COMPILING IMPERATIVE AND FUNCTIONAL LANGUAGES

JEREMY W. SHERMAN

Submitted to the
Division of Natural Sciences
New College of Florida
in partial fulfillment of the
requirements for the degree
Bachelor of Arts
under the sponsorship of
Dr. Karsten Henckell

Sarasota, Florida

May 2008

Jeremy W. Sherman: *Compiling Imperative and Functional Languages*

THESIS ADVISOR

Dr. Karsten Henckell

LOCATION

Sarasota, Florida

DATE

May 2008

To my family, whose unwavering support and strong belief in the value of education made this possible.

To my fiancée, whose confidence in me never faltered.

COMPILING IMPERATIVE AND FUNCTIONAL LANGUAGES

Jeremy W. Sherman

New College of Florida, 2008

ABSTRACT

Computers operate at a very low level of abstraction. People think in terms of higher-level abstractions. Programming languages let people describe a computation using higher-level abstractions in such a way that the description can be translated into something a computer can execute. This translation is performed algorithmically by a program called a compiler.

This thesis looks at how a compiler carries out this translation for two very different types of programming languages, the imperative and the functional. Imperative languages reflect the concept of computation that is built into modern von Neumann computers, while functional languages conceive of computation as a process of symbolic rewriting. The functional model of computation is utterly different from the von Neumann model, but programs written in functional languages must ultimately run on von Neumann machines.

The thesis focuses throughout on optimizing program representation for execution on modern von Neumann computers. A case study of the Glasgow Haskell compiler provides a concrete example of functional language compilation.

Dr. Karsten Henckell

Division of Natural Sciences

ACKNOWLEDGMENTS

This thesis is the capstone of four years of education. I would like to thank the various people and organizations that helped make this education possible: my parents for their constant support; AT&T (and before them, the SBC Foundation), the CWA Joe Beirne Foundation, First Data Corporation, the German–American Club of Sun City Center, and the New College Foundation for their financial support; and Saint Louis Priory School of Creve Cœur, Missouri for providing me with a solid grounding in the liberal arts.

I would also like to thank my baccalaureate committee, Drs. David Mullins, Patrick McDonald, and Karsten Henckell, for their participation in my baccalaureate examination, for the time they spent reading this work, for their comments, for the many wonderful classes they taught in which I had the chance to take part, and for their advice.

The advice of Dr. Karsten Henckell in particular proved invaluable while I was writing this thesis. Without his counseling on the direction and shape of my thesis, as well as his careful reading and comments, I would never have finished writing this thesis. Katelyn Weissinger and Marvin Sherman read drafts of parts of this work, and their comments and corrections did much to improve it. Any faults that remain are mine alone.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Computers and Programs

Computers are everywhere. Their rapid development, penetration into more and more aspects of our daily lives, and increasing effect on our world are the subjects of rumor, discussion, and daily news. What computers can do is simple: we can capture the essential elements of a computer in an abstract machine whose description takes up maybe ten pages. Within that concise, abstract specification hide volumes of details necessary to bring to life the efficient machines that are the wonder of our time. Computers do very little, but they do it extraordinarily well.

With computers came programming languages. The computer itself supports only a rudimentary, primitive language. This language describes everything in terms of the computer's hardware. It provides very few abstractions that hide these implementation details. The development of higher-level languages that support a much richer set of abstractions has been essential to realizing the potential of the computer.

The term PROGRAM is overloaded with meanings. It can refer to a computer-executable file, a specific application that can run on a variety of computers using a variety of computer-specific executables, or the code written in a programming language that is meant to become one of these other sorts of programs.

The concept that hides behind these different uses is that of the program as an idea of a computation, which could be something as abstract as "find derivatives of polynomials." In reifying this idea, one must make many implementation decisions. What algorithms should

be employed? How should we represent the information we are working with? In answering these questions, we draw on other, more elementary programs.

But, eventually, one must commit to a form for these programs, some sort of concrete representation. In their most authoritative form, these representations consist of executable specifications of the computation: "code" written in some language that can be made to run on a computer. Languages with a richer set of abstractions – higher-level languages – are a natural choice for the concrete representation, as they admit a more direct translation from the abstract idea.

A Tale of Two Stories

But, in the end, computers still speak computer, not these other, more human- and idea-friendly languages. The story of how a program represented in a higher-level language is transformed into a representation that a computer can not only carry out but that is well-suited to this purpose is an amazing, rich, nuanced story. The architecture of the computer determines whether a representation is well-suited for execution by it or not, and so this plays a part in this story. The abstractions provided by the higher-level language determine what sorts of transformations must be performed, so these too play a part in the story.

This story is the story of the compiler, the program that is responsible for carrying out the translation from higher-level language to machine language. It is also the central story of this thesis. We tell it by describing the major players: the computer, the compiler, and the languages. We discuss them, in fact, in roughly that order. It might seem backward to talk of compilers before languages. We actually assume throughout that you have at least a basic reading knowledge of a programming language such as C or Java, though we also provide analogies to natural language (such as English) where possible

in our discussion of compilers. Ultimately, we talk of compilers before languages because the job of a compiler is roughly similar for all languages, but the languages themselves differ in interesting ways that have a significant impact on the specifics of their compilers.

The birth, life, and death of programming languages also make for fascinating reading.* New languages build on and refine older languages while introducing novel ideas of their own. We can even talk of programming language genealogy and draw out family trees.†

Two of the oldest and most prolific trees belong to the imperative and functional families of programming languages. The abstractions offered by these families are sometimes quite similar, but the overall combination of abstractions differ in significant ways. These differences are truly fundamental: the two trees are rooted in different notions of the fundamental process of computation.

These notions are embodied in two different formalisms, the Turing machine and the lambda calculus. The universal Turing machine, a Turing machine capable of carrying out the computation of any other Turing machine, was the inspiration for the von Neumann machine that led to today's computers. The von Neumann machine, in turn, engendered the birth of the imperative language family. Thus, the translation from a higher-level, imperative language to a von Neumann computer's very low-level language can be looked at as a translation from one imperative language to another.

The lambda calculus, on the other hand, embodies a radically different notion of computation. Its heritors, the functional family, can be thought of in good part as higher-level versions of the lambda calculus. Translating these languages into the lambda calculus, then, is similar to translating imperative languages into machine language.

---

* If you are interested, you might want to start with the proceedings of the few history of programming languages (HOPL) conferences that have taken place.

† Lambda the Ultimate (`http://lambda-the-ultimate.org/`) has a good collection of links to genealogical diagrams.

But lambda calculus is not machine language, and so an important element of compiling all functional languages is effecting this paradigm shift: taking the representation of a computation defined in terms of the lambda calculus and turning it into a representation executable by a von Neumann machine.

The Neverending Story

The conclusion of this thesis is about a story that has yet to be written. Or perhaps it would be more exact to say, that we are writing now. For the last part of this thesis is about what is to become of our two families. In it, we will put the families side by side. We have seen where they have been, and some of where they are now. The final question is one you can help answer: what are they to become?

Part I

BACKGROUND

1

OVERVIEW

Before we can discuss compiling functional languages, we must set the scene.

BEGINNINGS looks into where the imperative and functional paradigms began.

COMPUTERS outlines the structure of modern computers with an emphasis on those features that particularly affect the design of compilers.

COMPILERS introduces compilers, including their architecture and associated theory, and concludes with a discussion of bootstrapping a compiler.

2

BEGINNINGS

2.1 A STICKY ENTSCHEIDUNGSPROBLEM

The DECISION PROBLEM was an important problem in twentieth-century mathematical logic.* It addresses the fundamental question of what we can and cannot know. There are many ways to pose the decision problem, or ENTSCHEIDUNGSPROBLEM as it was often called. One formulation was given by Hilbert and Ackermann in their 1928 book *Principles of Theoretical Logic.* They call the dual problems of determining the universal validity and determining the satisfiability of a logical expression the DECISION PROBLEM. The problem is solved when one knows a "process" that determines either property of any given logical expression in first-order logic. The particular first-order logic they had in mind was that propounded in their book on the restricted function calculus, later called the restricted predicate calculus.They were not able to be so clear about what they meant by "process."

By the 1930s, not only was the nebulous idea of a process formalized, but the decision problem had been solved in a way unanticipated by Hilbert: it was impossible to provide such a process.

The idea of a process was formalized three ways:

- the theory of RECURSIVE FUNCTIONS

* For example, it is intimately bound up with Hilbert's tenth problem:

> Given a Diophantine equation with any number of unknown quantities and with rational integral numerical coefficients: *To devise a process according to which it can be determined in a finite number of operations whether the equation is solvable in rational integers.*

- the LAMBDA CALCULUS
- the TURING MACHINE.

From lambda calculus springs the functional paradigm, while the Turing machine inspires the imperative paradigm.

2.2 CHURCH AND HIS CALCULUS

Church developed the lambda calculus in hope of providing a logical basis for all of mathematics. While he was ultimately frustrated in this, he succeeded in creating a rich framework for both logic and, eventually, computer programming.

The lambda calculus formulates computation as term rewriting and distills the concept of the function to textual substitution. The text comes in the form of LAMBDA TERMS; the rewriting comes as REDUCTION RULES. To define the set of lambda terms Λ, we seed it with an infinite set of variables $V = \{v, v', v'', \ldots\}$ and then further admit all expressions built using two operations, APPLICATION and ABSTRACTION:

$$\begin{aligned} x \in V &\implies x \in \Lambda \\ M, N \in \Lambda &\implies (MN) \in \Lambda \quad \text{(application)} \\ M \in \Lambda,\, x \in V &\implies (\lambda x M) \in \Lambda \quad \text{(abstraction)} \end{aligned}$$

The fundamental reduction rule of the lambda calculus is that of β-REDUCTION: the application of an abstracted term $\lambda x M$ to another term N can be replaced by M with N substituted for every occurrence of x throughout M, or, written more symbolically, $\forall M, N \in \Lambda$,

$$(\lambda x M)N = M\,[x := N]\,.$$

We will have more to say about the lambda calculus later in Chapter 12, THEORY. For now, we will content ourselves with pointing out that N in $(\lambda x M)N$ may be *any* other lambda term, including another abstracted term; the only distinction between "functions" $(\lambda x M)$ and "literals" v, v', etc. is that "functions" provide opportunities for β-reduction.

2.3 TURING AND HIS MACHINES

Turing was working expressly to address the *Entscheidungsproblem*. He formalized computation by way of an abstract machine. A "process" is embodied in a machine. In the case of the decision problem, it would accept logical expressions – instances of the decision problem – as input.* If there were an algorithm for the decision problem, the machine would then be able determine the answer for all instances. Instead, Turing found that any such machine would never be able to decide whether all possible input instances are or are not satisfiable; the decision problem is fundamentally UNDECIDABLE, which is another way of saying it is not computable.†

Turing's machines look very much like a high-level sketch of our modern von Neumann machines. They consist in a finite control (the program), a read-write head, and an infinitely long tape (the memory). The tape is divided into cells: each cell is either marked with a symbol or blank. The problem instance is written on the tape and the machine started; if it comes to a halt, the state it is in indicates the yea-nay–result of the computation. The final contents of the tape can be used to communicate actual details of the answer: for the decision problem as given above, the final state could be used to indicate that, yes, the in-

* These expressions would, of course, have to be suitably encoded for its consumption.
† This is not to say that some individual instances of the problem are not decidable, but that there is no solution to the problem as a whole.

put instance is satisfiable, while the tape could contain Boolean values satisfying the equation.

Formally, we can treat a Turing machine as a six-tuple $(Q, \Sigma, B, \delta, q_0, F)$:

Q is the finite set of states the control can be in.

Σ is the finite alphabet available for writing the input on the tape.

B is a distinguished blank symbol that cannot be part of the input; prior to placing the input on the tape, the tape is nothing but an endless sequence of cells filled with B.

δ is a state transition function, $\delta\colon (\Sigma \cup B) \times Q \to (\Sigma \cup B) \times Q \times D$, describing how the Turing machine reacts to reading a symbol σ in state q:

- it writes some symbol, either the blank symbol or an input symbol;
- it moves from its current state to some state in Q, possibly the same state; and
- its head moves some direction, either left L or right R (that is, $D = \{L, R\}$).

q_0 is the initial state of the machine.

F is the set of ACCEPTING STATES; $F \subset Q$, and if the machine concludes its computation, that is, HALTS in some state in F, this indicates an affirmative answer to the question posed it. The computation is concluded when the machine can make no further move, which occurs when $\delta(\sigma, q)$ is undefined.*

In the Turing machines' sequential operation and reliance on changes in their state and data store to perform computation, we find the roots of the imperative paradigm. Even more plain is the resemblance to our modern-day von Neumann computers.

* This makes δ a partial function. We can restore its totality by introducing the possibility of transitioning to a distinguished HALT action, but this is not really necessary.

3

COMPUTERS

3.1 FROM ABSTRACT TURING MACHINES TO CONCRETE COMPUTING MACHINES

A Turing machine takes some input, acts on it, and, if its computation terminates, produces some output. For example, we can specify a Turing machine that takes as input a natural number and checks whether that number is even or odd, and we can guarantee that it will always halt with an answer. To compute the solution to another problem, we must specify another Turing machine. This is fine when we are working with paper and pencil, but Turing machine computations executed via paper and pencil offer no advantage over any other work with paper and pencil and have the disadvantage of being exceedingly tedious. What if we wanted to move beyond paper-and-pencil machines and manufacture machines that perform these computations in the real world, machines that will not become bored and make a mistake, and, further, can carry out the computations much faster than we? In that case, producing a separate machine for every computation would not be of much use. Indeed, what we need is a universal machine, a single machine capable of computing anything any Turing machine can compute.

This UNIVERSAL TURING MACHINE would accept as input the description of another Turing machine and data for that machine to operate upon and then simulate the operation of the input machine on the input data. By devising an encoding for the description of a Turing

machine that can be processed by a Turing machine, we can build this abstract machine. What remains is to build the concrete machine.

What parts would such a machine need? From a user's perspective, any Turing machine performs three primary activities:

- accept input
- perform the computation for this input
- produce output.

Two of these steps involve communicating with the user; one is entirely internal to the machine. When we move to a universal Turing machine, what was once internal becomes external: the need to simulate the action of another Turing machine demands some way to store the description of the Turing machine while simulating it.

Considering Turing machines has in fact brought us to the essential parts of a modern computer:

- means of accepting input and communicating output
- storage for input, both programs and data
- facilities to process instructions and data.

This chapter will describe these three fundamental divisions of a computer with a particular emphasis on aspects of their implementation that affect compilation.

Before we move on, let us take one last look at Turing machines in light of this list. The processing facilities of the universal Turing machine are its transition function and states operating per the definition of Turing machines. Input-output facilities are not actually part of the Turing machine: input appears on the tape, computation occurs, and we somehow observe the final state and tape contents of the Turing machine. The universal Turing machine's storage is its tape. It is interesting that both data and program (the description of the machine

to be simulated) lie on the same tape. A single memory for both instructions and data is the hallmark of the VON NEUMANN ARCHITECTURE and distinguishes it from the HARVARD ARCHITECTURE, which uses separate memories for program and data.* While Turing machines are a tremendously useful model for computation in the abstract, and while they also serve surprisingly well to bridge the conceptual distance from the abstract model to the concrete machine, that is as far as they will bring us. In the rest of this chapter, we will be talking about time-bound, space-bound, hardware computers, the clever methods used to reduce the limitations introduced by reality, and how those methods affect compilation.

3.2 PROCESSOR

The processor is the brain of the computer. It is responsible for reading instructions, decoding and dispatching them for execution, and directing the other functional units in their operation. It does this repeatedly in a rapid instruction-fetch–decode–dispatch cycle: processor performance is often measured in terms of millions of instructions per second (MIPS), as well as in terms of the average cycles per instruction (CPI). The time to execute an instruction varies from a few cycles (simple arithmetic operations) to millions of cycles (loading values from memory). The processor keeps track of which instruction to fetch next via its PROGRAM COUNTER (PC). Every time it fetches an instruction, it increments the PC to the address of the location of the next instruction.

* To be fair, it is possible to define universal Turing machines naturally homologous to both of these architectures; it is simply our exposition that makes the von Neumann appear the more natural.

3.2.1 *The Fetch-Decode-Dispatch Cycle*

The processor only understands binary digits, or BITS. The instructions themselves are simply distinguished sequences, or STRINGS, of bits with an agreed upon meaning. The bit strings are given meaning as part of an INSTRUCTION-SET LANGUAGE (ISL). Each instruction-set language is used to talk with a specific INSTRUCTION-SET ARCHITECTURE (ISA). Each processor is an implementation of some instruction-set architecture and understands the instruction-set language designed for that architecture. In a sense, the instruction-set architecture is a description of an interface between an instruction-set language and a particular processor. It leaves the details unspecified, and this is where the various processors implementing a particular instruction-set architecture differentiate and distinguish themselves. As a loose analogy, consider a caller ID unit. It has to be able to connect to the phone system, and it has to speak the same language as the phone system to be able to receive the information it displays, but beyond that it is free to vary its shape, size, and the number of callers it can remember, among other things.

The processor computes according to the instructions it is given. It executes the instructions one after another. Before it can follow an instruction, it first has to get it, that is, the processor must FETCH the instruction. Next, it must read and understand it. This process of looking over and understanding an instruction is called INSTRUCTION DECODING. The first step of decoding an instruction is to recognize what sort of instruction has been fetched. Various sorts of instructions have various parts (called OPERANDS) relevant to what the processor is supposed to do; for example, an instruction to read in a location in memory will have to specify the location. After the processor understands these parts of the instruction, the processor has completed

decoding the instruction. The instruction can then be DISPATCHED for execution and the next instruction can be fetched by the processor.

What sorts of instructions are there? Instructions generally cover arithmetic operations (add, subtract, multiply, divide) on integers and floating point numbers,* shifts (left and right – the bitstring `001` shifted left by 2 becomes `100`), logical operations (and, or, not, exclusive or), jumps (instructing the processor to change its PC and begin fetching instructions at another location) and conditional branches (jumps that are predicated on a particular condition, like a register's being nonzero), and memory operations (load from memory into a register, store from a register into memory, and possibly copy from register to register or memory location to memory location). The way conditional branches are supported and the types of conditional branch instructions provided vary from processor to processor, as do the ways that memory locations can be specified. Many other operations may be provided, such as those meant to deal particularly with strings of alphabetic characters coded into bits in one way or another or instructions meant to deal with numbers encoded as binary-coded decimal rather than as binary numbers. Each instruction set is different.

3.2.2 *Functional Units*

The instructions themselves are carried out by other functional units. Many of the arithmetic operations will eventually be routed through an arithmetic logic unit (ALU). Those dealing with floating point numbers, however, are likely to be sent to either a floating point unit or even a floating point co-processor.

Storage for operands is frequently provided by REGISTERS. Registers may be either special-purpose (available for use only in floating

* Floating-point numbers are the computer equivalent of scientific notation. The "point" of "floating point" is the decimal point, whose position relative to the significant digits (those that we actually bothered to write down) can be changed by varying the exponent.

point operations, for example, or devoted to storing the PC or the number zero), in which case they are likely to be divided into REGISTER CLASSES, or general-purpose (available for any use in any instruction). Others may be functionally general-purpose but reserved for use by the operating system or assembler. The trend has been to greater numbers of general-purpose registers. Certain registers are exposed to the programmer via theinstruction-set language and guaranteed by the instruction-set architecture, but the implementation is likely to make use internally of far more registers. If all the operands of an instruction must be in registers, the instruction is said to be register-register. Some instruction sets have register-memory or even memory-memory instructions, where one or even all operands of the instruction are in memory. This was particularly common in the past.

3.2.3 *Assembly Language*

Bit-string instructions are fine for machines, but they are difficult for humans to work with. For this reason, ASSEMBLY LANGUAGES were developed. Assembly languages represent the instructions with alphabetic abbreviations such as `j` for *jump*, `beq` for *branch if equal*, or `add` for *add*. They will often allow the use of textual labels for the specification of branch and jump targets and the use of names for registers as opposed to purely numbers. They will accept directives as to the alignment of data in memory and convert character strings to bit strings in a particular encoding rather than requiring the programmer to perform the conversion. They might also provide greater levels of abstraction, such as providing pseudoinstructions like a richer set of branch conditionals that can be readily translated into the processor-

provided instructions or allowing the programmer to define macros* for common code sequences.

The ASSEMBLER is responsible for translating this symbolic instruction language into the binary instruction set language. It assembles assembly language code into OBJECT CODE executable by the processor. Its action is that of a compiler, though its job is generally much simpler than that of compilers for higher-level languages whose level of abstraction is much farther from the underlying processor.

3.2.4 *Types of Processors*

There have been many types of processors, but the two dominant types are the COMPLEX INSTRUCTION SET COMPUTERS (CISC) and the REDUCED INSTRUCTION SET COMPUTERS (RISC).

CISCs were developed when computing resources were very limited and most programming was done in assembly languages. Since the instruction set language itself was programmers' primary interface to the machine, it seemed worthwhile to provide higher-level instructions that accomplished more complex goals, such as copying an entire string of characters from one place in memory to another or repeating an instruction a number of times determined by a counter register. They also frequently used variable-length instructions, to minimize the amount of space required by instructions – more complex instructions would frequently require more information, and so more bits, than simpler instructions.

The complex instruction sets of CISCs were meant make it easier for people to program in assembly language. With the development of higher-level languages and compilers, these features were no longer necessary. In fact, compilers of the time were unable to take full ad-

* So-called by abbreviation of "macro-instruction." These are "big" instructions that stand in for and eventually expand out into a longer or more complicated sequence of instructions. The instructions might also be capable of accepting arguments to be substituted for placeholders throughout the expanded sequence.

vantage of these complex instructions and often generated instead equivalent sequences of simpler instructions. More complex instructions also require more complex logic and so more space within the processor. Supporting the complex instruction set slowed the processor and made it difficult to take advantage of more advanced processor design ideas. By reducing the complexity of the instruction set, the amount of space needed in the processor to implement the instruction set could be reduced, enabling the inclusion of greater numbers of fast registers. RISCs capitalized on these observations. One of the most noticeable simplifications of their instruction sets is the elimination of memory-memory, register-memory, and memory-register operations beyond those necessary to load data from memory to registers and store data to memory from registers, leading to the characterization of RISC architectures as LOAD-STORE ARCHITECTURES.

RISC ideas have been highly influential, and RISC processors are often used in embedded situations, such as in cell phones, PDAs, washing machines, automobiles, and microwaves. However, when IBM elected to go with Intel's 8086 series CISC processors rather than Motorola's 68000 series RISC processors for its personal computers, it set the stage for the x86 and its successors to become the most common non-embedded processors. Modern CISC processors, such as those made by Intel and AMD, integrate elements of the RISC philosophy into their designs while preserving compatibility with older x86 software. The CISC instructions are often translated by the processor into RISC microcode, which is then executed by the processor. Many of the more "CISCy" instructions have been preserved for compatibility but allowed to "atrophy" to where they will execute much more slowly than a longer sequence of simpler instructions accomplishing the same result. This blending of RISC and CISC ideas, which eliminates any clear distinction between the two approaches to processor design, has brought us to what might be called a post-RISC era.

3.2.5 *Issues Particularly Affecting Compilation*

When you compile software, you want it to run as well as possible on your computer. Often, this means as fast as possible, and as much work has gone into making the processors themselves run as fast as possible, the processor provides a lot for a compiler writer to worry about.

Registers

The number of registers available for compilation affects the usefulness of various optimizations as well as the speed of the compiled code. Storing to memory is slow, while registers are fast: the more data that can be kept in register, the better. Thus, the more general purpose registers available to the compiler when it comes time to allocate the generated code among the available registers, the better. Some processors provide a means to utilize the greater number of implementation-supplied registers; the REGISTER WINDOWS of Sun Microsystem's SPARC (Scalable Processor ARChitecture) machines are one example. As mentioned above, some architectures will specify that various registers are reserved for various purposes and unavailable to the compiler. The architecture might also specify how registers are to be treated during a procedure call by defining a CALLING CONVENTION.* All of this directly affects the code generated by a compiler targeting the architecture.

Historical Accidents and Implementation-Specific Extensions

As mentioned above, while the instruction set language might provide a special-purpose instruction for handling, say, string copying, this might actually execute slower than a sequence of simpler instruc-

* Even if an architecture has nothing to say about register usage in procedure call, a programming language specification might specify a calling convention in an attempt to guarantee interoperability between programs written in the language.

tions accomplishing the same thing. Thus, it is not sufficient to be familiar with the specification of the instruction set language or even of the instruction set architecture: decisions made in the implementation of the specific processor can affect choices made during compilation (or at least in some cases should). Other operations retained for compatibility might also be best avoided.

At the same time, various implementations will extend the instruction set in ways their designers hoped would be of use to compiler writers. Examples are the streaming SIMD extensions* added to various successors of the x86 architecture by Intel and AMD meant to speed up code compiled for multimedia applications.

Pipelining and Speculation

In an attempt to speed up instruction execution in general, modern processors are deeply PIPELINED. Pipelining exploits INSTRUCTION-LEVEL PARALLELISM. Rather than decoding an instruction, dispatching it, and waiting for its execution to complete before beginning to execute the next instruction, instructions are decoded immediately, one after another, and dispatched, so that their execution overlaps. Pipelining does not decrease how long it takes an instruction to execute, but it does increase the number of instructions that can be executed per unit of time, making it appear that instructions are executing faster.

The depth of the pipeline places an upper limit on the number of instructions whose execution can overlap. However, various hazards of an instruction sequence can prevent an instruction from completing every cycle, and so prevent a new instruction from being dispatched each cycle. This causes a decrease in the number of instructions that can be executed in a given time period, a quantity referred to as INSTRUCTION THROUGHPUT.

* Generally further abbreviated to SSE, SSE2, etc. for the various generations of extensions. The SIMD part stands for "single instruction, multiple data." An example of such an instruction would be an add operation that specifies the addition of two vectors (likely representing a point in three-dimensional space), each made up of several data components.

STRUCTURAL HAZARDS occur when multiple instructions require simultaneous use of the same functional units.

DATA HAZARDS occur when an instruction requires data that is not yet available.

CONTROL HAZARDS occur when the program counter is to be altered, but how it will be altered is not known sufficiently in advance to keep the pipeline full.

All of these hazards cause STALLS, also known as BUBBLES – space in the pipeline where an instruction would be executing, except that it cannot.

Various methods are employed to address these hazards. Structural hazards can be addressed by carefully designing the instruction set and the processor. Within the processor, data hazards are addressed by FORWARDING. Forwarding diverts data to where it is needed as soon as it is available, rather than waiting for it to become available through the usual means. For example, if an instruction is waiting on a value to be loaded from memory to a register, rather than waiting for the value to enter the datapath and finally be committed to register before loading it from the register, forwarding will make the value available as soon as it enters the datapath. It will eventually be stored to the targeted register, but in the mean time, the instruction that was waiting on the value can continue execution.* Outside the processor, data hazards are partially addressed by the introduction of a memory hierarchy, which we will discuss in Section 3.3, Memory.

Control hazards are addressed by BRANCH PREDICTION. This can be as simple as always assuming a branch will not be taken or more complex, such as dynamically tracking whether the branch is taken

* Some instruction sets expose the delay following branch (and load-store) instructions to the user in what is called a DELAYED BRANCH: the next (or next several) instructions following a branch instruction are always executed. As pipelines deepened, exposing the delay to the user became less and less feasible, and successors of such instruction sets have often phased out such instructions in favor of non-delayed versions.

more often than not during execution and acting accordingly. The processor begins executing instructions as if the branch had gone the way it predicted; when the branch is finally decided, if it goes the way that was predicted, there is no stall. If it goes the other way, there must still be a stall. Further, the processor must be able to undo all the instructions executed along the path not taken.

This is a specific example of a more general technique called SPECULATIVE EXECUTION, in which a guess is made about some property of a part of the instruction stream, execution continues as if the guess were correct, and then is either confirmed or undone depending on whether the guess was correct or not. This is useful not only for branches, but for reordering the instruction stream in general to increase instruction-level parallelism.

Multiple Issue Processors

Another way to address these hazards and improve performance in general is to move to MULTIPLE ISSUE processors. Rather than issuing at most a single instruction each cycle, multiple issue processors issue multiple instructions whenever possible. They are able to do this because much of the datapath has been duplicated, perhaps multiple times. Instructions can then truly execute simultaneously rather than simply overlapping. Clever methods are employed to ensure the appearance of sequential execution is not violated and to resolve dependencies between instructions. However, instruction sequences that have been optimized to maximize instruction-level parallelism will run faster; an optimizing compiler will take advantage of this.

In fact, in STATIC MULTIPLE ISSUE processors, compilers have no choice but to take advantage of this, as the processor itself merely performs multiple issues by following instructions. The burden of scheduling instructions to maximize instruction-level parallelism and taking advantage of the architecture falls entirely on the compiler. This has

the advantage of enabling the compiler to leverage its knowledge of the properties of the entire program to demonstrate that reordering and simultaneously scheduling instructions is safe, but it has the disadvantage of directly exposing much of the internal design of such a processor, so that a program is more likely to have to be recompiled in order to run on even a slightly different processor.

DYNAMIC MULTIPLE ISSUE, or SUPERSCALAR, processors attempt to exploit instruction-level parallelism at runtime. As they read in instructions, they reorder them, issue multiple instructions whenever possible, and speculatively execute instructions when they can. Since all of this goes on "behind the scenes," a compiler can completely ignore it and still produce runnable code. At the same time, a sequence of instructions tailored for a specific processor can maximize the amount of instruction-level parallelism exploitable by that processor. Thus, unlike with static multiple issue processors, knowledge of the specific implementation of an instruction-set architecture using dynamic multiple issue is advantageous to the compiler but is not necessary to produce code that will run, and code that runs on one implementation of the instruction-set architecture should run sufficiently well on all other implementations of the same, regardless of whether or not dynamic multiple issue is employed.

Not only do pipelining, speculation, and multiple issue greatly complicate the development of a processor, they also make it more difficult to predict how generated code will be executed, as well as placing great emphasis on optimizing for instruction level parallelism. Examples of the effect these have on compilers are the efforts taken to minimize the number of branches and keep values in register as long as possible, though this latter is even more severely motivated by the presence of a memory hierarchy.

3.3 MEMORY

A processor is nothing without memory. In the course of computing, the processor needs some way to store the data it is computing with. This necessitates some amount of temporary memory. Moreover, we should like for this memory to be very fast, so that computations not take an inordinate amount of time. However, we should also like to introduce a more persistent form of memory. This could take the form of something rather non-technical, such as punched cards, or something more technically sophisticated, such as a hard drive. We should also like this persistent storage to be as fast as possible, but we would be willing to sacrifice speed for capacity.

It is not surprising, then, that there should arise a definite memory hierarchy. This hierarchy is organized as a pyramid with the processor at its apex: memory closer to the processor can be accessed more quickly but can store much less (for reasons of both cost and space), while memory further from the processor is slower but much more capacious; by the time we reach the lowest level, SECONDARY STORAGE, the capacity has become virtually unlimited. This severe difference – fast and small on the one hand, slow and large on the other – is fine so long as we can restrict our computations to use the faster memory and leave the slower purely for storage. But this is rarely possible.

In order to reduce the need to use slow, secondary storage, we exploit the frequent fact of spatial and temporal locality in memory accesses to promote contiguous blocks of secondary storage into CACHES – faster but less capacious memory that mirrors the contents of secondary storage. Caches are organized into levels based on how close they are to the processor, and so how fast and small, generally from level one (L1) to at most level three (L3). Memory accesses first attempt to hit on the desired memory in cache; only if that fails do they have to resort to disk. The time to realize that the attempt to hit the memory in

the cache has failed is called the MISS PENALTY. This penalty is only introduced because of caching, but without caching, one would pay the far higher price of always having to wait for the disk to respond.

One common way to reduce both the miss penalty and the time to hit is to restrict the number of places a given block of memory can be placed in the cache. If the data at an address can be stored in any cache line, one must check every cache line to be sure that the data is not there. Such a cache is called FULLY ASSOCIATIVE. If the data at an address can only go in exactly one line, one can readily determine whether or not it is in the cache. Such a cache is called ONE-WAY SET ASSOCIATIVE. However, because each level of the memory hierarchy is smaller than the one below it, there are always fewer cache lines than there are blocks of memory that could need to be stored in the cache. Limiting each block to being stored in only one, or only two, or any number of places fewer than the number of lines in the cache introduces a competition for those limited number of places among the blocks of memory that can only be stored in those places. This is in addition to the necessary competition for being in any line at all of the space-limited cache that occurs even in a fully associative cache.

While caches seek to exploit spatial and temporal locality, precisely how is a matter of some delicacy, with no clear best solution. One can attempt to reduce the miss penalty or time to hit, increase the capacity of the cache, improve its strategy for loading in a new CACHE LINE (the unit of storage within the cache, amounting to a fixed number of bits) from disk and its selection of a line to evict, but one cannot do all of these at once. Multilevel caches only further complicate things.

To help in thinking about such issues, one can characterize the types of cache misses through the "three Cs":

COMPULSORY MISSES occur on first access to a block of memory because a memory access has to miss before its line can be brought into the cache. They cannot be avoided, though they can be pre-

vented from causing a delay by PREFETCHING, which provides a way to indicate that a given block of memory might be needed in the future and should be loaded now.

CAPACITY MISSES occur when a previously evicted block is requested again because a cache can only store so many lines of memory. They can be decreased by increasing the size of the cache.

CONFLICT MISSES occur only in non-fully associative caches, when a block that was evicted by another block competing for the same set of cache lines is requested again. They can be decreased by increasing the associativity of the cache.*

Of all of these, the one that has the most direct effect on compilation is compulsory misses, provided prefetching is available. Otherwise, it is simply the existence of a memory hierarchy and its workings that affect a compiler. These details can make some data structures more efficient than others, affecting how the compiler codes the runtime behavior of the program. It also makes some uses of memory to store data during the program more efficient than others: use of one memory layout or another also falls to the compiler.

The existence of a memory hierarchy has a major effect on both compilation and compiler design. It affects compilation by increasing the need for and desirability of optimizations to increase spatial and temporal locality of memory accesses, reduce the need for storage, and confine space needs to registers internal to the processor as much as possible. It affects compiler design not only because of its effects on the code a compiler must generate, but also because the memory hierarchy has an effect on the behavior of the data structures and algorithms used to implement the compiler. Most algorithms are developed, either intentionally or naïvely, in a FLAT MEMORY MODEL that assumes unlimited fast memory. As soon as one begins considering the effect

* While competition occurs between all blocks for all cache lines in a fully associative cache, the misses that occur due to that competition are classed as capacity misses.

of the memory hierarchy on the data structures and algorithms used, formerly optimal implementations may no longer be so.

Early attempts to develop algorithms and data structures within the context of a memory hierarchy used the DISK-ACCESS MODEL, which parameterizes its algorithms based on properties of the memory hierarchy such as the block size of and number of blocks in the cache (also called the width and height of the cache). These parameters are often not available and difficult, if not impossible, to determine at runtime. Introducing this explicit parameterization also makes code less portable and maintainable. Further, the model presumes fine-grained control over the behavior of the cache and storage that frequently is not available.

The later CACHE-OBLIVIOUS MODEL addresses these problems: while proofs of the behavior of its algorithms and data structures are by necessity parameterized, its data structures and algorithms are not, and behave well so long as at least a two-level memory hierarchy exists that can be modeled in the fast-slow small-large fashion appropriate to the two levels of cache and storage. Such a relationship exists generally between all levels of a memory hierarchy, so this suffices to guarantee the desired performance. This guarantee can and has been made precise in a formal fashion; for details, consult the Bibliographic Notes for this chapter.

3.4 INPUT-OUTPUT

Input-output is the computer's interface with the outside world. It encompasses everything from keyboards and monitors to network connections, disk access, and, in some sense, communication and synchronization between multiple PROCESSES (that is, currently running programs with their own memory context and position in their instructions) and between multiple THREADS of execution. (Threads make

up a process: each process has at least one thread of execution, but it might also divide its work between multiple threads.) Input-output requires implementation support. Input-output is primarily supported in one of two ways, through busy-wait ("spin blocking") protocols and through the use of interrupts.

In a BUSY-WAIT input-output protocol, the computer provides a means for a program to indicate it wishes to perform output or receive input from a device. A signal is used to indicate the status of the input-output "channel": once it is ready to accept output or provide input, the signal is set to indicate this. On performing the desired action, the signal is reset. This requires constant polling of the signal to see whether its status has changed. The process performing this polling is unable to proceed while it cycles between checking whether the status has changed and waiting between checks. A process behaving in such a way is said to be SPIN BLOCKING. Signals are also often provided by the processor for use in synchronizing the actions of processes using shared-memory concurrency. Atomic operations such as test-and-set might also be provided to help support concurrent execution.

Input-output can also be synchronized via INTERRUPTS. Interrupts are a sort of "unexceptional exception" in that they frequently make use of the processor's facilities for handling EXCEPTIONS (such as division by zero and numeric over- and underflow) but are only exceptional in the sense that they require an immediate, real-time response and a switch in context from the currently running process to a special exception-handler that will perform the input-output. (Interrupts are also often used by operating systems to provide access to the system routines they provide.) How interrupts and exceptions are supported and the types of interrupts and exceptions supported vary from processor to processor, but such facilities are common in most, if not all, modern processors because of the advantage of such an implementation over busy-wait. While use of interrupts is preferable to busy-wait,

it is not always possible. For example, communication over a network as occurs in using the Internet is frequently handled via busy-wait since most of the protocol stack is implemented in software.

Input-output is highly implementation-dependent and is a frequent source of complexity and difficulty both in the design of programming languages (particularly functional languages, for reasons to be discussed later) and processors. It is also slow and time-consuming. Due to the complexity of input-output, however, many of the issues are often exposed to and expected to be managed at the program-level rather than the language-level, so input-output is not a frequent target of optimizations in the back-end, nor would it be likely to be a very fruitful target. Handling concurrency and parallelism, on the other hand, may be the responsibility of the compiler, particularly in parallel programming languages. This is frequently true of the quasi-concurrency of EXCEPTIONS included in many newer languages. Exceptions are programmatic realizations of exceptional conditions that can arise at runtime, such as division by zero or the inability to access a given file. Whereas programs would generally give up and abort execution on encountering such a problem in the past, exceptions make it possible to recover from exceptional conditions and continue execution. They also mean that execution might suddenly need to jump to an exception handler at unpredictable times. Where permitted by the language, choices made by the compiler in how to support concurrency and parallelism can affect program runtime and safety.

3.5 BIBLIOGRAPHIC NOTES

The universal Turing machine was introduced by Alan Turing in his seminal paper Turing [127]. If the idea particularly intrigues you, you might enjoy Herken [52], a collection of short papers related to and inspired by Turing's own.

Patterson and Hennessy [98] is a good introduction to the design and architecture of modern computers, including an influential RISC assembly language and familial variations thereof. Another work by the same authors, Hennessy and Patterson [51], goes into far more detail. Information on specific instruction-set architectures and languages is generally freely available online from the manufacturer.

For an introduction to the cache-oblivious memory hierarchy model, you could do no better than to start with Demaine [39]. This paper briefly surveys the history of memory hierarchy models, formally introduces the cache-oblivious model, and explores some of its essential data structures with proofs of their space behavior. It provides pointers to the relevant literature throughout.

4

COMPILERS

A COMPILER is a translator. As in translation, there is a source language and a target language, and it is the translator's job to analyze the source text and produce "equivalent" text in the target language. When translating human languages, the equivalency of the original and a translation is only rough because there are so many factors on which to evaluate equivalency. When it comes to programming languages, however, we are not interested in the niceties of meter or alliteration, nor do we care about any subtleties of connotation: we want to map a computation expressed in one language to a computation expressed in another such that both produce identical outputs when given identical inputs.

Beyond that requirement, everything else about the computation is fair game for alteration. Since the source language is often unaware of the peculiarities of the target language, many of the details of how exactly the computation should be carried out are unspecified and open to interpretation: 5×4 can be calculated by straightforward multiplication of 5 and 4, but if 5×2 is already known, we need only multiply that quantity by 2, and since these multiplications happen to be powers of 2, we could instead employ bit shifts in place of multiplication, especially as a bit shift is likely to take less time than multiplication. Even if the source language is also the target language, the original source code might still be improved by careful translation without altering its behavior in the high-level sense of input-output mapping discussed above.

Alterations that will preserve observable behavior are called SAFE: correspondingly, alterations that might not are called UNSAFE. A com-

piler will never voluntarily perform unsafe alterations, though some allow the human user to instruct them to do so, as might at times seem desirable for runtime checks that serve only to catch errors that slipped past the programmer or when the human is able to determine that a transformation that appears to the compiler to be unsafe will, in fact, be safe in this case. The compiler must act conservatively: it can only consider safe those transformations that it can prove to be safe, and it must assume the worst where it cannot prove the behavior to be any better. The user need not make such assumptions.

You might be wondering whether, if compilers are translators, is there a similar programming language analogue for human language interpreters? There is, and they are even called INTERPRETERS. They perform on-the-fly interpretation: rather than translating the code for future execution, they directly execute it. They are not able to perform the extensive analysis of the whole program that compilers perform. It is, in fact, this degree of analysis that particularly distinguishes a compiler, though translation for the future rather than for the present is also frequently a characteristic. This latter characteristic, however, is less apparent in more recent developments such as JUST-IN-TIME COMPILATION, which might be coupled with an interpreter as in the Java Hotspot virtual machine. Our subject is neither interpreters nor just-in-time compilation, however, so this concludes our first and last words on the two subjects.

A compiler is a translator: it analyzes the source code and produces equivalent target code. This suggests a decomposition of the compiler into two parts, one performing analysis and the other generating target code. The analysis part is commonly called the FRONT END, while the code generation part is called the BACK END. From our discussion of alterations and the relative speeds of multiplication versus bit shifts, you might also infer that the compiler also attempts, following its analysis, to improve the code in some fashion. This OPTIMIZATION

is at times folded into the front and back ends, but as the number of optimizations employed rises and the complexity of performing them alongside the work of the front and back ends increases, it becomes wise to dedicate part of the compiler to optimization alone. Optimization can only be performed following analysis: we cannot improve what we do not understand. At the same time, we should like to generate optimized code as directly as possible. This suggests placing the optimizer between the front and back ends. Due to this common decomposition of concerns and arrangement of the flow of control between the parts, the part concerned with discovering and performing optimizations is sometimes wryly referred to as the MIDDLE END of the compiler.

4.1 FRONT END: ANALYZING SOURCE CODE

The front end of the compiler is responsible for analyzing the source code. It takes a long string of characters (the program), discerns what each string is meant to be in terms of the "types of speech" of the programming language, figures out how the various parts fit together to form a valid program (or that they do not form a valid program, if the programmer has made an error!), and tries to infer the meaning of the parts. The first two steps are similar to the spell-checking and grammar-checking performed by a wordprocessor. The last step is one wordprocessors have not quite achieved just yet. An analogy along the same lines as the others, however, would be "sense-checking" or "sanity-checking," which would answer the question, "This is a syntactically valid sentence, yes, but does it mean anything, or is it nonsense?" As a whole, the front end of a compiler represents one of the great achievements of computer science: we have powerful formalisms that can be used to specify and automatically generate it.

4.1.1 *Lexical Analysis*

As presented to the compiler, the source code is a very long sequence of characters. This is the domain of LEXICAL ANALYSIS. A long sequence of characters does not mean much at the character-level, so the first thing the front end must do is proceed from characters to a more meaningful level of abstraction. The LEXER, which performs lexical analysis (and is also called, quite naturally, the LEXICAL ANALYZER), reads in characters and chunks them into TOKENS, strings of characters having some meaning at the level of the programming language's structure. These tokens are akin to parts of speech in spoken language – while the specific details of the token ("this identifier is formed by the string `engineIsRunning`") might be recorded for use in later stages, they are subsumed by the token, which treats, in a sense, all nouns as nouns, regardless of whether one is "cat" and one is "dog."

This tokenization is performed systematically by simulating the operation of a FINITE AUTOMATON that recognizes tokens. A finite automaton is, like a Turing machine, an abstract machine, but it is far simpler and far less powerful: a Turing machine can do everything a finite automaton can, but a finite automaton cannot do everything a Turing machine can.

Regular Languages

It turns out that we can describe all decision problems as LANGUAGE PROBLEMS. A language is a (potentially countably infinite) set of WORDS, and words are made up of characters from a finite ALPHABET by CONCATENATION, the "chaining together" of characters denoted by writing them without intervening space: concatenating a and b in that order gives ab. The decision problem recast as a language problem becomes, "Given a word and a language (and, implicitly, an alphabet), determine whether the word is or is not in the language." The languages

for which a Turing machine can solve this problem are known variously as RECURSIVE, DECIDABLE, and TURING-COMPUTABLE languages. The languages whose membership problems can be solved by a finite automaton, on the other hand, are known as the REGULAR LANGUAGES and form a proper subset of the recursive languages.

Finite Automata

A finite automaton is a constructive way to describe a regular language. Each finite automaton is associated directly to a language, the language whose membership problem it solves. Given a word, it solves this problem by examining the word one character at a time. After it has consumed all its input, it halts operation. Based on the state in which it halts, we say either that it ACCEPTS the word or rejects it. We build a finite automaton by specifying its makeup. A finite automaton is made up of a finite set of states and a transition function that describes how, in each state, the finite automaton responds to consuming the characters of the alphabet. In specifying a finite automaton, we also specify the alphabet of its language, the finite automaton's initial state, and the set of FINAL or ACCEPTING STATES, those states which, when the finite automaton halts in them, indicate acceptance of the word.

We can specify the states and transition function in two ways: either in a table, as in Fig. 1, or graphically through a TRANSITION DIAGRAM. A transition diagram has circular nodes for states, typically labeled with the state name, and arrows between states, which indicate the transition function. The arrows are labeled with the character causing the state transition indicated by the arrow. Accepting states are indicated by circling the node representing their states, so that they appear as two concentric circles. Fig. 2 provides three examples of transition diagrams.

The form of the transition function distinguishes between several varieties of finite automata. A transition function that, on any character,

Figure 1: Specifying FA states and functions: Tables
These tables present the same information as the transition diagrams of Fig. 2. They represent three finite automata recognizing the language (a | b)*b. We have adopted the convention that all finite automata begin in state 0, asterisks indicate final states, and empty entries represent undefined transitions.

(a) Deterministic

	INPUT	
STATE	a	b
0	0	1
*1	0	1

(b) Non-Deterministic

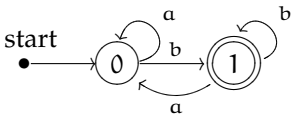
	INPUT	
STATE	a	b
0	0	0, 1
*1		

(c) ϵ–Non-Deterministic

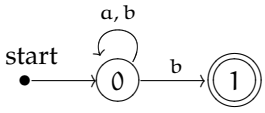
	INPUT		
STATE	ϵ	a	b
0	1		
1	2, 3		
2		4	
3			5
4	6		
5	6		
6	1, 7		
7			8
*8			

Figure 2: Specifying FA states and functions: Figures
These transition diagrams present the same information as the tables of Fig. 1. They represent three finite automata recognizing the language $(a \mid b)^*b$.

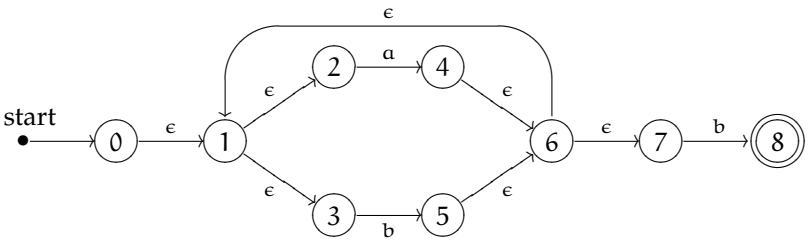
(a) Deterministic



(b) Non-Deterministic



(c) ϵ–Non-Deterministic



permits a transition to only one state is known as a DETERMINISTIC FINITE AUTOMATON (DFA). A transition function that permits a transition to a set of states on any character is known as a NON-DETERMINISTIC FINITE AUTOMATON (NFA). It accepts if any state out of the set of states it halts in is an accepting state. A final variety of finite automaton is distinguished by admitting not only transitions to a set of states, but "autonomous" transitions – transitions that occur without consuming any of the input. These are known as ϵTRANSITIONS because transitioning along them "consumes" only the empty word ϵ made up of no characters. This variety of finite automaton is known accordingly as an ϵ–NON-DETERMINISTIC FINITE AUTOMATON (ϵNFA). These varieties of finite automata are all equivalent in power – it is possible to convert a finite automaton of one type into another type such that both recognize the same language – but some sorts describe a language more naturally or concisely than others. Finite automata are unique in that, for a given regular language, there is a MINIMAL DETERMINISTIC FINITE AUTOMATON, a deter-

ministic finite automaton with the fewest number of states possible that is unique up to renaming of states.

Figs. 1 and 2 describe the same three finite automata in two ways, both in a table and through a transition diagram. All three finite automata recognize the same language. We can describe this language through finite automata as done here or through the regular expression (a | b)*b, which will be discussed in the next section. From the transition diagrams, what do you think this regular expression means?

The ϵ–non-deterministic finite automaton is the most visually complex. It was constructed algorithmically from the regular expression given above by patching together simpler finite automata by way of ϵ transitions. The many ϵ transitions make it highly non-deterministic. The simple non-deterministic finite automaton was created by identifying states joined solely by ϵ transitions. It is the most elegant of the three. Its sole non-determinism consists in state 0 having transitions to two different states on the character b, both to itself and to the final state 1. The deterministic finite automaton was constructed from the non-deterministic. Its state 1 behaves like the non-deterministic finite automaton when it is in the set of states $\{0, 1\}$, which it enters after encountering the character b.

Regular Expressions

We can also describe regular languages declaratively, using REGULAR EXPRESSIONS. These do not describe how to recognize a given language, but rather describe the language directly. This is done by augmenting the alphabet with a direct linguistic interpretation and by adding special symbols representing operations on this linguistic interpretation.

The linguistic interpretation associated to a character is direct and intuitive: the character a represents the language consisting of that single character, $\{a\}$. It is natural to generalize this direct representa-

tion to words: the word w represents the language consisting of that single word, $\{w\}$. Words are built up by concatenation. To aid in describing many concatenations of a simple structure, we can introduce some notation. Iterated concatenation of a regular expression w with itself is represented by superscripts: w^0 is the language of only ϵ, the empty word; w^1 is just $\{w\}$ itself; and, as a rule, $w^n = w^{n-1}w$. We can represent unlimited concatenation using the KLEENE STAR *: $w^\star$ represents the set of all concatenations of the language represented by w with itself, including w^0: $w^\star = \{w^0, w^1, w^2, \ldots\}$. If we wish to exclude the possibility of w^0, we can use the otherwise equivalent POSITIVE CLOSURE operator $^+$: $w^+ = \{w^1, w^2, \ldots\}$. To represent choice or ALTERNATION in the language – either regular expression w or regular expression v is acceptable – we can introduce a corresponding operator; $+$ and $|$ are both popular choices for representing it: we shall use $|$ here. Thus, the regular expression $\mathtt{a} \mid \mathtt{b}$ represents the language $\{\mathtt{a}, \mathtt{b}\}$, while, more generally, the regular expression $w \mid v$ constructed by the alternation of the regular expressions w and v represents the language $L(w) \cup L(v)$, where we use $L(w)$ to represent the language associated to the regular expression w.* Finally, to allow unambiguous composition of regular expressions, we can introduce clarifying parentheses. These let us describe, for example, the language $(\mathtt{a} \mid \mathtt{b})^\star\mathtt{b}$, the language comprising all strings of zero or more $\mathtt{a}$s or $\mathtt{b}$s followed by a $\mathtt{b}$.

While regular expressions are very useful for describing regular languages, they do not provide a way to recognize the languages they describe. Fortunately, regular expressions happen to be readily interconvertible with finite automata.

* In general, where X is any description of a language, whether by Turing machine or finite automaton or regular expression or by any other description aside from the sets representing the languages themselves directly, we write $L(X)$ for the language described by X.

Lexers

With regular expressions to describe the lexical structure of tokens and finite automata to perform the actual work of recognizing tokens, we have a ready way to perform tokenization. Simply scan through the character stream till every recognizing finite automaton will begin to fail; of those that make it this far and will accept, select the token of the highest priority as that summing up the scanned text. This introduction of prioritization provides an intuitive way to resolve ambiguity deriving from our wishing to chunk an input, the program, that in truth belongs to a language unrecognizable by a finite automaton, into words belonging to various token-languages recognized by finite automata.

For example, consider developing a lexer for the input

```
if ifPredicate;
then
    echo "True.";
else
    echo "False.";
fi
```

intended to report whether the provided predicate is true or false.* The desired tokenization is illustrated in Fig. 3 on page 43 along with a sequence of lexing rules that leads to this tokenization.

Before you can understand the rules, you must first understand some common extensions to the regular expression notation introduced so far:

A. A set of characters enclosed in square brackets is equivalent to the alternation of those characters, so $[abc] = (a \mid b \mid c)$.

B. Within square brackets, an inclusive range of characters is indicated by interposing a dash between the two endpoints. A regu-

* The syntax of the example is basically that of the Bash shell, except we have eliminated the prefix sigils that make it easy to recognize variables.

Figure 3: Tokenizing

(a) Desired Tokenization
Each token is enclosed by a pair of angle brackets. The subscript text following each closing bracket indicates the preceding token's type. ↲ and ␣ have been used to represent the whitespace characters newline and space.

⟨if⟩KW⟨␣⟩WS⟨ifPredicate⟩ID⟨;⟩SEQ⟨↲⟩WS
⟨then⟩KW⟨↲
␣␣␣␣⟩WS⟨echo⟩KW⟨"True."⟩STR⟨;⟩SEQ⟨↲⟩WS
⟨else⟩KW⟨↲
␣␣␣␣⟩WS⟨echo⟩KW⟨"False."⟩STR⟨;⟩SEQ⟨↲⟩WS
⟨fi⟩KW

(b) Lexical Rules
Each rule specifies how to lexically distinguish one type of token in terms of a corresponding regular expression. When more than one rule matches the input, the earliest is used.

TOKEN TYPE	REGULAR EXPRESSION
WS	[␣↲]+
KW	if | then | else | fi | echo
ID	[A-Za-z]+
STR	"[^"]*"
SEQ	;

lar expression matching any capital letter A through Z, then, is [A − Z], which is equivalent to (A | B | ··· | Z).

c. When a caret immediately follows the opening square bracket, this inverts the sense of the bracket-alternation: the listed characters are excluded from matching, but any other character of the alphabet will match the expression. Thus, [^"] matches any single character of the alphabet other than ".

The rules' order is important: earlier rules are assigned a higher priority than later. For example, since the rule recognizing a keyword, KW, precedes the rule recognizing an identifier token, IDENTIFIER, if is tokenized as a keyword rather than an identifier, even though both rules match the two alphabetic characters i followed by f.

In truth, the lexer is responsible for more than simply recognizing tokens. It works in cooperation with the parser (which we shall describe next) by feeding it a stream of tokens. Further, it records information associated with the tokens, often in a global, shared SYMBOL TABLE associating to each token, or symbol, some information, such as the text or value of the token and the file and line number where it was first encountered. It might even use information in the symbol table or information provided by the parser to make a distinction between tokens that it is impossible or exceedingly difficult to make with regular expressions alone.

For example, if the language of the text being scanned in Fig. 3 (page 43) required that all functions and variables be declared before use, the lexer would be able to eschew the `ID` token in favor of distinct `FUNCID` and `VARID` tokens by using information about the class of the identifier already stored in the symbol table to distinguish between the two.

Fig. 4 on page 45 is an example of the earlier scanning rules of Fig. 3 (page 43) adapted to use this approach. We have introduced two new keywords, `var` and `func`, and completely changed the identifier rule. Indeed, our lexing method has become more sophisticated: there is no longer is simple one-to-one correspondence between regular expressions and tokens. Instead, matching the input to a regular expression binds the matched text to a variable, `match`, and executes an associated action. This action can affect the environment in which lexing occurs and use that environment to decide how to classify the matched text. This occurs here through the variable `context`, which is used to determine whether we are declaring a new variable or function identifier, or whether we should be able to lookup which of the two the current match is in the symbol table.*

* We have assumed that the parser, discussed next, has taken care of updating the symbol table so that the lookup will succeed if the variable or function identifier was previously declared.

Figure 4: Using the symbol table to tokenize

(a) Desired Tokenization

A new line declaring `ifPredicate` to be a variable precedes the input text of Fig. 3. The tokenization changes accordingly.

⟨var⟩KW⟨␣⟩WS⟨ifPredicate⟩VARID⟨;⟩SEQ
⟨if⟩KW⟨␣⟩WS⟨ifPredicate⟩VARID⟨;⟩SEQ⟨↲⟩WS
⟨then⟩KW⟨↲
␣␣␣␣⟩WS⟨echo⟩KW⟨"True."⟩STR⟨;⟩SEQ⟨↲⟩WS
⟨else⟩KW⟨↲
␣␣␣␣⟩WS⟨echo⟩KW⟨"False."⟩STR⟨;⟩SEQ⟨↲⟩WS
⟨fi⟩KW

(b) Lexical Rules

When a rule's regular expression has the highest priority of all those matching the input, the input matched is stored in `match` and the associated action is taken. Each action ends by returning the token type of the matched input.

REGULAR EXPRESSION	ACTION
[␣↲]+	`return WS;`
`if` | `then` | `else` | `fi` | `echo`	`return KW;`
`var`	`context=VARDECL, return KW;`
`func`	`context=FUNCDECL, return KW;`

[A-Za-z]+	`if (context == VARDECL)`
	`  context=PLAIN, return VARID;`
	`else if (context == FUNCDECL)`
	`  context=PLAIN, return FUNCID;`
	`else return lookup(match);`

"[^"]*"	`return STR;`
;	`return SEQ;`

4.1.2 *Syntax Analysis*

SYNTAX ANALYSIS follows lexical analysis. If lexical analysis is concerned with categorizing words by part of speech, then syntax analysis is concerned with understanding how these parts of speech are grammatically related and whether the sentences so formed are grammatical or not.

Context-Free Languages

In fact, "grammatical" is precisely the word, for the formalism affording ready syntax analysis is that of context-free grammars. As with the regular languages, we are able to describe a given context-free language either constructively or declaratively. The context-free languages are a proper superset of the regular languages and a proper subset of the recursive languages. Roughly, the context-free languages are distinguished from the regular languages by their ability to describe "matching bracket" constructs, such as the proper nesting of parentheses in an arithmetic expression, while the recursive languages are distinguished from the context-free languages in part by their ability to cope with context.

Context-Free Grammars

We use CONTEXT-FREE GRAMMARS to specify context-free languages declaratively. As with regular expressions and finite automata, context-free grammars operate in the context of a specific alphabet. The letters of the alphabet are called TERMINALS or TERMINAL SYMBOLS. Context-free grammars augment this alphabet with a finite set of NONTERMINALS (NON-TERMINAL SYMBOLS) to be used in specifying grammatical PRODUCTIONS, which function as REWRITE RULES. Together, the set of terminal and non-terminal symbols are called GRAMMAR SYMBOLS, as they specify all the symbols used by the grammar.

Analogous to the start state of the finite automaton is the context-free grammar's distinguished START SYMBOL. All words in the language described by the context-free grammar are derived from the start symbol via the productions in a manner to be described shortly.

Putting these rules together, one arrives at a grammar specification like the following:

$$G = (N, T, \Sigma, P, S) \qquad N = \{A, B\} \qquad T = \{a, b\} \qquad \Sigma = a, b$$

$$P = \{S \to A, \quad S \to B, \quad S \to aABb, \quad A \to a \mid \epsilon, \quad B \to b \mid \epsilon\}$$

where N is the set of non-terminals, T the set of terminals, Σ the alphabet, and P the set of productions, where $\to$ is read as "produces." The symbol to the left of the arrow is called the HEAD of the production, while those to the right are called the BODY. For example, in the production $S \to aABb$, S is the head of the production and $aABb$ is the body:

$$\underbrace{S}_{\textit{head}} \to \underbrace{aABb}_{\textit{body}}$$

Derivation proceeds by substitution of production bodies for production heads: for example,

$$S \overset{S \to aABb}{\Longrightarrow} aABb \overset{A \to a}{\Longrightarrow} aaBb \overset{B \to b}{\Longrightarrow} aabb \tag{4.1}$$

where $\Rightarrow$ is read as "derives in one step" and the rule justifying the derivation is written above the arrow. Taking a cue from regular expressions, we can also write $\overset{\star}{\Rightarrow}$ for "derives in zero or more steps" (all grammar symbols derive themselves in zero steps) and $\overset{+}{\Rightarrow}$ for "derives in one or more steps," where the productions justifying the derivation are implicit in the superscript star; the keen reader should perhaps like to construct their own explicit, step-by-step derivation. The language

defined by the grammar is defined to be those strings made up only of terminal symbols that can be derived from the start symbol.

PARSE TREES We can use a PARSE TREE to represent the derivation of a word in the language without concern for unnecessary sequencing of derivations imposed by our sequential presentation. For example, our choice to derive a from A prior to deriving b from B above is irrelevant, but that we first derived aABb from S before performing either of the remaining derivations is not, since the heads of these derivations are introduced by the derivation from S. We define parse trees constructively:

A. Begin by making the start symbol the root.

B. Select a non-terminal on the leaves of the tree with which to continue the derivation and a production for which it is the head.

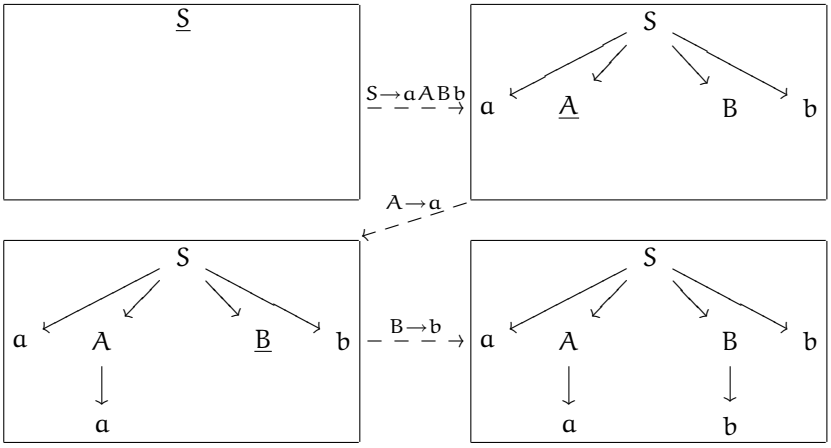
C. Create new child nodes of the chosen head symbol, one for each symbol in the body.

D. Repeat from B.

At any point in time, the string of symbols derived thus far – those on the leaves, read in the same order applied to the child nodes in the body of a production – is called a SENTENTIAL FORM. The process terminates when a word in the language is derived, as no non-terminal leaf nodes remain. Fig. 5 on page 49 gives the parse trees created in deriving aabb from our example grammar.

AMBIGUITY Parse trees represent the derivation of a word without regard to unnecessary sequencing. A given tree represents a given parse. If more than one parse tree can derive the same word in the language, the grammar is said to be AMBIGUOUS. This corresponds to the use of a significantly different ordering of productions and potentially even of a different set of productions. The grammar is called

Figure 5: Growing a parse tree
These figures represent the construction of a parse tree corresponding to the leftmost derivation of aabb given in (4.1). The nonterminal chosen in step B is underlined, and we have labeled the arrows between trees with the chosen production.



ambiguous because, given such a word, it is uncertain which productions were used to derive it. The grammar we gave above is ambiguous when it comes to the empty word ϵ, because $S \Rightarrow A \Rightarrow \epsilon$ and $S \Rightarrow B \Rightarrow \epsilon$ are both valid derivations with corresponding significantly different valid parse trees of ϵ. However, if we were to eliminate the productions $S \Rightarrow A$ and $S \Rightarrow B$ from the grammar, we would then have an unambiguous grammar for the context-free language comprising $\{ab, aab, abb, aabb\} \equiv \left\{a^i b^j \mid 1 \leqslant i, j \leqslant 2\right\}$. The only sequential choices are insignificant: in deriving aabb, we must have derived aABb from S, but following that, did we first derive the second a or the second b?

DERIVATION ORDER While the parse trees for a word in a language factor out differences between possible derivations of the word other than those reflecting ambiguity in the grammar, when performing a derivation or constructing such a parse tree, we must employ such "insignificant" sequencing. There are two primary systematic ways to do so: always select the leftmost nonterminal symbol in step

B of the parse tree construction process, or always select the rightmost. These ways of deterministically choosing the next symbol to replace in the derivation give rise to what are unsurprisingly known as a LEFTMOST DERIVATION and a RIGHTMOST DERIVATION; to indicate the use of one or the other, the derivation arrow in all its forms is augmented with a subscript of $\mathtt{lm}$ for leftmost derivation and $\mathtt{rm}$ for rightmost, giving $\underset{\mathtt{lm}}{\Longrightarrow}$ and $\underset{\mathtt{rm}}{\Longrightarrow}$. Since this is only a matter of choice in constructing the parse tree, it should be clear that, for any given parse tree, there exist both leftmost and rightmost derivations of its sentential form.

Pushdown Automata

We can also specify context-free languages constructively using an abstract machine called a PUSHDOWN AUTOMATON. A pushdown automaton is a finite automaton augmented with a stack and associated stack alphabet. It has an initial stack symbol as well as an initial state. Its transition function and behavior is complicated by its being inherently non-deterministic. As might be expected, its transition function is parameterized by the current input symbol, current state, and the symbol currently on top of the stack. However, for each such triple, the transition function specifies a set of pairs. Each pair consists of a state and a sequence of stack symbols with which to replace the current top of stack. For a given triple, the pushdown automaton simultaneously transitions to all the states indicated by the transition function and replaces the symbol on top of the stack with the corresponding symbols for each new state it is in. Each one of these can be treated as a new pushdown automaton. To "move," each member of the family of pushdown automata consults the current input, its state, and the top of the stack, and then transitions accordingly. We again have a choice of rep-

resenting this either with a table or graphically. While finite automaton transition diagrams had arrows labelled

⟨*input symbol*⟩

the arrows of pushdown automaton transition diagrams are labelled

⟨*input symbol*⟩, ⟨*stack symbol to pop*⟩/⟨*stack symbols to push*⟩

where the convention used for the stack operations is that the symbol that is to be on top of the stack after pushing is leftmost (that is, the stack conceptually grows to the left).

There are some casualties of the transition from finite automata to the increased descriptive power of pushdown automata. pushdown automata are inherently non-deterministic: they always admit ϵ-transitions and can be in a set of states at any given time. This non-determinism is essential for them to define the context-free languages. The languages described by deterministic pushdown automata, while still a proper superset of the regular languages, are only a proper subset of the context-free languages. Further, there is no algorithmic way to produce a minimal pushdown automaton for a given language. This poses a particular problem for parsing: as with lexing, we would like to use grammars to describe the syntactic structure and pushdown automata to perform parsing by recognizing that structure, but we must now find some way for our inherently deterministic computers to cope with this inherent non-determinism in a reasonable amount of time.

Parsers

As exaggeratedly hinted at above, while grammars define a language, parsers are faced with an input that they must characterize as either of that language or not. They must, in fact, do more than simply check that their input is grammatical: they must construct an intermediate

representation of their input to pass on to the next part of the compiler.*

We also mentioned the problem of the non-determinism inherent to context-free grammars and pushdown automata. So long as we only face insignificant questions of sequencing, we will have no problem determining what to do next. Realistic inputs do not require truly non-deterministic parsing. A program is meant to have a single meaning: to correspond to a parse tree, not a parse forest. Non-determinism occurs in parsing a programming language when the available context is insufficient to predict the shape of the parse tree, and it becomes necessary to entertain several possibilities simultaneously. Eventually, more context will be available to resolve the ambiguity, and we can return to building a single parse tree and abandon the others as false starts. Problems such as these are likely to affect only part of the input, and methods have been developed that handle such "temporary non-determinism" gracefully.

The remainder of our discussion of parsers will focus on several of the more common of their many types. The level of our discussion will be one of summary, not of definition; for details, the interested reader is referred to the literature discussed in Section 4.6, Bibliographic Notes.

RECURSIVE DESCENT PARSERS Recursive descent parsers discover a *leftmost derivation* of the input string during a *left-to-right scan* of the input, whose alphabet, thanks to the lexer, will be tokens rather than individual letters and symbols. One function is responsible for handling each token; parsing begins by calling the function associated to the start symbol. They discover the derivation by recursively calling themselves as necessary. The parser is aware of the current input symbol via what is known as LOOKAHEAD. Since we are dealing with an

* If, indeed, there is a next pass: it is possible to construct one-pass compilers that translate from source to target in a single pass over the source code.

actual machine, however, we are not restricted to lookahead of a single symbol, though we might prefer to do with only a single symbol's lookahead for effiency' sake. Those grammars parsable by a recursive descent parser with k tokens of lookahead are known as LL(k): leftmost derivation by left-to-right scan employing k tokens of lookahead.

When recursive descent parsers use one token of lookahead, they act much like a pushdown automaton. The implicitly managed function call stack acts as the pushdown automaton's stack. However, since they trace out a leftmost derivation with only a limited number of tokens of lookahead, they must anticipate the proper derivation with minimal information about the rest of the input stream. This makes recursive descent parsers one of the most limited forms of parsers, though they might be the parser of choice in some cases because of the naturalness of expression they can admit and the simplicity and compactness of their parsers. Many of the disadvantages of recursive descent parsers can be overcome by admitting variable tokens of lookahead, with more tokens being used as needed to disambiguate the choice of production.

PRECEDENCE PARSING Recursive descent parsers are sometimes coupled with precedence parsers in order to facilitate parsing of arithmetic expressions. The order in which operations should be carried out is determined by a frequently implicit grouping determined by operator associativity and precedence. For example, multiplication is normally taken to have higher precedence than addition, so that $3 \times 5 + 4$ is understood to mean $(3 \times 5) + 4 = 19$ and not $3 \times (5 + 4) = 27$. The left associativity of multiplication determines that $2 \times 2 \times 2$ should be understood as $(2 \times 2) \times 2$. This becomes important, for example, in cases where the operands have side effects: suppose `id` is a unary function printing its input and then returning its input unchanged. Assume further that arguments to operators are evaluated left-to-right. Then `id(1) + id(2) + id(3)` will print `123` if addition is understood to be left-

associative, but it will print `231` if addition is understood to be right-associative, even though the result of the additions will be identical due to the associativity property of addition.*

Operator precedence parsing is preferred over the use of LL(k) grammar rules not only because it is somewhat unobvious how to enforce the desired associativity and precedence in an LL(k) grammar, but also because doing so introduces a chain of productions that exist solely to enforce the desired associativity and precendence relations between the expression operators. Beyond its use in concert with recursive descent parsers, precedence parsing has mostly been subsumed by the class of grammars we shall describe next. The central idea of using precedence and associativity to disambiguate an otherwise ambiguous choice of productions has lived on in implementations of parser generators for this later class. Without recourse to a way other than grammatical productions to indicate precedence and associativity, grammars would often have to take a form that unnecessarily obscures their meaning simply to grammatically encode the desired precedence and associativity relations.

LR(k) PARSERS The LR(k) – left-to-right scan, rightmost derivation with k tokens of lookahead – family of parsers is perhaps the most commonly used in practice. I say "family" because a number of subtypes (to be discussed shortly) were developed to work around the exponential space and time requirements of the original LR(k) algorithm. The class of grammars recognizable by an LR(k) parser is known as the LR(k) grammars, and it is possible to give a reasonably straightforward LR(k) grammar for most programming languages. However, it was some time before clever algorithms that avoided unnecessary requirements of exponential space and time were developed, and so other, more restrictive classes of grammars with less demanding parsers

* If argument evaluation proceeded right-to-left, `213` and `321` would be printed instead.

were developed and deployed. Parser generators targeting these classes are more limited in terms of the grammars they can generate parsers for, not in terms of the languages such grammars can recognize: all parsers of the LR(k) family, where $k > 0$, accept the same class of languages; they simply place different, more or less restrictive demands on the form of the grammars describing those languages.

Where LL(k) parsers create a derivation from the top down by starting with the goal symbol and eventually building a derivation for the input, LR(k) parsers build a rightmost derivation in reverse by reading in the input till they determine that they have seen the body of a production and then reducing the body to the head. They eventually reduce the entire input to the start symbol (often in this context called the GOAL SYMBOL), at which point parsing is complete. They use a stack to store the symbols seen and recognized so far, so in the course of parsing they carry out a very limited set of actions: shifting input onto their stack, reducing part of the stack to a single symbol, accepting the input as a valid word in the grammar, and indicating an error when none of the above applies. Because of this behavior, such a bottom-up parser is often called a SHIFT-REDUCE PARSER.

SLR(k) PARSERS The earliest and most restricted such class is known as the SIMPLE LR(k), or SLR(k). These parsers use a simplistic method of determining what action to take while in a given state and reading a given input that introduces conflicts that more sophisticated methods would be capable of resolving. In a shift-reduce parser, there are two possible types of conflicts:

SHIFT/REDUCE CONFLICTS where the parser has seen what it considers the body of a valid production at this point in the parse but has also seen a viable prefix of yet another production, so it cannot determine whether to reduce using the former or shift

further symbols onto the stack in an attempt to recognize the latter.

REDUCE/REDUCE CONFLICTS where the parser has seen the entirety of the body of two productions that appear to be valid at this point in the parse and is unable to determine which to reduce to.

LALR(k) PARSERS More sophisticated parsing methods are more discriminating about what productions are still valid at a given point in the parse by taking into account more or less of the parsing process and input seen thus far, so called LEFT CONTEXT as it is to the left of where the parser presently is in consuming the input. (In this analogy, the lookahead symbols could be considered right context, though that term is never used.) One such method is known as LOOK-AHEAD LR (LALR). These parsers can be seen as "compressed LR parsers," though this compression can introduce spurious reduce/reduce conflicts that would not occur in a full LR parser. This has historically been seen as an acceptable tradeoff for the reduction in table size and construction time, since any LR grammar can be reformulated as an LALR grammar, but with more sophisticated LR algorithms developed later that retained the full power of full LR parsers while producing comparable levels of compression wherever possible (meaning that parsing an LALR grammar with such an LR parser would require the same space as parsing it with an LALR grammar), such a tradeoff became unnecessary, though it remains widespread.

TABLE-DRIVEN PARSERS Whereas recursive descent parsers and operator descent parsers can be hand-coded, many of the other parsing algorithms were developed to operate by way of precomputed tables.* They explicitly model a finite automaton, called the CHARACTERIS-

* That is not to say that the others cannot also be implemented through tables, simply that the table method is not felt to be the necessity that it is for these others.

TIC FINITE AUTOMATON; the tables allow the transition function to be implemented purely by table lookup. As hand-creation of tables is time-consuming and error-prone, tables for parsing are generally created algorithmically and the resulting tables used with a DRIVER that simply does little more than gather the information necessary to perform the operations specified by the table.

DIRECT-CODED PARSERS Parsers implemented entirely in code (rather than as a set of tables with a driver) were long seen as something to be generated only by humans, while parsers generated from a higher-level grammar description were to be implemented by way of tables. However, another possibility, often faster and smaller because of its lower overhead and its lack of a need to encode a rather sparse table, is to have the parser generator create a direct-coded parser, a parser that is not table-driven but yet is generated from a higher-level description rather than being written by hand.

GLR PARSERS LR parsers are restricted to parsing only LR languages. However, a very similar technique can be used to parse all context-free languages. GENERALIZED LR (GLR) PARSERS are more general than LR parsers in two senses:

- They are able to parse all context-free grammars, not just LR grammars.
- Their method of parsing is a generalization of that used in LR parsers.

They generalize the parsing method of shift-reduce LR parsers by coping with ambiguity in the grammar by duplicating the parse stack and pursuing competing parses in parallel. When they determine a particular parse is in fact invalid, it and its stack are destroyed. If the grammar is in fact ambiguous and multiple parses are possible, this might

lead to a PARSE FOREST instead of a parse tree. Making such parsers feasible requires some effort, and part of that effort was to replace several duplicate parse stacks by what amounts to a "parse lattice" that share as many grammar symbols as possible as parses converge and diverge, much reducing the space requirements of the parser as well as time spent repeating the identical shifts and reduces on different parse stacks. It is also important to employ similar compression methods as with the newer LR parser generation algorithms, so that extra space and time is only employed as strictly necessary to deal with non-LR constructs.

SEMANTIC ACTIONS We generally desire to know more than that a given input is grammatical: we want to create a representation of the information discovered during parsing for later use. This is done by attaching SEMANTIC ACTIONS, to productions in parsers and to recognized tokens in lexers. Such actions are invoked when the production is reduced or the token recognized, and they are used to build the representation and, in the lexer, to emit the recognized token for the parser's use. They also can be used to compute attributes of the nodes in the parse tree, as discussed next.

4.1.3 *Semantic Analysis*

SEMANTIC ANALYSIS, also known as CONTEXT-SENSITIVE ANALYSIS, follows scanning and parsing. Its job is to "understand" the program as parsed. Not all elements of the language can be checked by either regular expressions or context-free grammars; checking these falls to the semantic analyzer. Approaches to semantic analysis vary widely; while a formalism that permits generating semantic analyzers from a higher-level description, as is done for lexers and parsers, exists,

its use has yet to become widespread. Frequently, semantic analysis is done purely through *ad hoc* methods.

A program in truth has two aspects to its semantics, the static and the dynamic. STATIC SEMANTICS are those aspects of the program's meaning that are fixed and unchanging. A common example is the type of variables (though there are languages that employ dynamic typing). These aspects are particularly amenable to analysis by the compiler, and information derived from understanding them can be used to optimize the program. A program's DYNAMIC SEMANTICS are those aspects of the program that are only determined at runtime.

Nevertheless, a compiler can attempt to prove through analysis certain properties of the running program, for example, that an attempt to access an array element that does not exist (the eleventh element of a ten-element array, for example) can never occur. Some languages require that the compiler guarantee certain runtime behavior: if it is unable to provide that guarantee at compile time through analysis, the compiler must insert code to check the property at runtime. Java, for example, requires that no out-of-bounds array access occur: any such attempt must be refused and raise an error. Since these runtime checks can slow down a program, a frequent point of optimization in languages requiring such checks is proving at compile-time properties that enable the omission of as many such checks as possible from runtime. Many languages, particularly older languages, do not require runtime checks even where they might be worthwhile, while some compilers might permit disabling the insertion of runtime checks, an option favored by some for the generation of final, production code after all debugging has occurred.

Attribute Grammars

The formalism mentioned above for generating semantic analyzers is that of ATTRIBUTE GRAMMARS. Attribute grammars can be seen as

an evolution of context-free grammars. They begin by taking the grammar symbols of the grammar and associating to each one a finite set of ATTRIBUTES. They next take the grammatical productions and associate to each one a similarly finite set of SEMANTIC RULES. Each rule is a function that describes how to calculate a single attribute of one of the symbols of the production (which attribute we shall call the TARGET of the semantic rule) in terms of any number of attributes of any of the symbols of the production. Where the same symbol occurs multiple times in the same production, subscripts are used to differentiate the different occurrences of the symbol. To refer to a symbol's attribute, we follow the name of the symbol with a dot and the name of the attribute, so that $A.x$ would refer to the attribute x of the symbol A. These conventions are amply illustrated in Fig. 6b on page 61.

INHERITED AND SYNTHESIZED ATTRIBUTES Recall that each grammatical production has two parts, a single symbol called the head and a body of some symbols that is derived from the head. The set of attributes of the symbols are likewise partitioned into two disjoint sets, those that can be the target of a semantic rule when the symbol is in the head of the production, called SYNTHESIZED ATTRIBUTES, and those that can be the target of a semantic rule when the symbol is part of the body of the production, called INHERITED ATTRIBUTES.

Consider as an example the second production in Fig. 6b on page 61, $\texttt{ADDER}_1 \rightarrow \texttt{DEF}\ \texttt{'.'}\ \texttt{ADDER}_2$. Any attributes of $\texttt{ADDER}_1$ targeted by a semantic rule in this production must be by definition synthesized attributes, while any targeted attributes of the other three symbols DEF, '.', and $\texttt{ADDER}_2$ must be inherited attributes. If you check the attributes that are in fact targeted against the table of symbols and attributes in Fig. 6a (also on page 61), you will find that this is indeed the case.

You might notice in the same table that some entries are prohibitory dashes. That is because every symbol can have both synthesized and in-

Figure 6: An attribute grammar

The nonterminals are `GOAL`, `ADDER`, `VAL`, `SUM`, and `DEF`. The terminals are `NAME`, `NUM`, and the single characters `.`, `+`, and `=`. The start symbol is the head of the first production, `GOAL`.

(a) Symbols and Attributes

Start and terminal symbols are prohibited from having inherited attributes, hence the dashes. An empty entry indicates the symbol has no attributes of that type, though it could.

	ATTRIBUTES	
SYMBOL	INHERITED	SYNTHESIZED
`GOAL`	–	`total`
`ADDER`	`defs`	`total`
`SUM`	`defs`	`total`
`DEF`	`defs`	`pair`
`VAL`	`defs`	`amt`
`NUM`	–	`amt`
`NAME`	–	`txt`
`.`	–	
`+`	–	
`=`	–	

(b) Productions and Rules

An unquoted string of non-whitespace characters represents a single grammar symbol. Characters between single quotation marks represent themselves as symbols in the production.

PRODUCTION	SEMANTIC RULES
`GOAL` $\longrightarrow$ `ADDER`	`GOAL.total` $\longleftarrow$ `ADDER.total`
$\texttt{ADDER}_1 \longrightarrow$ `DEF '.' ADDER`$_2$	`DEF.defs` $\longleftarrow$ $\texttt{ADDER}_1$`.defs`
	$\texttt{ADDER}_2$`.defs` $\longleftarrow$ `DEF.defs` $\cup$ `DEF.pair`
	$\texttt{ADDER}_1$`.total` $\longleftarrow$ $\texttt{ADDER}_2$`.total`
`ADDER` $\longrightarrow$ `SUM '.'`	`SUM.defs` $\longleftarrow$ `ADDER.defs`
	`ADDER.total` $\longleftarrow$ `SUM.total`
`VAL` $\longrightarrow$ `NAME`	`VAL.amt` $\longleftarrow \{\mathrm{snd}\, p \mid p \in$ `VAL.defs` $\wedge\, \mathrm{fst}\, p =$ `NAME.txt`$\}$
`VAL` $\longrightarrow$ `NUM`	`VAL.amt` $=$ `NUM.amt`
$\texttt{SUM}_1 \longrightarrow$ `VAL '+' SUM`$_2$	`VAL.defs` $\longleftarrow$ $\texttt{SUM}_1$`.defs`
	$\texttt{SUM}_2$`.defs` $\longleftarrow$ $\texttt{SUM}_1$`.defs`
	$\texttt{SUM}_1$`.total` $\longleftarrow$ `VAL.amt` $+$ $\texttt{SUM}_2$`.total`
`SUM` $\longrightarrow$ `VAL`	`SUM.total` $\longleftarrow$ `VAL.amt`
`DEF` $\longrightarrow$ `NAME '=' SUM`	`SUM.defs` $\longleftarrow$ `DEF.defs`
	`DEF.pair` $\longleftarrow$ (`NAME.txt`, `SUM.total`)

herited attributes, except terminal and start symbols, which are not allowed inherited attributes. Terminal symbols are often prohibited from having inherited attributes so that attribute grammars can be readily composed to form larger attribute grammars by identifying a terminal symbol of one grammar with the start symbol of another. Likewise, they are allowed to have synthesized attributes with the assumption that the values of these attributes will be provided by some source external to the attribute grammar itself, such as another attribute grammar or the lexer. In terms of a single grammar, the start symbol cannot have any inherited attributes since it is never part of a production body. If we compose grammars, the start symbol will still have no inherited attributes, since we have barred terminal symbols from having inherited attributes.

We could, for example, compose the `ADDER` grammar of Fig. 6, page 61, with a `NUM` grammar for parsing a variety of numerical formats, such as signed integers and scientific notation. The sole modification we might have to make to the `NUM` grammar is to convert the information stored in its attributes for storage in the sole `amt` attribute of the `NUM` symbol of the `ADDER` grammar. This easy composition is made possible by the conventions barring terminal and start symbols from having inherited attributes. If terminal symbols had to take into account inherited information, more extensive modifications of the grammars would often be required before they could be composed.

THE ATTRIBUTED TREE In terms of a parse tree, a synthesized attribute is computed from attributes at or below itself in the parse tree, while inherited attributes are computed from attributes at their own level or above them in the parse tree. Thus, the computation of synthesized and inherited attributes can be viewed respectively as information flowing up and down the parse tree.

In fact, the parse tree is central to how attribute grammars are used. A bare parse tree is the fruit of a context-free grammar. With an attribute grammar, we produce a parse tree wherein every node is decorated with its own instances of the attributes associated to its symbol. Every node where a given symbol appears will have the same attributes, but the values of the different instances of the attributes can differ. A parse tree annotated as described with attribute occurrences is called an ATTRIBUTED TREE. Fig. 7 on page 64 provides a small example of an attributed tree for a word in the language of the attribute grammar of Fig. 6 (page 61).

ATTRIBUTE EVALUATION With the bare parse tree become an attributed tree, the stage is set for us to use the semantic rules to assign concrete values to the tree's attribute instances. This process of computation is known as ATTRIBUTE EVALUATION. Provided attribute evaluation terminates, the attribute grammar formalism defines the meaning of the program (which now makes up the leaves of an attributed tree) to be the values of the attributes of the start symbol.

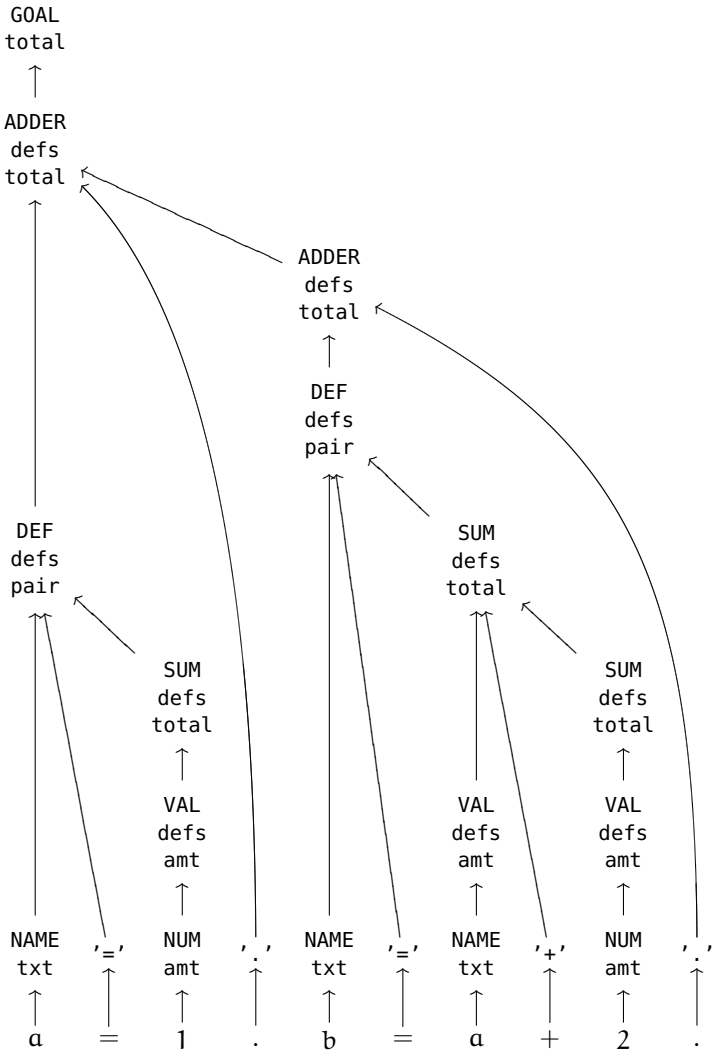
This definition of attribute evaluation, however, is purely descriptive. When we look to perform attribute evaluation, things are not so simple. Attribute evaluation will not necessarily terminate,* and determining an appropriate order for evaluation such that evaluation can be performed efficiently is nontrivial.

Part of making this formalism usable involves, as with context-free grammars, finding restricted classes of attribute grammars that are sufficiently powerful to capture the semantic information desired while still allowing efficient evaluation. Two such classes are the S-ATTRIBUTED GRAMMARS and the L-ATTRIBUTED GRAMMARS. S-attributed grammars admit only synthesized attributes. They can thus be evaluated during a simple bottom-up walk of the parse tree like that performed

* A simple example is the production $A \to B$ together with the semantic actions $A.x \leftarrow B.x + 1$ and $B.x \leftarrow A.x + 1$, which together cause a loop that repeatedly increments $A.x$ and $B.x$.

Figure 7: An attributed tree

The bottom-most row is external to the grammar. It would be processed by the lexer and transformed into the terminal symbol tokens shown immediately above that row.



by a shift-reduce parser. L-attributed grammars loosen the restrictions imposed by S-attributed grammars somewhat. In addition to synthesized attributes, they allow semantic rules targeting the attributes of a symbol B_k in a production $A \to B_1 B_2 \cdots B_n$ to use any attributes of A or $B_1 \cdots B_{k-1}$. In terms of the attributed tree, this allows a symbol's attributes to be computed in terms of those of either its children (in the case of a synthesized attribute) or those of its parent and the siblings to its left (in the case of an inherited attribute). Like the S-attributed grammars, L-attributed grammars admit information flow from bottom-to-top within the parse tree, but they also allow for left-to-right information flow. This is a natural match for a left-to-right, depth-first walk of the parse tree, as occurs during recursive descent parsing.

Problems faced by practical implementations of the attribute grammar formalism include the management of storage for the multitude of attribute instances used during evaluation and the amount of attributes that exist solely to share non-local information. Non-local information is in general a problem with attribute grammars, and while a symbol table can be used alongside the grammar to avoid this issue, it is also an end-run around the formalism.

4.2 INTERMEDIATE REPRESENTATIONS

Translation begins with a source language and ends with a target, but those are rarely the only representations of the program used during compilation. Between the initial representation of the source code input to the compiler and the final representation of the target code output from the compiler, the compiler will use various INTERMEDIATE REPRESENTATIONS (IRS). These need not resemble either the initial or final representation in the least, and the compiler is not restricted to use only one intermediate representation. Intermediate representa-

tions are, in a sense, common, private languages used within and between different parts of a compiler that support the operation of those parts.

The intermediate representations chosen affect all parts of the compiler, both on the superficial level of simple representation of the code and on the deeper level of how the compiler carries out its translation and even how much the compiler can prove about the runtime behavior of the code to exploit in optimizing it.

For all its importance, intermediate representations remain more a matter of craft than science. Many IRs have been used – estimates of two for every compiler ever created are likely conservative – , but this myriad of IRs nevertheless is susceptible to categorization along various axes. Two such axes are the form of the intermediate representation and its level of abstraction.

4.2.1 *Form*

Intermediate representations divide broadly into classes based on their structure: those whose structure is linear, and those whose structure is GRAPHICAL, an artificial term meaning "graph-like" that has nothing to do with graphics or visual display.

Linear

Linear IRs resemble the structure of most programming languages, in that they have an implicit sequencing: begin at the beginning and process each instruction in turn till the last instruction is processed. Jump instructions of some form or another – either as higher-level, structured control flow constructs such as `while` and `for`, or as lower-level jumps and branches to labeled statements (or, at an even lower level, to statements a certain offset away) – can be used to explicitly alter this implicit order.

Linear IRs have the advantage of being easy to represent for debugging or otherwise observing the actions of the compiler. They can also be easily written out to a text file. They simply become lines of text. Their flat, linear structure can also be a disadvantage. They have no easy way to share identical lines between sections beyond threading through them again via jumping. This can inflate the size of the IR code and hide redundant computations. At the same time, because of their similarity to most target languages, a linear IR can be a very good choice for when a compiler must finally perform target code generation.

Graphical

Graphical IRs are so-called because they represent the program as a graph with arcs and nodes rather than as a large, linear sequence. Depending on the graphical IR used, this can obscure control flow, but it can also represent higher-level structure than is possible in a linear IR. Tree-based IRs suffer from the same issues of size and repetition of common substructure as textual IRs. Graphical IRs based on DIRECTED ACYCLIC GRAPHS, which can be thought of as trees that admit merging of branches,* can avoid both of these faults, though since, in imperative programming languages, `x` at one point in the program need not be the same as `x` at another, the textual identity of two repetitions of `compute(x)` may not actually be a sign of redundant computation. Graphical IRs always introduce a question of representation: many data structures can be used to represent graphs and many algorithms can be used to carry out the same operation, and each choice of data structure and algorithm has its own tradeoffs.

It is also not convenient to represent a graphical IR as some form of output for human consumption; the IR must either be sequenced

* Or, if you are more mathematically inclined, can be thought of as directed graphs restricted not to have cycles, that is, a sequence of arcs leaving one node that can be traversed obeying their direction in order to return to the initial node. It is clear which viewpoint prevailed in the name of the structure.

and encoded into a linear form, or more complex and time-consuming techniques must be employed to create a pictorial representation. This latter is not an option for storing information for the compiler's consumption: the IR must then be encoded into a linear representation, though the compiler does not require a textual representation – a novel binary representation developed to suit the compiler's needs might in this case be the better choice. Regardless of problems of representation, many operations performed by the compiler are best expressed as operations on a graph, and a graph is often the most natural form to view the code from, as in the CONTROL FLOW GRAPH that graphically depicts blocks of sequentially executed code (so-called BASIC BLOCKS) connected by directed arcs to blocks that control might transfer to.

4.2.2 *Level of Abstraction*

intermediate representations can also be classified by their level of abstraction. Some levels of abstraction are more appropriate for the application of some optimizations than others. Some optimizations can usefully be used at many levels of abstraction, while others can only be used at a certain level of abstraction: for example, optimizations dealing with register usage require that register usage be exposed by, expressed in, and directly manipulable through the intermediate representation. In this case, only a low-level intermediate representation will do.

High-level intermediate representations are frequently very close to the source language. They often include direct representations of structured control flow and indexed array accesses. However, much like the source language itself, they are not very suitable for the application of many optimizations, so they see only limited use within a compiler. An example of a high-level linear intermediate representation would basically be a simple high-level programming language. A com-

mon high-level graphical intermediate representation is the ABSTRACT SYNTAX TREE (AST). An abstract syntax tree is something of an abbreviated parse tree; it omits "uninteresting nodes" and eliminates the lower-level information of the parse tree in favor of a more semantically relevant and concise form.

Mid-level intermediate representations are much like high-level intermediate representations, except that they will generally require explicit computation of array accesses and eliminate structured control flow in favor of labels, jumps, and branches. It is very possible to blend high- and mid-level intermediate representations.

Low-level intermediate representations expose many more details about the target language and target machine. While this strongly suggests use of what is virtually the assembly language of the target machine, it is still possible to employ a graphical intermediate representation. Such an intermediate representation will have to provide a way to indicate indirection through memory addresses (in the jargon of C and its relatives, this would be called "pointer dereferences").

4.2.3 *Static Single Assignment Form*

STATIC SINGLE ASSIGNMENT (SSA) FORM is something of a hybrid intermediate representation. It can be used at any level of abstraction that represents explicit variables. It aims to make explicit which definition of a variable each use refers to. It does this by treating the redefinition of a variable as the definition of a new variable; the different variables created by variables that were multiply defined in the original intermediate representation are often presented as subscripted variable names, so that a_0 would correspond to the original definition of the variable a, and a_1, a_2, and so forth to subsequent redefinitions of this variable.

To put a program in SSA form, one begins with a linear intermediate representation. One next constructs around this a control flow graph. If you draw out a few control flow graphs, this will demonstrate an obstacle to putting a program in static single assignment form: what do we do when two different definitions of the same variable could prevail at the same point in the program? This problem motivated the most peculiar element of SSA, PHI FUNCTIONS. These are used to deal with control flow that would otherwise defeat the aim of every variable in every block having a single, unambiguous referent. Whenever two definitions of a variable can reach the same block, the variable is considered to be redefined with the appropriate definition based on which in-arc control flow enters the block from. Thus, if both a_0 and a_1 could reach the same subsequent use of a in the initial intermediate representation, a phi function would be introduced to supply the appropriate definition of a:

$$a \leftarrow \phi(a_0, a_1)$$

4.2.4 *Symbol Tables*

We include along with the intermediate representation the tables of information maintained by the compiler. The most prominent of these is the SYMBOL TABLE, which records information on all symbols – variables, function names, and the like – in use in the program. The type of information in the symbol table reflects where in the compilation process the compiler is and partially determines the level of the current intermediate representation. Basic information is usually gathered through cooperation between the scanner and parser and is often necessary for and augmented during static semantic analysis. Other parts of the compiler will introduce further annotations to the symbols. The information stored for a symbol might include details such

as the name, storage class (statically allocated, dynamically allocated, or created and destroyed along with a procedure), type, size, and much more. Use of a symbol table is in some senses analogous to allowing all semantic rules access to the attributes of the goal symbol: the table provides a way to readily aggregate information collected from a variety of places, in a variety of ways, at a variety of times.

4.3 MIDDLE END: OPTIMIZING THE IR

The middle end comes, as one might expect, between the front and back ends. Since it follows the front end, it has at hand the program in some form of intermediate representation that encodes, not only the program, but useful information about the program. It precedes the back end, since its efforts can go some way towards easing the work of the back end. The purpose of the middle end is, given the program in some form, to work with that, possibly by manipulating it through various intermediate forms, to optimize the program. This is a rather vague aim with many possible interpretations, and this variety of interpretations is reflected in the variety of middle ends.

What does it mean, to optimize a program? In some sense, the entire compiler's work is an optimization of the program: it is given source code, a static, lifeless description of a program, and it produces a directly executable description of a program. This is the most significant improvement of all that a compiler makes. It is not surprising, then, that it took some time for the middle end to become a distinguished part of compiler architecture.

In general, though, when we speak of optimization without any clarification, we are talking about optimization for runtime speed. A program that is slow to respond is frustrating, and a program that is slower than necessary is wasteful of its users' time. In the past, when space was at a premium, it was acceptable to trade off speed for a

smaller code size: a program you cannot fit in the available memory is no more executable than the original source code and is even more useless; you can at least read and learn something from the source code. Optimizations for code size are still important when it comes to limited memory situations like those encountered in embedded systems and in situations where the resultant program must be transmitted through a low-bandwidth channel. In addition to code size, space optimization can also attempt to minimize runtime usage of space. A software developer working on a program will want a completely different sort of optimization, one that enables the easiest debugging and most helpful profiling.

Speed, space, debugging, profiling – doubtless you could come up with more optimization goals. While these goals of optimizing one or another property of the program sometimes align, they also frequently compete. Different compilers will be better or worse at different kinds of optimization. Some will let you configure the optimizations they perform, though their inbuilt preference for one or another kind of optimization will still show itself in the diversity and quality of optimizations of the different kinds available. Sometimes, the available optimizations can be switched on and off in ways that would mean something to the casual user, like letting the user specify different levels of optimization or that the end product should support debugging, for example, the GNU compiler collection's (GCC's) family of optimization flags `-O`, `-O0`, `-O1`, `...`, `-O3` and its debugging family beginning with `-g`. Sometimes, the optimizations can only be switched on and off in their own cryptic language; some examples from GCC would be `-ftree-dominator-opts` and `-fgcse`.

We will be focusing on optimizations for speed, but even there, there is much room for variation. Optimization happens piecemeal: each optimization attempts to produce a specific sort of improvement. Some improvements can hinder others. Some complement each other. Of-

ten, the interaction of two optimizations cannot be predicted, since the specifics of their interaction will depend on the target architecture and the function being compiled. When you consider that a whole host of optimizations is going to be performed, some of them multiple times, it becomes clear that the question of which optimizations should be performed when is not a trivial problem.

Optimizations are particular to their purpose and the type of program representation they work with. We discuss optimizations for imperative languages in Chapter 9, OPTIMIZING and the optimizations used in compilers for two different functional languages in [sections to be written].

4.4 BACK END: GENERATING TARGET CODE

The BACK END is responsible for completing the work of a compiler. It receives the program in some intermediate representation, itself might construct various further intermediate representations of the program, and ultimately produces the final representation, the program in the target language. The intermediate representation expresses a computation in a form understood by the compiler. The back end must take this and express it in the target language. This requires finding translatable units and recording their translation, then sequencing these translations for the best effect. This translation must obey whatever resource limits exist in the target language.

Here, we will focus on an instruction-set language as the target language. In this setting, the task of choosing how to represent the elements of the intermediate representation in the target language corresponds to instruction selection; ordering the translations corresponds to instruction scheduling; and working within the limits of the target language corresponds to register allocation.*

* This is true if the intermediate representation treats all data as being in registers except when it cannot. If the intermediate representation instead leaves all data in

These tasks are not cleanly separated. Choices made in each can (and when they cannot because of particular architectural decisions, they perhaps should) affect the others. The instructions selected to express a particular subcomputation can increase or decrease the demand on registers, which can require instructions be inserted to free up registers for other computations. The introduction of new instructions would strongly suggest that the whole sequence of instructions be rescheduled, which can again introduce problems with register load. Nevertheless, we will discuss them separately, because that is how they are best dealt with.

4.4.1 *Instruction Selection*

Instruction selection provides the basic material of the program in the target language. While instruction scheduling and register allocation are necessary for correctness, they simply rework the instructions generated in instruction selection.

Instruction selection is tasked with bridging from an intermediate representation to the actual target language. As with bridges, the nearer one side is to the other, the easier it will be to bridge the gap: the closer the intermediate representation is to the target language, the easier the job of instruction selection. If the intermediate representation is not very low-level, it will be necessary to convert it to something low-level. This will likely not be a very clean conversion if left to so late in compilation; there will be a lot of code meant to work around possible problems that may or may not be present because the results of earlier analyses that matter at a low level were not represented in the inter-

memory and moves it into registers for only as long as necessary, then REGISTER PROMOTION, which is the process of figuring out what data can be promoted from storage in memory to storage in register and then promoting it, is a better word for what occurs than register allocation. This promotion step is more a matter of taking advantage of the power of the language rather than one of restricting the translation to obey the language's limits. We will discuss register allocation here, but similar techniques apply to register promotion.

mediate representation. If the intermediate representation is low-level, but its architectural model differs from that of the target platform – the intermediate representation is stack-based or resembles the assembly language of a RISC machine, while the target platform is a CISC machine, say – it will be more difficult to perform instruction scheduling.

However difficult it might be, the same basic ideas suffice for instruction selection. To avoid clouding the exposition, we will assume the low-level intermediate representation that enters instruction selection is tree-based. A simple approach would simply walk the tree and generate, whenever possible, general instructions that ignore related nodes. A more complex approach would attempt to use local optimization and awareness of related nodes to build up a sequence of instructions.

A rather different approach uses a technique called peephole optimization that was originally developed to perform some last optimizations on the target code. It used a library of patterns to simplify a small window, or peephole, of a few instructions at a time. By scrolling this window through the entirety of the target code, less efficient code patterns could be replaced with more efficient counterparts. The limited window size keeps the process very quick, but all the same, it is able to perform some useful optimizations.

A Simple Tree-Based Approach

The simplest approach would generate instructions during a single tree walk. This would not be much more complicated than flattening the tree. It would also not produce very good code, since it would either make no use of context or only very limited use. Context is essential to producing a decent instruction sequence. A number of instruction sequences can be used to encode even straightforward arithmetic statements. Consider `x = y + 4`. Focusing on one node at a time, a RISC-like instruction sequence might load `y` into a register, load `4`

into a register, sum those values and store the result in yet another register, which becomes `x`.

$$
\begin{aligned}
r_y &\leftarrow y \\
r_4 &\leftarrow 4 \\
r_{y+4} &\leftarrow \texttt{add} \quad r_y \quad r_4 \\
r_x &\leftarrow r_{y+4}
\end{aligned}
$$

If we consider a bit more of the tree, we might load the value of `y` into a register, then use an immediate addition operation to compute `y + 4` and store the result into a register that represents `x`.

$$
\begin{aligned}
r_y &\leftarrow y \\
r_x &\leftarrow \texttt{addi} \quad r_y \quad 4
\end{aligned}
$$

Naïvely generating code to access array elements (which is how local variables are frequently represented at a low level in a program) can result in many redundant computations as part of the offset from the start of the array is calculated and recalculated and shared elements of the computations are not reused. Trying to eliminate redundant computations significantly complicates the code with special-case optimizations. Traveling any distance along this route of attempting to hand-optimize a simple scheme strongly suggests that more complex methods be employed. Fortunately, more complex methods are available.

Tree Pattern-Matching

Tree pattern-matching methods are instruction selection methods that use a store of tree patterns to build instructions. A common approach is that of BOTTOM-UP REWRITE SYSTEMS (BURS). Bottom-up rewrite systems work by tiling the tree with a stock of predefined patterns. As

each node is subsumed by a pattern tile, a choice is made based on the tiles of its subtrees that minimizes the cost of the tiling. The costs can be fixed or allowed to vary dynamically during the rewriting, say, to reflect the demand on registers introduced thus far. The costs can represent whatever it is one wishes to optimize for during instruction selection: code size, power consumption, execution time, or whatever else.

The patterns and costs used by a bottom-up rewrite system in tiling the tree can readily be represented in a table, which suggests the use of "code generator generators" similar to the lexer and parser generators used in producing the front-end, and such do exist. The rewrite rules make use of context-free grammars. Productions represent the shape of the tree. Costs are associated to each production, along with code templates. This is quite similar to an attribute grammar, and a bottom-up rewrite system likely could be described in that framework.

To tile the tree, we work from the bottom up, considering one node (as the root of a subtree) at a time. The tiling proceeds by identifying productions whose bodies match the subtree headed by the node currently under consideration. The least costly production is selected, and we move on to another node and its subtree. All the information we need know about a subtree is encoded in the head of the production selected when its root was considered. Once we have tiled the entire tree, a traversal fills in the code templates and records the instruction sequence.

This tree pattern-matching process is highly suggestive of number of other processes, which suggests adapting their techniques to fit this purpose. There are the classic pattern matchers, the finite automata; the context-free grammar component as well as the bottom-up method suggests adapting parsing techniques; a tree flattened into a string could perhaps be attacked using string matching methods (which, in many cases, ultimately end in use of finite automata); or, now that

the problem is better understood, we can hand-code a tree pattern-matching program.

Peephole

Peepholes are generally thought of in terms of PEEPHOLE OPTIMIZATION, as briefly described at the start of our discussion of the back end. However, their methods can also be used for instruction selection alongside optimization. The problem again reduces to pattern matching, but unlike in our discussion of tree pattern-matching, we assume the intermediate representation used for pattern matching with a peephole is linear, like the assembly code instruction sequences that peepholes were intended to optimize.

Instruction selection through a peephole begins by transforming the intermediate representation to an especially low-level form that models all the side-effects of instructions on the target machine. The peephole is used to simplify this instruction sequence and then to match patterns in this simplified sequence. These patterns are associated with code in the target language. Unlike the bottom-up methods used for trees, here the patterns are matched linearly and sequentially (visually, from top to bottom in the normal way of writing code).

What kind of simplifications can be seen through a peephole? Within a peephole, we can avoid unnecessary storage of temporary values in registers by substituting the value itself in place of its register in operations using the register. We can recognize a store followed by a load of the same value. Some simplifications might enable other simplifications. If we give the simplifier knowledge of when a given value is used by preprocessing the expanded low-level IR, we can jump back up and eliminate computation and storage of a value if later simplifications eliminate all its uses. Control flow complicates matters: should we use a strictly static window, or should our window include instructions that might follow during execution? Should we look at all uses

of a value together, ignoring intervening instructions, and proceed that way? The basic idea is amenable to considerable sophistication; the pattern-recognition and simplification part is, as with tree pattern-matching, also producible through a generator, at least as far as its basic elements go.

Clever implementation can enable the instruction selector to learn about simplification patterns. One can use well thought-out heuristics to quickly generate and test a variety of instruction sequences of various costs, simply by pasting together operations. Sequences that do no have the same effect as that identified for improvement are quickly discarded. Guided by a skilled compiler implementor and a suitable sampling of programs, this exhaustive search approach can be used during development to generate a sophisticated library of simplification patterns for later use.

4.4.2 *Instruction Scheduling*

Instruction selection produces a sequence of instructions, but its concern is generating instructions that can carry out the needed computations, not making sure all the instructions will work together: Does a use of a value come too soon after its definition, while the value is still being computed and not yet available? Will this introduce an error in the program, or simply unnecessary delay? Instruction scheduling worries about problems like these. It tries to ensure the selected instructions will work well together by reordering them. Its prime directives are to enhance instruction-level parallelism and reduce the time required by the program. It works at the block-level so that it does not have to deal with the consequences of control flow. It is hoped that stitching the scheduled blocks together will result in a good enough overall schedule, and this hope is generally realized.

Listing 4.1: Example of control dependence

```
bgtz r_x → TARGET
instructions...
TARGET: instructions...
```

Listing 4.2: Example of data dependences

```
1 r_x ← 4
2 r_y ← r_x + 1
3 r_z ← r_x × 5
4 r_x ← 5
```

What limits are placed on reordering? These limits are best expressed in terms of DEPENDENCES between instructions.* We always speak of the later instruction as depending in some way on the earlier. There are a variety of ways one instruction can depend on another. Perhaps the most obvious sort of dependence is CONTROL DEPENDENCE, when a sequence of instructions only executes if some condition is met. For example, in listing 4.1, the instructions between the first line and the line labeled `TARGET` are control dependent on the first line's instruction, which says to branch to the instruction labeled `TARGET` if the value in register r_x is greater than zero, since their execution depends on whether the first line sends the flow of control immediately to `TARGET`, skipping over them, or not.

Reordering also must respect DATA DEPENDENCES in the initial instruction sequence. To understand data dependence, it is necessary to think of an instruction as receiving input and producing output, which it stores in an output location, frequently a register. For example, the addition instruction in line two of listing 4.2 takes as input r_x and 1, produces their sum, and stores that value in the register r_y.

There are several types of data dependence. A TRUE DATA DEPENDENCE exists between two instructions when one requires a value created by another, that is, an input of the one is the output of the other:

* We do indeed mean "dependence" (plural "dependences") and not "dependency" (plural "dependencies").

we cannot use a value before it has been defined. Such is the case in lines one and two of listing 4.2, since the value stored in r_y on line two depends on the value of r_x defined in line one. Two instructions are OUTPUT DEPENDENT when both modify the same resource, that is, when both have the same output location. In listing 4.2, the instructions on the first and fourth lines are output dependent, since both store to r_x.

You might wonder whether it is also possible for instructions to be input dependent. It is not, as instructions are not kept from having their relative orders inverted simply because they share an input: whether line two of listing 4.2 precedes line three or line three precedes line two, both orders will result in the same values being stored to r_y and r_z. However, the idea can be useful, so some compilers will track it nevertheless as a sort of INPUT PSEUDO-DEPENDENCE.

In addition to these various kinds of data dependence, reordering must respect or eliminate ANTIDEPENDENCES. These are dependences between instructions that exist, not because of data flow, but because of conflicting uses of the same resources: specifically, one instruction is antidependent on a preceding instruction when its output location is used as input to the earlier instruction. The dependence is created solely by the reuse of the location: if the later instruction were to output to a different location and no other dependence existed between the two instructions, then they could be freely reordered with respect to each other. In listing 4.2, the instruction on line four is antidependent upon both preceding instructions.*

These dependences can be used to create a DEPENDENCE GRAPH (also called a PRECEDENCE GRAPH) representing the program, where each instruction is a node and there is a directed edge from a first node to a second whenever the second depends on the first. The graph is used along with information about the target platform to produce a

* Looping structures present a mess of dependence problems of their own. However, we do not discuss them here, as they are generally the target of analysis and optimization in the middle end.

schedule, which associates each instruction-node with a positive integer specifying the cycle in which it should be issued. The information needed is the functional units required by the instruction and the number of cycles the instruction takes to execute, called the DELAY of the instruction. If no value is required in the schedule before it is ready, the schedule will be correct. If there are never more instructions executing than the functional units can handle, and there are never more instructions dispatched in a cycle than is possible for the target platform, the schedule will be feasible. Within these constraints, we must attempt to schedule the instructions so that all dependences are respected and the cost of the schedule (often, the amount of time it requires) is minimized. This, of course, is an ideal that we cannot guarantee in practice.

The graph can be usefully annotated with the cumulative delay to each node starting from a root. The path from a root of the dependence graph to the highest-numbered leaf is called the CRITICAL PATH and critically determines the length of the schedule: no matter what, it can take no less time than the annotation at the leaf endpoint of the critical path.

List Scheduling

The dominant paradigm for scheduling is called LIST SCHEDULING. The basic idea of the method is, first, to eliminate antidependences by renaming the contested resources; next, to build the dependence graph; then, to assign priorities to each operation (for example, as determined by the cumulative delay at that operation's node); and, finally, to schedule the instructions in priority order as their dependences are fulfilled. This last step simulates the passage of cycles in order to track when operations can safely be scheduled and record the resulting schedule.

Clearly, there is a lot of detail missing from this sketch. The priority scheme used, for example, has an important effect on the resulting schedule, as does the tiebreaking method between operations with

identical priorities. There is no consensus on the best priority scheme, likely because there is no such thing. Further, we can perform scheduling either forward or backward. Working forward, we first schedule the instruction we want to execute in the first cycle, then repeatedly update the cycle counter and select the next instruction to execute. Before scheduling an instruction, we must check that sufficient cycles have passed that all instructions it depends on have completed. In the dependence graph, then, forward scheduling works from the leaves to the roots. Working backward, roots are scheduled before leaves, and the first operation scheduled executes last. We are then scheduling each instruction before the instructions it depends on. When we were working forward, before scheduling an instruction we would check that all instructions it depended upon had been completed in the simulation; now, working backward, we first schedule an instruction and then delay scheduling each instruction it depends on until we are far enough in advance of that instruction in our simulation that its result will be available to the already scheduled instruction. Neither forward nor backward scheduling is always best; since scheduling is fairly easily done, often a block will be scheduled both forward and backward, possibly a few times using different priority schemes, and the best of the schedules produced is then chosen.

There is also a lot of room to elaborate the method. Why limit ourselves to scheduling block-by-block? We can produce a better overall schedule if we look beyond a basic block. If either of two blocks can follow a single block, each successor block will work best with one or another scheduling of the predecessor, but we can only schedule the predecessor in one way. How should we decide which successor should determine the schedule of the predecessor? If we were to generate code for the region we wish to schedule, run it several times, and track which blocks execute most frequently, we could make an informed

decision. Some schedulers take this tack, called TRACE SCHEDULING since it makes use of an EXECUTION TRACE, or record of execution.

4.4.3 *Register Allocation*

Register allocation is the final step of code generation. Instructions have been generated and scheduled. Now, it is time to ensure they meet the platform's register constraints. The most fundamental constraint imposed on registers is the number available, but others must also be taking into account. These include constraints on register usage imposed by calling conventions and those imposed by register classes.

Register allocation is, in fact, an umbrella term for two closely related tasks: REGISTER ALLOCATION, which is the task of deciding which values should reside in registers when each instruction is issued, and REGISTER ASSIGNMENT, which takes the values to be allocated to registers and decides which register should hold which value when each instruction is issued.

Often, all values cannot be kept in registers. Dumping the register's contents to memory is called REGISTER SPILLING. Spilling a register is necessary but expensive. At each use, the data must be loaded into a "scratch register," and any changes to the data must be stored back to memory in order to free up the scratch register to load other spilled values. Sometimes, it is cheaper to recompute a value at each use than to go through the expense of spilling it and loading it back. Recomputation in place of register spilling is referred to as REMATERIALIZATION: rather than using previously provided "material," we are recreating it as needed.

Clearly, register allocation directly affects register assignment. Unfortunately, the interaction of the two concerns – what values should be kept in registers, and which registers should they be kept in – are not cleanly separated. Since we might be bound to assign particular

sorts of values to particular registers, issues of assignment can affect register allocation: we might be unable to use floating point registers for anything except floating point values, and a calling convention will likely specify that arguments to the procedure must be stored in specific registers, not just in some registers. Since marshaling data to and from register and memory itself requires registers, we do not even have the whole register set available.

The overall process of register allocation (which is what we shall mean by "register allocation" from now on), then, is nontrivial.* As with much in compiler design, we must resort to heuristics. We wish to minimize the amount of register-memory traffic by maximizing the amount of data kept in registers. A simple register allocator would consider only a block at a time. At the end of a block, it would spill all its registers. (A following optimization pass could attempt to remove unnecessary spills.) A top-down approach would estimate how many times a value is used in the block, allocate those to registers throughout the entire block, and spill the rest of the values used in the block to memory. A bottom-up approach would work through the block, instruction by instruction, and ensure that the operands of each instruction are in register. Where possible, it will use values already in register and load values into free registers. When all registers are full and a value not in register is needed, it will spill the value whose next use is farthest from the current instruction.

The top-down approach works, in a sense, by using detailed information about the block to set an overall policy, which it then follows. The bottom-up approach also uses detailed information about the block, but it makes its decisions instruction by instruction, rather than following an overall plan for the block. Its only plan is the same

* In fact, it ends up being NP-complete for any realistic formulation of the problem. A polynomial-time algorithm exists for the simplest of cases, as well as for SSA form (see Section 4.2, Intermediate Representations for a description of SSA form), but virtually any additional complexity – including the translation from SSA into the processor's instruction set language – promotes the problem to NP-completeness. Naturally, any time you actually find yourself needing to perform register allocation, you likely will not be dealing with the polynomial-time case.

for all blocks: make the tough decisions (which values to spill, which registers to use?) when it has to. This top-down–bottom-up dichotomy persists through all types of register allocators, though much the same effect can be achieved either way.*

More complex algorithms are required to handle register allocation across greater regions than single blocks. Modern register allocation algorithms often draw their inspiration from the graph coloring problem. Because of this, they are called GRAPH-COLORING REGISTER ALLOCATORS.

Graph-coloring register allocators begin by reformulating the problem of register allocation in terms of LIVE RANGES. A definition of a variable is LIVE until it is KILLED by a redefinition of the same variable. For example, if I assign 5 to the variable n, that definition is live until I later assign another value to n, say 6. The extent of the program where the definition is live is its live range.† All uses of the variable within a definition's live range refer to that definition. These live ranges are in competition for the limited supply of registers. Where they overlap, they are said to interfere with each other. From this, it is simple to construct an INTERFERENCE GRAPH: each live range is a node, and two nodes are joined by an edge whenever their live ranges interfere. Coloring a node corresponds to assigning its live range to a register.

After we have constructed the interference graph for a region, the nodes divide into two fundamental groups. Those nodes with more neighbors than there are registers are CONSTRAINED, while those with fewer are UNCONSTRAINED. This captures a basic distinction:

* It is an artifact of our simple description that the top-down allocator will dedicate a register throughout the entire block to a value heavily used in the block's first half but unused in its second, while the bottom-up allocator will choose to spill the value once it is no longer needed. Getting the top-down allocator to behave similarly, however, would make it less simple.

† In this case, we are concerned with the STATIC EXTENT, or SCOPE, of the definition. This is made explicit when the program is represented in static single assignment form. There is a corresponding notion of DYNAMIC EXTENT, which is the period of time when the definition is live at runtime, but this does not concern us here except as it is reflected in the definition's static extent.

the live ranges represented by unconstrained nodes can always be assigned to a register; those represented by constrained nodes must compete with their neighbors in the interference graph for the limited number of registers.

A top-down graph-coloring register allocator will use this graph to prioritize the live ranges. It will first try to color the constrained nodes based on the estimated cost of having to spill their associated live ranges. After that is done, it is trivial to color the unconstrained nodes. The devil lies in how to estimate spill costs and how to handle cases where this process results in nodes that cannot be colored.

Rather than using an overall estimate to determine the nodes' coloring priorities, a bottom-up allocator will work directly from the interference graph, node by node, to decide the order in which it will try to color the nodes. For example, it might pluck them out one by one, beginning with the unconstrained nodes, and place them on a stack.* To color the nodes, it works through the stack from top to bottom, gradually rebuilding the interference graph. It removes a node from the stack, reinserts it and its edges into the graph, and attempts to color it in the graph as it stands then.

If this process succeeds in coloring all nodes, register allocation is complete; otherwise, the bottom-up allocator must select nodes to spill and then insert the code to handle the spilled value.† If it has reserved registers to deal with this as we assumed earlier, allocation is complete, though such reservation might create a need to spill. On the other, if it has not, the changed program, which now incorporates the spill code, must undergo analysis and allocation anew.

---

* The bottom-up allocator would remove unconstrained nodes first for two reasons. For one, removing them first puts them at the bottom of the stack, which delays coloring them till the end. For another, removing them reduces the DEGREE, or number of neighbors, of neighboring nodes in the resulting graph. A node that was previously constrained might thus become unconstrained.

† An alternate tactic is LIVE RANGE SPLITTING. Instead of spilling an entire range, we split the range into two smaller ranges. This might divide the uncolorable node into two colorable nodes. If one split does not, further splitting eventually will: spilling the entire range corresponds to the finest splitting of all, where each use occurs in its own range.

Unfortunately, the interference graph does not capture all aspects of the problem, and so graph coloring does not provide a complete solution. These weak spots are also the points where graph-coloring register allocation can most be improved. This lack of a perfect fit also leaves room for other approaches with other inspirations, such as jigsaw puzzles: what is the best way to assemble the live-range pieces?

In the end, we do not need to find the absolute best register allocation. To carry out the computation specified by the original source code, it suffices to find a register allocation acceptable to the user. With that done, compilation can be considered complete. The program's odyssey through the compiler, its journey in many guises through the many parts, is at an end.

4.5 BOOTSTRAPPING, SELF-HOSTING, AND CROSS-COMPILING

Compilers have their own chicken-and-egg problem: Compilers are programs written in some language that compile programs written in a language, potentially programs written in the same language in which they themselves are written. Compilers written in their own source language are known as SELF-HOSTING COMPILERS and are a particularly puzzling instance of this problem. Further, compilers run on a variety of machines: where did the first compiler for a new machine come from? These problems have several solutions. One can go about growing a compiler incrementally, by way of another compiler, by way of an interpreter, or by cross-compiling.*

To grow a compiler incrementally, one implements a compiler for a subset of the source language in a language understood by an existing compiler (or even in machine language) and then uses this core language to write a compiler that can translate a greater subset of the

* T-DIAGRAMS are frequently employed to explain these methods, but I have always found them more confusing than helpful and omit them here. The T encodes the three relevant issues of a compiler: the source language to the left, the target language to the right, and the machine the compiler runs on at the base.

source language; this can be repeated as many times as necessary to encompass the entire language.

One can implement a compiler for the desired source language in a language already understood by a running compiler. Once that compiler has been used to generate a compiler for the new source language, a compiler for that language can be written using the language and compiled with this compiler to obtain a self-hosting compiler.

If an interpreter for the language exists, a self-hosting compiler can be written immediately and run in the interpreter on its own source code to create a compiled version of itself. Due to the comparative slowness of interpreted code next to compiled code, it may be necessary to interpret only a skeleton of the compiler and use that to compile only the same skeleton. This skeleton, for example, might omit all optimization and use only the simplest of algorithms for code generation. Once a skeleton compiler exists, it can be run on code for the full compiler, producing a compiler capable of optimization and clever code generation. This compiler, however, will not be as efficient as possible, since it was compiled with the skeleton compiler: this can be resolved by having the slow-running, full compiler compile its own source code, at which point the desired faster, full compiler will be obtained.

CROSS-COMPILATION is a quick way to produce compilers for new machines.* A cross-compiler is a compiler that compiles across target machines, for example a C compiler running on a SPARC machine but generating code runnable by a PowerPC. By cross-compiling the compiler itself, a compiler for the new machine is readily obtained. This allows one to leverage all the work put into creating the compiler for the original machine.

Clearly, a variety of solutions to this problem exist, but I hope the central idea of BOOTSTRAPPING has come through, that of using what

* Cross-compilers are also essential in embedded situations, where the target does not have the resources to run a compiler and it is impossible to develop an application for the machine using the machine itself.

you have now to pull yourself up to where you would like to be. The particular approach employed depends very much on the circumstances and on the preferences of the compiler writers.

4.6 BIBLIOGRAPHIC NOTES

Hopcroft, Motwani, and Ullman [55] is a standard textbook covering languages and Turing machines and touching on computational complexity. Its emphasis is on the abstract machines and the languages themselves as opposed to scanning and parsing. The classic reference for compiler design is Aho, Sethi, Ullman, and Lam [6], known affectionately as "the dragon book" for its cover art. (The color of the dragon is sometimes used to specify the edition.) Many more recent texts still refer the reader to it for its detailed information on scanning and parsing, which is dealt with more cursorily in more modern texts to allow more discussion of optimization. However, the dragon books, including the 2006 edition [6], preserve the original 1977 edition's [5] presentation of LALR parsers as the ultimate LR parser. The dragon books present LALR parsers as an improvement on LR parsers because they avoid the exponential space–time problem of the original LR algorithm. However, this problem had been addressed by Pager [96] in 1973. Pager's method was later illustrated and explained more clearly and briefly, though less formally, in Spector [120]. It is unfortunate that the dragon books appear not to take this not exactly recent development into account.

A good, modern, introductory textbook on compiler design is Cooper and Torczon [31]. Muchnick [93] picks up where a course using that book would leave off by giving more advanced information on the basic content and covering optimization and analysis in great detail. As one implements more optimizations in a compiler, the problem of optimization phase ordering, mentioned in 4.3 on page 73, grows in

importance. One cannot escape the problem even by forgoing compilers to code directly in assembly, as it affects even even hand-optimized code [54]. Kulkarni, Hines, Whalley, Hiser, Davidson, and Jones [68] describes an interesting attack on the problem by way of genetic algorithms.

Textbooks on compilers often seem to give the impression that scanning and parsing are solved problems and the world has moved on. While that might be the case for scanning, parsing is still an active area of research. The Purdue Compiler Collection Tool Set bucked the trend of providing LR-style parser generators in favor of developing an LL parser generator. This parser generator is now a project in itself, ANTLR (ANother Tool for Language Recognition) [97]. Other areas of research are implementing practical full LR parsers (see Menhir [114] for an example) and GLR parsers (for example, Elkhound [85]), as well as addressing problems of development of domain-specific languages and composable grammars; see, for example, Wyk and Schwerdfeger [137] and Bravenboer and Visser [23] and other work by those authors. A more thorough and up to date reference than Aho et al. [6] for parsing is Grune and Jacobs [48].

Another research direction in parsing theory is that of scannerless parsers. We have presented the lexer and parser as distinct but interacting modules of a compiler with their own theoretical bases. This division is standard in compiler design: from a software engineering perspective, it allows for the two to be tested and debugged independently; from a theoretical perspective, it allows research into the twin topics of regular and context-free languages to be utilized each to its utmost. Merging the two into a single, scannerless parser brings advantages, but it also presents problems. These are described in Visser [133]. A specific aspect of the solution to some of these problems is the subject of van den Brand, Scheerder, Vinju, and Visser [132]. Braven-

boer, Éric Tanter, and Visser [24] presents an example of a successful, practical application of scannerless parsers.

Attribute grammars augment context-free grammars with attributes and semantic rules in order to describe the program's semantic meaning. We developed attributed trees from parse trees; we next described how semantic rules are used to perform attribute evaluation in the context of an attributed tree; finally, we discussed the problems inherent in this theoretical framework, such as the difficulty it has handling nonlocal information such as that often stored in a symbol table in *ad hoc* methods of semantic analysis. There's more to be said about attribute grammars than this, though, and their uses extend beyond compilers. They can be put to good use in the generation of debuggers, syntax-aware editors, and entire interactive development environments. They are also useful in language design, while the semantics of the language are still changing rapidly. A good survey of attribute grammars as they are actually used is Paakki [95], which, in addition to explaining attribute grammars and giving examples of their use, introduces a taxonomy classifying the various attribute grammar paradigms that have developed.

5

CONCLUSION

This part provided background information essential to understanding the remainder of this work.

- In Chapter 2, BEGINNINGS, we introduced the basic ideas of lambda calculus and Turing machines. These provide the fundamental models of computation for the functional and imperative paradigms, respectively. This connection will be made clearer in the following parts.

- In Chapter 3, COMPUTERS, we used Turing machines as a bridge to modern computers. Succeeding sections described the three major parts of a computer: processor, memory, and input-output. Roughly, the processor is what lets a computer compute, memory provides storage, and input-output is what makes computers useful by allowing them to affect and interact with the world. We stressed the variety of processor architectures while giving some taste of that variety. We explained the existence of a memory hierarchy as well as the obstacle it presents to execution speed. We gave a rough sketch of how input-output is implemented in computers. We did not have much to say beyond this, since many of the details of input-output are more pertinent to programming languages themselves rather than their compilers.

- In Chapter 4, COMPILERS, we surveyed compiler architecture and design. We introduced the three-part structure of a compiler and discussed each part. Along the way, we sketched the theory that lies at the basis of each part and how it is used in practice. We also briefly surveyed intermediate representations

and their importance to the compiler. Lastly, we broached the chicken-and-egg issue of developing a compiler for a new programming language, implementing a compiler in its own source language, and similar compiler construction problems. The important point is that compilers neither develop in a vacuum nor spring fully-formed from the pregnant void, but evolve gradually, though this evolution may involve the seemingly contradictory device of the compiler effectively "pulling itself up by its own bootstraps."

Part II

IMPERATIVE LANGUAGES

6

OVERVIEW

The family of imperative programming languages is large. It represents the most commonly employed programming paradigm, and it is in light of it and its accomplishments that all other language families are judged. Before we begin to discuss functional languages, it behooves us to examine their less exotic imperative cousins.

DEFINING looks at what we mean by "imperative language," discusses how imperative languages have developed over time, highlights central concepts of imperative programming, and mentions several problems inherent in the paradigm.

COMPILING discusses issues particular to compiling imperative languages.

OPTIMIZING describes how optimization is performed and introduces basic analyses and optimizations used with imperative programs.

7

DEFINING

7.1 HISTORY AND CONCEPTS

We earlier said that Turing machines inspired the imperative paradigm. What is paradigmatic about Turing machines? It is their sequential operation and reliance on state in order to compute. A Turing machine computes move by move, at each step writing to its tape, updating its internal state, and shifting its read–write head a tape cell to the left or right. Its behavior is time-determined: depending on how it has modified its tape and the state it finds itself in after passing over the cells along its computational trajectory, it makes one more move, move after move, in sequence.

We then anchored our understanding of modern computers in Turing machines. These elaborate, complex machines grow out of this simple seed, but they have not left their roots. We find again a reliance on state and sequence. The program counter advances cycle by cycle as the contents of registers and memory change.

We mentioned assembly languages in passing then and described their simple form, abbreviated mnemonics for machine instructions and operands. The imperative language family grows out of these and so inherits the Turing machine spirit. They have become more elaborate and complex over time, much as computers elaborated on the fundamental concept of Turing machines.

The first imperative language was FORTRAN , which is short for "the IBM Mathematical Formula Translating System." It was developed at IBM in the 1950s as a language for scientific computing and pioneered

compiler design and optimization, since no-one would trouble to make the switch from writing hand-written, hand-optimized assembly code to FORTRAN unless it ran nearly as fast as such assembly code. The language theory we introduced in the chapter on compilers had not yet been developed, and *ad hoc* techniques were used instead. The problems encountered in inventing and applying these techniques spurred the development

FORTRAN was a child of its time. It relied on a fixed format for code entry that was based on the punch cards used at the time. Each line of eighty characters was broken into fields of different, fixed widths, such as for numeric statement labels that were used in branching instructions, for the "continuation character" that indicated that the line it was on continued the previous line rather than beginning a separate statement, and for the program code itself. All of its control flow statements (other than the sequencing implied by one line of code following another) relied on the numeric labels. An example, peculiar to FORTRAN, is the ARITHMETIC IF STATEMENT. The arithmetic if transferred control to one of three statement labels depending on whether a provided arithmetic expression was less than, equal to, or greater than zero: `IF (`⟨`expression`⟩`)` ⟨`statement label`$_1$⟩, ⟨`statement label`$_2$⟩, ⟨`statement label`$_3$⟩.

FORTRAN II was the first FORTRAN to support procedures. Procedures allow algorithms to be specified in isolation from the rest of the program and reused. Within the PROCEDURE BODY, FORMAL PARAMETERS specified in the procedure's declaration are manipulated in place of the actual arguments that must be manipulated in the program. The formal parameters are "dummy variables" like the x in the mathematical function $f(x) = x^2$ and exist to give a name within the function's body to all values substitutable for the formal parameter. When the procedure is called from the program, actual arguments are supplied; these are bound to the formal parameters by their position,

so that the first argument is referred to by the first formal parameter, the second argument by the second formal parameter, and so forth; the computation specified by the procedure body is carried out; and, at the end of the procedure, control returns to the caller and continues where it left off. This is a powerful abstraction, and the style of programming it gives rise to is sometimes called PROCEDURAL PROGRAMMING.

FORTRAN procedures used what is known as a CALL BY REFERENCE EVALUATION STRATEGY. An evaluation strategy describes the way in which expressions are evaluated; the expressions that the various names for evaluation strategies focus on are, conveniently enough, functions. In call by reference evaluation, the arguments are bound to the formal parameters in such a way that modifications to the formal parameters affect the arguments' values. Suppose we have defined a function `MAKE-SIX` that expects a single parameter and simply sets that parameter equal to 6. If the value of `x` is 5, and we call that function with `x` as its argument, then, following the function call, `x` will have the value 6.

FORTRAN continues to be used and updated today. It remains a language meant for scientific computing, but now, it is trailing the innovations of other imperative languages. One of these innovations is STRUCTURED PROGRAMMING, which does away with goto-based control flow in favor of STRUCTURED CONTROL FLOW. The structure is one based on higher-level control flow constructs like logical if statements that execute their body depending on the truth value of the provided expression, various looping constructs that repeat their body in a predictable way, and function calls. Rather than having to deduce that one of these common patterns is being used by deciphering the various uses of labels, it is plain; rather than having to frame these relatively intuitive forms of control flow in terms of statement labels and jumps, one can express these patterns directly. This addressed the problem of "spaghetti code" with convoluted, goto-based control flow

that made it difficult to understand and predict the operation of a program.

Another early imperative language family was ALGOL , short for algorithmic language. ALGOL was the product of a joint effort between European and American computer scientists to produce a common language for specifying algorithms. Each version is named by its debut year, starting with ALGOL 58. The development of ALGOL saw the birth of BACKUS NORMAL FORM, abbreviated BNF and now read as BACKUS-NAUR FORM, a notation for specifying grammars that has been slightly extended and used extensively since.

There were many official, numbered ALGOL versions, and even more extensions and variations developed in practice, and we will ambiguously collapse them all into the single identifier ALGOL. ALGOL featured CALL BY VALUE, CALL BY NAME, and CALL BY REFERENCE evaluation strategies. With call by value, the value of the argument is provided as the value of the formal parameter, but modification of the formal parameter affects only the parameter within the function body and not the original argument. Call by name is similar to call by reference, except that each use of the parameter causes reevaluation of the associated argument. If the argument is simply a value, this is no different, but if it has side-effects, say, is a function that increments some global counter, this will be carried out each time the value of the parameter is used. This allowed for some confusing behavior, and call by reference was preferred and enforced by later versions of ALGOL.

A notable descendent of ALGOL is the C programming language, whose development began in the 70s with the UNIX OPERATING SYSTEM and continues today. As its development alongside an operating system might suggest, C was intended as a SYSTEMS PROGRAMMING LANGUAGE offering relatively detailed, low-level control of the computer it is operating on. This is reflected in its use of unrestricted MEMORY POINTERS which refer to addresses in the computer's memory.

As UNIX rose in prominence, so too did C. C is still widespread and fairly widely used, and it functioned in some sense as a *lingua franca* of computing, as the ALGOL family did before it. C's mark on the imperative language family is most notable in its visual appearance, where semicolons are used as statement separators and curly braces (`{` and `}`) are used to delimit blocks in place of other textual indicators such as `BEGIN` and `END`.

Following structured programming, the next development in the imperative family was OBJECT-ORIENTED PROGRAMMING. Object-oriented programming introduces the concept of an OBJECT as a higher-level unit of abstraction than the function. An object is a bundle of state and METHODS (functions) that operate on that state. Objects are instances of a CLASS or type of object, by which we mean that the objects are structured as described by the class, though each might be in a different state. Object-oriented programming entered the C family by way of C++ and, later, Java, of which the latter has perhaps replaced C as computing's *lingua franca*. These languages also introduced powerful MODULE-like abstractions that allowed definitions to be grouped at an even higher level into packages or namespaces. This helps to avoid naming conflicts, which become more and more of a problem as a program grows larger and larger and requires more and more variable names, which prevents one body of code from interfering with another and eases reuse.

Java is notable for providing GARBAGE COLLECTION, sometimes known as AUTOMATIC MEMORY MANAGEMENT. This means that the programmer is no longer responsible for indicating when some storage referred to by a variable is no longer needed. Instead, the runtime system attempts to discover when some storage is no longer reachable or no longer needed and reclaims this by freeing up the space for use with other variables. There is a cost associated to this at run time, since the run time system must track this storage and reclaim it and lacks

the knowledge of the program that the programmer has. This cost prevented its widespread adoption prior to Java. But there is also a cost to not providing garbage collection, since manual storage management has proven to be a difficult and time-consuming issue that can cause subtle problems in a program. The price is paid in development rather than at run time.

Java is also notable as being designed to run on its own associated platform, the Java VIRTUAL MACHINE (VM), rather than in a specific machine environment. This enables programs written in Java to run on any platform for which a Java virtual machine has been implemented. This too comes at a cost: the program is interpreted by the virtual machine which then uses the underlying machine to carry out the specified operation. This can be a slow process. Innovations in interpreter design and compilers have done much to ameliorate this, but it is still an issue.

Thus, imperative languages developed, in a sense, as abstractions of assembly language. They continue to rely on sequencing and state to perform computation and explicitly describe the process of computation: do this, then do that, then.... Mathematical notation can be used to specify formulas. Functions abstract common operations. Modules abstract over related definitions. Classes abstract over functions (now called methods) and state and allow programs to better resemble the real-life objects they are modeling. They frequently employ call by value and call by reference evaluation strategies alongside procedural and structured control abstractions. They are generally statically scoped: the extent of a variable depends primarily on where it falls in the textual description of a program. These scopes can, and do, nest, both statically (if blocks within if blocks, say) and dynamically (a procedure call places one in a new scope).

7.2 PROBLEMS

The imperative paradigm has problems dealing with architectures that do not reflect its heritage. Its reliance on statements that execute in sequential order, affect the associated program state, and use that state to determine their sequencing has serious problems dealing with CONCURRENT EXECUTION, in which several threads of execution can be acting simultaneously and affecting each other in nontrivial, and sometimes problematic, ways. This same reliance also limits the ways in which imperative code can be composed and reused. Modules, classes, procedures, and scoping in general exists in part to address this problem by partitioning the namespace, so that one block of code's variable `global_name` can differ from another's. As more and more lines of code are added to a program, the interaction between various side effects on the environment and state that are implicit in different functions and statements compounds. To sum up the imperative paradigm's problems in a single word, the problem is with scale: growing larger, in programs, in number of executing threads and processors, in problem complexity, poses a serious problem to a language family with such humble, historical origins.

7.3 BIBLIOGRAPHIC NOTES

The proceedings of the Association for Computing Machinery's few History of Programming Languages (HOPL) conferences published much valuable and fascinating material on the development of programming languages. The strongly historically rooted description of the imperative languages reflects my own views on the subject. Much of the material on functional languages contrasts them with imperative languages and discusses failings of imperative languages that appear in contrast; the problems I mention spring from this sort of compari-

son. If one compares the many branches of the imperative family, other problems and tradeoffs come into view, but, while interesting in themselves, they are not relevant here. The appendices of Scott [117] feature an interesting family tree of the imperative languages as well as many capsule summaries of programming languages. The book itself is a good starting point if the diversity of programming languages catches your interest, though, as is usual, the bulk of its focus is on imperative languages.

8

COMPILING

The common overall approach to programming adopted by imperative languages gives rise to common issues in their compilation. A few such issues are scoping, data storage, and common data types such as arrays.

8.1 STATIC AND DYNAMIC LINKS

Static scoping means that the variables of enclosing scopes must be accessible from within their enclosed scopes. It also means that inner scopes must be able to MASK or SHADOW the variables of outer scopes, creating a variable with an identical name in an inner scope that then makes the same-named variable in the outer scope inaccessible from that inner scope.*

We mentioned the use of a symbol table as a form of intermediate representation. But a single symbol table cannot cope with this nesting. Instead, NESTED SYMBOL TABLES are used. As each block is entered, a new symbol table is created. On leaving, the symbol table is destroyed.

Listing 8.1: Scopes and shadowing

```
int x = 5;
std::cout << x; // prints 5
{ /* now in brace-delimited inner scope */
    int x = 6; // this x masks the outer
    std::cout << x; // prints 6
} /* back in outer scope: outer x no longer masked */
std::cout << x; // prints 5
```

* Some languages do provide a way to refer unambiguously to variables in enclosing scopes, such as Tcl's `upvar` and `uplevel`.

This corresponds to the definitions of the inner scope going out of scope.

The different blocks are connected in two ways: statically and dynamically. By statically, we mean lexically, in the textual representation of the source code. By dynamically, we mean at runtime, in the sense that the scope of a called function or other such block nests within that of the caller, regardless of where the block is located statically. The symbol tables are connected in two ways that reflect this distinction, via a STATIC LINK and a DYNAMIC LINK. If a variable's definition is not found in the local symbol table, the static link is followed up and that symbol table checked; this process is repeated till there is no enclosing scope, at which point we must conclude the variable is simply undefined. Similarly, when a function is called, its scope is dynamically linked to that of its caller: at the end of the function, control will return to the caller; exiting a function corresponds to returning along the dynamic link.

Object-oriented programming introduces one more kind of link through its class hierarchies. Class hierarchies allow subclasses to inherit the definitions and so state and behavior of their ancestors. Often, this is used to create multiple specializations of a more general class, as might be used to treat the relationship between cars and boats with the more general class of vehicle. The superclass link can be thought of much like an extension of the static link to handle the class structure of object oriented programs.

8.2 STACKS, HEAPS, AND STATIC STORAGE

Symbol tables hold symbols that allow us to access data. Where should this data be placed? The way inner scopes nest within outer can be seen as analogous to a piling up of definitions. Those on top mask those below; when we leave a scope, we pop it off the stack to reveal

the scope below. Data that comes and goes with a scope can then be allocated on a stack. The behavior of functions and their data is similar, as suggested by the use of the dynamic link. Data can be allocated as part of the call stack; once we return to a lower level in the stack, that is, once we return from the function, its local data is no longer needed. Thus, local variables are STACK-ALLOCATED.

Data whose extent is not related to these ideas, such as that stored in space allocated by the programmer and freed either by the programmer or automatically during garbage collection, cannot be allocated on the stack. Instead, it must be stored safely away till needed later or till we know it is no longer needed. The region where such data is allocated is known as the HEAP, and such data is said to be HEAP-ALLOCATED.*

Data that must persist throughout the program's execution is called STATIC and is typically STATICALLY ALLOCATED. Such data is allocated when the program is initialized and not freed till the program terminates.

Stack-allocated data is handily managed implicitly by the call and return sequence and nesting of scopes, while heap-allocated data can be more troublesome. A typical layout in memory for the stack and heap places their starting points at opposite ends of the space available, so that they both have the most space possible. The total stack and heap size is limited, but neither is arbitrarily limited beyond that, as both grow towards each other from opposite ends of the memory space.

8.3 ARRAYS

Arrays are the most common COMPOUND DATA TYPE, so called because they are a data type, array, of some other data type, the base

* This heap has nothing to do with the heap data structure, which is often used to implement priority queues.

type. An example would be an array of integers, `int[] array = {0, 1, 2, 3}`.

With arrays, there are questions of central interest to the programmer, since they are matters of syntax and convenience. How are the elements indexed? Is the first element number 1 or number 0? Can the lower bound be changed? What information is available at runtime about the array, for example, its size, lower bound, or base type?

Then, there are questions that face the compiler writer but are less directly a language user's concern. The most obvious of these is how to lay out a multi-dimensional array in one-dimensional memory. There are three popular approaches, each used by a major programming language. Their names are biased to two-dimensional arrays, but the ideas generalize to higher-dimensional arrays.

COLUMN-MAJOR ORDER places elements in the same column in adjacent memory locations. This order is used by FORTRAN.

ROW-MAJOR ORDER places elements in the same row in adjacent memory locations. This order is used by C.

INDIRECTION VECTORS use pointers to one-dimensional arrays. This is the approach adopted by Java.

The choice of column-major or row-major order influences which is the best order to traverse an array: traversing a column-major array by row requires jumping through memory, while traversing it by column simply requires advancing steadily, bit by bit through memory. For higher dimensional arrays, row-major means the rightmost index varies fastest, while column-major means the leftmost index varies fastest. Accessing an element is done by arithmetic, which is used to calculate an offset from the BASE ADDRESS of the array.

An example should clarify this. Suppose we represent something's address by prefixing an at-sign `&` to it. Say we want to access the element `a[3][5][7]` of a $10 \times 10 \times 10$ row-major array of integers where

each dimension's lower bound is 0. We calculate its address as an offset from `&a` in terms of the size of an integer s as follows:

A. Find its offset from the start of the highest dimension. Here, that is the third dimension, and we will call the offset o_3. We want to know the start of the eighth element, which is the length of seven integers beyond the start of this dimension. Thus, $o_3 = 7 \cdot s$.

B. Find where that dimension begins in terms of the start of the next lower. Here, that would be o_2, and that would place us past five runs of ten integers apiece, since each dimension is ten integers long. Thus, $o_2 = 5 \cdot 10 \cdot s$.

C. Repeat the previous step until we run out of lower dimensions. Here, we need only repeat it once more, to find the offset to the start of the second dimension. That is past three runs of ten runs of ten integers apiece, thus, $o_1 = 3 \cdot 10 \cdot 10 \cdot s$.

D. Finally, sum all the offsets. This is the offset from the base address. Added to the base address, it gives the address of the desired element. Thus, our element is the s amount of data starting at $\&a[3][5][7] = \&a + o_1 + o_2 + o_3$.

It is not hard to see how this computation can be simplified:

$$\begin{aligned}\&a[3][5][7] &= \&a + o_1 + o_2 + o_3 \\ &= \&a + 3 \cdot 10 \cdot 10 \cdot s + 5 \cdot 10 \cdot s + 7 \cdot s \\ &= \&a + s(7 + 10(5 + 10(3)))\end{aligned}$$

The computation can also be generalized to handle nonzero lower bounds and arrays of more dimensions.

Indirection vectors are similar, but different. Every n-dimensional array with n greater than 1 is simply an array of addresses of arrays representing the $n-1$ dimension, except the final dimension, which is

a 1-dimensional array of the base type. To access `a[3][5][7]` as we did above, we would:

A. Find the address stored at index 3 of the array of addresses.

B. Dereference that address to access the next dimension. Repeat the previous instruction with the index at that dimension, and so on till we arrive at the last dimension, at which point we proceed by simple one-dimensional array arithmetic to retrieve the value.

If we represent dereferencing by a star $*$, then we can write this:

```
/* a is a multidimensional array's name */
base = &a;
b = (*base)[3];
c = (*b)[5];
value = (*c) + 7 * sizeof(int);
// equivalently,
//     value = (*(*((*base)[3])[5])) + 7 * sizeof(int);
```

Indirection vectors replace the cost of array arithmetic with pointer dereferencing. Rather than calculate a complicated offset, they acquire the next needed address directly from a trivial, one-dimensional offset of a known address.

8.4 BIBLIOGRAPHIC NOTES

For further information on these topics, almost any text on compilers will do, though only more recent texts such as Cooper and Torczon [31] will discuss issues germane to object-oriented languages.

9

OPTIMIZING

Our discussion of optimization in Section 4.3, Middle End: Optimizing the IR focused on the variety of properties of a program one might wish to optimize and gave examples such as speed, size, and power consumption and the problem of optimization phase ordering. Now, we will describe how one optimizes a program's representation, along with some examples of common optimizations applied to imperative programs.

Optimization comprises two closely-related tasks: analysis, which gathers the information needed for an optimization, and application of the optimization. Each application of an optimization changes the representation of the program, so a single analysis is likely to be repeated several times. Optimizations and their analyses range from being quite generally applicable to being quite specific to the language, even to specific uses of the language. The source code might even be written in order to ease optimization, possibly through annotations with no meaning in the source language but helpful to the optimizer. Program analysis and optimization is a fruitful area of research, with new analyses and optimizations being developed constantly and older ones refined. The details of implementation are specific to each compiler and its chosen intermediate representations; as such, the intermediate representation itself contributes in important ways to optimization. Indeed, static single assignment form was developed, and is extensively employed, to ease optimization.*

* For more on SSA, see Section 4.2, Intermediate Representations.

9.1 ANALYSIS

Analysis is integral to optimization. The variety of analyses are often loosely classified based on their subject. Perhaps the broadest class is DATA FLOW ANALYSIS. It can be distinguished from classes such as ALIAS ANALYSIS, which deals with attempting to discover which names in the program refer to the same data (that is, are ALIASES for the same data), control flow analysis, and dependence analysis.

9.1.1 *Control Flow*

Control flow analysis is necessary to perform almost all other analyses. The aim of control flow analysis is to deduce the control flow relationships between the elements of the intermediate representation. In a linear IR, this makes explicit the sequential flow between adjacent statements as well as that created by jump statements, goto statements, and more structured control flow statements.

There are several approaches to control flow analysis varying in their applicability, speed, and the type of information they provide. Some methods can produce a relatively high-level structural analysis of control flow that recognizes the type of control flow created by the use of structured programming statements such as `while`, `if-then`, `if-then-else`, and `case`. Others can do little more than recognize the presence of some sort of loops as opposed to simple sequential control flow.

Structural Units

It is necessary to understand the structure of the control flow graph in order to understand the various scopes of analysis and optimization. The fundamental element of a control flow graph, typically constituting the nodes of the graph, is the BASIC BLOCK (BB), a maximal se-

quence of instructions that must be executed from start to finish. This bars the possibility of either entering or exiting from the middle of a basic block, so that, for example, labeled statements can only begin a basic block. Procedure calls are a matter of some delicacy, and whether they are treated as interrupting a basic block or not depends on the purpose of the control flow analysis being performed. They might even be treated in both ways. Delayed branches also introduce problems as to how the instructions in the delay slots should be treated; fortunately, this issue can largely be ignored except for very low-level representations on architectures that make such delays visible.

We say a basic block with more than one predecessor in the control flow graph is a JOIN POINT, since several flows of control come together in that block. A basic block with more than one successor is similarly called a BRANCH POINT. A single basic block can be both a join point and a branch point.

A slightly larger structural unit is the EXTENDED BASIC BLOCK (EBB). Extended basic blocks comprise a rooted control flow subgraph. Its root is a join point. An EBB is the largest connected set of basic blocks reachable from the join point that are not themselves join points. Thus, if control reaches any of the blocks in the EBB, it must have gone through the root.

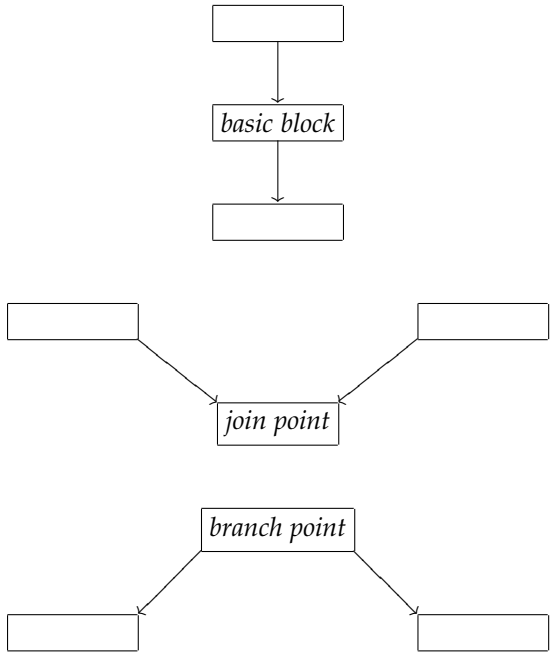
The procedure itself forms the next largest generally recognizable structural unit, though this is defined not in terms of the graph but rather by the program itself. The largest unit is the entire program. In between extended basic blocks and an entire procedure sit regions of various sorts, defined as suitable for different analyses and optimizations.

Scopes of Analysis and Optimization

Corresponding to these structural units are the different scopes of analysis and optimization. These names are used to describe the subgraphs

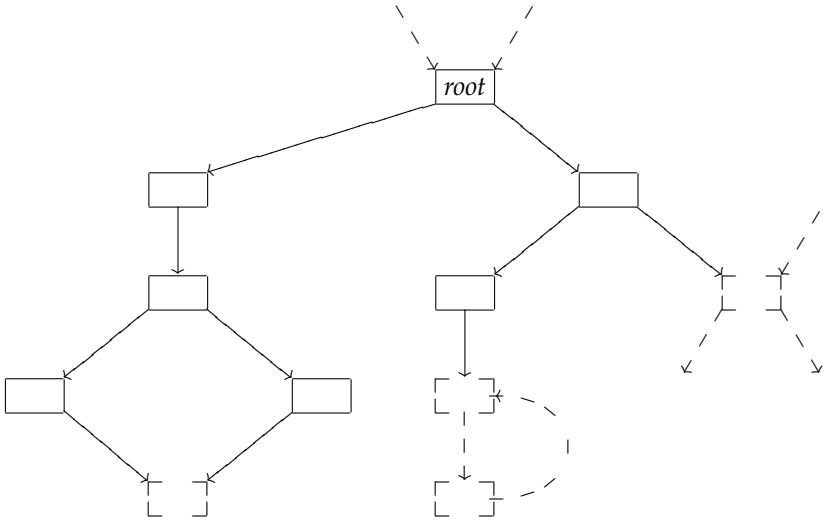
Figure 8: Structural unit examples

(a) Basic blocks



(b) Extended basic block

Blocks with a solid outline are part of the extended basic block. Blocks with a dashed outline are not.



of the control flow graph considered during a given analysis or optimization.

LOCAL SCOPE corresponds to a single basic block.

SUPERLOCAL SCOPE corresponds to a single extended basic block.

REGIONAL SCOPE corresponds to a region not otherwise specified.

GLOBAL SCOPE (also called INTRAPROCEDURAL SCOPE) corresponds to an entire procedure.

WHOLE-PROGRAM SCOPE is unambiguous; you might sometimes see it called INTERPROCEDURAL SCOPE as well, particularly in the phrase "interprocedural analysis," which describes a variety of often rather intractable analyses.

"Global scope" might appear to be a misnomer for anything less than the entire program, but it is generally preferred to "intraprocedural analysis," since that sounds altogether too much like "interprocedural analysis." Global analysis encompasses a procedure's entire control flow graph; interprocedural analysis must cope with a number of control flow graphs, one for each procedure.*

9.1.2 *Data Flow*

Data flow analysis, together with control flow analysis, is the bread and butter of optimization. It can frequently be performed alongside control flow analysis: both require similar techniques for building and propagating information. Where control flow analysis concerns how basic blocks are related, data flow analysis concerns how various kinds of data are communicated along those relations.

* Attempts to introduce "universal" as a synonym for "interprocedural" as "global" is used for "intraprocedural" were unsuccessful, but the contrast between the two names might help to remember the distinction.

Data flow analyses are posed as data flow problems. An example is the REACHING DEFINITIONS PROBLEM: What definitions of a variable could still be in force (LIVE) at the point of a given use of that variable? Similar is the problem of UPWARD EXPOSED VARIABLES: Control flow graphs are generally drawn so control flows from top to bottom, and this question asks, what variables must have been defined upward of a given basic block?

These two problems typify two major classes of data flow problems, the FORWARD DATA FLOW PROBLEMS, like reaching definitions, and the BACKWARD DATA FLOW PROBLEMS, like upward-exposed variables. These are so called because they require propagating information either forward along the control flow graph's edges or backwards. A third, rarer, and more troublesome class is that of the BIDIRECTIONAL DATA FLOW PROBLEMS. This class is troublesome enough that it is often either not bothered with or reformulated in terms of the other two, as was the case for PARTIAL-REDUNDANCE ELIMINATION, which seeks to discover computations of the same value that are performed multiple times along certain paths through the control flow graph.

Data flow analysis is well enough understood that it can be automated in good part by the appropriate tools. This understanding is based theoretically upon LATTICES and FLOW FUNCTIONS. Lattices are structured to correspond to the properties of the program under analysis. Flow functions allow us to abstractly model the effect of parts of the representation on those properties. Together, these let us abstractly simulate the effects of executing the program and discover intrinsic properties of the program, frequently independent of input values and control flow. Data flow problems can be posed in terms of these lattices and flow functions. Solutions to the problems become the FIXED POINTS of properly formulated data flow equations, which can be solved by iteration and, often, several other quicker and more

clever methods. Solvability of such problems can be guaranteed for a variety of flow functions.

An issue that is not dealt with directly by these abstractions is the correctness of transformations based on these analyses. The analysis must be performed with an eye towards the optimizing transformation that will be based on its results. We wish to be as aggressive as possible in optimizing the program, but we cannot be so aggressive that we do not preserve its behavior. In developing and implementing optimizations, we walk a fine line between aggressiveness and conservatism: if we are too conservative in drawing conclusions from our analysis, we will fail to improve the program as much as possible; if we are overly aggressive, we will be unfaithful to the original program, and the results will be something related but different. This dilemma has its parallel in translation: a word-for-word translation is stilted and awkward, but without great care a more natural, fluid translation risks departing from the meaning of the source text.

9.1.3 *Dependence*

Dependence analysis aims to discover computational dependences of various sorts between the elements of the representation, often the individual statements in a low-level linear representation. It is central to instruction scheduling and is discussed in detail in Section 4.4, Back End: Generating Target Code.

9.1.4 *Alias*

Alias analysis is concerned with determining when and which different names can refer to the same data. The way aliases can be created and used varies greatly from language to language. Alias analysis is often quite difficult. Making the most conservative assumptions possi-

ble – namely that all data whose addresses have been made available can be affected by any alias in the program – can guarantee correctness at the expense of preventing possible optimizations. The cost of these hyperconservative assumptions varies depending on the program. In programs that do not make extensive use of aliases, this assumption might not pose much of a problem. In programs that do use aliases extensively, the assumption could bar almost all optimization. Thus, alias analysis is necessary to creating an aggressively optimizing compiler.

There are two parts to alias analysis: ALIAS GATHERING, which discovers which variables are aliases and basic information about what data is aliased, and ALIAS PROPAGATION, which propagates this information throughout the program and completes the analysis. The propagation phase can be modeled as a data flow problem.

There are a few distinguishable types of alias information. MAY INFORMATION describes what may happen but does not always happen. This information must be accounted for and its effects allowed, but it cannot be depended on to occur. MUST INFORMATION is information about what must happen. This information is very useful in performing optimizations. Alias information (and the analysis that produces it) can also be FLOW SENSITIVE and FLOW INSENSITIVE. The flow here is control flow. Flow insensitive analysis is simpler and generally can be performed as several local analyses that are then pieced together to form the whole of the information. Flow sensitive analysis is more difficult, both computationally and conceptually, but also more detailed. It requires understanding and, to some extent, simulating the program's control flow. The combinations of these factors – must, may, flow sensitive, and flow insensitive – determine how long the analysis takes, the usefulness of the information, and both the optimizations that can be based on the information and the extent of those optimizations.

9.2 OPTIMIZATION

9.2.1 *Time*

As discussed above, optimizations can be classified by their scope. They can also be classified by when they are generally applied: early in compilation, somewhere in the middle, or later. Time generally corresponds to the level of abstraction of the program representation. Early on, the representation is much closer to the source language than later, when it generally becomes much closer to assembly language.

9.2.2 *Examples*

We will now give several examples of optimizations. We will name the optimization, briefly describe it, and then give an example of some source language code. We then demonstrate the results of the optimization as transformed source code and provide a description of the transformations performed.

COMMON SUBEXPRESSION ELIMINATION There are several varieties of common subexpression elimination depending on the scope and approach. They all have the aim of avoiding redundant computation by reusing existing values. Common subexpression elimination can be usefully applied both early and late in compilation.

Listings 9.1 and 9.2 provide a simple example of common subexpression elimination. The initial assignments of both `i` and `j` require the computation of `a * b`. Common subexpression elimination will factor out this computation into a temporary value so that it need only be computed once. Notice that common subexpression elimination tends to increase register pressure, since the values of the common subexpressions must now be preserved beyond their first, immediate use.

DEAD AND USELESS CODE ELIMINATION This optimization is easy to describe, though in practice it ends up being applied in a variety of cases. It simplifies the representation, which speeds the following analyses and transformations. It is commonly run many times during compilation. It aims to eliminate code that is USELESS, that is, that computes a result no longer used, and code that is DEAD or unreachable. Dead and useless code can be the result of textual substitution as done by the C preprocessor in expanding macros or the result of other optimizations eliminating all uses of a definition or all statements in a block.

The example in listings 9.3 and 9.4 on page 123 is a very artificial example in C. Code portability is often achieved in C by writing suitable preprocessor macros that are then configured based on the environment in which the code is compiled. Environment-dependent code that cannot be dealt with abstractly through macros is included conditional on other macros representing the platform. The preprocessor's conditional statements are normally used to selectively include code for compilation by the compiler as opposed to the conditional statements of the language. Here, we instead use the C language's conditional statements to include code. This results in dead and useless code that would be removed by dead and useless code elimination, as shown in the transformed code.

CODE HOISTING Code hoisting is so called because it corresponds visually to lifting a computation up in the control flow graph, which is usually drawn so that control flows from top to bottom. Rather than specifying that a computation occur in all branches of an extended basic block, we might be able to hoist the computation up to the common ancestor of all those blocks so that it is specified only once. This reduces code size. Code hoisting is a type of CODE MOTION.

Listing 9.1: Common subexpression elimination: Source code

```
int i, j;
i = a * b + 3;
while (i < 10) {
    i = i + 10;
}
j = a * b + i;
```

Listing 9.2: Common subexpression elimination: Transformed code

```
int i, j, t1;
t1 = a * b;
i = t1 + 3;
while (i < 10) {
    i = i + 10;
}
j = t1 + i;
```

Listing 9.3: Dead and useless code elimination: Source code

```
#include "location.h"
#include "transport.h"
#include "platformConfig.h"
Transport deliveryMethod;
Location from = location_getHere();
Location to = location_getThere();
deliveryMethod = transport_nextDayAir;
/* Suppose platformConfig.h declares
 * SUPPORTS_WORMHOLES true,
 * SUPPORTS_FTL false. */
if (SUPPORTS_WORMHOLES) {
    deliveryMethod = wormhole_open();
    /* this renders the earlier definition of
     * deliveryMethod useless */
} else if (SUPPORTS_FTL) {
    /* this branch can never execute, so it is dead */
    deliveryMethod = ftl_getShip();
}
```

Listing 9.4: Dead and useless code elimination: Transformed code

```
#include "location.h"
#include "transport.h"
#include "platformConfig.h"
Transport deliveryMethod;
Location from = location_getHere();
Location to = location_getThere();
/* useless code eliminated */
deliveryMethod = wormhole_open();
/* dead code eliminated */
```

Listings 9.5 and 9.6 provide a trivial example. As with other such examples, it is likely the programmer would perform such a painfully obvious optimization in the source code. Nevertheless, it is instructive. The computation of `x`, which occurs in all branches of the switch...case statement, is hoisted from the cases to before the switch. You can see this hoisting visually in terms of the corresponding control flow graphs in Fig. 9.

LOOP UNSWITCHING Here, switching refers to if-then-else control flow or, more generally, switch...case control flow. When this occurs within the loop, the switching occurs with each passage through the loop. If the condition determining which case of the switch is executed is loop invariant, then we can move the switch to surround the loop and then duplicate the loop within each case. Then the switch is encountered only once, when we select which variety of the loop to use. This trades code size against execution speed: there are fewer branches, so the code will run faster, but the loop body must be repeated in each case.

In listing 9.7, we find a loop with a nested if-then-else statement. If we assume that `warnlevel` remains unchanged throughout the loop, then we would, each time we go through the loop, have to test `warnlevel` in order to select the same, still appropriate branch. Listing 9.8 shows the results of applying loop unswitching to the code in listing 9.7. The branch is now selected prior to looping, which eliminates many tests and jumps.

Note how loop unswitching obscures the basic intent of the code, namely, "tell everyone on the team a certain message depending on the current `warnlevel`," and reduplicates the code governing control flow (the for-loop header). If the programmer were to apply loop unswitching manually to the source code in this case, obscuring the code's purpose would harm its long-term maintainability, and reduplicating the

Listing 9.5: Code hoisting: Source code

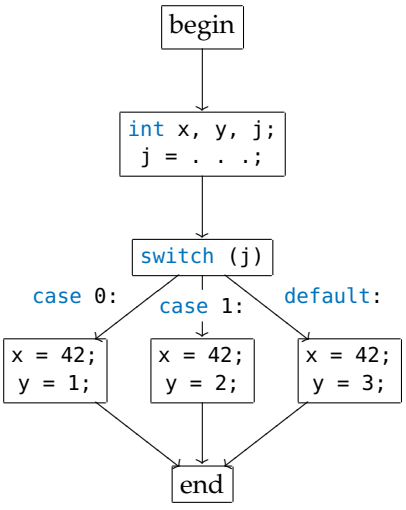
```
int x, y, j;
j = . . .;
switch (j) {
    case  0: x = 42; y = 1; break;
    case  1: x = 42; y = 2; break;
    default: x = 42; y = 3; break;
}
```

Listing 9.6: Code hoisting: Transformed code

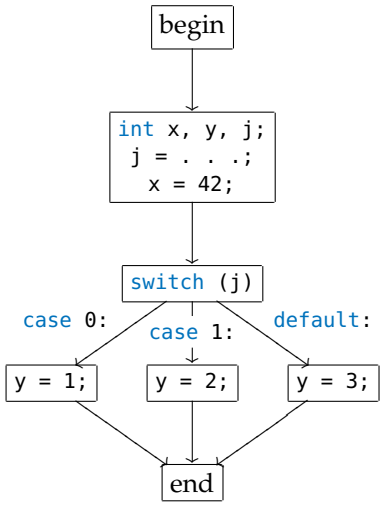
```
int x, y, j;
j = . . .;
x = 42;
switch (j) {
    case  0: y = 1; break;
    case  1: y = 2; break;
    default: y = 3; break;
}
```

Figure 9: Code hoisting: Control flow graphs

(a) Source code



(b) Transformed code



control flow code introduces the opportunity of updating it in one branch but failing to update it it in the other. Thus, it is inadvisable for the programmer to manually perform this optimization. Since the execution time saved by unswitching the loop could be significant, it is important that the compiler perform this optimization.

9.3 BIBLIOGRAPHIC NOTES

While Cooper and Torczon [31, chapters 8–10] provides an introduction to analysis and optimization, Muchnick [93] concerns itself almost exclusively with analysis, optimization, and the construction of a compiler that executes these. (This chapter is heavily indebted to both books.) It leaves untouched matters of optimizations for parallel architectures and other such optimizations needed by demanding scientific computing programs. For discussion of those issues, it recommends Bannerjee [14, 15, 16], Wolfe [136] and Zima and Chapman [138].

The LLVM (Low-Level Virtual Machine) project [71] is notable among other things for its extensive use of SSA form in its compiler architecture. It uses SSA as its primary representation of the program through most of compilation.

Static analysis is valuable for much more than compile-time optimization. It is necessary for development of advanced IDEs (interactive development environments), source code style-checking tools, and bug-finding tools, among other things. Static analysis and its many uses is an active topic of research that has led to several commercial ventures, such as Klocwork,* Coverity,† and Fortify,‡ as well as open-source research projects seeing industrial use such as Findbugs [56].

* http://www.klocwork.com/
† http://www.coverity.com/
‡ http://www.fortifysoftware.com/

Listing 9.7: Loop unswitching: Source code

```
for (Person p = team.head; p->next != NULL; p = p->next) {
    if (warnlevel >= 2000) {
        p.klaxon << "Warning!";
    }  else {
        p.spywatch << "All clear.";
    }
}
```

Listing 9.8: Loop unswitching: Transformed code

```
if (warnlevel >= 2000) {
    for (Person p = team.head; p->next != NULL; p = p->next) {
        p.klaxon << "Warning!";
    }
} else {
    for (Person p = team.head; p->next != NULL; p = p->next) {
        p.spywatch << "All clear.";
    }
}
```

10

CONCLUSION

Imperative languages developed to replace assembly languages for general programming purposes. They have dominated the programming language landscape, and their long history and wide use have made them the target of much research. They provide the conventional backdrop against which other programming language families, such as the functional languages discussed next, play their part. Unconventional ideas are often explained in terms of concepts familiar from the imperative paradigm. Alternative paradigms are judged in light of the successes and failures of the imperative. Thus, in addition to technical background, this part serves to communicate something of a common cultural background, as well.

- In Chapter 7, DEFINING, we quickly surveyed the development of the imperative programming paradigm through the growth of the imperative language family, focusing on FORTRAN, ALGOL, C, and Java as examples of goto-based, procedural, structured, and object-oriented programming. We concluded by giving several drawbacks of the imperative paradigm.
- In Chapter 8, COMPILING, we introduced common issues encountered in developing a compiler for imperative programming languages, in particular:
  - scope
  - data storage
  - array layout.

- In Chapter 9, OPTIMIZING, we described the process of optimization in terms of a variety of analyses and transformations and gave examples of several common optimizations applied to imperative programs.

Part III

FUNCTIONAL LANGUAGES

11

OVERVIEW

We have discussed imperative languages at some length. Now, we move on to functional languages.

THEORY discusses some theory basic to functional programming.

HISTORY sketches the history of the functional language family by way of several of its defining languages.

COMPILING describes in broad terms how functional languages are compiled.

CASE STUDY: THE GLASGOW HASKELL COMPILER addresses the question of how programs in modern functional languages are actually compiled to run on modern computers. We answer this by studying the Glasgow Haskell compiler, the principal compiler for the Haskell language.

12

THEORY

12.1 TYPES

When we mentioned types earlier, we took for granted that the meaning was clear enough based on shared experience of types in common programming languages. Now, we wish to discuss the functional languages. Here, we cannot rely on shared experience. Functional languages have seen some use as testbeds for practical applications of developments in type theory. Modern functional languages are founded not only on the lambda calculus, but on typed lambda calculi. But the lambda calculus is a poor way to introduce the terminology and concepts of types, and so we shall first discuss types in order to develop an intuitive understanding of some concepts that we will later introduce into the lambda calculus. Before we go on to discuss the languages themselves, then, we would do well to look a bit more carefully at the concept of *type.*

Types are introduced into programming languages in order to make the language safer and easier to compile efficiently. Types make the language safer by making it much easier for the compiler to catch nonsensical operations, such as trying to add a string to a pointer and store the result in a structure. If a language can guarantee that operations of one type will not be applied to operands of an incompatible type, then we say the language is TYPE SAFE. In a type safe language, a type inconsistency becomes a critical error. Consequently, programs, especially large, complex programs, written in type-safe languages are easier to debug than those written in non–type-safe languages. Types

can also make the language easer to compile efficiently. Different types of data can, or sometimes must, be dealt with in different ways at the machine level. Types increase the amount of knowledge about the elements of the program available to the compiler for optimization and allow the resulting code to be specialized to the types involved.

Providing type information, however, can be burdensome on the programmer. It is desirable that the programmer need not explicitly specify the types involved in the program, but rather that the types be implicit in the values used and behaviors specified. This is done through TYPE INFERENCE. Once all types have been inferred, type checking can proceed; once the program successfully passes type checking, the type information can be used in optimization and code generation.

Now that we have clarified how types are used and why they matter, it is time to be clearer about what types are. Quite simply, we can look at a TYPE as an identified set of values. These sets can overlap or be disjoint. The integer `6` falls into both the integer subranges `1..10` and `5..15`. But the set of all integers and the set of all strings is distinct; even the integer `5` and the string "5" containing the character *5* can be made readily distinguishable by introducing the lexical convention of writing strings within quotation marks. Sets can also be related by inclusion. All integers and reals are also numbers at the same time. From this, we can see that a given value can belong to a set of types. To express that a given value is of a given type, we write $\langle value \rangle : \langle Type \rangle$.

We can also construct types from other types. With $\rightarrow$, we can build the type of functions from one type to another from the types of its domain and codomain. The integer successor function `succ`, defined such that `succ x : Int = x + 1`, that is, it takes an integer as input and outputs that integer incremented by `1`, would then have type `Int → Int`. PRODUCT TYPES are tuples with the various components of the tuple capable of taking on values of various types. To represent a specific tuple, we write its elements as a comma-separated list within paren-

theses. For example, `(1, 'A')` is a specific tuple to which we could assign the type `Int` × `Char`. If we allow the components of the tuple to be referenced by name rather than ordinal position, we find that we have reinvented structures (in the terminology of C) or records (in the terminology of Pascal).

The TYPE SYSTEM of a language is often specified in this way, by enumerating the BASE TYPES and then describing how those types can be combined to form other types. In this way, the type system itself represents a small language of types with its own syntactic and lexical rules and its own semantic content embedded within the larger context of the programming language itself.

12.1.1 *Polymorphism*

Parametric and Ad Hoc

POLYMORPHISM is the property of being of many types. It stands opposed to MONOMORPHISM, the property of being of a single type. Both concepts are broadly applicable to the typed elements of programming languages – variables, functions, operators, sometimes even modules – and, by extension, to programming languages themselves: a polymorphic language exhibits polymorphism wherever possible, an almost polymorphic language has fewer polymorphic features, a nearly monomorphic language has virtually none, and a monomorphic language has purely monomorphic features.

Traditionally, polymorphism is informally divided into two kinds based on the polymorphism exhibited by a function: PARAMETRIC POLYMORPHISM, where the function behaves uniformly across all types, and AD HOC POLYMORPHISM, where the behavior of the function is specified separately for different types.

In parametric polymorphism, the parametric type of the function is best expressed by introducing a TYPE VARIABLE. For example, if we

use the notation [⟨*type*⟩] to represent a list of the given type, then a generic length function parameterized on the type of the list would have type $\forall\alpha.[\alpha] \to$ `Int`.* This expresses that *length* has type "function from list of type α to `Int` for all types α ($\forall\alpha$)." It is possible to define such a function because of the common structure of all types of lists. This is not unusual: parametric polymorphism is frequently achieved by exploiting some common property of the types involved.

Ad hoc polymorphism, on the other hand, requires separate definitions for all types involved. The addition operator `+` is frequently ad hoc polymorphic. When given two integers, it returns an integer; when given two real numbers, it returns a real number; in some languages, when given two strings, it returns their concatenation, so that `"to" + "day"` returns `"today"`. It is the nature of ad hoc polymorphism that the function will not be defined for all possible types and will not be uniformly defined even for those types to which it can be applied, as in the dual uses of `+` for both numerical addition and string concatenation.

It is, in fact, possible to regard ad hoc polymorphism as monomorphism together with the OVERLOADING of function names. From this point of view, `+` is not a single function applicable to two values both either integers, reals, or strings, but is in fact three different monomorphic functions that share a single name. By examining the types of the supplied arguments, the overloading can be resolved, so that, for example, `1 + 2` can be turned into a call to `addInt` and `1.0 + 2.0` to a call to `addReal`, while `"to" + "day"` can be turned into a call of `concatString`.

Matters become more confused when we introduce COERCION, the implicit forcing of a value from one type into another. This is common with numeric arguments: if a function of type `Real` $\to$ `Real` is applied to an `Int`, the `Int` value might be coerced to a value of type `Real`, so that `floor 4` becomes `floor (toReal 4)`, as if the programmer had writ-

* In fact, we can view the square bracket notation [T] for a list of type T as a syntactic convenience for expressing the application of the parametric type constructor List α to the type T.

Table 1: Ad hoc polymorphism as overloading

OVERLOADED CALL	RESOLVED TO
`1 + 2`	`addInt 1 2`
`1.0 + 2.0`	`addReal 1 2`
`"to" + "day"`	`concatString "to" "day"`

ten `floor 4.0`. With coercion and overloading in force, what happens when `1 + 2.0` is encountered? Would it be as if the programmer had instead written `addIntReal 1 2.0`, an addition function expecting an integer and a real as its two inputs, or would the integer be coerced so that `addReal` could be used? Does `addInt` even exist, or are integers always coerced to be of type `Real` before invoking `addReal`?*

Subtype

SUBTYPE POLYMORPHISM is a restricted kind of parametric polymorphism in which the universal quantification of the type variable is restricted to the universe of those types that are subtypes of some other type. For example, a parametrically polymorphic function for sorting lists relies on the fact that the type of the lists is ordered in some way. Thus, what is desired is to express that `sort` is a function from lists of some ordered type to lists of the same ordered type, which is to say that it is a function from lists of all types where the type is ordered to lists of the same type. If we introduce `Order` as the type of all ordered types and write $A \subseteq B$ to express that A is a subtype of B, then we can assign `sort` the type $\forall\alpha \subseteq \texttt{Order}.[\alpha] \to [\alpha]$. This combination of universal quantification and subtyping is referred to as BOUNDED UNIVERSAL QUANTIFICATION.

* We owe this example to Cardelli and Wegner [29, p. 476].

12.2 LAMBDA CALCULUS

In Chapter 2, BEGINNINGS, we briefly sketched the lambda calculus. Now, we shall take the time to do it justice. The fundamentals of the lambda calculus are simple, unassuming, and somewhat unintuitive. By extending the lambda calculus, we can make it more "natural" for reasoning about and even doing programming, but this comes at the cost of reducing its power. It is, however, partly this reduction in power that makes these extensions so appealing.

The rest of this chapter assumes you are familiar with the first-order predicate calculus, in particular the treatment of quantifiers and free and bound variables. If you are not, some of the finer details of the presentation will elude you, but you should still come away with an intuitive understanding of many of the concepts of this chapter.

12.2.1 *Pure Untyped Lambda Calculus*

Pure untyped lambda calculus is the original form of the lambda calculus. When someone speaks of "the lambda calculus" without qualification, this is what is meant. Its purity is due to its conceptual simplicity and elegance. We call it untyped because it makes no distinction between types: everything is of the same type.

The building blocks of the lambda calculus are LAMBDA TERMS. We will call the set of all lambda terms Λ. Λ is readily defined using a context-free grammar:

$$\Lambda \to V \mid P \mid B$$

$$V \to v \mid V'$$

$$P \to (\Lambda\ \Lambda)$$

$$B \to (\lambda V\ \Lambda)$$

Let us go through this, line by line, to ensure we understand it. The first production

$$\Lambda \to V \mid P \mid B$$

says that we build terms in Λ using three different rules, V, P, and B. If we peek ahead at the productions for P and B, we can see that these depend on Λ. Thus, the first rule, V, is critical. It generates the most basic lambda terms: VARIABLES. The essential properties of variables are:

A. They have no substructure: they are ATOMIC.

B. Each variable is distinguishable from each other.

The first property is plain from the definition. The second property becomes clear once we write out the terms that V produces. These are none other than the infinite set $\{v, v', v'', \ldots\}$, namely, an infinite set of variables, each built from the same basic symbol (v) by the addition of more and more primes ($'$).

Now that we have some terms in Λ, we are free to describe how to form new lambda terms from other lambda terms. That is what the next two rules do. Each describes one way to generate, given two lambda terms, one more lambda term.

The first way is to juxtapose two terms and enclose the result in parentheses. This is described in the context free grammar as

$$P \to (\Lambda\ \Lambda)$$

and is known as APPLICATION. If M and N are lambda terms, then an application of M to N looks like $(M\ N)$.

The second way is to follow a λ with a variable and another term and enclose the whole in parentheses. The grammatical production corresponding to this is

$$B \to (\lambda V\ \Lambda)$$

and is known as ABSTRACTION. If M is a term, then $(\lambda v\ M)$ is an example of an abstraction. We shall call the variable v in such a term the VARIABLE OF ABSTRACTION and the term M the abstractionbody. We will sometimes adopt the point of view of constructing this term from M and v. In that case, we say that we are "abstracting the variable v over the term M."

As it stands, the readability of this notation degrades rapidly as lambda terms become more complex. The number of parentheses grows rapidly, and it becomes difficult to tell which variables are identical and which different as the number of primes in use grows. Thus, we introduce some conventions:

- Lowercase letters (x, y, z, and so forth) represent atomic variables.
- Capital letters (M, N, and the like) represent arbitrary lambda terms.
- Application is considered to associate to the left. This allows us to omit the parentheses introduced by abstraction except when we must override this rule for a specific term. Thus, MNOP should be read as $(((M\ N)\ O)\ P)$.
- Abstraction is considered to associate to the right. This allows us to omit many of the parentheses introduced by abstraction. Thus, λx λy λz M should be read as $(\lambda x\ (\lambda y\ (\lambda z\ M)))$.
- The variable of abstraction will be separated from the abstraction body by a dot, so that we write λx.M instead of λx M.

Table 2: Notational conventions

TYPE OF TERM	BECOMES	ORIGINALLY
Variables	$x, y, z, \ldots$	$v, v', v'', \ldots$
Terms	$M, N, \ldots$	$M, N, \ldots \in \Lambda$
Application	$MNOP$	$(((M\ N)\ O)\ P)$
Abstraction	$\lambda x.\lambda y.\lambda z.M$	$(\lambda v\ (\lambda v'\ (\lambda v''\ M)))$ where $M \in \Lambda$

These conventions are summarized in Table 2 on p. 143

We shall write $M \equiv N$ to state that M and N are syntactically equivalent. Intuitively, when we say that they are syntactically equivalent, we mean that they were "built" the same way and, though they might use different variable names, they can be considered to be the "same term in a different guise." For example, thanks to the infinity of basic variables, we can readily construct an infinity of syntactically equivalent lambda terms by picking a variable, picking another variable and applying the first to the second, then abstracting over the second:

1. Pick a variable.

$$x \equiv y \equiv z \equiv \ldots$$

2. Pick another.

$$y \equiv z \equiv w \equiv \ldots$$

3. Apply the first to the second.

$$xy \equiv yz \equiv zw \equiv \ldots$$

4. Abstract over the second.

$$\lambda y.xy \equiv \lambda z.yz \equiv \lambda w.zw \equiv \ldots$$

At each step, all the terms listed are syntactically equivalent. We claim that this is justified because all these terms behave the same way, as we shall see quite soon. This makes the notion of syntactic equivalence a powerful and useful one. But this concept is more sophisticated than it might at first sound, and there are some pitfalls to watch out for in defining and using it, which we shall come to shortly.

There is one more ingredient of the pure untyped lambda calculus. So far, we have established a static universe of lambda terms. We can conceive of larger and larger terms, but we cannot simplify them or do anything beyond list them. The missing spark that puts these terms in motion and enables computation is called β-REDUCTION. β-reduction resembles the application of grammatical productions in context-free grammars, and the notation is similar, though both the behavior and notation are slightly and significantly different.

We begin by defining SINGLE-STEP β-REDUCTION, written $\rightarrow_\beta$. This relates lambda terms to lambda terms; specifically, it says that we can replace the application of an abstraction term to another term with the term formed from the abstraction body by substituting the other term for the variable of abstraction wherever it occurs in the body:

$$(\lambda x.M)N \rightarrow_\beta M[x := N]$$

We can also describe how β-reduction behaves with other sorts of terms:

- β-reduction is allowed in either half of an application.

$$\frac{M \rightarrow_\beta M'}{MN \rightarrow_\beta M'N} \qquad \frac{M \rightarrow_\beta M'}{NM \rightarrow_\beta NM'}$$

- β-reduction is allowed within an abstraction.

$$\frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'}$$

The notation $\frac{\langle \textit{upper statements} \rangle}{\langle \textit{lower statements} \rangle}$ means that, if we know the upper statements to be true, then the lower statements must also be true. The horizontal line between the two levels can be read "implies."

We can similarly define MANY-STEP β-REDUCTION, $\twoheadrightarrow_\beta$: $M \twoheadrightarrow_\beta N$, where N is not necessarily distinct from M, if there is some chain of zero or more β-reductions beginning with M and terminating with N, that is, $M \to_\beta M' \to_\beta \cdots \to_\beta N$.* Unlike single-step β-reduction, which only relates different terms, many-step β-reduction also relates a term to itself: $M \twoheadrightarrow_\beta M$ always.

β-Reduction and the Perils of Names

What is going on here? We can think of it this way: abstraction binds its variable. Once we have abstracted a term over a given variable, we cannot do so again. The variable is no longer free. When we apply an abstracted term to another term, β-reduction simultaneously performs the binding of the variable to the other term and substitutes that term for the bound variable throughout the abstraction body. Now that its purpose has been fulfilled and the abstraction made concrete, the abstraction disappears. An example will make this clearer. Take $\lambda y.xy$. Applying it to some lambda term, M, and β-reducing it gives:

$$(\lambda y.xy)M \to_\beta xy\,[y := M] = xM$$

But here there be dragons. This is where the convenient identification of syntactically equivalent terms returns with a vengeance. Suppose we take the doubly-abstracted term $\lambda x.\lambda y.xy$ and apply it to the

* Note that we are using a two-headed arrow $\twoheadrightarrow$ here instead of the starred arrow $\stackrel{*}{\Rightarrow}$ that we used for the similar concept of derivation in multiple steps with context-free grammars. As with derivation, where a subscript $\mathtt{lm}$ or $\mathtt{rm}$ indicated whether leftmost or rightmost derivation was used, a subscript β here indicates that β-reduction was employed.

seemingly innocent term wyz. Let us also apply it to the syntactically equivalent term tuv. What happens?

$$(\lambda x.\lambda y.xy)(wyz) \to_\beta \lambda y.xy\,[x := wyz] = \lambda y.wyzy$$
$$(\lambda x.\lambda y.xy)(tuv) \to_\beta \lambda y.xy\,[x := tuv] = \lambda y.tuvy$$

But, should we apply this to yet another term, say s, something unexpected occurs:

$$(\lambda y.wyzy)s \to_\beta wyzy\,[y := s] = wszs$$
$$(\lambda y.tuvy)s \to_\beta tuvy\,[y := s] = tuvs$$

The results are obviously no longer syntactically equivalent!

This problem should be familiar to anyone acquainted with the first-order predicate calculus. The abstraction symbol λ in the lambda calculus behaves exactly the same as do the quantifiers $\exists$ and $\forall$ in that both bind their associated variable in the body of the term that we say is abstracted over in the lambda calculus and quantified in the first-order predicate calculus. The problem is one of VARIABLE CAPTURE: we substituted wyz, which has as its free variables $\{w, y, z\}$, into a term in which y was bound, thus incidentally binding the y of wyz. When we substituted the syntactically equivalent tuv, however, all three variables remained free in the result. Thus, naïve β-reduction does not necessarily preserve syntactic equivalence, contrary to our intent in establishing that equivalence. The only way to ensure that syntactically equivalent terms produce equivalent results is to be very careful to avoid variable capture. This is more difficult to fully specify than it sounds, and we refer you to any text on the first-order predicate calculus for the details. For the purposes of this example, it suffices that we always rename bound variables to some variable that does not occur free in the term about to be substituted. Here, that would mean renaming the bound variable y to some variable other than those of wyz, say

v. If we perform this renaming and then repeat our experiment, we get the appropriate results:

$$(\lambda x.\lambda y.xy)(wyz) \equiv (\lambda x.\lambda v.xv)(wyz) \quad \text{(renaming } y \text{ to } v\text{)}$$
$$(\lambda x.\lambda v.xv)(wyz) \rightarrow_\beta \lambda v.xv\,[x := wyz] = \lambda v.wyzv \quad \text{(first application)}$$
$$(\lambda v.wyzv)s \rightarrow_\beta wyzv\,[v := s] = wyzs \quad \text{(second application)}$$
$$wyzs \equiv tuvs \quad \text{(the results)}$$

As you can see, the result is now syntactically equivalent to that reached when we use tuv instead of wyz.

α-Reduction

There are two ways around the variable capture problem. One is to simply assume that all this renaming takes place automatically and get on with the theory. This is very convenient if all you are interested in is developing the theory associated with the lambda calculus and is a favorite choice of theoreticians. The other option is to formalize this renaming process by introducing another type of reduction, α-REDUCTION, and modify the rules surrounding β-reduction to explicitly forbid its use where variable capture would occur, thus forcing the invocation of α-reduction before β-reduction can continue. This is somewhat messier, but it better reflects what must occur in a practical implementation of the lambda calculus. While waving our hands and saying that we identify syntactically equivalent terms and all renaming occurs as necessary for things to come out as desired in the end works fine on paper and fine with humans, we must be a bit more explicit if we are to successfully implement β-reduction in a compiler.

Doing so gets even messier. We must continually come up with unique names, make repeated textual substitutions, and keep checking to ensure we're not about to capture a variable. But this mess was foisted upon us by our choice of variables. We have already made clear

by our treatment of these variables that their names serve as nothing more than placeholders. They are just ways for the abstraction to point down into the term and indicate at which points we should make the substitutions called for by β-reduction. What if, instead, we reversed the relationship between the abstraction and its variable?

De Bruijn Indices

The central insight of DE BRUIJN INDICES is to eliminate the use of corresponding variable names in the abstraction in favor of numbers "pointing" to the appropriate lambda. Free variables are considered to point to lambdas that have yet to be introduced. Thus, instead of writing $\lambda x.xy$, we would write $\lambda.12$, since the x in the former notation is bound by the first enclosing λ, and 2 is the first number greater than the number of enclosing lambdas. The more complex term $\lambda x.(\lambda y.xy)(\lambda z.\lambda y.xyz)$ would become $\lambda.(\lambda.21)(\lambda.\lambda.321)$.

This demonstrates that we are not simply renaming variables-as-letters to variables-as-numbers. Instead, we are using two closely-related notions to assign the numbers: the level and depth of a given variable occurrence. To determine a variable occurrence's LEVEL, we think of starting from the outside of the expression, at level 1, and descending through it to the occurrence. Each time we enter the scope of another lambda along the way, we drop down another level in the term, from level 1 to level 2 and so on. We ultimately replace the bound variable with its DEPTH. The depth is how many levels above the current level of nesting the corresponding binding λ is located. If the occurrence and its binder are at the same level, we consider the occurrence's depth to be 1. Each level we must ascend from the variable after that to reach the binder adds one to the depth. We conceive of the free variables as sitting one above the other over the outermost lambda of the term, so that we must ascend past the top level in counting out their depth.

We can thus readily identify free variables, since their depth is greater than their level.

Take the xs in the last example, which became respectively 2 and 3. Let us visually display the level of an term by dropping down a line on the page; then the conversion between variable names and de Bruijn indices becomes easier to see:

Table 3: Converting to de Bruijn indices

LEVEL	VARIABLE NAMES	DE BRUIJN INDICES
1	λx.	λ.
2	(λy.xy) (λz.	(λ.21) (λ.
3	λy.xyz)	λ.321)

This conversion can be performed algorithmically by keeping track of which variable names were bound at which level of nesting. We can also readily convert from de Bruijn indices back to variable names. This allows for the entry and display of lambda terms using variable names while reduction proceeds in terms of de Bruijn indices. Since de Bruijn indices give a unique representation for each syntactically equivalent lambda term, they sidestep the problems with variable binding and the like that we encountered earlier.*

Currying

But variable names are more convenient for humans both to write and read,† and so we return to using the more conventional notation. We can even do a bit more to increase the readability of our lambda terms.

* You might have noticed that, since the indices do depend on the level of nesting, they must be adjusted when substitution occurs under abstraction. But this is only a slight problem compared to the mess brought on by names, and it can be readily and efficiently dealt with.

† De Bruijn suffered no confusion on this count: he intended his namefree notation to be "easy to handle in a metalingual discussion" and "easy for the computer and for the computer programmer" and expressly not for humans to read and write [38, pp. 381–82].

We have already eliminated excess parentheses: let us now eliminate excess lambdas.

Consider the lambda term $\lambda x.(\lambda y.(xy))$. According to our conventions, abstraction associates to the right, and so this can be unambiguously written as $\lambda x.\lambda y.xy$. But, look again: the distinction between the variable bound by abstraction and the term within which it is bound is clear, since each abstracted term has the form $\lambda\langle \textit{variable} \rangle.\langle \textit{term} \rangle$. When we come across nested abstraction, as above, it is clear that those variables closer to the nested term but left of a dot are the result of abstraction at a deeper level of nesting. So let us write, instead of $\lambda x.\lambda y.xy$, rather $\lambda xy.xy$. This applies wherever we would not require that parentheses intervene between nested lambdas due to convention. Thus, we could rewrite $\lambda x.(\lambda y.xy)(\lambda z.\lambda y.xyz)$ as $\lambda x.(\lambda y.xy)(\lambda zy.xyz)$.

Notice now how a lambda term such as $\lambda xy.xy$ resembles a function of multiple arguments. If we apply it to two terms in succession, then it eventually β-reduces precisely as if we had supplied two arguments to a binary function: $(\lambda xy.xy)MN \twoheadrightarrow_\beta MN$. But, what happens if we apply it to a single term? Well, $(\lambda xy.xy)M \to_\beta \lambda y.My$, that is, we get back a term abstracted over y. It is as if, on supplying only one argument to an n-ary function, we got back a function of arity $n-1$. When we apply this term to, say, N, we arrive at MN yet again, precisely as if we had immediately supplied both "arguments" to the original term.

This is not mere coincidence. We can, in fact, represent *all* n-ary functions as compositions of n unary functions. Each such function simply returns a function expecting the next argument of the original, n-ary function. This form of an n-ary function is known as its CURRIED form, and the process of transforming an uncurried function into a curried function is called CURRYING.*

* The name is a reference to the logician Haskell B. Curry, though the idea appears to be due to Moses Schönfinkel.

From Reduction to Conversion

Let us now step back a ways to where we had just defined β-reduction. β-reduction is a one-way process: it can never make a term more complicated than it was before, though we cannot go all the way to claiming that β-reduction always results in a less complicated term. For example, $(\lambda x.xx)(\lambda x.xx)$ always and only β-reduces to itself and thus becomes neither more nor less complicated: $(\lambda x.xx)(\lambda x.xx) \to_\beta \lambda x.xx\,[x := xx] = (\lambda x.xx)(\lambda x.xx) \to_\beta \cdots$.

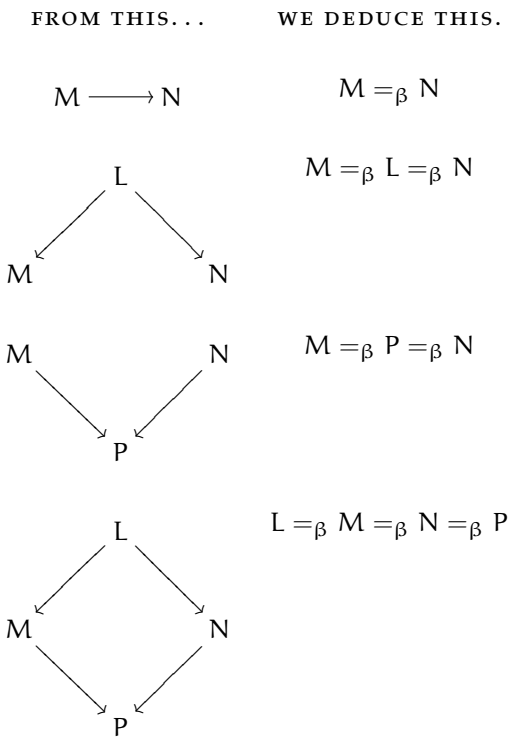
But β-reduction does relate terms: one term is related to another if it can eventually β-reduce to that term. In some sense, all terms that are the result of β-reduction of some other term are related in that very way. We thus name this relation by saying that such terms are β-CONVERTIBLE, so that if $M \twoheadrightarrow_\beta N$, or $N \twoheadrightarrow_\beta M$, or $L \twoheadrightarrow_\beta M$ and $L \twoheadrightarrow_\beta N$, or $M \twoheadrightarrow_\beta P$ and $N \twoheadrightarrow_\beta P$, then M and N (and additionally L and P, if such is the case) are β-convertible.* This is illustrated in Fig. 10, p. 152. We notate this relation with a subscripted equals sign: $M =_\beta N$. Note that, as a consequence of the definition of β-convertibility, $M =_\beta M$ for all lambda terms M.

(Now that we have introduced β-conversion, it is time to emend our earlier comments on α-reduction. What we have called α-reduction is more commonly, and more properly, called α-conversion, since the relation between names is inherently bidirectional: x converts to y as readily as y converts to x.)

THE CHURCH–ROSSER THEOREM An important property of the lambda calculus is related to this. It is known as the CHURCH–ROSSER THEOREM, and it says that the lambda calculus together with $\to_\beta$ has two properties.

* To be more exact, β-conversion is the equivalence relation generated by many-step β-reduction.

Figure 10: β-conversion
Letters represent lambda terms. A directed arrow $M \to N$ means that M β-reduces to N in some number of steps, that is, $M \twoheadrightarrow_\beta N$.



- Firstly, it says that if any lambda term can be reduced in one or more steps to two different lambda terms, then it is possible to reduce each of those lambda terms in some number of steps to the same term. More symbolically, this can be put as follows: if $M \twoheadrightarrow_\beta M_1$ and $M \twoheadrightarrow_\beta M_2$, then there exists an M_3 such that $M_1 \twoheadrightarrow_\beta M_3$ and $M_2 \twoheadrightarrow_\beta M_3$. What this means graphically is that, given the figure from the second row of the table in Fig. 10 (p. 152), we can infer the existence of the term P in the the last row of that table.

- Secondly, it states that if two terms M and N are β-convertible into each other, then there is some other common term P to which the first two can both be reduced. This is to say that, if $M =_\beta N$, then we can find some P such that $M \twoheadrightarrow_\beta P$ and $N \twoheadrightarrow_\beta P$ as well.

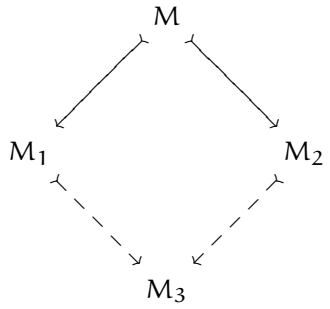
The first of these properties is known as the CHURCH–ROSSER PROPERTY. The second property does not have a name of its own. Since the Church–Rosser theorem is so important, we will sketch a proof that it holds for the pure untyped lambda calculus. We defer a discussion of the theorem's implications for the lambda calculus till after we have introduced the idea of a normal form.

There are numerous proofs of the Church–Rosser theorem. The one sketched here is due to Tait and Martin–Löf by way of Barendregt.* After proving a set of lemmas, the Church–Rosser theorem becomes a simple corollary.

We begin by defining the DIAMOND PROPERTY. A binary relation $\rightarrowtail$ on lambda terms satisfies the diamond property if for all lambda terms M, M_1, and M_2, such that $M \rightarrowtail M_1$ and $M \rightarrowtail M_2$, there also exists a term M_3 such that both $M_1 \rightarrowtail M_3$ and $M_2 \rightarrowtail M_3$. In a reduction diagram, the term M_3 appears to complete the diamond

* See Barendregt [17, §3.2].

Figure 11: The diamond property
The binary relation $\rightarrowtail$ satisfies the diamond property if, for every three terms M, M_1, and M_2 such that $M \rightarrowtail M_1$ and $M \rightarrowtail M_2$, there exists some fourth term M_3 such that $M_1 \rightarrowtail M_3$ and $M_2 \rightarrowtail M_3$. In this diagram, solid arrows indicate assumed relations, while dashed arrows indicate inferred relations.



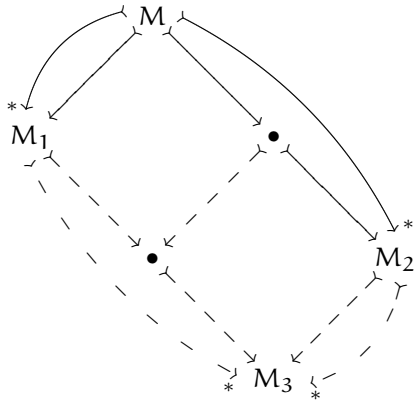
begun by $M \rightarrowtail M_1$ and $M \rightarrowtail M_2$; this is illustrated by Fig. 11 on page 154.

From this definition, it is clear that, if we can prove that $\twoheadrightarrow_\beta$ satisfies the diamond property, then we have proved that $\twoheadrightarrow_\beta$ has the Church–Rosser property. The first lemma shows that, if a binary relation $\rightarrowtail$ on a set (such as β-reduction $\rightarrow_\beta$ is on the set of lambda terms) satisfies the diamond property, then so too does its transitive closure $\rightarrowtail^*$. This is suggested by the diagram of Fig. 12 on page 155.

This lemma is not quite what we need. Single-step β-reduction $\rightarrow_\beta$ is not reflexive, but many-step β-reduction is. Many-step β-reduction $\twoheadrightarrow_\beta$ is in fact the *reflexive* transitive closure of $\rightarrow_\beta$. The solution to this mismatch is to define a reflexive binary relation on lambda terms similar to $\rightarrow_\beta$ such that many-step β-reduction is this new relation's transitive closure. Once we prove that this new relation has the diamond property, we have proven that its transitive closure $\twoheadrightarrow_\beta$ has the Church–Rosser property.

Once we have that many-step β-reduction $\twoheadrightarrow_\beta$ satisfies the diamond property, it becomes simple to prove that β-convertible terms can be β-reduced to a common term (the second property specified): The result follows readily from the definition of $=_\beta$.

Figure 12: Transitive diamonds
No matter how many $\rightarrowtail$ steps $\rightarrowtail^*$ might put M_1 and M_2 away from M, repeated application of the diamond property of $\rightarrowtail$ lets us show that its transitive closure $\rightarrowtail^*$ also satisfies the diamond property: there is always some M_3 such that both $M_1 \rightarrowtail^* M_3$ and $M_2 \rightarrowtail^* M_3$.



Normal Forms

Now that we have defined lambda terms and β-reduction, we can give a normal form for lambda terms. This is important if we are to explain the impact of the Church–Rosser theorem. A lambda term is said to be in NORMAL FORM if it cannot be further β-reduced, that is to say, the lambda term M is in normal form if there is no N such that $M \rightarrow_\beta N$.

Not all terms have normal forms. The lambda term $(\lambda x.xx)(\lambda x.xx)$, sometimes called Ω, has no normal form, since it reduces always and only to itself. Some terms do have normal forms, but it is possible to β-reduce the term an arbitrary number of times without reaching this form. What this means practically that it is important to select the right REDUCIBLE EXPRESSION (REDEX) to β-reduce, otherwise one might continue β-reducing without ever terminating in the normal form. It is easy to produce examples of such terms by throwing Ω into the mix: $(\lambda xy.y)\Omega z$ has as its normal form z, but of course one could repeatedly select Ω for reduction and thereby never realize this. Terms that *always* reduce to normal form within a finite number of β-

reductions, regardless of the reduction strategy employed, are called STRONGLY NORMALIZING.

Now that the concepts of normal forms and strongly normalizing terms is clear, we can explicate the Church–Rosser theorem. Altogether, it means that, if a term has a normal form, then regardless of the reduction steps we use to reach that form, it will always be possible from anywhere along the way to reach the normal form: there's no way we can misstep and be kept from ever reaching the normal form short of intentionally, repeatedly making the wrong choice of expression to reduce. If the term is strongly normalizing, we can go one better and state that the reduction steps we use to reach its normal form are completely irrelevant, as all chains of reductions will eventually terminate in that normal form. This also shows that the normal form is unique, for if N_1 and N_2 were two distinct normal forms of a given term, then they would have to share a common term N_3 to which they could both be β-reduced.

Recursion and Y

The pure, untyped lambda calculus is Turing complete. An important part of achieving this degree of expressive power is the ability to make recursive definitions using the lambda calculus. The way to do so is surprisingly succinct. One common means is what is known as the PARADOXICAL COMBINATOR, universally designated by Y. Here is its definition:

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

It is also known as the FIXED-POINT COMBINATOR, since for every F, we have that $F(YF) =_\beta YF$.* Let us see how exactly this works:

$$\begin{aligned}
YF &\equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F \\
&\to_\beta (\lambda x.F(xx))(\lambda x.F(xx)) \\
&\to_\beta F((\lambda x.F(xx))(\lambda x.F(xx))) \\
&=_\beta F((\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F) \\
&\equiv F(YF)
\end{aligned}$$

At each step, we have used color to indicate the terms that are involved in producing the next step. We have indicated at each step whether syntactic equivalence, β-reduction, or full-fledged β-conversion was employed. As you can see, we begin by using simple β-reduction. The breakthrough that permits us to arrive at the desired form is replacing the two instances of F in the inner term $((\cdots F\cdots)(\cdots F\cdots))$ by the application of $\lambda f.(\cdots f\cdots)(\cdots f\cdots)$ to F.

If you look at the steps leading up to that abstraction, you can see how Y leads to recursion. Let us carry this process out a bit further:

$$\begin{aligned}
YF &\equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))F \\
&\to_\beta (\lambda x.F(xx))(\lambda x.F(xx)) \\
&\to_\beta F(\lambda x.F(xx))(\lambda x.F(xx)) \\
&\to_\beta F(F(((\lambda x.F(xx))(\lambda x.F(xx))))) \\
&\to_\beta F(F(F(((\lambda x.F(xx))(\lambda x.F(xx)))))) \\
&\ \ \vdots
\end{aligned}$$

As you can see quite plainly, YF leads to repeated self-application of F. This iteration of F is how it produces a fixed point.

* Y is not unique in producing fixed points. Turing's fixed point operator $\Theta = (\lambda ab.b(aab))(\lambda ab.b(aab))$ will do just as well.

A Brief Word on Reduction Strategies

Before we move on to extend the lambda calculus with some concepts a bit more elaborate and convenient than the low-level but elegant pure, untyped lambda calculus, it behooves us to put in a brief word about reduction strategies.

In the context of the lambda calculus, we call them reduction strategies. In the context of programming languages in general, we use the phrase ORDER OF EVALUATION. They both come down to the same thing: where do we want to focus our efforts? And, perhaps more importantly, where should we focus our efforts?

The choices we make in terms of lambda calculus reduction strategy have equivalents in terms of order of evaluation of functions and their arguments. The fundamental question is this: should we evaluate the arguments before passing them on to the function, or should we call the function and just point it at its arguments so it has access to them and their values as needed?

If we first deal with the arguments and only then with the function as a whole, then we are employing a CALL-BY-VALUE evaluation strategy, so called because the function call takes place with the values of the arguments provided to the function. The arguments are evaluated, and the resulting value is bound to the formal parameters of the function.

If we instead begin by evaluating the body of the function itself and only evaluate the arguments as necessary, we are pursuing a CALL-BY-NAME evaluation strategy. Call-by-name gets its name from the way that the formal parameters are bound only to the names of the actual arguments. It is only when the value of one of the arguments is required to proceed with evaluating the function's body that the argument is evaluated.

The lambda calculus equivalent of call-by-value is known as the APPLICATIVE ORDER reduction strategy. We can describe the applicative

order strategy quite simply: reduction is first performed in the term to which the abstraction is being applied. Only when we have exhausted this possibility do we perform the application.

If we strengthen this preference into a hard and fast rule that no β-reduction is to be performed under an abstraction, then we can define an alternative normal form, known as WEAK NORMAL FORM:

- Variables are defined to be in weak normal form.
- An application MN is in weak normal form if and only if both M and N are in weak normal form.
- An abstraction $\lambda x.M$ is in weak normal form.

It is in the treatment of abstractions that weak normal form differs from normal form: weak normal form considers all abstractions $\lambda x.M$ to already be in weak normal form, while normal form requires that the term M being abstracted over also be in normal form.

The lambda calculus equivalent of call-by-name, on the other hand, is known as the NORMAL ORDER reduction strategy and corresponds to always select the leftmost–outermost reducible expression for reduction. Normal order is so called because, if the term has a normal form, we can *always* reduce it to normal form by employing the normal order reduction strategy. The same cannot be said for applicative order, which can fail to reduce a term to normal form even when one exists. Again, the Ω term makes it easy to give an example. Something as simple as $(\lambda xy.y)\Omega z$ suffices. Under normal order evaluation, we first β-reduce $(\lambda xy.y)\Omega \rightarrow_\beta (\lambda y.y)$ and then apply the result to z, giving the normal form z. With applicative order evaluation, however, we begin by β-reducing the first argument, Ω – and that is as close as we shall ever get to the normal form z, since Ω has no normal form.

Strictness

If a computation never terminates, as with the attempt to β-reduce Ω to normal form, then we say that it DIVERGES. An important concept for functions is that of STRICTNESS. We say a function is strict in a given parameter if the evaluation of the function itself diverges whenever the evaluation of the parameter diverges. The function $\lambda xy.y$ given above is strict only in its second argument, y, since we can evaluate the function even when a divergent term is substituted for x, as recently demonstrated. A function that is strict in all arguments is called a STRICT FUNCTION. The function $\lambda x.x$ can readily be seen as strict, since this function is simply the identity function.

η-*Conversion*

Strictness is important to understanding the appropriateness of a different sort of convertibility relationship between terms. This type of conversion expresses the equivalence of a function expecting some number of arguments and a function that "wraps" that function and provides it with those arguments. We call two such expressions M and N η-CONVERTIBLE, written $M =_\eta N$, and we define η-conversion as

$$\lambda x.Fx =_\eta F, \quad x \notin \mathcal{FV}(F) \ .$$

The last part, $x \notin \mathcal{FV}(F)$, should be read as "where x is not among the free variables of F" and serves to exclude abstraction over variables that would capture a free variable in F from being defined as η-convertible with F.*

η-conversion is useful because it identifies a host of identically-behaving terms. We can produce an infinite number, even when we identify α-convertible expressions, starting with something as simple as

* If you are familiar with the concept of *extensional equality,* you should have no trouble remembering the definition of η-conversion if you think of it as "η for *extensional.*"

the identity $\lambda x.x$. All we need do is repeatedly abstract this term with respect to x. This gives rise to the sequence I of terms

$$\begin{aligned} I_1 &= \lambda x.x \\ I_2 &= \lambda x.(\lambda x.x)x \\ I_3 &= \lambda x.(\lambda x.(\lambda x.x)x)x \\ I_4 &= \lambda x.(\lambda x.(\lambda x.(\lambda x.x)x)x)x \\ &\vdots \end{aligned}$$

Since x is always bound within the body of the abstraction, this causes no problems with variable capture. But all the terms in this sequence behave the same way; in fact, the application of any one of them to a term M is β-convertible with the application of any other to the same term M and, ultimately, with M itself: $I_n M \to_\beta I_{n-1} M \to_\beta \cdots \to_\beta I_1 M \to_\beta M$.

But η-conversion might nevertheless be an inappropriate notion of equality in some cases. If we are reducing only to weak normal form, for example, then reduction of $\lambda x.\Omega$ terminates immediately, while reduction of Ω will never terminate. The introduction of types can also pose problems for the notion of η-conversion. But where we can employ η-conversion, we can at times drastically reduce the number of substitutions required due to β-reduction by simplifying the terms involved using η-conversion first. In the end, whether we decide it is appropriate or not to add η-conversion to our lambda calculus, the operational equivalence between terms that it highlights is well worth keeping in mind.

12.2.2 *Extending the Lambda Calculus*

While we have taken some time to explain the lambda calculus, the system itself is quite lean. It is also powerful. Indeed, we can use it to

describe any computation we could describe with a Turing machine. Such a description is also similarly lengthy and inconvenient. Thus, we will extend the lambda calculus in two major ways:

- We will introduce a new set of normal-form lambda terms as constants, so that the numbers, booleans, and so forth become primitive concepts of the calculus rather than needing to be encoded in terms of lambda terms.
- We will introduce typing into the system.

These extensions also serve to raise the level of abstraction of the lambda calculus closer to that of a functional programming language.*

Untyped Lambda Calculus with Constants

We first extend the pure untyped lambda calculus by adding CONSTANTS. As when we introduced variables, we can formally define the constant terms as normal form lambda terms built from a base symbol c distinct from the v used for variables and many primes:

$$\Lambda \rightarrow V \mid C \mid (\Lambda\ \Lambda) \mid (\lambda\ V\ \Lambda)$$
$$V \rightarrow v \mid V'$$
$$C \rightarrow c \mid C'$$

and then our use of, say, c to represent the integer 0 becomes purely a matter of convention. But just as we established variable naming conventions to make our notation more readable, so too can we establish conventions for writing constants that allow 0, 1, *True*, and so forth to appear directly in our notation.

But we are not restricted to adding only static constants such as the numbers and booleans. We can also take some to be operators, such as a test for equality $=$, addition $+$, subtraction $-$, or even *If*. These

* This section follows the development of the lambda calculus in Hudak [57] closely, including use of some of the same examples.

clearly operate on other terms, but how remains to be defined. That is the purpose of δ-RULES, which are basically ad hoc reduction rules for dealing with constant terms much as β-reduction describes more generally how to reduce expressions of any lambda terms.

For example, we can give a set of δ-rules that make $+$ operate on the particular constants that we have identified with the integers in a way consistent with our intuitive understanding of addition:

$$(+\ 0)\ 0 \to_\delta 0$$
$$(+\ 0)\ 1 \to_\delta 1$$
$$\vdots$$
$$(+\ 1)\ 0 \to_\delta 1$$
$$\vdots$$

We can deal with *If* likewise:

$$\mathit{If\ True}\ e_1 e_2 \to_\delta e_1$$
$$\mathit{If\ False}\ e_1 e_2 \to_\delta e_2$$

In extending the system with δ-rules, we must be very careful to preserve properties we consider essential to the system, such as the Church–Rosser properties. For example, if we were to add δ-rules such that *Or* becomes a left-to-right, short-circuiting operator, we would be fine:

$$\mathit{Or\ True}\ e \to_\delta \mathit{True}$$
$$\mathit{Or\ False}\ e \to_\delta e$$

But if we wanted to add δ-rules truer to our intuitive understanding of *Or*, namely that it yields true if either of its operands is true, regardless of the value of the other operand, we might add rules such as:

$$\textit{Or True } e \rightarrow_\delta \textit{True}$$

$$\textit{Or } e \textit{ True} \rightarrow_\delta \textit{True}$$

$$\textit{Or False False} \rightarrow_\delta \textit{False}$$

But, with the addition of these rules, it would no longer be true that a normal order reduction strategy guarantees reduction of a term to normal form if it has one. In fact, no deterministic reduction strategy would suffice to regain that property! Any deterministic strategy, on encountering $\textit{Or } e_1 \ e_2$, would have to always reduce e_1 or always reduce e_2 before reducing the other term. If it always first reduces e_1, then it will fail to reduce $\textit{Or } \Omega \textit{ True}$ to normal form; if it always first reduces e_2, then it will fail to reduce $\textit{Or True } \Omega$ to normal form.

Typed Lambda Calculus with Constants

The addition of constants to the lambda calculus merely made it easier for us to express concepts already expressible in the pure lambda calculus. The central notions of abstraction, application, and of the various kinds of reduction and conversion that we had introduced for the lambda calculus remained untouched. The only definition we really had to modify was that of a normal form, and we modified that implicitly with our introduction of constants as distinguished, normal form lambda terms. The introduction of types, however, fundamentally changes the lambda calculus.

To develop the typed lambda calculus with constants, we begin by adopting a lambda term grammar identical to that used for the lambda calculus with constants:

$$\Lambda \to V \mid C \mid (\Lambda\ \Lambda) \mid (\lambda\ V\ \Lambda)$$
$$V \to v \mid V'$$
$$C \to c \mid C'$$

We then introduce alongside this a parallel set T of TYPES comprising type variables, constants, and function types:

$$T \to V \mid C \mid F$$
$$V \to \alpha \mid V'$$
$$C \to \zeta \mid C'$$
$$F \to (T \to T)$$

Notational conventions accompany this introduction:

- σ, τ, υ represent arbitrary types.
- α, β, γ represent arbitrary type variables.
- The function type arrow $\to$ is considered to associate to the right. Thus, $\sigma \to \tau \to \upsilon$ should be read as $\sigma \to (\tau \to \upsilon)$.

We will not need to refer to arbitrary type constants, so no convention addresses them. Just as we named certain constant lambda terms 0, 1, and so forth, so too can we introduce names for various constant types, such as `int`, `real`, and `bool`.

Lambda terms and types come together in STATEMENTS. A statement $M : \sigma$ says that a given term M, the SUBJECT of the statement, can be assigned type σ, the PREDICATE of the statement.

Whereas lambda terms formed the basis of the pure lambda calculus, statements make up the basis of the typed lambda calculus. When

we introduced constants into the pure lambda calculus, we had to introduce δ-rules that formally related those constants within the system. We used these δ-rules to establish relationships that agreed with our intuitive understanding of how the constants were interrelated, but we had to be sure not to introduce a careless rule that changed the very properties of the lambda calculus that make it so useful to us. The particular statements that make up the basis of the typed lambda calculus are also left to our discretion, and we can use them in a similar way. Thus, we will assume, for example, that *True* : `bool` and *False* : `bool`, and that 0 : `int`, 1 : `int`, and so on. Formally, the basis – let us call it $\mathcal{B}$ – is composed of a set of statements whose subjects are distinct variables or constants.

Using this basis, we can assign types to other lambda terms. If we can derive a statement $M : \sigma$ from the basis $\mathcal{B}$, then we write $\mathcal{B} \vdash M : \sigma$. All statements can be derived using three rules:

- BASIS. If $x : \sigma$ is an element of the basis, then we can make the statement that $x : \sigma$.

$$\frac{x : \sigma \in \mathcal{B}}{\mathcal{B} \vdash x : \sigma}$$

- $\rightarrow$ INTRODUCTION. Abstraction is analogous to creating a function by transforming a variable in an expression into a parameter. The type resulting from abstraction reflects this.

$$\frac{\mathcal{B} \vdash x : \sigma \quad \mathcal{B} \vdash M : \tau}{\mathcal{B} \vdash (\lambda\, x\, M) : (\sigma \rightarrow \tau)}$$

- $\rightarrow$ ELIMINATION. Application is analogous to applying a function to an appropriate argument, and this is reflected in the type of an application.

$$\frac{\mathcal{B} \vdash M : (\sigma \rightarrow \tau) \quad \mathcal{B} \vdash N : \sigma}{\mathcal{B} \vdash (M\, N) : \tau}$$

The introduction of types has important implications for the central properties of the lambda calculus. The Church–Rosser property persists, but we gain several other powerful properties:

- SUBJECT REDUCTION. Type persists unchanged through β-reduction.

$$\frac{\mathcal{B} \vdash M \twoheadrightarrow_\beta M' \quad \mathcal{B} \vdash M : \sigma}{\mathcal{B} \vdash M' : \sigma}$$

- STRONG NORMALIZATION. If a term can be assigned a type, then it is strongly normalizing.*

- DECIDABILITY OF TYPE-CHECKING. Given a basis $\mathcal{B}$ and a statement $M : \sigma$, it is decidable whether $\mathcal{B} \vdash M : \sigma$.

- DECIDABILITY OF TYPE INFERENCE. Given a basis $\mathcal{B}$ and a term M, we can decide whether there is any σ such that $\mathcal{B} \vdash M : \sigma$. If there is, then we can use $\mathcal{B}$ and M to compute such a σ.

These are indeed powerful, useful properties, but the overall expressive power of the lambda calculus in fact *decreases* with the introduction of types. Recall that, in the pure lambda calculus, there were terms without normal forms, such as Ω. The term that we used to introduce recursion into the lambda calculus, Y, has no normal form. Both Ω and Y apply a term to itself:

$$\Omega \equiv (\lambda x.xx)(\lambda x.xx)$$

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

As you can see, they actually make use of nested self-application: both terms contain the application $(x\ x)$ within the application of one abstraction to the selfsame abstraction. In Ω, this abstraction is $\lambda x.xx$; in Y, it is $\lambda x.f(xx)$.

* Recall that a term is strongly normalizing if and only if it always β-reduces to normal form after a finite number of reductions.

But these terms cannot be assigned a type, for the fundamental reason that self-application is not typable. Let us see what happens when we attempt to assign a type to $\lambda x.xx$. We know that x must have some type, say, σ. The rule of $\rightarrow$ introduction specifies that, to apply one term to another, the first term must be able to be assigned a function type and the second term must be able to be assigned the same type as that to the left of the arrow in the function type. But this means that the type of x must be a solution to the type equation $\sigma = (\sigma \rightarrow \tau)$, and there is no such type in our typed lambda calculus with constants.*

We are not so sad to see Ω become untypable. A term that does nothing except lead to endless β-reduction is useless to us. But we needed Y to define recursive functions. Regaining Y is the point of our next extension.

Typed Recursive Lambda Calculus with Constants

After the effort of the past two extensions to the lambda calculus, extending the typed lambda calculus with constants to encompass recursion is surprisingly simple. All we need do is introduce a polymorphic fixed-point operator among our constants, introduce an appropriate type into our basis, and craft a δ-rule to make this operator behave as we desire.

Thus, we anoint Y our fixed-point operator name of choice. The only functions we want to apply Y to are those that must recurse upon themselves. As such, they must consume the very type of value they produce, that is, the type of any such function must be $(\sigma \rightarrow \sigma)$. A fixed point of such a function must have the type of the argument of the function. Since Y, given a function, produces a fixed point of that function, we assign Y the family of types $Y : (\sigma \rightarrow \sigma) \rightarrow \sigma$, which is to say that we add a $Y_\sigma : (\sigma \rightarrow \sigma) \rightarrow \sigma$ to our basis $\mathcal{B}$ for every type σ that can be formed according to our grammar for types.

* It is possible to extend the lambda calculus so that self-application becomes typable; see the discussion in Barendregt and Hemerik [19, Section 3.2, pp. 14–17] of recursive types and the $\lambda\mu$-calculus.

That Y in fact is a fixed-point operator is represented in the lambda calculus by the fact that $YF =_\beta F(YF)$. The final element of our extension, then, is a family of δ-rules corresponding to the family of typed fixed-point operators Y_σ that reintroduces this convertibility:

$$\frac{\mathcal{B} \vdash M : (\sigma \to \sigma)}{(Y_\sigma\ M) \to_\delta (M\ (Y_\sigma\ M))} \qquad \frac{\mathcal{B} \vdash M : (\sigma \to \sigma)}{(M\ (Y_\sigma\ M)) \to_\delta (Y_\sigma\ M)}$$

This has the effect that, if $M : (\sigma \to \sigma)$, then $(Y_\sigma\ M)$ and $(M\ (Y_\sigma\ M))$ are interconvertible. We call this type of conversion TYPED Y-CONVERSION.

With this final extension, we now have a lambda calculus that closely resembles the lambda calculi that underlie modern functional languages. But, before we talk of them, perhaps we ought to go over the developments and languages that led to today's functional languages.

12.3 BIBLIOGRAPHIC NOTES

De Bruijn indices were first described in de Bruijn [38]. Other notations for variable binding have been developed more recently; McBride and McKinna [80] describes an interesting hybrid that uses de Bruijn indices for bound variables and (ideally, meaningful) names for free variables.

We have not made as fine a distinction between the different reduction strategies as that made by Sestoft [118]. Our description equates call by value with applicative order and call by name with normal order, while he carefully distinguishes these in terms of where reduction can occur.

We have barely scratched the surface of the lambda calculus and type theory. Those interested in the impact of the lambda calculus on logic and computer science would find Barendregt [18] interesting reading. Turner [130] explains the significance of the lambda calculus

and Church's thesis in general for functional programming in but one chapter of a book [94] dedicated to examining Church's thesis seventy years after its postulation. If the theory of the lambda calculus itself is more in line with your interests, Barendregt [17] is the standard reference for the untyped lambda calculus and develops many aspects of the topic in tremendous detail. The lambda calculus in one form or another is also often hurriedly introduced as more or less new material where needed in books and articles dealing with functional programming.

The concept of *type* can itself be introduced into the lambda calculus in a variety of ways and then subsequently elaborated. Cardelli and Wegner [29] provides a very readable introduction to practical issues of types and programming languages – the presentation of types in this chapter is significantly influenced by the presentation therein – while Barendregt and Hemerik [19] looks more carefully at the ways typing can be introduced formally into the lambda calculus; our formal definition of the pure, untyped lambda calculus follows in part the two-page summary given near the beginning of this article. The process we followed of gradually extending the pure untyped lambda calculus into the typed recursive lambda calculus with constants follows that of Hudak [57]. Thompson [125], developed from lectures given at the University of Kent and the Federal University of Pernambuco, Recife, Brazil, introduces types in the context of programming languages and constructive logic, while Pierce [112] is a full-length textbook on type theory, and Pierce [111] an edited collection of research papers on the topic.

13

HISTORY

13.1 PREDECESSORS

There does not appear to be a consensus on which language was the first, truly functional language. This is because the argument inevitably ends up being about how the terms of the argument should be defined. What is a functional language? Are there elements it must have? Elements it must not? Do research languages count, or does a language have to have seen significant "real world" use? Is a given "language" really a language, or is it simply a dialect?

You will have to make up your own mind about these matters, possibly on a case-to-case basis. Regardless of your decisions, there *is* a good consensus on which languages contributed to the development of the functional family, regardless of whether or not they truly belong to it. To dodge the whole issue, we will simply characerize them as the predecessors of modern functional languages.

13.1.1 *Lisp*

The earliest predecessor is the list processing language. Known as LISP when it first appeared in the late 1950s (it was all the rage then to capitalize the names of programming languages), and since grown into a diverse family of Lisps, it appeared shortly after Fortran. It originated with McCarthy, and, in fact, elements of its list processing facilities were first implemented as extensions of Fortran [122].

Lisp grew out of artificial intelligence, particularly the expert systems and their need to perform list processing and limited theorem proving. In fact, the list is its primary and most characteristic data type. Lisp programs themselves can be characterized and represented as lists, and this lends Lisp its most distinctive feature: its heavy use of parenthesization. The ability of Lisp to represent itself in itself – Lisp lists are Lisp programs are Lisp lists – is known as HOMOICONICITY, and this lends Lisp much of its power and extensibility.

This focus on lists is unlike the lambda calculus, which features functions as its sole data type, and even in its extensions remains solidly anchored by its focus on the function. McCarthy draws on the lambda calculus solely to provide a notation for "functional forms" as opposed to functions – basically, to indicate which positional argument should be bound to which variable name in a function's definition. In introducing Lisp, he in fact states that the "λ-notation is inadequate for naming functions defined recursively" and introduces an alternate notation [83]. Many languages today get by using the lambda term Y that we introduced earlier for this purpose; the impact of the lambda calculus on Lisp was superficial, and this is in good part why one might want to exclude Lisp from a list of functional languages.

Much of the spirit of functional languages, however, first appeared in Lisp: functions as "first-class citizens" and the use of recursive functions as opposed to step-variable–based loops, as well as an elegant, remarkably simple definition characterize both Lisp and the modern functional languages. As far as elegance goes, it is possible to write a Lisp interpreter in not very many lines of Lisp.

Lisp flourished as artificial intelligence flourished, and it weathered the cold AI winter, perhaps even better than AI did itself. It was readily implemented by many groups and extended in many different directions, so Lisp soon became more a family of languages than a single language. From the 1960s on, there were two major Lisps (Interlisp

and MacLisp) and many other significant Lisps. Today, the two primary Lisps are Common Lisp, the result of a standardization effort in the 1980s, and Scheme, which in the mid-1970s sprang out of ongoing research in programming language theory and so was inspired more immediately by the lambda calculus.

13.1.2 *Iswim*

Iswim (for "if you see what I mean") is a family of programming languages developed in the mid-1960s by Peter Landin. It is the first language that really looks like modern functional languages. In stark contrast to Lisp, it is not all about lists, it uses infix notation, and it is thoroughly based on the lambda calculus. It also features `let` and `where` clauses for creating definitions local to a given scope. This is one of the most immediately visually distinctive elements of modern functional languages. Iswim also allowed the use of indentation (significant whitespace) for scoping alongside the more common punctuation-based delimiters.

13.1.3 *APL and FP*

APL ("a programming language"), developed in the early 1960s by Kenneth Iverson, was never intended to be a functional programming language, but rather an array programming language. Thus, it provided built-in support for operating on arrays in terms of themselves rather than in terms of their elements, as well as ways of composing these array operations. It is also notable for its concision: it was intended to be programmed in using a specialized alphabet. This, coupled with its approach to handling arrays, led to very compact programs.

It appears to have influenced John Backus in his development of FP. FP itself never saw much, if any, use. It was advocated in Backus' 1978 Turing award lecture in which he warned of the "von Neumann bottleneck" that ultimately constrains imperative programming languages to "word-at-a-time programming." FP intended to do for functional programming what structured programming did for imperative programming with its standard control flow constructs by providing a few higher-order functions ("functional forms") that he considered essential and sufficient for whatever one might want to do.

FP is most notable in the history of functional languages for the credibility it lent to the field – Backus received the Turing award in good part because of his fundamental role in the development of Fortran – and the interest it generated in functional programming. While the development of modern functional programming languages took a different road than that defined by FP, FP's emphasis on algebraic reasoning and programming using higher-order functions is very much of the same spirit.

13.2 MODERN FUNCTIONAL LANGUAGES

While the who's in, who's out of older languages is up for debate, most modern functional languages bear a close family resemblance. The central features of a modern functional language are:

- first-class functions and a firm basis in the lambda calculus;
- static typing coupled with type inference and polymorphism;
- algebraic data types and pattern matching.

Most modern functional languages also feature:

- abstract data types and modules;
- equational function definitions and boolean guards.

We will discuss each of these in turn.

13.2.1 *Central Features*

First-Class Functions and the Lambda Calculus

It is quite easy to represent functions in the lambda calculus and to create functions of functions. Such HIGHER-ORDER FUNCTIONS are unusual in imperative languages. Among the provided data types, they are usually second-class citizens: they have no literal representation, but can only be created through statements, nor can they be assigned to variables, passed into or returned from other functions. They are not on par with the integers or even characters.

Functional languages make functions first-class citizens. This means that:*

- Functions are denotable values: there is some way to describe a function literally, just as you would write `5` to denote the integer five without having to give it a name.
- Functions can be passed into functions: such functions with functional arguments are known as HIGHER-ORDER FUNCTIONS.
- Functions can be returned from functions.
- Functions can be stored in data structures: you can create lists of functions as readily as you would lists of integers.
- Storage for functions is managed by the system.

With functions as first-class citizens, it easy to create and employ higher-order functions, and functional programming has a rich vocabulary describing common, heavily-used higher-order functions and common

* This particular list is due to Mody [92]; others provide similar lists of "rights" characteristic of first-class data types. Some authors go further and describe the rights typical of second- and third-class data types, as well [for example 117, §3.5.2].

types of higher-order functions. First-class functions are also employed extensively in the form of curried functions.

First-class functions is the most striking result of functional languages' basis in the lambda calculus, and it heavily influences the entire style of programming in functional languages. But taking the lambda calculus as the starting point of the entire programming language is the most radical characteristic of modern functional languages, and the effects of this choice are felt throughout the resulting languages.

Static Typing, Type Inference, and Polymorphism

Modern functional languages are statically typed. They are based, not on the untyped lambda calculus, but on some variety of the typed lambda calculus. The introduction of types has advantages from the software engineering point of view. It also has advantages from the point of view of compiler performance.

Static typing in imperative languages is often regarded as a burden because of the need to declare the type of all variables and functions. Modern functional languages relieve this burden through type inference. This means that code written in functional languages is free to omit redundant type declarations: if you state that `x = 5`, then there is no need to reiterate that `x` is an `Integer` for the sole benefit of the compiler. Modern functional languages are designed to allow type inference, and their compilers are designed to perform it.

A surprising result of type inference is that it makes polymorphism the standard behavior for functions. Whenever a function could be construed as taking operands of a more general type, it is, unless an explicit type declaration is supplied that restricts this.

The standard higher-order function `map` is a good example of this. `map` takes as its arguments a function and a list and produces a list containing the results of applying the function to each element of the original

list in order. That description is somewhat complex; an example would perhaps be simpler. If we take for granted a boolean function `isNonZero` that takes an integer argument and returns either `True` if the number is nonzero or `False` if it is zero,* then

```
map isNonZero [0, 1, 2, 3]
```

evaluates to

```
[isNonZero 0, isNonZero 1, isNonZero 2, isNonZero 3]
```

and thence to

```
[False, True, True, True]
```

The type of map is `(a -> b)-> [a] -> [b]`, where `a` and `b` here are type variables as discussed in Polymorphism, p. 137.

Algebraic Data Types and Pattern Matching

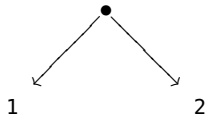
A distinctive characteristic of the type systems of modern functional languages is their support for creating and using ALGEBRAIC DATA TYPES (ADTS). Algebraic data types are so called because they can be looked upon as a sum of products of other data types. What this means practically is that algebraic data types function as discriminated (tagged) unions; the tags are called DATA CONSTRUCTORS and serve to wrap the supplied data in the algebraic data type. Pairs and lists are simple examples, but since special syntax is often supplied to make their use more natural, they are not very good examples of creating ADTs.

Let us instead consider an algebraic data type representing a tree with values of some unspecified type stored in the leaves. The declaration of such a data type might look like

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
```

* We can define `isNonZero` in Haskell as `isNonZero x = not (x == 0)`.

(The unspecified type that is being wrapped is represented here as `a` .) This also happens to be a *recursive* data type: each branch wraps a pair of subtrees. The declaration `tree = Branch (Leaf 1)(Leaf 2)` gives the variable `tree` the value of a branch with two leaves of integers. Thus, we have values of type `Integer` substituting for the type variable `a` in `Tree a`. The variable `tree` thus has type `Tree Integer`, read "tree of integer," and corresponds to the tree



We have seen that it is simple to create an algebraic type and build instances of that type. But how do we get at the wrapped information? To decompose algebraic data types, modern functional languages support PATTERN MATCHING.

The fundamental pattern-matching construct is the CASE expression. Its basic form indicates the variable for which cases are being enumerated and then sets up a correspondence between patterns and expressions to evaluate as the value of the *case* expression in the event the corresponding pattern matches the provided variable. The patterns are checked in the order they are listed; the first matching pattern decides which expression is evaluated. Still informally, but somewhat more symbolically, we could represent the form of the *case* expression as

`case` ⟨*variable*⟩ `of` (⟨*pattern*⟩ `->` ⟨*expression*⟩)$^+$

As an example, let us suppose we wish to count the number of branches in a tree. An example of such a function is `countBranches` of Listing 13.1 on page 179. The patterns are analogous to the expressions we would use to construct the type of data that the pattern matches; the variables of the patterns, rather than passing data into the construc-

Listing 13.1: Pattern-matching via `case`

```
countBranches tree = case tree of
                       Leaf   _   -> 0
                       Branch a b -> 1 + countBranches a
                                       + countBranches b
```

tors, instead are used as names for the data that was initially supplied as parameters to the constructor.

The type of this function together with its name provide an excellent summary of its behavior. It also provides us with another example of polymorphism and our first example of subtyping. The type of `countBranches` is `(Num t1)=>Tree t -> t1`. Here, `(Num t1)=>` expresses a restriction on the type of the type variable `t1` used in the rest of the type expression. It says that the type of `t1` must be some subtype of the type class `Num`. The underscore you see in the definition of this function is used in patterns as a "don't care" symbol: it indicates the presence of a value that we choose not to bind to a name, since we do not intend to refer to the value.

13.2.2 *Other Features*

Abstract Data Types and Modules

ABSTRACT DATA TYPES are data types that hide their concrete representation from the user. In this way, the representation of the type becomes internal to it: the fact that, say, a stack is actually implemented as a list is hidden, and only operations dealing with stacks as stacks are exposed. This means that the implementation of the abstract type can be changed as necessary. For example, if lists proved too slow to support the heavy use we wished to make of stacks, we could move instead to some other representation without having to change any of the code that used our stacks.

This kind of implementation hiding together with interface exposure is frequently accomplished through a module system. The existence of a powerful and usable module system is an important part of the "coming of age" of functional languages, because modules are necessary to support "programming in the large" as is necessary in real-world environments where complex problems must be solved and large amounts of code are involved. In terms of modules, an abstract data type's representation is hidden by not exporting representation-specific definitions for use in the program importing the module.

In the context of abstractions of algebraic data types, this takes the form of not exporting the data constructors. Instead, other functions are exported that make use of the data constructors without exposing this fact to the user of the abstract data type. A simple version of such a function would simply duplicate the data constructor. More complex versions can build in bounds-checking, type-checking, or normalization of the representation – for example, such a "smart constructor" could be used to ensure an internal tree representation remains balanced.

Equations and Guards

Modern functional languages support a very readable, compact notation for defining functions that builds on the pattern matching performed by *case* statements. They allow functions to be defined as a sequence of equations. Listing 13.2 on page 181 reimplements the functionality of Listing 13.1 (p. 179) using an equational style of function definition. If you compare this new definition to the earlier definition, which used the *case* expression, you will see that the pattern matching is implicit in the syntax used to define functions equationally.

Another feature of modern functional languages that simplifies function definition is GUARDS. Guards are boolean predicates that can be used in function definitions and *case* statements. Guards block the ex-

Listing 13.2: Pattern-matching via equational function definition

```
countBranches2 (Leaf   _   ) = 0
countBranches2 (Branch a b) = 1 + countBranches a
                                + countBranches b
```

Listing 13.3: Cases with guards

```
isLeaf t | countBranches t > 0 = False
         | otherwise           = True
```

pression they precede from being used when they evaluate to false, even if the pattern preceding the guard matches. The first pattern and guard successfully passed determines the case that applies to the given value.

An an example, we could use one of the `countBranches` functions given earlier to define an `isLeaf` predicate for use with our trees. If the tree has zero branches, it must be a leaf. If it has one or more branches, it must not be. In Listing 13.3 on page 181, we use a guard that applies this number-of-branches test in order to prevent the function `isLeaf` from evaluating to `True` when its argument `t` is a tree with more than zero branches.

We can also describe guards in terms of how the same effect could be accomplished using other expressions. Guards used with function definitions can be seen as equivalent to chained *if* expressions where each successive guard appears in the *else* branch of the preceding guard.* The expressions being guarded in the function definition become the contents of the *then* branch that is evaluated if their guard evaluates to true. A translation along these lines of the `isLeaf` function of Listing 13.3 (p. 181) is:

```
isLeaf2 t = if countBranches t > 0 then False else True
```

* We do indeed mean *if expressions*, not *if* statements. An *if* expression can be used anywhere an expression is expected. The *else* branch is always required, which means the expression will always have some value, either that of the true or the false branch.

Listing 13.4: Guards as chained if-then-else–expressions

```
isLeaf3 t = if countBranches t > 0
               then False
               else if True
                       then True
                       else True
```

But, since the `otherwise` of Listing 13.3 (p. 181) is simply another name for `True`, we can produce a more faithful (and redundant) translation of the original `isLeaf` function, as shown in Listing 13.4 on page 182.

We can similarly transform a *case* statement that uses both patterns and guards, but this requires a significant amount of nesting and duplication. We must first attempt to match the patterns. As before, if a pattern does not match, the next pattern is tried. Each guard migrates to the corresponding expression. The original expression is wrapped in an *if* expression that tests the corresponding guard condition. If the test succeeds, the *then* branch is the expression corresponding to the pattern just matched and guard just passed is evaluated. Otherwise, we must duplicate the remaining patterns and guards, and transform them similarly.

An example should clarify this. We will not use descriptive names as before, because they would obscure the transformation and motivating such descriptive names would unduly prolong this discussion. The case expression with three guarded branches of Listing 13a (p. 183) can be transformed as described above into the nested case expressions without guards of Listing 13b (p. 183).

These transformations can be adapted to handle multiple guarded expressions per pattern, but transformation process only becomes more tedious. The examples given should suffice to demonstrate how much the use of guards simplifies both reading and writing of functional programs using pattern matching.

Figure 13: Transforming a *case* statement to eliminate guards

(a) With Guards

```
case E of
    p1 | g1 → e1
    p2 | g2 → e2
    p3 | g3 → e3
```

(b) Without Guards

```
case E of
    p1 → B1
    p2 → B2
    p3 → B3
where
    B1 = if g1
           then e1
           else
               case E of
                   p2 → B2
                   p3 → B3
    B2 = if g2
           then e2
           else
               case E of
                   p3 → B3
    B3 = if g3
           then e3
           else error ("Patterns not"++
                       " exhaustive")
```

13.3 CLASSIFIED BY ORDER OF EVALUATION

Functional languages developed along two branches. These branches are distinguished by their evaluation strategy: one branch pursued the applicative order, call-by-value evaluation strategy; the other pursued the normal order, call-by-name evaluation strategy. Languages belonging to the applicative order branch are called EAGER LANGUAGES because they eagerly reduce functions and arguments before substituting the argument into the function. Presented with the application $f\,M$, where $f \to_\beta f'$ and $M \to_\beta M'$, an eager language will reduce $f\,M$ to $f'\,M'$ and only then substitute M' into f. Languages that are part of the normal order branch are called LAZY LANGUAGES, because they delay reducing functions and arguments until absolutely necessary. When a function is applied to an argument, they simply substitute the argument wholesale and proceed with reduction of the resulting lambda

term. When a lazy language encounters an application f M of f to M, it immediately performs the substitution of M into f.

The branches have also diverged along the lines of purity and strictness. Eager languages have historically been IMPURE, meaning that they allow side effects of evaluation to affect the state of the program. DESTRUCTIVE UPDATE (also known as mutation) is a prime example. Using destructive update, we can sort a list in place simply by mutating its elements into a sorted order. Without destructive update, we would be forced to use the old list to produce a new list, which requires us to allocate space for both the original list and its sorted counterpart.

While destructive update might lead to local improvements in efficiency, it and other impurities destroy REFERENTIAL TRANSPARENCY, since the same expression no longer evaluates to the same value at all times and places in the program. Consider the list `ell = [3, 2, 1]`. With this definition, `head ell` evaluates to `3`. But if we sort it in place, later occurrences of `head ell` will evaluate to `1`. As you can see, `head ell` is no longer always equivalent to `head ell`: the reference `head ell` is no longer transparent.

Losing referential integrity complicates reasoning about the behavior of the program and the development of any proofs about its behavior. While impurity makes it easier to rely on knowledge of data structures and algorithms gained while using imperative languages, it also undermines one of the strengths of functional programming, that its programs are easier to reason about. The ability to fall back on imperative algorithms also stunts the development of purely functional data structures and algorithms. This is impurity as crutch.

While lazy languages have remained pure, this is in good part due to necessity. Lazy reduction makes it difficult to predict when a particular term will be reduced, and so it is hard to predict when the side effects of a particularly reduction would occur and difficult to ensure they occur when you wish.

The decision between strict and non-strict semantics has also frequently fallen along family lines. Eager languages are almost always strict, by which we mean that the functions of that language default to being strict.* If they are going to pursue an applicative order reduction strategy, unless they investigate some sort of concurrent pursuit of several reductions simultaneously, then they will be stuck reducing a divergent argument regardless of whether it would be needed by the function once the substitution of the argument into the function is made. This is the case when functions that ignore their argument are applied to a divergent term: $(\lambda x.\lambda y.y)\Omega \rightarrow_\beta \lambda y.y$, but if you attempt to evaluate Ω prior to substituting it for x in the function, the evaluation will diverge. Lazy languages, on the other hand, will not fall into this trap. Their evaluation strategy makes them non-strict.

The way that laziness forces a language to take the "high road" of purity has been referred to as the "hair shirt of laziness" [102]. The purity that results from adopting non-strict semantics has a pervasive effect on the entire language. For example, one is forced to discover a functional way to cope with input–output, and computation with infinite data structures becomes feasible. Infinite data structures are usable in a lazy language because, so long as only a finite amount of the structure is demanded, evaluation continues only until that amount has been evaluated.

We have provided some background on the two primary branches of the modern functional family. Now we will briefly summarize their history.

13.3.1 *Eager Languages*

The most influential eager languages have fallen under the umbrella of the ML family. ML originally began as the metalanguage (hence the

* Whether it is even possible to avoid strictness in a particular case, and the particular methods for doing so where it is possible, will differ from language to language.

name) for the LCF theorem prover project under way at the University of Edinburgh in the early 1970s. It bears a resemblance to Landin's proposal for Iswim. It is a modern functional language, and as such supports the features discussed earlier. Its strong emphasis on type inference was groundbreaking. Type inference was made possible by Milner's rediscovery of a type system earlier described by Damas and Hindley that walked the fine line between a too powerful type system in which type inference is infeasible and an overly restrictive type system.

In the late 1980s, ML was standardized under the name Standard ML. Standard ML is unusual among programming languages in that the entire language has a formal definition, first published in 1990. Standard ML's support for modules (called structures in Standard ML) is unusually extensive and complex; module signatures (interfaces) can be specified separate from the modules themselves, and it is possible to define functions over modules (such functions are known in ML as functors). A revised edition of the definition was published in 1997. Along with some slight changes to the language, the revision introduced the Standard Basis Library in order to specify a common set of functionality that all conforming Standard ML implementations should provide.

ML's background as a metalanguage for a theorem prover is reflected in its continuing use in programming language research and theorem proving. This research is greatly aided by the published standard: extensions of the language have a solid basis on which to build. But SML was not the only outgrowth of ML.

The Caml languages are another branch of the ML family. This branch has arguably eclipsed Standard ML, particularly in the number of non-research uses to which its languages have been put. Caml was originally an acronym for "Categorical Abstract Machine Language"; the name has been retained, though the abstract machine has long

been abandoned in its implementation. The language began development in 1987 for use in projects of the *Formel* project at INRIA; the primary outgrowth of this has been the Coq proof assistant. Because the language was meant for internal use, it was not synchronized with Standard ML, since adhering to a standard would make it difficult to adapt the language as needed to suit the problems faced in the group's work.

The start of the 1990s saw the reimplementation of the Caml language. This version of Caml was called Caml Light and featured a bytecode compiler. The interpreter for this bytecode was written in C so as to be easily portable. A bytecode-compiled program can run without changes on any platform to which the interpeter has been ported. Caml Light was promoted as a language for education.

In 1996, Objective Caml made its debut. Objective Caml adds support for object-oriented programming to Caml Light, strong module support, an extensive standard library, and compilation to native code in addition to continuing support for bytecode compilation. In the mid-2000s, Objective Caml became the inspiration for Microsoft's F# programming language meant to be used with their .NET framework.

The Caml family of languages provides a marked contrast to the Standard ML family. While Standard ML was published as a formal document with clear roots in programming language research, the development of the Caml languages is driven by their continued use for day-to-day programming to support other interests. Standard ML is a single language with many independent implementations. The Caml family, on the other hand, is defined by its provided compiler: whatever the compiler will accept is what the language is at any given time. Thus, a Caml language is defined more by its REFERENCE IMPLEMENTATION than by any formal document. Objective Caml is not just a language, but a compiler and a host of other tools (such as a preprocessor, profiler and debugger, and tools for performing lexing

and parsing) that come together to make up the current version of Objective Caml.

13.3.2 *Lazy Languages*

Our description of modern eager languages focused on the prominent ML family. Modern lazy languages developed a bit differently. Much of the early work in lazy languages was done by Turner in a series of languages developed during the late 1970s and early 1980s.

1976 saw the appearance of SASL, the St. Andrews Static Language. It introduced the equational style of function definition and the use of guards. Functions were automatically curried, and indentation could be used in place of semicolons. The type system was rudimentary.

KRC, the Kent Recursive Calculator, made its debut in 1981. It made lists easier to use by introducing a shorthand notation and list comprehensions. Both will seem quite familiar to anyone acquainted with higher mathematics. Shorthand notation allowed the use of ellipsis dots to express ranges. Thus, `[1..5]` is equivalent to `[1, 2, 3, 4, 5]`, and `[1..]` creates the infinite list `[1, 2, 3, 4..]`. LIST COMPREHENSIONS* provide a compact notation for generating lists from other lists.

For example,

```
[ 2*x | x <- [0..], 2*x < 100 ]
```

can be read as "the list with elements $2 * x$, where $x = 0, 1, \ldots$ and $2 * x < 100$." This is similar to the set expression $\{x \mid x \in \mathbb{N} \wedge 2 \cdot x < 100\}$. As you can see, the notation for the list comprehension has two sides. The right hand side contains generating expressions and filters. Generating expressions such as `x <- [0..]` introduce a name. In the course of evaluating the list comprehension, this name will be bound to each

* We now call them *list comprehensions*. At the time, they were described as both *set expressions* and *ZF expressions*.

value of the generating list in turn. The left arrow `<-` can be read as "drawn from." Filters evaluate to either true or false. If they all evaluate to true, then the current bindings of the names introduced by the generating expressions are used to evaluate the expression on the left hand side of the list comprehension. The result of evaluating the left hand side expression is appended to the output list, new bindings are made, and the process repeats. (While this description has not gone over all the details, it should be enough to communicate the flavor and expressiveness of list comprehensions.)

We likened list comprehensions to the set expressions used in mathematics, but a list comprehension differs from a mathematical set expression in two important ways:

- It can contain duplicates.
- It is produced algorithmically and its results are ordered.

A simple list comprehension containing duplicates is `[1 | x <- [1..3]]`, which produces `[1, 1, 1]`. That the results are ordered is necessitated by our drawing values from lists and putting the results in a list. That the result is produced algorithmically is important when we use infinite lists, as above where `x` is drawn from a list of natural numbers. Even though we know that, once the output list has had 98 appended to it, no greater value of x will satisfy the predicate $2 \cdot x < 100$, the interpreter will continue to evaluate the list comprehension until as many values as we ask for have been produced; if we ask for all values, it will continue forever, since there is always one more x to try in the infinite list `[0..]`. Thus, thinking of these as set expressions rather than attractively concise ways to generate lists can lead to trouble.

Turner's language design efforts culminated in Miranda. Miranda was the first of his languages to feature a Hindley–Milner type system. It included user-defined types and polymorphism. It also featured SECTIONS, which solve a notational problem with infix operators. If

we want to map a normal, prefix function `f` down a list `L`, then `map f L` suffices. But if we wish to halve each element of the list, we must resort to either defining a throw-away function, say `half x = x / 2`, or defining an anonymous function using lambda notation, such as `\ x -> x / 2`. Sections make it possible to refer to use the infix operator directly here. Sections are written by surrounding the infix operator in parentheses. If the operator alone is in parentheses `(/)`, it is called a section; if a value is supplied to the left or right, it is known as a left or right section, respectively. Thus, we could express "halve each element" by composing `map` with a right section of `/`: `map (/2)L`. Likewise, we could generate a list of the reciprocals of all elements of the list `L` using `map` and a left section of `/`: `map (1/)L`.

Turner founded a company in 1983 to commercialize Miranda. He attempted to transfer lazy functional programming into industry. Miranda was the most developed lazy functional programming language of its time, but Miranda was not free. Distribution of derivatives was prohibited without the company's consent in order to avoid a proliferation of dialects and to keep Miranda programs portable, which led to some conflicts with other researchers.

The late 1970s and early 1980s had seen a proliferation of similar lazy, purely functional languages. The syntax differed, but the semantics were virtually identical, so that researchers had no problem understanding each other's papers. But the lack of a common language was seen as a problem for both research, through the duplication of effort, and for promoting use of lazy functional languages outside of research, since no single language was supported and most had been developed for research rather than industrial application.

This situation was resolved through the creation of a freely available, purely functional, lazy language called Haskell intended for use in research, education, and industry. Over the course of the 1990s, implementations of the Haskell language matured and Haskell eventually

displaced Miranda in both education and research. Miranda continues to be used and taught in some places, but the niche it once filled is now occupied by Haskell.

While Haskell was intended to standardize the state of the art in lazy functional languages, it did end up introducing new ideas. Type classes were developed for the first version of Haskell. A type class can be looked at as a named description of the functions that an instance of the type class must support and the types of those functions. Types that are declared to be instances of a specific type class then must provide implementations of the type class's functions. In a surprising parallel to object-oriented programming, those functions thus become overloaded in a way that is resolved through a hierarchy of types. (Object-oriented programming's overloaded methods are resolved on the basis of the object's identity, an important distinction.) Type classes have been extended in various ways as Haskell evolved, and Haskell has become a playground for "type hackery" such as implementations of Peano arithmetic at the type level.

The other new idea that Haskell embraced was the use of monads. Monads entered Haskell some years after it was first standardized. They came by way of denotational semantics; practical experience with them in Haskell led to their extensive use to constrain side effects to well-defined regions of a program so that referential transparency is not destroyed. This led to a somewhat better solution to the problems of input–output that have long plagued functional programs (about which we will say more in the last part of this thesis).

Haskell has remained the state of the art in lazy functional languages. Its policy of allowing a published language definition to coexist with extensions of the language and various dialects has enabled further research to be carried out by extending or modifying Haskell. Extensions that are embraced by the community of Haskell users are subsequently standardized and included in the next revision of the

standard. Use of Haskell outside research continues to grow, as does Haskell's influence in the world of programming languages.

13.4 BIBLIOGRAPHIC NOTES

Hudak [57] surveys the history of functional programming languages through the 1980s. It develops the concepts of the lambda calculus and its extensions in parallel to the history. This survey particularly influenced the overall shape of our history.

Lisp made its debut in McCarthy's seminal paper "Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I" [83].* There is a good body of literature on the history of Lisp. McCarthy gives a recounting of its early history [81]. Stoyan [122] covers much the same time period, concluding their history a bit before McCarthy, but where McCarthy's history was based primarily on his recollection, theirs is based on written records. It is very interesting to watch the elements of Lisp gradually fall into place here and there throughout various documents. McCarthy's Lisp retrospective [82] provides a very concise recounting of the most significant innovations and characteristic elements of Lisp. Steele Jr. and Gabriel [121] gives a fascinating recounting of the tumultuous history of the Lisp family that transpired between the early history as described by McCarthy and Stoyan and the standardization of Common Lisp. Layer and Richardson [72] describes the novel elements of the Lisp programming environment, including some information on Lisp machines, computers that were specially developed to support Lisp and its environment. As for Scheme, its community recently (2008) ratified *The Revised^6 Report on the Algorithmic Language Scheme*.†

* If you read this paper, you will find that we have fudged some of the technical details of Lisp's description and omitted recounting some significant innovations that were not relevant to the body of functional programming. This was intentional.

† Affectionately known as the R^{6}RS; the R^{6}R part stands for the *Revised Revised ... Revised Report*. For the report itself as well as details on the process that led to its ratification, see http://www.r6rs.org/.

Iswim was introduced by Landin [69] as language framework meant to support creation of full-featured domain-specific languages. FP was first described in Backus's Turing award lecture [13]. APL is described in a book [61] by its creator, Iverson.

Gordon gives a brief history [47] of the LCF theorem prover project that led to ML and of LCF's successors. The type system and inference algorithm described by Milner [87] was also independently developed by Curry [34] and Hindley [53]. Milner's work was subsequently extended by Damas [35]. The type inference algorithm is known as both the Hindley–Milner algorithm and the Damas–Milner algorithm and centers around the unification of type variables. The algorithms can also be expressed in terms of generating and subsequently solving a system of constraints [113]. Kuan and MacQueen have described [67] how two compilers, one for Standard ML and one the Objective Caml compiler, have improved the efficiency of the algorithm by ranking type variables.

Standard ML [88, 89, 90] incorporated a module system developed by MacQueen [74, 75, 76, 77]. Unlike the language definition itself, part of the documentation of the Standard Basis is available online (`http://sml.sourceforge.net/Basis`) as well as in a book [45]. The website provides only the formal specification; the book includes tutorials and idioms, as well. An initiative (`http://sml.sourceforge.net/`) is under way to support the development of common tools and test suites and more coordination overall between Standard ML implementors and implementations.

The recollections of a member of the team that developed Caml [32] provided much of the material for our description of the Caml language family. Information on the current status of the various Caml languages can be found online (`http://caml.inria.fr/`).

Documentation of SASL and KRC is sparse. Very little on SASL was published outside technical reports and user manuals. A later version

of the user manual [131] indicates that SASL was extended with KRC's list comprehensions and support for floating point numbers. Another paper [116] introduces the implementation of SASL at the Austin Research Center, which went by the name ARC SASL. ARC SASL also included list comprehensions, though there is no indication of floating point support. KRC [129] was introduced as part of a paper explaining why functional programming languages are superior to others, where it is described succinctly as "(non-strict, higher order) recursion equations + set abstraction."

Miranda was created by Turner in the 1980s [128] and heavily influenced the design of Haskell. Miranda can now be freely downloaded for personal or educational use from `http://www.miranda.org.uk`. The history of Haskell, including its use of type classes and monads, is thoroughly described [58] by several members of the committee that developed the language. Current information, including an up-to-date version of the published Haskell Report [100] defining the language, is available online (`http://haskell.org/definition`).

14

COMPILING

14.1 FROM INTERPRETERS TO COMPILERS

Higher-level imperative languages in a sense grew out of assembly language. They were designed with compilation in mind. Functional languages, however, have a history of interpretation. Lisp began solely as an interpreted language, and ML programs were first run using an interpreter written in Lisp. It was not immediately clear how to compile such languages. The interpreters supplied a programming environment (for example, the interpreter might support both editing and debugging of code from within the interpreter) and were used interactively. The interpreter managed the runtime system, including in particular GARBAGE COLLECTION, which freed space allocated to data that had become useless. Later, such interpreters were extended to support on-the-fly compilation of function definitions.

When it comes to functional languages, the distinction between interpreter and compiler becomes blurry. Interpreters can perform compilation, and compilers for functional languages frequently provide an interactive interface in addition to simple compilation facilities. When a program written in a functional language is interpreted, the interpretation manages garbage collection, storage allocation, and other such issues. All this bookkeeping is the province of the RUNTIME SYSTEM, and it does not go away simply because one wants to compile a program instead of interpret it. In order to work, the compiled representation of a program written in a functional language must be coupled with a runtime system. It is a short step from providing each compiled

representation with a runtime system to providing each with both a runtime system and its own, private interpreter. This code–runtime-system–interpreter package is self-sufficient: it is directly executable. Such a package is, in fact, what some compilers for functional languages produce.

Other implementations offer the option of compiling a program to a bytecode representation for interpretation afterwards by a virtual machine. The same bytecode program can then be run on a variety of platforms, so long as its minimal platform – the virtual machine – is available to host it.

14.2 THE RUNTIME SYSTEM

The runtime system is an essential part of compiling functional languages. It provides a common set of functionality needed by all compiled programs. The two most critical services it provides are primitive operations and storage management. Primitive operations are what actually perform the computation specified by δ-rules. They also enable programs to interface with their environment by providing access to functionality related to the operating system. Storage management is essential, since space is allocated as needed and must be reclaimed in order to prevent excessive memory use. The run time system might also handle threading, bytecode interpretation, and runtime linking of object code. Because the runtime system oversees primitive operations and memory management, it also plays an important part in profiling the runtime behavior of the compiled code.

Every compiler for functional languages will contain a run time system of some sort. A runtime system provides several distinct, major services. Storage management can become quite involved; primitive operations are a necessary part of producing working code. Since each of these services can to some degree be dealt with independently of

the others, the compiler's structure may not reflect a concept of a runtime system as such, but the basic services will nevertheless be in place and recognizable; they simply will not be grouped together.

14.3 THE PROBLEM

We have seen that functional languages are based on the lambda calculus and provide the programmer with a variety of higher-level abstractions such as algebraic data types, pattern matching, and higher-order functions. These have few parallels in imperative languages, and they create new problems for compilation.

New abstractions are not the only source of problems encountered when compiling functional languages. We compiled imperative languages by progressively lowering the level of abstraction of the program's representation till finally we were representing the program in machine instructions. If we try the same lowering process with a functional language, we run into a snag: instead of bottoming out in machine language, we bottom out in the lambda calculus. Functional languages are based on a model of computation fundamentally different from the von Neumann model at the root of the imperative languages. To compile a functional language, we must somehow model the lambda calculus's computation-through-reduction using the von Neumann computer that is the target platform.

We will return to these problems as we go through the phases of compilation.

14.4 THE FRONT END

The problems confronted by the front end do not change when we move from imperative to functional languages. However, if we choose to implement the front end in the functional language itself, we can

take advantage of the abstractions offered by functional languages in constructing the lexer and parser. If we instead implement generators for lexers and parsers using the functional language, it becomes a simple matter to integrate lexing and parsing into a compiler written in the language itself. Other programs written in the language then also have ready access to a lexer and parser. Perhaps this goes some way to explain why many compilers for functional languages are distributed along with both a lexer generator and parser generator written in the language of the compiler.*

As you might imagine, semantic analysis takes on a new importance in languages where type inference is taken for granted and the programmer can create new data types. Type inference can be treated as a pass in itself. Type inference replaces type checking, since once the compiler has reconstructed a valid type for a term, the type has been checked. If the term cannot be assigned a valid type, type inference has failed: either the program is not well-typed, or the programmer must supply type annotations for some term that is valid but for which a type cannot be inferred.

14.5 INTERMEDIATE REPRESENTATIONS

Compilers for functional languages employ some intermediate representations not used by imperative language compilers. Functional languages are on the whole a sugaring of the lambda calculus, and so it is possible to represent a program written in a functional language using smaller and smaller subsets of the full language. Thus, source-to-source transformations, where code in the source language is rewritten in the source language in an altered form, play a more important role in functional languages than is common in imperative languages.

* That this allows the compiler writer to avoid the complexities associated with defining and then using a foreign-function interface to programs produced using an imperative language could also be a motivating factor.

Transformations into a core language are in fact sometimes used in definitions of functional languages in order to explain the behavior of more complex constructs.

Just as programs represented in imperative languages are translated into SSA form because this facilitates static analysis, optimization, and proof of a given optimization's correctness, it was popular around the 1980s to translate a functional language program into CONTINUATION PASSING STYLE (CPS). In CPS, control flow and argument passing is made explicit. Every function is augmented with a further argument that serves as the CONTINUATION. The function is then called with another function that is to use the result of its computation as the continuation argument. Rather than returning the result `x` of evaluating the function to a caller, the function instead invokes the continuation with `x` as the continuation's argument. In compiling call-by-value languages, translation into CPS has been proven to enable more transformations than are possible in the source language.

However, since the translation to CPS is ultimately reversed during code generation, recent compilers have moved to carefully performing some of the transformations developed for use with CPS directly in the source representation. CPS is still used locally for some optimizations in a process known as "contification" or local CPS conversion. This can be used alongside SSA to enable further optimizations during functional language compilation.

Graph representations of the program also play a bigger part in some compilers. A large class of compilers build their back end around graph representations of the program; reduction is performed in terms of the graph. The development of such GRAPH REDUCTION machines played an important part in making lazy evaluation feasible, since they provide a ready way to conceive of substitution in reduction without copying. If all terms are represented by a collection of linked nodes, rather than copying the term to each location in order to substitute it,

we instead make multiple links to the single original term. When one of the substituted terms is reduced, all terms immediately share the result: no reduction is performed more than once.

Some compilers employ *typed* intermediate languages. This allows them to use the additional information provided by types throughout compilation. Instead of simply performing type checking to ensure the program is valid and subsequently ignoring types, the type of terms becomes additional fodder for analysis and optimization.

14.6 THE MIDDLE END

Just as in imperative compilers, the middle end is where the most effort is expended. Optimization is the key to producing good code and a good compiler. (Naturally, different kinds of optimization will be required depending on your idea of good.) Compilers for functional languages are typically the subjects and results of research. Different compilers are frequently based around entirely different intermediate representations and back ends, so work on optimization for functional languages is much more balkanized than research on optimization for imperative languages. Optimizations described in the literature are generally described in terms of improvements of an existing compiler in the context of a particular language, set of intermediate representations, and back end. It is not always clear which parts of this work applies in general to functional language compilation, and which parts are inextricable from the particular context in which they were developed.

While the particular optimizations that can be performed might differ from compiler to compiler, all compilers for functional languages confront a set of common problems due to the features that modern functional languages offer. These problems are partially addressed through enabling and performing specific kinds of optimizations and

partially through design of the back end. They can also be addressed through extensions to the language itself that allow the programmer to provide further information to the compiler.

Compiler-specific language extensions are not confined to functional language compilers, of course. An imperative example would be a C compiler adding support for an equivalent of the `restrict` keyword added by the C99 standard prior to the standard's publication. The `restrict` keyword is a type qualifier meant to be used with pointers. It is used to declare that the object pointed to by the pointer will be accessed only through the pointer. This shortcuts the need for the compiler to perform difficult alias analysis to determine whether this is the case by allowing the programmer to advise the compiler that this relationship between the pointer and the pointed to holds. More importantly, this allows the programmer to declare that this restricted relationship holds even when the compiler would be unable to infer the relationship through analysis, which enables previously impossible optimizations.

Such extensions are not without peril. The `restrict` keyword also provides one more way for C programmers to shoot themselves in the foot. If an optimization relies on the fact that a pointer is declared with the `restrict` type qualifier, but the relationship indicated by the qualifier in fact does not hold, then optimization could introduce erroneous behavior into an otherwise correct program. The same difficulty is a matter of concern for other extensions that provide information relied on in optimization that cannot be verified independently through analysis; at the same time, an extension that does not also extend the potential for optimization would be redundant.

Listing 14.1: Creating a closure
A closure is a function together with an environment providing bindings for the function's free variables. The binding used for each variable is the one lexically closest to where the function is defined. In this example, the variable `n` is free in the definition of `addNto`. The closest definition of `n` is that made by `makeAdder`. Thus, evaluating `makeAdder 3` results in a closure containing the function `addNto` and an environment in which `n` is bound to `3`.

```
makeAdder n = addNto
    where addNto m = n + m
```

14.6.1 *Common Problems*

Whether the compiler chooses to extend the language or not, it still faces some common problems.

- First-class functions require the construction of closures (which are defined below). A lazy evaluation strategy requires the creation of even more closures.
- The immutability required to preserve referential transparency can require significant amounts of copying. For example, sorting a list recursively produces a multitude of new lists. Lists can be expensive to construct and manipulate, but they are used extensively in functional programming, as are algebraic data types and pattern matching in general.
- Polymorphism is desirable, but it also requires that all arguments be treated identically regardless of their type: no matter whether an argument is an integer or a list, it has to fit in the same argument box.

Closures and Suspensions

A CLOSURE is formed by taking a function definition and binding any free variables to their existing definition in the closest enclosing (generally lexical) environment. The code in Listing 14.1 (p. 202) returns a

closure that can be used to produce a function that always adds two to its argument:

```
add2to = makeAdder 2
```

With this definition, evaluating `map add2to [1, 2, 3]` results in `[3, 4, 5]`. Note that the definition of the function `addNto` uses a variable `n` that is not passed to it. This variable is defined in the immediately enclosing environment of `makeAdder`. When `makeAdder 2` is evaluated, `n` is bound to `2` and a closure of `addNto` is returned wherein `n` is bound to the value `n` had when the closure was created. Evaluating `makeAdder 3` results in a closure where `n` is bound to 3. If there were no definition for the free variable `n` in the function definition, it would be impossible to produce a closure and the definition would be in error. This would be the case if we attempted to define `g x = x + y` in an environment without any binding for `y`.

A closure could at times be formed by partially evaluating the closure by directly substituting the definition, particularly in eagerly evaluated languages, but it is tricky to ensure this specialization of the function takes into account the already available definitions and preserves the semantics of evaluation of the unspecialized function. Instead, a closure is most often implemented as an unevaluated function together with its own environment of definitions. Only once all arguments have been provided to the function will evaluation actually occur. In this sense, a closure represents a frozen, or suspended, computation: a promise to perform some evaluation once all arguments are available to the function. Dealing with closures efficiently thus becomes an important part of enabling heavy use of higher-order functions in programs written in a functional language – and any functional language that encourages currying encourages frequent use of higher-order functions.

Since lazy languages only evaluate a term when necessary, they must make extensive use of suspended computations and only force

their evaluation as needed. Optimizing the implementation of such suspensions thus becomes an important part of optimizing a compiler for a lazy language. Indeed, a common optimization is to introduce STRICTNESS ANALYSIS, which attempts to eliminate the construction of suspensions that will perforce be evaluated in the course of evaluating the program. As an example, a request to display the result of a computation requires that the entire computation be carried out to produce the result. There is no question of some part of the result not being required, since the entire result is supposed to be output. Such a display function is strict in its argument.

Referential Transparency and Copies

The immutability required to preserve referential transparency can require significant amounts of copying. For example, sorting a list recursively produces a multitude of new lists. This has nothing to do with strictness. The solution to this problem is a combination of deforestation, also called fusion, and update analysis. DEFORESTATION attempts to eliminate data structures that are created only to be immediately consumed. This can be considered to some extent as a special case of UPDATE ANALYSIS, which attempts to discover when functions accepting a data structure and returning a modified copy of that data structure can be implemented so that they instead update the original data structure without producing a copy. This can be done whenever the original data structure will not be accessed in the future. With this requirement satisfied, the in-place, destructive update can be done without destroying referential transparency, since there are no remaining references through which the update of the original data structure could be discovered. Mutable references, such as are allowed by the ML family of languages, might seem a way to allow the programmer to directly intervene to solve this problem, but mutable

references begin to return us to the complications of static analysis that we encountered when discussing imperative languages.

Pattern matching plays an important part in modern functional languages. A naïve implementation of pattern matching that goes through the patterns case by case, as we described the process of pattern matching earlier, is needlessly slow. More sophisticated implementations (using, for example, decision trees) can do much better.

Polymorphism and Boxes

In order for a function to be parametrically polymorphic, it must be able to accept any argument, regardless of the argument's type. For this to work, every argument must be superficially similar. Polymorphism forces a single, standardized representation of all arguments. Frequently, this takes the form of a pointer to heap-allocated structures with a common layout. Even arguments that could be directly represented, such as integers or floating point numbers, end up being allocated on the heap in order to look like all the other arguments.

Such a common representation is called a BOXED REPRESENTATION because we can think of putting every data type into a box that makes them all look the same. To actually use the data, we must unbox it; when we are done, we must box it again. This boxing can be expensive. The way to lessen this expense is to work at relaxing the constraint that required boxing in the first place by allowing functions to deal with UNBOXED arguments. Such arguments are cheaper to allocate and cheaper to work with and can lead to significant gains in efficiency. Enabling manipulation and use of unboxed arguments, and introducing an analysis that can discover when it is possible to substitute unboxed data for boxed data, is an important optimization for languages with many polymorphic functions. Unboxed types can even be added directly to the language, which allows the programmer

to directly manipulate unboxed types when efficiency is of particular concern.

14.7 THE BACK END

The middle end used analysis to optimize the representation of the program in preparation for translation to run on a von Neumann machine. The back end is responsible for effecting this translation. This translation consists in bridging the functional model of computation and the imperative model of computation. Conceptually, this is done by simulating functional computation-by-reduction in an imperative, von Neumann setting.

This simulation is typically performed by an ABSTRACT MACHINE. There are two criteria by which an abstract machine for a functional language should be judged [108, p. 184]:

- How well it supports the functional language.
- How effectively it can be mapped onto a real machine.

The ultimate goal of the back end and of the abstract machine is code generation. This can take the form of either code native to the given platform, or it can take the form of very low-level C code. When C code is the target language, it is in order to use the C compiler as a portable assembler. While the native assembly language can vary significantly between different platforms, the C language does not. Generating C code makes it easier to port the compiler to a new platform, since almost every platform will already have a working C compiler. Code produced in this way cannot compete with directly generated native code, but such a comparison misses the point of using C as target language. This choice is motivated by a desire to have a working compiler on as many platforms as possible as soon as possible. Native

code generation can always be added later, but code generation via C allows the compiler to produce code for unanticipated platforms.

14.7.1 *Types of Abstract Machine*

A large variety of abstract machines has been proposed. We can roughly divide these machines into three kinds:

- stack machines
- fixed combinator machines
- graph reduction machines

Stack Machines

Stack machines, such as the Functional Abstract Machine (FAM) [28] and the SECD (stack, environment, code, dump) machine [70], work by compiling the low-level intermediate language into stack instructions. The instruction set is customized to the functional language. Use of stack instructions makes it simple to apply peephole optimization to refine the stack code produced.

Stack machines are characterized by their representation of the code as stack instructions. They might also use a variety of other stacks: environment, control flow, and data. The environment stack stores environments of bindings mapping names to values. The control flow stack keeps track of the order of operations and is used to resume evaluating an expression after a detour to evaluate one of its subexpressions. The data stack is where data structures are allocated and stored and provides a way to access these structures, as well. Some of these other components might also be implemented as stacks in other kinds of abstract machines.

Fixed Combinator Machines

Fixed combinator machines eliminate the need to maintain an environment by transforming the entire program into a fixed set of combinators applied to known arguments. The bindings that would have been provided by the environment are instead made explicit through function application. The chosen set of combinators varies from machine to machine; a frequent subset of the chosen combinators are the S, K and I combinators, which are defined in terms of the lambda calculus as

$$S = \lambda xyz.xz(yz)$$

$$K = \lambda xy.x$$

$$I = \lambda x.x$$

The elimination of the environment is elegant, as is the very small number of primitive routines (one for each of the chosen combinators), but the code size can grow tremendously, and, since each combinator only performs a minimal amount of work, it requires the evaluation of many small combinators to accomplish anything. Thus, fixed combinator machines have a high functional call overhead. Adding more combinators that do more work to the fixed set of combinators can ameliorate this somewhat, but as different sets of combinators are more appropriate for different programs, the problem cannot be eliminated.

Graph Reduction Machines

Graph reduction machines conceive of the program as a graph of function, argument, and application nodes. Reduction takes place in terms of the graph at application nodes; the result of evaluation replaces the application node. The final value of the program is obtained by reducing the graph to the root node.

Where stack machines used an environment, and fixed combinator machines used transformation into a fixed set combinators, graph reduction machines take a third route: they transform the entire program into a set of combinators by what is known as LAMBDA LIFTING [37, 43, 63]. These combinators are extracted from the program itself by introducing additional abstractions over the free variables of function definitions. As in the fixed combinator machines, an environment is unnecessary, since binding via lambda replaces binding via the environment. But the use of these custom combinators, known as SUPERCOMBINATORS [59], avoids the problems associated with the fixed combinator machines.

Graph reduction machines naturally implement a call-by-need reduction strategy. Substitution occurs by substituting a pointer to the same, shared node. Whenever that node is evaluated and updated, all pointers immediately have access to the updated value. Thus, every node is reduced at most once; this at-most-once property is known as FULL LAZINESS.

Unfortunately, building and maintaining the graph structure is expensive. Since reduction is modeled in terms of the graph, the program-as-graph ends up being interpreted at runtime, which also limits execution speed. Because of the problems posed by conventional architectures for these abstract machines, there were attempts to implement hardware that provided direct support for graph reduction (sometimes extending support to *parallel* graph reduction).

Compiled graph reduction machines work around this problem in order to achieve good performance on conventional architectures. They replace an explicit graph structure with code that acts as if the graph were present. They thus preserve the conceptual simplicity afforded by viewing the program as a graph while avoiding the expense of building and updating a graph data structure. A variety of refinements and variations on this theme are possible [26, 62, 79, 99, 108, 115].

14.7.2 *The Abstract Machine Design Space*

We have briefly described abstract machines in terms of several different ways they encode the functional program and evaluate it. Another way to describe an abstract machine is in terms of the decisions made in its design. This requires giving an example of the other choices that could have been made.

As you might have deduced from our earlier discussion, the two most important choices are the evaluation strategy implemented and the way the environment is handled. The major decision for evaluation strategy is between call-by-value and call-by-name. Call-by-need evaluation is merely a variation on call-by-name that implements laziness.

There are many variations on these principal strategies in terms of how precisely they are implemented. The primary variations concern how function application is performed. The two options go by the suggestive names of push/enter and eval/apply. In the push/enter approach, a function call `f x y` is evaluated by pushing `x` and `y` on the stack and entering the code for `f`. It is up to `f` itself to determine whether sufficient arguments are available; if insufficient arguments are provided, the function returns a closure; if sufficient are available, it completes the application and returns the result. In the eval/apply model, the caller determines whether sufficient arguments are available and controls the application process. Both approaches can be used with any evaluation strategy; eval/apply has historically been favored for call-by-value and push/enter for call-by-name, but research [78] suggests eval/apply is preferable in both cases. Surprisingly, this conclusion is reached, not on the basis of a difference in performance (the differences are negligible), but in a reduction in the complexity of the compiler that accompanies use of eval/apply.

Environment management can be done using explicit environments or via combinators. If explicit environments are used, there is a choice

of the type of environment: should definitions be shared or copied? That is to say, when entering an enclosing lexical scope, should the closure direct accesses to free variables to enclosing closures, creating something of a tree-like structure, or should each closure receive copies of its needed values? Sharing eliminates time spent copying but increases the time spent traversing data structures to reach the point of variable definition; copying requires time and space be spent making copies, but each function closure then retains only needed bindings (there is no reason to copy over unneeded ones) and access to those bindings is possible in constant time.

If copying is chosen to implement environments, there is one more question: when should the copy be performed? Different abstract machines have different answers to this question. Some possibilities are:

- when a function is entered;
- when a closure is built and again when it is entered; and
- only when a closure is built.

We will not discuss these further; the interested reader is referred to an article by Douence and Fradet [42] and associated technical reports [40, 41].

We mentioned that call-by-need can be seen as a variation on call-by-name. Call-by-name necessitates sharing and updating of closures. This updating could be performed in two ways, either by the caller or by the callee, but all implementations use callee-update since this prevents every caller of the same callee from having to test the callee to see whether it has been evaluated or not. Instead, the callee, if not previously evaluated, sparks its evaluation and updates itself with the result. The callee of course also returns the result to the caller that forced its evaluation.

Graph reduction machines can be seen as implementing these same strategies but in terms of graphs. The details of transformation between the two perspectives are beyond the scope of this discourse.

14.8 BIBLIOGRAPHIC NOTES

Garbage collection is an interesting topic in itself comprising a variety of algorithms with a variety of purposes and no apparent optimum approach. The state of the art as of 1995 is described in a textbook by Jones and Lins [64]. An alternative to garbage collection is region inference, which is a static analysis that enables the compiler to hardcode at compile-time the work usually performed by a dynamic garbage collector [126].

McCarthy [81] describes the creation of the first Lisp interpreter. The first ML interpreter was implemented in Lisp [see 47, footnote 5]. By the start of the 1990s, Lisp offered a sophisticated programming environment [72]. Cardelli [28] describes how an ML compiler was developed based on the "implementation folklore" of various Lisps rather than using the style advocated by compiler textbooks directed towards imperative languages. The incremental, on-the-fly compilation used by Lisp systems and some interactive compilers for functional languages is also known as JUST-IN-TIME COMPILATION and has its own interesting history [12]. Virtual machines are also an active topic of research in themselves [119], as is how they relate to functional languages and abstract machines [1, 2, 3, 4, 36].

A brief history of CPS is given by Flanagan, Sabry, Duba, and Felleisen [44]. While they suggest the argument over whether compiling primarily through CPS is worthwhile has been settled against CPS, Kennedy [66] at least believes CPS provides distinct benefits in simplicity compared to other, later intermediate representations. The best resource on CPS, at least as of the early 1990s, is *Compiling with Continuations* [10].

Speaking of intermediate representations, we should note that SSA can actually be seen as functional programming [10].

The suspensions created to represent "frozen" computations during the compilation of lazy languages are also known as THUNKS [60]. Thunks can be used to simulate call-by-name within a call-by-value evaluation strategy [50]. Strictness analysis, which can be used to avoid the creation of thunks, can be seen as a case of order of evaluation (or "path") analysis [21]. Strictness optimizations can cause surprising, unwelcome behavior [20].

Deforestation can be performed via so-called short cuts [46]. Opportunities for deforestation can be recognized through higher-order type inference [30]. The elimination of unnecessary construction of intermediate data structures is addressed more generally by stream fusion [33]. Update analysis is discussed by Bloss [22]. Another surprising source of optimizations for functional aggregates are loop optimization techniques developed for use in scientific computing [9].

Call-pattern specialization [99] can be used to reduce the cost of the pervasive use of algebraic data types and function definition through pattern matching. Efficient pattern-matching nevertheless requires some finesse [73]. Unboxed representations and their benefits are discussed by Peyton Jones and Launchbury [106], Thiemann [124].

We did not discuss the problem of space leaks [134] and its partial solution through black-holing [65] for graph reducers. Black-holing makes it impossible to back up in the event of an interrupt or other exception, requiring another solution [115] if we wish to support interrupts while avoiding the space leaks otherwise prevented by black-holing.

Douence and Fradet [42] describe an overarching framework for describing and decomposing abstract machines. Our summary of the abstract machine design space drew heavily on their work. Another view [3] of abstract machines makes a technical distinction between

abstract machines, which operate directly on lambda terms, and virtual machines, which operate on lambda terms compiled into their own instruction set. It is also possible to go between functional evaluators and abstract machines that implement the evaluation strategy via a state transition system [2].

15

CASE STUDY: THE GLASGOW HASKELL COMPILER

The discussion of the previous chapter was limited to generalities. We now look at a specific implementation of functional compilation. The Glasgow Haskell compiler is an actively developed, mature compiler for the lazy functional language Haskell. It implements numerous extensions to the standard language and provides a variety of additional tools and libraries, many of which are used in developing the compiler itself.

We gave a brief history of the Haskell language towards the end of Chapter 13, HISTORY. The Glasgow Haskell Compiler (GHC) is today the principal Haskell compiler. It is used both to produce compiled Haskell programs doing real work as well as for research into functional languages and their implementation. GHC is written primarily in Haskell itself, though some parts (including most of the runtime system) are implemented in C.

Compilers are very complex programs made up of a number of interacting, complex parts. We make no pretense of describing the Glasgow Haskell compiler *in toto*. Our study is guided by two questions:

- How does GHC use the fact it is compiling a functional language to its advantage?
- How does GHC solve the problems a functional language poses for compilation?

Answering these questions entails looking at specific optimizations enabled by functional languages and optimizations required to efficiently compile functional languages. All optimizations are carried out

using specific intermediate representations, so we will describe the intermediate representations used in these compilers. In the course of discussing solutions to problems introduced by functional languages, we will also briefly discuss GHC's implementation of garbage collection and pattern matching. We will also look more closely at how it transforms the source functional program into something executable in the target, von Neumann environment.

15.1 INTERMEDIATE REPRESENTATIONS

The Glasgow Haskell compiler uses several progressively simpler intermediate languages. We can roughly equate each language with a certain phase in compilation:

- The front end uses a representation of Haskell itself.
- The middle end uses a much simpler core language called, unsurprisingly, Core.
- The back end uses the STG and Cmm languages.

The first intermediate representation is produced and used by the front end. This representation is a representation of Haskell itself using data types and constructors. GHC takes the unusual step of performing type inference using what is fundamentally the source language rather than a desugared, simpler, core language. This makes it easy for the compiler to report errors in terms of the code provided by the programmer.

The Haskell representation is then desugared into a very simple core language called CORE. Core encodes values alongside their types, and so optimizations using the Core representation can take advantage of type information. This type information includes information on type equality constraints and coercions. Further details do not concern us,

but can be found in an article by Sulzmann, Chakravarty, Jones, and Donnelly [123].

Possibly one of the greatest advantages of carrying through types into the intermediate language, however, is not that they become available for optimization, but that they make it easy to catch errors introduced during development, since such an error is likely to introduce an erroneous term that can be caught by a simple type check. As all optimizations are performed as Core-to-Core transformations, and optimizations can interact in complex ways, error recognition by cheap type checking is very helpful.

The back end uses the STG (for Spineless Tagless G-machine) language. Core is transformed into STG through an intermediate step. The Core representation is first transformed into a Core representation of the program that is closer in spirit to STG. Only after this transformation is the Core representation transformed into an STG representation.

STG is the language of an abstract machine, the Spineless Tagless G-machine. This machine was designed for efficient translation into imperative code executed by a conventional, von Neumann computer. However, GHC does not translate STG code directly into native code. It instead translates it into Cmm. Cmm is GHC's implementation of C-- (read "C-minus-minus"; see Peyton Jones, Ramsey, and Reig [110] for a description), a language that closely resembles C but is somewhat simpler and lower-level.*

Once the program is represented in Cmm, it can be compiled to native code in two different ways: directly, or through C. (The choice is the user's, though the default is direct generation of native code from the Cmm representation.)†

* GHC neither uses nor requires all capabilities of C--, and so Cmm does not implement those unneeded capabilities. Other small differences combine to make Cmm a dialect of C--, which is itself a moving target. Current information on C-- is available from http://www.cminusminus.org/.

† Another transformation that would turn the generated Cmm into continuation passing style Cmm code is currently under development.

Direct code generation proceeds by transforming the Cmm representation into a data type representation of assembly instructions. Where the front end began by transforming the source code into a data type representation of Haskell code, the back end finishes by printing out a representation of the data type encoding of the assembly instructions. The resulting code can then be assembled into an object file.

Compilation by C is messier, less elegant, and appears to be deprecated. Since Cmm is virtually a subset of C, it is not difficult to generate something that can be compiled as a C program. This is compiled with any available C compiler. Since the program is not really a C program but a representation in C of a Spineless Tagless G-machine program, some of the assumptions made by the compiler are erroneous and result in suboptimal code. The assembly language code produced by the C compiler is thus postprocessed as a last optimization. The primary effects of this postprocessing are the removal of many unneeded register save and restore sequences and the rearrangement of the memory layout of the assembly code. This results in a corresponding rearrangement of the object file produced when the assembly code is assembled.

15.2 GARBAGE COLLECTION

GHC implements generational garbage collection. Generational garbage collection is based on the assumption that "young" objects – those that have been recently allocated – are more likely to have died than older objects. Generational garbage collectors thus focus their garbage collecting efforts on younger objects. The age of an object is described in terms of generations. The garbage collector assigns all allocated objects to one of a set of generations. A newly created object belongs to the first generation. During collection, pointers in objects of those older generations that are not being collected are used as roots to determine which objects of the generations undergoing collection are live. Objects

that survive a certain number of collections are promoted to the next generation.

Garbage collection is invoked frequently to keep heap use under control; since most garbage collections only examine younger generations, such minor collections are inexpensive. When a minor collection will not suffice to reclaim sufficient memory, a major collection is performed using a mark-compact algorithm: this leads to all generations being examined, so that storage allocated to older objects that, by virtue of their age, had survived collection past the end of their lives is reclaimed.

GHC's garbage collector is of the so-called "stop the world" variety. During garbage collection, only the garbage collector is active. All computation ceases. This plays a surprisingly important role in ensuring that GHC interfaces well with the outside library (GMP, the GNU Multiple-Precision Library) that it uses to provide arbitrary-precision arithmetic. The arithmetic library is implemented in C; if garbage collection occurred while a library function was being executed, garbage collection could relocate the data the function was was working with out from under the function's pointer to that data. However, because the garbage collector requires that the world be stopped, it is only invoked when all running threads have reached a sequence point; since none of the functions provided by this library have such a stopping point, they cannot be interrupted by the garbage collector.

15.3 PATTERN MATCHING

Pattern matching is in fact a core part of the Core and STG representations. Even conditional expressions are handled through pattern matching: `if B then X else Y` is written using a case expression as `case B of {True -> X; False -> Y}`. `B` is called the expression scrutinized

by the case expression. In the Core and STG languages, case analysis forces evaluation of the expression scrutinized by the case analysis.

Expression evaluation is a necessary part of pattern matching in these intermediate languages. To explain why, we must describe how data type declarations are treated. As discussed earlier, a declaration such as `data Tree a = Leaf a | Branch (Tree a)(Tree a)` creates an algebraic data type with two constructors, `Leaf` and `Branch`. The ordering of these declarations is considered significant in Core and STG, because each constructor is assigned a discriminator integer, beginning with 0 and increasing by one for each constructor declaration. Thus, `Leaf` would be assigned 0 and `Branch` would be assigned 1.

Pattern matching uses this discriminator to decide which branch of the case expression should be chosen. After evaluation of the scrutinized expression completes, its discriminator is examined; a jump is then made directly to the corresponding branch of the case expression. The simplicity of this system is due to the Spineless Tagless G-machine's inbuilt support for algebraic data types.

15.4 OPTIMIZATIONS

All optimizations are performed as Core-to-Core transformations [109]. Some optimizations, such as strictness analysis and let-floating, require significant nonlocal analysis. Some can be done in the course of several local simplification passes.

Some of these local simplifications are specified using a rewrite rule syntax [105] that is available to all users of the Glasgow Haskell compiler. Others are more complex, such as function inlining. Inlining basically treats a function call as a macro; it replaces the call with an instance of the body of the function. (There are, as always, some complexities [103].) Inlining a function eliminates the need to perform a pipeline-unfriendly jump to the code for the function during evalua-

tion. The Glasgow Haskell compiler uses heuristics to determine when a function should be inlined; it will also inline a function when the programmer has specified through a source-code annotation that the function should be inlined.

Strictness analysis [107] attempts to discern which expressions will perforce be evaluated. The code can then be optimized to avoid the costs of unnecessary laziness: no suspension need be created in the first place for the expression, and so the expression will not need to be forced and updated later. Thus, strictness analysis can be used to avoid a fair amount of work.

The let-floating transformations are another class of non-local transformations. Let floating describes the effect of the transformation: a let or letrec binding is shifted from one place in the source code to another. Shifting here should be understood in terms of depth within an expression: within a lambda abstraction, the body of a branch of a case expression, or the expression scrutinized by a case expression. There are advantages and disadvantages to floating let-bindings both in and out, as well as some local transformations that are generally helpful; the specific application of let-floating to a given case is decided, as is usual in optimizations, through some heuristic rules. Further information on let-floating can be found in the article by Peyton Jones, Partain, and Santos [104].

Of all these optimizations, strictness analysis is the only one we can plainly categorize as an example of an optimization necessitated by the inherent inefficiency of a lazy language. The rest of the optimizations are little different from the code tuning transformations an imperative compiler might perform using static single assignment form;* indeed, the imperative optimization stalwart, common subexpression elimination, can also be applied to a Core program.

* This analogy is more accurate than you might at first think; SSA can actually be looked at as transforming an imperative program into an equivalent functional program to ease analysis [11].

15.5 GOING VON NEUMANN

The transformation that reroots the functional program in the imperative, von Neumann paradigm is the transformation from the program's STG representation to its Cmm representation. Of course, by the time the program has been transformed into the STG language, a significant amount of analysis and optimization has been performed with the aim of producing code that is more efficient (both in space and time) within the von Neumann setting. Since all the "heavy lifting" has been done using other, more complex representations, the actual translation from STG to Cmm is fairly direct. That is not to say that it is simple; there are many details concerning the precise memory layout of a closure, the accommodation of unboxed types, and heap and control flow management.

While we can identify the transformation from STG into Cmm as the moment that the functional program becomes a viable imperative program, this single aim influences the entirety of the compiler's design. There are many other compilers for many other functional languages, all complex and all implementing their own approach to functional compilation, but with this brief survey of the Glasgow Haskell compiler, we bring our discussion of functional compilation to a close.

15.6 BIBLIOGRAPHIC NOTES

Hudak et al. [58, §9] places the Glasgow Haskell compiler in the context of other implementations of the Haskell language. The version of GHC considered here is version 6.8.2. More information on GHC is available from its website, `http://haskell.org/ghc/`. The GHC Commentary, written by the developers for other developers and anyone else interested in the compiler's internals, is available at `http://hackage.haskell.org/trac/ghc/wiki/Commentary`; it was very helpful in prepar-

ing this chapter. The source code itself is also well-commented: if you should wish to explore functional compilation in more depth by reviewing the code for a compiler, you could scarcely hope for a compiler with better documentation.

The Spineless Tagless G-machine [101, 108] refines the Spineless G-machine of Hammond [49], which itself is a refinement of Johnsson's G-machine [62]. Around 2007, reconsideration of the *tagless* part of the Spineless Tagless G-machine led to the introduction of tags indicating whether or not a closure has been evaluated and, if so, the discriminator of its data constructor in order to reduce branch mispredictions encountered during case analysis and cache pollution caused by unnecessarily loading the info table of the closure. This work is described by Marlow et al. [79].

16

CONCLUSION

This chapter focused on the functional language family. After we introduced the theory at its roots, we sketched the history of the functional family, from early predecessors such as LISP and influential non-functional languages such as APL to today's mature functional languages. Many approaches to compiling functional languages have been advanced, and we discussed some of them and provided a somewhat more in-depth case study of an actual compiler.

- In Chapter 12, THEORY, we revisited the concept of *type* and explored some of its complexities. We then developed the lambda calculus and introduced constants and types into its framework. When we discovered this prevented us from employing recursive definitions, we introduced a family of typed fixed point operators.

- In Chapter 13, HISTORY, we looked at the history of the functional family through the lens of its influential languages. Among the predecessors of today's functional languages, we discussed McCarthy's LISP, Landin's Iswim, Iverson's APL and Backus's FP. We then turned to modern functional languages. After describing common defining features, we looked at the two primary branches of the functional family, the eager and the lazy languages. Eager languages, such as ML, use what amounts to call-by-value as their reduction strategy. Lazy languages, such as those created by Turner and their successor of sorts, Haskell, implement what amounts call-by-name.

- In Chapter 14, COMPILING, we described in broad terms how functional languages are compiled. Functional languages are primarily a "sugaring" of the lambda calculus, and by first desugaring them, we reduce them to a simple, core language that is only slightly more abstract than the lambda calculus itself.The core language representation is then compiled into instructions for an abstract machine of some variety, and it is this abstract machine that runs the program: the compiled representation, in some sense, encodes both the program and a virtual machine to run the program.

- In Chapter 15, CASE STUDY: THE GLASGOW HASKELL COMPILER, we looked at how the Glasgow Haskell compiler actually compiles a functional language. This case study provided concrete examples in contrast to the generalities of our discussion in Chapter 14, COMPILING.

EPILOGUE

The body of this work is through. This epilogue reflects on what you have just read about the functional and imperative families and offers some thoughts on future developments.

LOOKING BACK

Imperative and Functional Languages

The models of computation that underlie both the imperative and functional families were developed around the same time, but the development of the von Neumann machine set imperative languages on the path to ascendancy.

At first, these machines could be programmed only by manipulation of their hardware. The development of software brought assembly language, the prototypical imperative language, to the fore. Even today, assembly language cannot be beat for the control over the underlying machine it brings, but this control comes at great cost: in programming time, and in portability. It takes a long time to write a substantial amount of assembly code, and the code is then tied to the platform it was written for.

In the 1950s, Fortran brought imperative languages to a higher level of abstraction; later imperative languages brought more powerful abstractions. Still, early Fortran remains, though primitive, recognizably imperative. Within a decade, Lisp would be born. Lisp was an important predecessor for today's functional languages: Lisp made higher-order functions available, and so one faces similar problems compiling

Lisp as compiling today's functional languages. But Lisp started as Lisp and continues as Lisp: there is no mistaking Lisp code for anything but Lisp code, and Lisp style is quite distinct from the style of modern functional programming.

It was another decade before ML made its debut in the 1970s. It started as an interpreted language without the concept of algebraic data types, which was borrowed later from another language. The lazy branch would not begin to bear fruit until the 1980s. Over the next two decades, the functional language family would grow into its modern form.

In order to have any hope of displacing assembly as the dominant programming language, Fortran had to be fast, and it was designed from the outset with speed in mind. Lisp grew up with artificial intelligence and was adopted because it was very well-suited to programming in that domain. It competed on features and the power of its abstractions, not on speed. It pioneered garbage collection, but it took decades of research to get past the "stop the world" effect that scanning the heap and scavenging useful data can cause if done without sufficient sophistication. Since many application domains for programming languages demand speed, Lisp was only ever a marginal language outside symbolic processing. The imperative family would continue to look on garbage collection as an expensive and unneeded luxury until languages developed for object-oriented programming showed that it can bring new levels of programmer productivity.

ML grew out of work on a theorem prover, and it too was developed (using Lisp, no less) to serve its application domain. Its type system could provide guarantees for theorem proving that a weaker system could not. Significant work was required to make both Lisp and ML run decently fast on "stock hardware." Partly for this reason, many persons researched alternative computing architectures meant to support such languages directly, just as the von Neumann architec-

ture naturally supports programs written in imperative languages. But stock hardware eventually won out, and it was only in the 1990s that optimizations were discovered to make lazy functional languages at all competitive with compiled imperative languages on stock hardware.

The bottom line, for all programming languages, is the machine they must eventually run on. This has been a blessing for imperative languages (at least when uniprocessors were the standard) and a curse for functional. Functional languages also suffer from requiring of the programmer a fundamentally different style of programming than other kinds of languages.

Backus's criticisms of imperative languages, leveled during his Turing award lecture, continue to be valid. Imperative programming is still not high-level enough – to use Backus's phrase, it still amounts to "word-at-a-time programming" – and many of the dominant imperative languages continue to require programmers to supply redundant type annotations. This is ameliorated to some degree by the rise of dynamically-typed scripting languages, some of which (Groovy, Beanshell, Pnuts, Jython, and JRuby, for example) are implemented on top of the very platform designed to host the more heavy-weight Java, but dynamic typing gives up the benefits of static typing offered by functional languages.

Functional programming languages stand in stark contrast to imperative languages. The contrast might be too severe: their strangeness might put off more programmers than it attracts. It requires a significant investment of time and effort to transition from imperative to functional programming, especially since many of the techniques learnt in an imperative setting cannot be transferred directly to the functional, including even common data structures.

Today's functional programming languages have finally begun to overcome the slowness inherent in simulating β-reduction on a von Neumann machine, but Backus's primary criticism of them was not based

on their being slow, but on their not being "history sensitive." They seem to have no way to store information from one run of the program to the next; the lack of state cripples their usefulness. Backus rightly pointed out that this has been a major source of trouble; he gave the example of pure Lisp becoming wrapped in layers upon layers of von Neumann complexity.

Today's functional languages have tried to solve the problem of state either by limiting its use and making it explicit, in ML via reference cells and in lazy languages through either monads or streams. The ML concept of reference cells is virtually identical to the concept of a variable in imperative languages. We will not say anything of monads here. Streams can be used in lazy languages to represent interaction with the world outside: the program is provided with an infinite stream of responses as input and produces an infinite stream of requests. Since the language is lazy, it is possible to avoid evaluating any input responses until a request has been made, such as to open a file. This often leads to "double-barreled" continuation passing style, where one barrel is used if the request succeeds and the other is used if the request garners an error response. These solutions avoid layers of von Neumann complexity, but at the cost of a different variety of obtuseness.

Imperative and Functional Compilers

Imperative and functional compilers have no trouble with lexing and parsing. It's in the middle and back ends that problems present themselves. Here, they must go head to head with the problems inherent in their languages.

Imperative languages make extensive use of pointers and other aliases and frequently reassign names (variables) to different values. This complicates analysis of data flow in the program and limits the optimizations that can be performed. Imperative compilers have historically

Listing 16.1: A tail-recursive factorial function

```
fac n = fac' n 1
    where fac' 1 accu = accu
          fac' n accu = fac' (n - 1) (n*accu)
```

had difficulty handling recursion, as it is difficult to tell when a given recursive call can reuse the same stack space rather than requiring allocation of a new stack frame. Recursive calls that pass all needed information to the function itself for the next call do not require allocation of a new stack frame, as the return value will have been computed by the time the recursion bottoms out. This kind of recursion is known as TAIL RECURSION. Listing 16.1 on page 231 gives a common example of this: a version of the factorial function that makes use of an accumulator argument to store the in-progress computation of the final value. The function `fac` serves to hide this accumulator function from the user; it simply calls the actual worker function with its argument and the initial value of the accumulator.

Imperative languages also have difficulty dealing with concurrency and parallelism. It is here that the von Neumann bottleneck becomes most apparent. The reliance programs written in imperative languages have on constant access to a common store leads, in a concurrent setting, to problems with too many threads needing access to the same part of the store. Locking mechanisms can keep this model workable, but they require a significant amount of trouble on the programmer's part.

Functional languages have their own problems. The most obvious ones boil down to the mismatch between their computational model and that of the machine their programs must run on. It is hard to carry out reduction efficiently. The necessity of closures leads to a significant amount of overhead for running programs and a significant amount of added complexity in the implementation of compilers for functional languages.

Functional programmers' extensive use of lists and similar data structures can also lead to insufferably slow code without optimization. Either the programmer must be very careful and aware of the code that will be produced by the compiler for a given function, or the compiler must perform clever optimizations. Lazy languages only complicate this with their unpredictable evaluation order. Slight differences in the way a function is written can lead to completely different time and space complexities. Lazy languages also require the development of strictness analyses and associated optimizations.

Purity can in fact be considered a burden from the compiler's point of view. Referential transparency simplifies analysis and transformation, but it also necessitates a new class of optimizations. Update analysis attempts to discover when a given reference will never be used and reuse its associated data structure. Operations must be implemented such that the greatest amount of each data structure possible is shared and reused, lest space be wasted. For lazy languages, a new class of problems, space leaks, rears its ugly head, surprising programmers and leading to *ad hoc* analyses and optimizations meant to squash some of the most egregious examples.

Needless to say, work on compiling both functional and imperative languages continues.

LOOKING FORWARD

Functional languages are growing up. They are beginning to see increasing use in industry and increasing interest among programmers. They also hold out promise as a way to deal with the rise of ubiquitous symmetric multiprocessors, which brings the problems of concurrent programming out of scientific and network programming and into programming in general.

Functional languages also continue to influence imperative languages. Java brought garbage collection into the mainstream. Several imperative languages, including Microsoft's C#, now allow anonymous functions; the programming language Python borrowed list comprehensions from Haskell; the spirit of declarative programming, if not explicitly functional programming, shows through in the language-integrated query (LINQ) facilities added to the .NET platform. Functional languages have been implemented for both the Java virtual machine (Scala) and Microsoft's .NET platform (F#, developed and promoted by Microsoft itself).

Functional programming, and declarative programming in general, appears to promise increased programmer productivity. As programming time continues to become the limiting factor in what is doable in software, this could lead to increasing adoption of functional languages. At the same time, imperative languages have begun to assume more elements of functional programming. It is possible that something like Objective Caml or Microsoft's F# will become the dominant programming language in the next couple of decades.

These families are merging in some ways, but they also continue to develop in their own peculiar ways. Object-oriented programming has led to aspect-oriented programming; there is continuing research in the functional programming community on more powerful type systems, including those embracing dependent types and observational type theory. There are also attempts to extend functional languages in the direction of logic languages.

The families of programming languages continue to diversify, branch out, and join together. Their fortunes also change as new languages grow in popularity and old ones fall out. Old languages are sometimes made new again through a revised definition or new extensions that breathe life back into them. These are exciting times.

SUGGESTIONS FOR FURTHER RESEARCH

This thesis touched on a wide variety of subjects, from computer architecture to formal language theory and automata, parsing, optimizations, imperative and functional languages, the lambda calculus and type theory. Each of these subjects has already had much said about it. References to related work have been given in the bibliographic notes at the end of each chapter.

If paradigms have captured your fancy, you might want to investigate a paradigm we did not have time to explore, logic programming. This paradigm is exemplified by the language Prolog. Work is ongoing to make constraint programming, an offshoot of logic programming, a viable paradigm. We mentioned attempts to blend functional and imperative features in a single language; there have also been attempts to create so-called functional logic languages, such as Curry.

Work to date on virtual machines for functional languages has approached them from a formal point of view. There is a significant body of literature treating virtual machines in themselves that explores optimizing and improving them. Applying this literature to the virtual machines used with functional languages could perhaps yield interesting results.

BIBLIOGRAPHIC NOTES

Algebraic data types were borrowed by ML from Burstall's Hope [27]. We referred frequently to Backus's influential 1978 Turing award lecture [13].

Functional languages have seen significant use in industry. Development of Objective Caml is funded in part by a consortium including Intel, Microsoft, Jane Street Capital [91], and LexiFi; the last two companies are involved in trading and finance. Hudak et al. [58] give a

list of companies using Haskell along with descriptions of how they use the language in addition to examples of the language's impact in higher education. Wadler* maintains an extensive list of applications of functional programming. Appel† keeps up a smaller list of implementation work done using ML. Wadler [135] provides an insightful analysis of why functional languages are not used more.

Observational type theory [8] is an interesting and powerful idea, while dependent types are powerful enough to express a variety of concepts that must otherwise be built into a language or done without [7, 84]. Meijer has worked to introduce concepts from functional programming into the imperative programming world, and he provides an excellent overview [86] of this work.

* "Functional Programming in the Real World," http://homepages.inf.ed.ac.uk/wadler/realworld/.
† "Implementation work using ML," http://www.cs.princeton.edu/~appel/smlnj/projects.html

BIBLIOGRAPHY

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. Technical Report BRICS RS-03-13, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Mar 2003. (Cited on page 212.)

[2] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *PPDP '03: Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19, New York, NY, USA, 2003. ACM. ISBN 1-58113-705-2. doi: http://doi.acm.org/10.1145/888251.888254. (Cited on pages 212 and 214.)

[3] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: A functional derivation. Technical Report BRICS RS-03-14, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, Mar 2003. (Cited on pages 212 and 213.)

[4] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between call-by-need evaluators and lazy abstract machines. Technical Report BRICS RS-04-3, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, February 2004. (Cited on page 212.)

[5] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Mass., USA, 1977. ISBN 0-201-00022-9. Known as the "green dragon book". (Cited on page 90.)

[6] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, and Monica S. Lam. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Higher Education, second edition, August 2006. ISBN 978-0-321-48681-1. Known as the "purple dragon book". (Cited on pages 90 and 91.)

[7] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. Manuscript, available online, April 2005. (Cited on page 235.)

[8] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-677-6. doi: http://doi.acm.org/10.1145/1292597.1292608. (Cited on page 235.)

[9] Steven Anderson and Paul Hudak. Compilation of haskell array comprehensions for scientific computing. *SIGPLAN Not.*, 25(6): 137–149, 1990. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/93548.93561. (Cited on page 213.)

[10] Andrew W. Appel. *Compiling with continuations*. Cambridge University Press, New York, NY, USA, 1992. ISBN 0-521-41695-7. (Cited on pages 212 and 213.)

[11] Andrew W. Appel. Ssa is functional programming. *SIGPLAN Not.*, 33(4):17–20, 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/278283.278285. (Cited on page 221.)

[12] John Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/857076.857077. (Cited on page 212.)

[13] John Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Com-*

mun. ACM, 21(8):613–641, 1978. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/359576.359579. (Cited on pages 193 and 234.)

[14] Utpal Bannerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, Mass., USA, 1988. (Cited on page 126.)

[15] Utpal Bannerjee. *Loop Transformations for Restructuring Compilers*. Kluwer Academic Publishers, Boston, Mass., USA, 1993. (Cited on page 126.)

[16] Utpal Bannerjee. *Loop Parallelization*. Kluwer Academic Publishers, Boston, Mass., USA, 1994. (Cited on page 126.)

[17] H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, revised edition, 1984. (Cited on pages 153 and 170.)

[18] H. P. Barendregt. The impact of the lambda calculus in logic and computer science. *Bull. Symbolic Logic*, 3(2):181–215, June 1997. URL http://www.cs.ru.nl/~henk/papers.html. (Cited on page 169.)

[19] Henk Barendregt and Kees Hemerik. Types in lambda calculi and programming languages. In *ESOP '90: Proceedings of the 3rd European Symposium on Programming*, pages 1–35, London, UK, 1990. Springer-Verlag. ISBN 3-540-52592-0. (Cited on pages 168 and 170.)

[20] Marcel Beemster. Strictness optimization for graph reduction machines (why id might not be strict). *ACM Trans. Program. Lang. Syst.*, 16(5):1449–1466, 1994. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/186025.186040. (Cited on page 213.)

[21] Adrienne Bloss. Path analysis and the optimization of nonstrict functional languages. *ACM Trans. Program. Lang. Syst.*, 16(3):328–369, 1994. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/177492.177497. (Cited on page 213.)

[22] Adrienne Bloss. Update analysis and the efficient implementation of functional aggregates. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 26–38, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: http://doi.acm.org/10.1145/99370.99373. (Cited on page 213.)

[23] Martin Bravenboer and Eelco Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 365–383, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-9. doi: http://doi.acm.org/10.1145/1028976.1029007. (Cited on page 91.)

[24] Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for aspectJ. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 209–228, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: http://doi.acm.org/10.1145/1167473.1167491. (Cited on page 92.)

[25] Martin Bravenboer, Eelco Dolstra, and Eelco Visser. Preventing injection attacks with syntax embeddings. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 3–12, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-855-8. doi: http://doi.acm.org/10.1145/1289971.1289975.

[26] G. L. Burn, S. L. Peyton Jones, and J. D. Robson. The spineless g-machine. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 244–258, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62717. (Cited on page 209.)

[27] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. In *LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143, New York, NY, USA, 1980. ACM. doi: http://doi.acm.org/10.1145/800087.802799. (Cited on page 234.)

[28] Luca Cardelli. Compiling a functional language. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 208–217, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: http://doi.acm.org/10.1145/800055.802037. (Cited on pages 207 and 212.)

[29] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/6041.6042. (Cited on pages 139 and 170.)

[30] Olaf Chitil. Type inference builds a short cut to deforestation. *SIGPLAN Not.*, 34(9):249–260, 1999. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/317765.317907. (Cited on page 213.)

[31] Keith D. Cooper and Linda Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers, San Francisco, 2004. (Cited on pages 90, 112, and 126.)

[32] Guy Cousineau. A brief history of caml (as i remember it). Manuscript, available online, 1996. URL http://www.pps.jussieu.fr/~cousinea/Caml/caml_history.html. (Cited on page 193.)

[33] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. *SIGPLAN Not.*, 42 (9):315–326, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1291220.1291199. (Cited on page 213.)

[34] Curry. Modified basic functionality in combinatory logic. *Dialectica*, 23(2):83–92, June 1969. doi: 10.1111/j.1746-8361.1969.tb01183.x. (Cited on page 193.)

[35] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM. ISBN 0-89791-065-6. doi: http://doi.acm.org/10.1145/582153.582176. (Cited on page 193.)

[36] Olivier Danvy. A journey from interpreters to compilers and virtual machines. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering*, pages 117–117, New York, NY, USA, 2003. Springer-Verlag New York, Inc. ISBN 3-540-20102-5. (Cited on page 212.)

[37] Olivier Danvy and Ulrik Pagh Schultz. Lambda-lifting in quadratic time. In *FLOPS '02: Proceedings of the 6th International Symposium on Functional and Logic Programming*, pages 134–151, London, UK, 2002. Springer-Verlag. ISBN 3-540-44233-2. (Cited on page 209.)

[38] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church–Rosser theorem. *Indagationes Mathematicæ*, 34 (5):381–392, 1972. (Cited on pages 149 and 169.)

[39] Erik D. Demaine. Cache-oblivious algorithms and data structures. In *Lecture Notes from the EEF Summer School on Massive*

Data Sets, Lecture Notes in Computer Science, BRICS, University of Aarhus, Denmark, June 27–July 1 2002. Springer-Verlag. URL http://erikdemaine.org/papers/BRICS2002/. To appear. (Cited on page 32.)

[40] Rémi Douence and Pascal Fradet. A taxonomy of functional language implementations. Part I: Call-by-value. Research Report 2783, INRIA, Rennes, France, 1995. (Cited on page 211.)

[41] Rémi Douence and Pascal Fradet. A taxonomy of functional language implementations. Part II: Call-by-name, call-by-need, and graph reduction. Research Report 3050, INRIA, Rennes, France, 1995. (Cited on page 211.)

[42] Rémi Douence and Pascal Fradet. A systematic study of functional language implementations. *ACM Trans. Program. Lang. Syst.*, 20(2):344–387, 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/276393.276397. (Cited on pages 211 and 213.)

[43] Adam Fischbach and John Hannan. Specification and correctness of lambda lifting. *J. Funct. Program.*, 13(3):509–543, 2003. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796802004604. (Cited on page 209.)

[44] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 39(4):502–514, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/989393.989443. Includes a retrospective on the article, which was originally presented at PLDI 1993. (Cited on page 212.)

[45] Emden R. Gansner and John H. Reppy. *The Standard ML Basis Library*. Cambridge University Press, 2002. ISBN 978-0521791427. (Cited on page 193.)

[46] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: http://doi.acm.org/10.1145/165180.165214. (Cited on page 213.)

[47] Michael J. C. Gordon. From LCF to HOL: A short history. In G. Plotkin, Colin P. Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing, chapter 6, pages 169–186. MIT Press, 2000. ISBN 978-0262161886. URL http://www.cl.cam.ac.uk/~mjcg/papers/HolHistory.html. A version is available from the author at http://www.cl.cam.ac.uk/~mjcg/papers/HolHistory.html. (Cited on pages 193 and 212.)

[48] Dick Grune and Ceriel J.H̃. Jacobs. *Parsing Techniques: A Practical Guide*. Springer-Verlag, London, UK, second edition, 2007. ISBN 978-0-387-20248-8. Complete first edition available online at http://www.cs.vu.nl/$\sim$dick/PTAPG.html. (Cited on page 91.)

[49] K. Hammond. The Spineless Tagless G-machine — not, 1993. URL citeseer.ist.psu.edu/hammond93spineless.html. (Cited on page 223.)

[50] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *J. Funct. Program.*, 7(3):303–319, 1997. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796897002748. (Cited on page 213.)

[51] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, fourth edition, Sep 2006. (Cited on page 32.)

[52] Rolf Herken, editor. *The Universal Turing Machine: A Half-Century Survey*. Springer-Verlag, London, UK, second edition, May 1995. (Cited on page 31.)

[53] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 149:29–60, Dec. 1969. ISSN 0002-9947. URL http://www.jstor.org/stable/1995158. (Cited on page 193.)

[54] Stephen Hines, Prasad Kulkarni, David Whalley, and Jack Davidson. Using de-optimization to re-optimize code. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 114–123, New York, NY, USA, 2005. ACM. ISBN 1-59593-091-4. doi: http://doi.acm.org/10.1145/1086228.1086251. (Cited on page 91.)

[55] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Higher Education, third edition, July 2007. ISBN 978-0-23-146225-1. URL http://infolab.stanford.edu/~ullman/ialc.html. (Cited on page 90.)

[56] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4. doi: http://doi.acm.org/10.1145/1028664.1028717. URL http://findbugs.sourceforge.net/docs/oopsla2004.pdf. *See http://findbugs.sourceforge.net/.* (Cited on page 126.)

[57] Paul Hudak. Conception, evolution, and application of functional programming languages. *ACM Comput. Surv.*, 21(3):359–411, 1989. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/72551.72554. (Cited on pages 162, 170, and 192.)

[58] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, 2007. ACM. ISBN 978-1-59593-766-X. doi: http://doi.acm.org/10.1145/1238844.1238856. (Cited on pages 194, 222, and 234.)

[59] R. J. M. Hughes. Super-combinators: A new implementation method for applicative languages. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 1–10, New York, NY, USA, 1982. ACM. ISBN 0-89791-082-6. doi: http://doi.acm.org/10.1145/800068.802129. (Cited on page 209.)

[60] P. Z. Ingerman. Thunks: a way of compiling procedure statements with some comments on procedure declarations. *Commun. ACM*, 4(1):55–58, 1961. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/366062.366084. (Cited on page 213.)

[61] Kenneth E. Iverson. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. ISBN 0-471430-14-5. (Cited on page 193.)

[62] Thomas Johnsson. Efficient compilation of lazy evaluation. *SIGPLAN Not.*, 39(4):125–138, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/989393.989409. (Cited on pages 209 and 223.)

[63] Thomas Johnsson. Lambda lifting: transforming programs to recursive equations. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 190–203, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4. doi: http://dx.doi.org/10.1007/3-540-15975-4. (Cited on page 209.)

[64] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996. ISBN 978-0-471-94148-4. (Cited on page 212.)

[65] Richard E. Jones. Tail recursion without space leaks. *Journal of Functional Programming*, 2(1):73–79, January 1992. (Cited on page 213.)

[66] Andrew Kennedy. Compiling with continuations, continued. *SIGPLAN Not.*, 42(9):177–190, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1291220.1291179. (Cited on page 212.)

[67] George Kuan and David MacQueen. Efficient type inference using ranked type variables. In *ML '07: Proceedings of the 2007 workshop on Workshop on ML*, pages 3–14, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-676-9. doi: http://doi.acm.org/10.1145/1292535.1292538. (Cited on page 193.)

[68] Prasad A. Kulkarni, Stephen R. Hines, David B. Whalley, Jason D. Hiser, Jack W. Davidson, and Douglas L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim.*, 2(2):165–198, 2005. ISSN 1544-3566. doi: http://doi.acm.org/10.1145/1071604.1071607. (Cited on page 91.)

[69] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/365230.365257. (Cited on page 193.)

[70] Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, April 1964. ISSN 0010-4620. (Cited on page 207.)

[71] Chris Lattner. LLVM: An Infrastructure for Multi-Stage Optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, Ill., USA, Dec 2002. *See* `http://llvm.cs.uiuc.edu`. (Cited on page 126.)

[72] D. Kevin Layer and Chris Richardson. Lisp systems in the 1990s. *Commun. ACM*, 34(9):48–57, 1991. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/114669.114674. (Cited on pages 192 and 212.)

[73] Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. *SIGPLAN Not.*, 36(10):26–37, 2001. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/507546.507641. (Cited on page 213.)

[74] David MacQueen. Modules for standard ml. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 198–207, 1984. ISBN 0-89791-142-3. (Cited on page 193.)

[75] David B. MacQueen. Structure and parameterization in a typed functional language. In *Symp. on Functional Languages and Computer Architecture*, pages 525–537, Gothenburg, Sweden, June 1981. (Cited on page 193.)

[76] David B. MacQueen. Using dependent types to express modular structure. In *Proceedings of the 13th annual ACM symposium on Principles of programming languages*, pages 277–286. ACM Press, 1986. doi: http://doi.acm.org/10.1145/512644.512670. (Cited on page 193.)

[77] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 409–423, London, UK, 1994. Springer-Verlag. ISBN 3-540-57880-3. (Cited on page 193.)

[78] Simon Marlow and Simon Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 4–15, New York, NY, USA, 2004.

ACM. ISBN 1-58113-905-5. doi: http://doi.acm.org/10.1145/1016850.1016856. (Cited on page 210.)

[79] Simon Marlow, Alexey Rodriguez Yakushev, and Simon Peyton Jones. Faster laziness using dynamic pointer tagging. *SIGPLAN Not.*, 42(9):277–288, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1291220.1291194. (Cited on pages 209 and 223.)

[80] Conor McBride and James McKinna. Functional pearl: I am not a number – I am a free variable. In *Haskell '04: Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017477. (Cited on page 169.)

[81] John McCarthy. History of LISP. *SIGPLAN Not.*, 13(8):217–223, 1978. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/960118.808387. (Cited on pages 192 and 212.)

[82] John McCarthy. Lisp - notes on its past and future. In *LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming*, pages .5–viii, New York, NY, USA, 1980. ACM. doi: http://doi.acm.org/10.1145/800087.802782. (Cited on page 192.)

[83] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/367177.367199. (Cited on pages 172 and 192.)

[84] James McKinna. Why dependent types matter. *SIGPLAN Not.*, 41 (1):1–1, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1111320.1111038. (Cited on page 235.)

[85] Scott McPeak. Elkhound: A fast, practical GLR parser generator. Technical Report UCB/CSD–2–1214, University of California, Berkeley, Berkeley, California, USA, December 2002. URL

http://www.cs.berkeley.edu/~smcpeak/elkhound/. (Cited on page 91.)

[86] Erik Meijer. Confessions of a used programming language salesman. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 677–694, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297078. (Cited on page 235.)

[87] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978. (Cited on page 193.)

[88] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, Nov 1990. ISBN 978-0-262-63137-2. (Cited on page 193.)

[89] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, May 1997. ISBN 978-0-262-63181-5. (Cited on page 193.)

[90] Robiner Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Feb 1990. ISBN 978-0-262-63132-7. (Cited on page 193.)

[91] Yaron M. Minsky. Caml trading. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–285, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: http://doi.acm.org/10.1145/1328438.1328441. (Cited on page 234.)

[92] R. P. Mody. Functional programming is not self-modifying code. *SIGPLAN Not.*, 27(11):13–14, 1992. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/141018.141021. (Cited on page 175.)

[93] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Academic Press, 1997. ISBN 978-1-55860-320-4. (Cited on pages 90 and 126.)

[94] Adam Olszewski, Jan Wole'nski, and Robert Janusz, editors. *Church's Thesis after 70 Years*, volume 1 of *Mathematical Logic*. ontos verlag, 2006. ISBN 3-938793-09-0. URL http://www.ontos-verlag.de/Buchreihen-ML-Olszewski.php. (Cited on page 170.)

[95] Jukka Paakki. Attribute grammar paradigms—a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255, 1995. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/210376.197409. (Cited on page 92.)

[96] David Pager. The lane tracing algorithm for constructing lr(k) parsers. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 172–181, New York, NY, USA, 1973. ACM. doi: http://doi.acm.org/10.1145/800125.804048. (Cited on page 90.)

[97] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, 1995. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/spe.4380250705. URL http://www.antlr.org/article/1055550346383/antlr.pdf. Current information on ANTLR is available from http://www.antlr.org/. (Cited on page 91.)

[98] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann Publishers, third revised edition, June 2007. (Cited on page 32.)

[99] Simon Peyton Jones. Call-pattern specialisation for haskell programs. *SIGPLAN Not.*, 42(9):327–337, 2007. ISSN 0362-1340.

doi: http://doi.acm.org/10.1145/1291220.1291200. (Cited on pages 209 and 213.)

[100] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, April 2003. ISBN 9780521826143. doi: http://dx.doi.org/10.2277/0521826144. Also available online from http://haskell.org/. (Cited on page 194.)

[101] Simon Peyton Jones. Implementing lazy functional languages on stock hardware: The Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992. URL http://research.microsoft.com/copyright/accept.asp?path=/users/simonpj/papers/spineless-tagless-gmachine.ps.gz&pub=34. (Cited on page 223.)

[102] Simon Peyton Jones. Wearing the hair shirt: A retrospective on Haskell. Published online, 2003. URL http://research.microsoft.com/~simonpj/papers/haskell-retrospective/index.htm. Invited talk at POPL 2003 (no paper in published proceedings). (Cited on page 185.)

[103] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(5):393–434, 2002. ISSN 0956-7968. doi: http://dx.doi.org/10.1017/S0956796802004331. (Cited on page 220.)

[104] Simon Peyton Jones, Will Partain, and André Santos. Let-floating: moving bindings to give faster programs. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 1–12, New York, NY, USA, 1996. ACM. ISBN 0-89791-770-7. doi: http://doi.acm.org/10.1145/232627.232630. (Cited on page 221.)

[105] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: Rewriting as a practical optimisation technique in GHC. In *Haskell '01: Proceedings of the ACM SIGPLAN workshop on Haskell*, 2001. URL http://haskell.org/haskell-workshop/2001/. (Cited on page 220.)

[106] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 636–666, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 0-387-54396-1. (Cited on page 213.)

[107] Simon L. Peyton Jones and W. Partain. Measuring the effectiveness of a simple strictness analyser. In J. T. O'Donnell, editor, *Glasgow Workshop on Functional Programming 1993*. Springer-Verlag, 5–7 July 1993. URL citeseer.ist.psu.edu/jones93measuring.html. (Cited on page 221.)

[108] Simon L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 184–201, New York, NY, USA, 1989. ACM. ISBN 0-89791-328-0. doi: http://doi.acm.org/10.1145/99370.99385. A much longer version is available from the website of Peyton Jones. (Cited on pages 206, 209, and 223.)

[109] Simon L. Peyton Jones and André L. M. Santos. A transformation-based optimiser for haskell. *Sci. Comput. Program.*, 32(1-3):3–47, 1998. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(97)00029-4. (Cited on page 220.)

[110] Simon L. Peyton Jones, Norman Ramsey, and Fermin Reig. C–: A portable assembly language that supports garbage collection.

In *PPDP '99: Proceedings of the International Conference PPDP'99 on Principles and Practice of Declarative Programming*, pages 1–28, London, UK, 1999. Springer-Verlag. ISBN 3-540-66540-4. URL http://www.cminusminus.org/. (Cited on page 217.)

[111] Benjamin C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. MIT Press, 2005. (Cited on page 170.)

[112] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002. (Cited on page 170.)

[113] François Pottier. A modern eye on ML type inference: old techniques and recent developments. URL http://cristal.inria.fr/~fpottier/publis/fpottier-appsem-2005.ps.gz. Lecture notes for the APPSEM Summer School, September 2005. (Cited on page 193.)

[114] François Pottier and Yann Régis-Gianis. *Menhir Reference Manual*. INRIA, December 2007. URL http://cristal.inria.fr/~fpottier/menhir/. (Cited on page 91.)

[115] Alastair Reid. Putting the spine back in the Spineless Tagless G-machine: An implementation of resumable black-holes. In *IFL '98: Selected Papers from the 10th International Workshop on 10th International Workshop*, pages 186–199, London, UK, 1999. Springer-Verlag. ISBN 3-540-66229-4. (Cited on pages 209 and 213.)

[116] H. Richards. An overview of ARC SASL. *SIGPLAN Not.*, 19(10): 40–45, 1984. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/948290.948294. (Cited on page 194.)

[117] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann Publishers, San Francisco, second edition, 2006. (Cited on pages 106 and 175.)

[118] Peter Sestoft. Demonstrating lambda calculus reduction. In Torben Æ. Mogenson, David A. Schmidt, and I. Hal Sudborough,

editors, *The essence of computation: Complexity, analysis, transformation; essays dedicated to Neil D. Jones*, volume 2566: Festschrift of *Lecture Notes in Computer Science*, pages 420–435. Springer-Verlag, New York, NY, USA, 2002. ISBN 3-540-00326-6. See http://www.dina.kvl.dk/~sestoft/lamreduce/ for a system implementing and visualizing the reduction strategies discussed in the paper. (Cited on page 169.)

[119] Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4):1–36, 2008. ISSN 1544-3566. doi: http://doi.acm.org/10.1145/1328195.1328197. (Cited on page 212.)

[120] D. Spector. Efficient full lr(i) parser generation. *SIGPLAN Not.*, 23(12):143–150, 1988. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/57669.57684. (Cited on page 90.)

[121] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of Lisp. *SIGPLAN Not.*, 28(3):231–270, 1993. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/155360.155373. (Cited on page 192.)

[122] Herbert Stoyan. Early lisp history (1956 - 1959). In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 299–310, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: http://doi.acm.org/10.1145/800055.802047. (Cited on pages 171 and 192.)

[123] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 53–66, New York, NY, USA, 2007. ACM. ISBN 1-59593-393-X. doi: http://doi.acm.org/10.1145/1190315.1190324. (Cited on page 217.)

[124] Peter J. Thiemann. Unboxed values and polymorphic typing revisited. In *FPCA '95: Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 24–35, New York, NY, USA, 1995. ACM. ISBN 0-89791-719-7. doi: http://doi.acm.org/10.1145/224164.224175. (Cited on page 213.)

[125] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991. ISBN 0-201-41667-0. URL http://www.cs.kent.ac.uk/people/staff/sjt/TTFP/. Available free of charge from the author's website, http://www.cs.kent.ac.uk/people/staff/sjt/TTFP/. (Cited on page 170.)

[126] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, 1998. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/291891.291894. (Cited on page 212.)

[127] Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42:230–265, 1937. (Cited on page 31.)

[128] D. A. Turner. Miranda: a non-strict functional language with polymorphic types. In *Proc. of a conference on Functional programming languages and computer architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4. (Cited on page 194.)

[129] D. A. Turner. The semantic elegance of applicative languages. In *FPCA '81: Proceedings of the 1981 conference on Functional programming languages and computer architecture*, pages 85–92, New York, NY, USA, 1981. ACM. ISBN 0-89791-060-5. doi: http://doi.acm.org/10.1145/800223.806766. (Cited on page 194.)

[130] David Turner. Church's thesis and functional programming. In A. Olszewski, editor, *Church's Thesis after 70 Years*, pages 518–544. ontos verlag, 2006. (Cited on page 169.)

[131] David A. Turner. Sasl language manual. Technical report, University of St. Andrews, 1976 (revised 1983). Available from the author's website http://www.cs.mdx.ac.uk/staffpages/dat/. (Cited on page 194.)

[132] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized lr parsers. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 143–158, London, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4. (Cited on page 91.)

[133] Eelco Visser. Scannerless generalized-lr parsing. Technical Report P9707, University of Amsterdam, Amsterdam, July 1997. (Cited on page 91.)

[134] Philip Wadler. Fixing some space leaks with a garbage collector. *Software Practice and Experience*, 17(9):595–608, September 1987. URL http://www.research.avayalabs.com/user/wadler/papers/leak/leak.ps. (Cited on page 213.)

[135] Philip Wadler. Why no one uses functional languages. *SIGPLAN Not.*, 33(8):23–27, 1998. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/286385.286387. URL http://homepages.inf.ed.ac.uk/wadler/papers/sigplan-why/sigplan-why.ps. (Cited on page 235.)

[136] Michael R. Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, Redwood City, Calif., USA, 1996. (Cited on page 126.)

[137] Eric R. Van Wyk and August C. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *GPCE '07: Proceed-*

ings of the 6th international conference on Generative programming and component engineering, pages 63–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-855-8. doi: http://doi.acm.org/10.1145/1289971.1289983. (Cited on page 91.)

[138] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Machines*. ACM Press/Addison-Wesley, Reading, Mass., USA, 1991. (Cited on page 126.)