Undergraduate Topics in Computer Science

Kent D. Lee

Foundations of Programming Languages

Second Edition

UTiCS

Springer

Undergraduate Topics in Computer Science

Kent D. Lee

Foundations of Programming Languages

Second Edition

Springer

Kent D. Lee
Luther College
Decorah, IA
USA

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-319-70789-1 ISBN 978-3-319-70790-7 (eBook)
https:\\doi.org\10.1007/978-3-319-70790-7

Library of Congress Control Number: 2017958018

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

A career in computer science is a commitment to a lifetime of learning. You will not be taught every detail you will need in your career while you are a student. The goal of a computer science education is to give you the tools you need so you can teach yourself new languages, frameworks, and architectures as they come along. The creativity encouraged by a lifetime of learning makes computer science one of the most exciting fields today.

There are engineering and theoretical aspects to the field of computer science. Theory often is a part of the development of new programming languages and tools to make programmers more productive. Computer programming is the process of building complex systems with those tools. Computer programmers are program engineers, and this process is sometimes called software engineering. No matter what kind of job you end up doing, understanding the tools of computer science, and specifically the programming languages you use, will help you become a better programmer.

As programmers it is important that we be able to *predict* what our programs will do. Predicting what a program will do is easier if you understand the way the programming language works. Programs execute according to a computational model. A model may be implemented in many different ways depending on the targeted hardware architecture. While there are currently a number of popular hardware architectures, most can be categorized into one of two main areas: register-based central processing units and stack-based virtual machines. While these two types of architectures are different in some ways, they also share a number of characteristics when used as the target for programming languages. This text develops a stack-based virtual machine based on the Python virtual machine called JCoCo.

Computer scientists differentiate programming languages based on three paradigms or ways of thinking about programming: *object-oriented/imperative programming*, *functional programming*, and *logic programming*. This text covers these three paradigms while using each of them in the implementation of a non-trivial programming language.

It is expected that most readers of this text will have had some prior experience with object-oriented languages. JCoCo is implemented in Java (hence the J), providing a chance to learn Java in some detail and see it used in a larger software project like the JCoCo implementation. The text proceeds in a bottom-up fashion by implementing extensions to JCoCo using Java. Then, a full-featured functional language called *Small* is implemented on top of the JCoCo virtual machine. The *Small* language is a subset of Standard ML. Standard ML is first introduced in this text and then used to implement the *Small* subset of the Standard ML language, which really is not that small afterall. Finally, late in the text a type inference system for *Small* is developed and implemented in Prolog. Prolog is an example of a logic programming language.

The text is meant to be used interactively. You should read a section, and as you read it, do the practice exercises. Each of the exercises is meant to give you a goal in reading a section of the text.

The text Web site http://www.cs.luther.edu/~leekent/PL includes code and other support files that may be downloaded. These include the JCoCo virtual machine and the MLComp compiler/type inference system.

I hope you enjoy reading the text and working through the exercises and practice problems. Have fun with it and get creative!

Acknowledgements

I have been fortunate to have good teachers throughout high school, college, and graduate school. Ken Slonneger was my advisor in graduate school, and this book came into being because of him. He inspired me to write a text that supports the same teaching style he used in his classroom. I'd also like to thank Eric Manley of Drake University for working with me by trying the projects with his students and for the valuable feedback he provided to me during the development of this text. Thanks, Eric.

I'm also fortunate to have good students working with me. Thanks go to Jonathan Opdahl for his help in building the Java version of CoCo, a virtual machine used throughout this text, and named JCoCo both because it is implemented in Java and because Jonathan helped me build it. Thank you Jonathan for your work on this project. It is greatly appreciated.

For Teachers

This book was written to fulfill two goals. The first is to introduce students to three programming paradigms: object-oriented/imperative, functional, and logic programming. To be ready for the content of this book, students should have some background in an imperative language, probably an object-oriented language such as Python, Java, or C++. They should have had an introductory course and a course in data structures as a minimum. While the prepared student will have written several programs, some of them fairly complex, most probably still struggle with predicting exactly what their program will do. It is assumed that ideas such as polymorphism, recursion, and logical implication are relatively new to students reading this book. The text assumes that students have little or no experience with the functional and logic programming paradigms.

The object-oriented language presented in this book is Java. C++ has many nuances that are worthy of several chapters in a textbook. The first edition of this text did cover C++ as the object-oriented language, but Java is better suited to the JCoCo virtual machine implementation presented in this text. However, significant topics of C++ are contrasted to Java in this text. Notably, the pass-by-value and pass-by-reference mechanisms in C++ create considerable complexity in the language. In addition, the ability of C++ programs to create objects both on the run-time stack and in the heap is contrasted to Java. Of course the standard object-oriented concepts including polymorphism and inheritance and a comparison of templates from C++ and interfaces from Java are covered in this text.

The text uses Standard ML as the functional language. ML has a polymorphic type inference system to statically type programs of the language. In addition, the type inference system of ML is formally proven sound and complete. This has some implications in writing programs. While ML's cryptic compiler error messages are sometimes hard to understand at first, once a program compiles it will often work correctly the first time. That's an amazing statement to make if your past experience is in a dynamically typed language such as Lisp, Scheme, Ruby, or Python.

The logic language used in this text is Prolog. While Prolog has traditionally been an Artificial Intelligence language, it originated as a metalanguage for expressing other languages. The text concentrates on using Prolog to implement a type inference system. Students learn about logical implication and how a problem they are familiar with can be re-expressed in a logic programming language.

The second goal of the text is to be interactive. This book is intended to be used in and outside of class. It is my experience that we almost all learn more by doing than by seeing. To that end, the text encourages teachers to actively teach. Each chapter follows a pattern of presenting a topic followed by a practice exercise or exercises that encourage students to try what they have just read. These exercises can be used in class to help students check their understanding of a topic. Teachers are encouraged to take the time to present a topic and then allow students time to reflect and practice the concept just presented. In this way, the text becomes a lecture resource. Students get two things out of this. It forces them to be interactively engaged in the lectures, not just passive observers. It also gives them immediate feedback on key concepts to help them determine whether they understand the material or not. This encourages them to ask questions when they have difficulty with an exercise. Tell students to bring the book to class along with a pencil and paper. The practice exercises are easily identified.

This book presents several projects to reinforce topics outside the classroom. Each chapter of the text suggests several non-trivial programming projects that accompany the paradigm being covered to drive home the concepts covered in that chapter. The projects and exercises described in this text have been tested in practice, and documentation and solutions are available upon request.

Decorah, USA Kent D. Lee

Contents

1 Introduction

This text on Programming Languages is intended to introduce you to new ways of thinking about programming. Typically, computer science students start out learning to program in an imperative model of programming where variables are created and updated as a program executes. There are other ways to program. As you learn to program in these new paradigms you will begin to understand that there are different ways of thinking about problem solving. Each paradigm is useful in some contexts. This book is not meant to be a survey of lots of different languages. Rather, its purpose is to introduce you to the three styles of programming languages by using them to implement a non-trivial programming language. These three styles of programming are:

- Imperative/Object-Oriented Programming with languages like Java, C++, Python, and other languages you may have used before.
- Functional Programming with languages like Standard ML, Haskell, Lisp, Scheme, and others.
- Logic Programming with Prolog.

The book provides an in-depth look at programming in assembly language, Java, Standard ML, and Prolog. However, the programming language concepts covered in this text apply to all languages in use today. The goal of the text is to help you understand how to use the paradigms and models of computation these languages represent to solve problems. The text elaborates on when these languages may be appropriate for a problem by showing you how they can be used to implement a programming language. Many of the problems solved while implementing a programming language are similar to other problems in computer science. The text elaborates on techniques for problem solving that you may be able to apply in the future. You might be surprised by what you can do and how quickly a program can come together given the right choice of language.

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_1

To begin you should know something about the history of computing, particularly as it applies to the models of computation that have been used in implementing many of the programming languages we use today. All of what we know in Computer Science is built on the shoulders of those who came before us. To understand where we are, we really should know something about where we came from in terms of Computer Science. Many great people have been involved in the development of programming languages and to learn even a little about who these people are is really fascinating and worthy of an entire book in itself.

1.1 Historical Perspective

Much of what we attribute to Computer Science actually came from Mathematics. Many mathematicians are programmers that have written their programs, or proofs in the words of Mathematics, using mathematical notation. In the mid 1800s abstract algebra and geometry were hot topics of research among mathematicians. In the early 1800s Niels Henrik Abel, a Norwegian mathematician, was interested in solving a problem called the quintic equation. Eventually he developed a new branch of mathematics called Group Theory with which he was able to prove there was no general algebraic solution to the quintic equation. Considering the proof of this required a new branch of mathematics, much of Abel's work involved developing the mathematical notation or language to describe his work. Unfortunately, Abel died of tuberculosis at twenty six years old.

Sophus Lie (*pronounced Lee*), pictured in Fig. 1.1, was another Norwegian mathematician who lived from 1842–1899 [20]. He began where Abel's research ended and explored the connection of Abstract Algebra and Group Theory with Geometry. From this work he developed a set of group theories, eventually named Lie Groups. From this discovery he found ways of solving Ordinary Differential Equations by

Fig. 1.1 Sophus Lie [21]

exploiting properties of symmetry within the equations [8]. One Lie group, the $E8$ group was too complicated to map in Lie's time. In fact, it wasn't until 2007 that the structure of the $E8$ group could be mapped because the solution produced sixty times more data than the human genome project [1].

While the techniques Lie and Abel discovered were hard for people to learn and use at the time, today computer programs capable of symbolic manipulation use Lie's techniques to solve these and other equally complicated problems. And, the solutions of these problems are very relevant in the world today. For example, the work of Sophus Lie is used in the design of aircraft.

As mathematicians' problem solving techniques became more sophisticated and the problems they were solving became more complex, they were interested in finding automated ways of solving these problems. Charles Babbage (1791–1871) saw the need for a computer to do calculations that were too error-prone for humans to perform. He designed a *difference engine* to compute mathematical tables when he found that human *computers* weren't very accurate [27]. However, his computer was mechanical and couldn't be built using engineering techniques known at that time. In fact it wasn't completed until 1990, but it worked just as he said it would over a hundred years earlier.

Charles Babbage's difference engine was an early attempt at automating a solution to a problem, but others would follow of course. Alan Turing was a British mathematician and one of the first computer scientists. He lived from 1912–1954. In 1936 he wrote a paper entitled, "On Computable Numbers, with an Application to the Entscheidungsproblem" [23]. The Entscheidungsproblem, or decision problem, had been proposed a decade earlier by a German mathematician named David Hilbert. This problem asks: Can an algorithm be defined that decides if a statement given in first order logic can be proved from a set of axioms and known truth values? The problem was later generalized to the following question: Can we come up with a general set of steps that given any algorithm and its data, will decide if it terminates? In Alan Turing's paper, he devised an abstract machine called the Turing Machine. This Turing Machine was very general and simple. It consisted of a set of states and a tape. The set of states were decided on by a programmer. The machine starts in the start state as decided by the programmer. From that state it could read a symbol from a tape. Based on the symbol it could write a symbol to the tape and move to the left or right, while transitioning to another state. As the Turing machine ran, the action that it took was dictated by the current state and the symbol on the tape. The programmer got to decide how many states were a part of the machine, what each state should do, and how to move from one state to another. In Turing's paper he proved that such a machine could be used to solve any computable function and that the decision problem was not solvable by this machine. The more general statement of this problem was named the *Halting Problem*. This was a very important result in the field of theoretical Computer Science.

In 1939 John Atanasoff, at Iowa State University, designed what is arguably the first computer, the ABC or Atanasoff-Berry Computer [28]. Clifford Berry was one of his graduate students. The computer had no central processing unit, but it did perform logical and other mathematical operations. Eckert and Mauchly, at the University of

Pennsylvania, were interested in building a computer during the second world war. They were funded by the Department of Defense to build a machine to calculate trajectory tables for launching shells from ships. The computer, called ENIAC for Electronic Numerical Integrator and Computer, was unveiled in 1946, just after the war had ended. ENIAC was difficult to program since the program was *written* by plugging cables into a switch, similar to an old telephone switchboard.

Around that same time a new computer, called EDVAC, was being designed. In 1945 John von Neumann proposed storing the computer programs on EDVAC in memory along with the program data [26]. Alan Turing closely followed John von Neumann's paper by publishing a paper of his own in 1946 describing a more complete design for stored-program computers [24]. To this day the computers we build and use are stored-program computers. The architecture is called the von Neumann architecture because of John von Neumann's and Alan Turing's contributions. While Turing didn't get the architecture named after him, he is famous in Computer Science for other reasons like the Turing machine and the Halting problem.

In the early days of Computer Science, many programmers were interested in writing tools that made it easier to program computers. Much of the programming was based on the concept of a stored-program computer and many early programming languages were extensions of this model of computation. In the stored-program model the program and data are stored in memory. The program manipulates data based on some input. It then produces output.

Around 1958, Algol was created and the second revision of this language, called Algol 60, was the first modern, structured, imperative programming language. While the language was designed by a committee, a large part of the success of the project was due to the contributions of John Backus pictured in Fig. 1.2. He described the structure of the Algol language using a mathematical notation that would later be called Backus-Naur Format or BNF. Very little has changed with the underlying computer architecture over the years. Of course, there have been many changes in the size, speed, and cost of computers! In addition, the languages we use have become

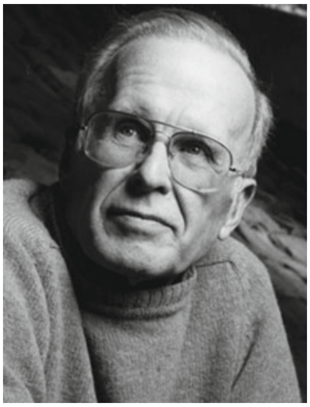


Fig. 1.2 John Backus [3]

even more structured over the years. But, the principles that Algol 60 introduced are still in use today.

Recalling that most early computer scientists were mathematicians, it shouldn't be too surprising to learn that there were others that approached the problem of programming differently. Much of the initial interest in computers was spurred by the invention of the stored-program computer and many of the early languages reflected this excitement. The imperative style was closely tied to the architecture of a stored program computer. Data was read from an input device and the program acted on that data by updating memory as the program executed. There was another approach developing at the same time. Back in 1936, Alonzo Church, a U.S. mathematician who lived from 1903–1995, was also interested in the decision problem proposed by David Hilbert. To try to solve the problem he devised a language called the lambda calculus, usually written as the λ-calculus. Using his very simple language he was able to describe computation as symbol manipulation. Alan Turing was a doctoral student of Church and while they independently came up with two ways to prove that the decision problem was not solvable, they later proved their two models of computation, Turing machines and the λ-calculus, were equivalent. Their work eventually led to a very important result called the Church-Turing Thesis. Informally, the thesis states that all computable problems can be solved by a Turing Machine or the λ-calculus. The two models are equivalent in power.

Ideas from the λ-calculus led to the development of Lisp by John McCarthy, pictured in Fig. 1.3. The λ-calculus and Lisp were not designed based on the principle of the stored-program computer. In contrast to Algol 60, the focus of these languages was on functions and what could be computed using functions. Lisp was developed around 1958, the same time that Algol 60 was being developed.

Logic is important both in Computer Science and Mathematics. Logicians were also interested in solving problems in the early days of Computer Science. Many problems in logic are expressed in the languages of propositional or predicate logic.

Fig. 1.3 John McCarthy [14]

Of course, the development of logic goes all the way back to ancient Greece. Some logicians of the 20th century were interested in understanding natural language and they were looking for a way to use computers to solve at least some of the problems related to processing natural language statements. The desire to use computers in solving problems from logic led to the development of Prolog, a powerful programming language based on predicate logic.

Practice 1.1 Find the answers to the following questions.

1. What are the origins of the three major computational models that early computer scientists developed?
2. Who were Alan Turing and Alonzo Church and what were some of their contributions to Computer Science?
3. What idea did both John von Neumann and Alan Turing contribute to?
4. What notation did John Backus develop and what was one of its first uses?
5. What year did Alan Turing first propose the Turing machine and why?
6. What year did Alonzo Church first propose the λ-calculus and why?
7. Why are Eckert and Mauchly famous?
8. Why are the history of Mathematics and Computer Science so closely tied together?

You can check your answer(s) in Section 1.8.1.

1.2 Models of Computation

While there is some controversy about who originally came up with the concept of a stored program computer, John von Neumann is generally given credit for the idea of storing a program as a string of 0's and 1's in memory along with the data used by the program. Von Neumann's architecture had very little structure to it. It consisted of several registers and memory. The Program Counter (PC) register kept track of the next instruction to execute. There were other registers that could hold a value or point to other values stored in memory. This model of computation was useful when programs were small. However, without additional structure, anything but a small program would quickly get hard to manage. This was what was driving the need for better and newer programming languages. Programmers needed tools that let them organize their code so they could focus on problem solving instead of the details of the hardware.

1.2.1 The Imperative Model

As programs grew in size it was necessary to provide the means for applying additional structure to them. In the early days a function was often called a sub-routine. Functions, procedures, and sub-routines were introduced by languages like Algol 60 so that programs could be decomposed into simpler sub-programs, providing a way for programmers to organize their code. Terms like top-down or bottom-up design were used to describe this process of subdividing programs into simpler sub-programs. This process of subdividing programs was often called *structured programming*, referring to the decomposition of programs into simpler, more manageable pieces. Most modern languages provide the means to decompose problems into simpler subproblems. We often refer to this structured approach as the imperative model of programming.

To implement functions and function calls in the von Neumann architecture, it was necessary to apply some organization to the data of a program. In the imperative model, memory is divided into regions which hold the program and the data. The data area is further subdivided into the static or global data area, the run-time stack, and the heap as pictured in Fig. 1.4.

In the late 1970s and 1980s people like Niklaus Wirth and Bjarne Stroustrup were interested in developing languages that supported an additional level of organization called Object-Oriented Programming, often abbreviated OOP. Object-oriented programming still uses the imperative model of programming. The addition of a means to describe classes of objects gives programmers another way of organizing their code into functions that are related to a particular type of object.

When a program executes it uses a special register called the stack pointer (SP) to point to the top activation record on the run-time stack. The run-time stack contains one activation record for each function or procedure invocation that is currently unfinished in the program. The top activation record corresponds to the current

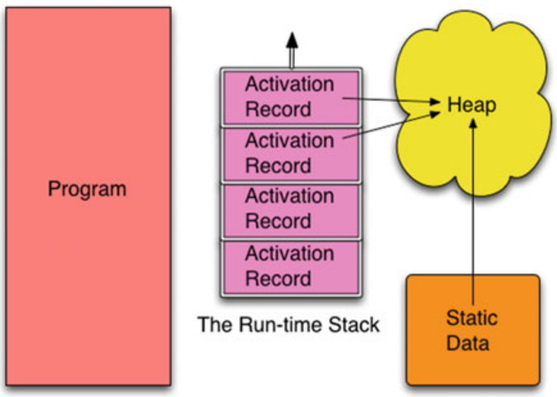


Fig. 1.4 Imperative model

function invocation. When a function call is made an activation record is pushed onto the run-time stack. When a function returns, the activation record is popped by decrementing the stack pointer to point to the previous activation record.

An activation record contains information about its associated function. The local variables of the function are stored there. The program counter's value before the function call was made is stored there. This is often called the return address. Other state information may also be stored there depending on the language and the details of the underlying von Neumann architecture. For instance, parameters passed to the function may also be stored there.

Static or global data refers to data and functions that are accessible globally in the program. Global data and functions are visible throughout the program. Where global data is stored depends on the implementation of the compiler or interpreter. It might be part of the program code in some instances. In any case, this area is where constants, global variables, and possibly built-in globally accessible functions are stored.

The heap is an area for dynamic memory allocation. The word dynamic means that it happens while the program is running. All data that is created at run-time is located in the heap. The data in the heap has no names associated with the values stored there. Instead, named variables called pointers or references point to the data in the heap. In addition, data in the heap may contain pointers that point to other data, which is also usually in the heap.

Like the original von Neumann architecture, the primary goal of the imperative model is to get data as input, transform it via updates to memory, and then produce output based on this imperatively changed data. The imperative model of computation parallels the underlying von Neumann architecture and is used by many modern languages. Some variation of this model is used by languages like Algol 60, C++, C, Java, VB.net, Python, and many other languages.

Practice 1.2 Find the answers to the following questions.

1. What are the three divisions of data memory called?
2. When does an item in the heap get created?
3. What goes in an activation record?
4. When is an activation record created?
5. When is an activation record deleted?
6. What is the primary goal of imperative, object-oriented programming?

You can check your answer(s) in Section 1.8.2.

1.2.2 The Functional Model

In the functional model the goal of a program is slightly different. This slight change in the way the model works has a big influence on how you program. In the functional model of computation the focus is on function calls. Functions and parameter passing are the primary means of accomplishing data transformation.

Data is generally not changed in the functional model. Instead, new values are constructed from old values. A pure functional model wouldn't allow any updates to existing values. However, most functional languages allow limited updates to memory in the imperative style.

The conceptual view presented in Fig. 1.4 is similar to the view in the functional world. However, the difference between program and data is eliminated. A function is data like any other data element. Integers and functions are both first-class citizens of the functional world.

The static data area is still present, but takes on a minor role in the functional model. The run-time stack becomes more important because most work is accomplished by calling functions. Functional languages are much more careful about how they allow programmers to access the heap and as a result, you really aren't aware of the heap when programming in a functional language. Data is certainly dynamically allocated, but once data is created on the heap it is not modified in a pure functional model. Impure models might allow some modification of storage but this is the influence of imperative languages creeping into the functional model as a way to deal with performance issues. The result is that you spend less time thinking about the underlying architecture when programming in a functional language.

Lisp, Scheme, Scala, Clojure, Elixir, Haskell, Caml, and Standard ML, which is covered in this text, are all examples of functional languages. Functional languages may be pure, which means they do not support variable updates like the imperative model. Scheme is a pure functional language. Most functional languages are not pure. Standard ML and Lisp are examples of impure functional languages. Scala is a recent functional language that also supports object-oriented programming.

Practice 1.3 Answer the following questions.

1. What are some examples of functional languages?
2. What is the primary difference between the functional and imperative models?
3. Immutable data is data that cannot be changed once created. The presence of immutable data simplifies the conceptual model of programming. Does the imperative or functional model emphasize immutable data?

You can check your answer(s) in Section 1.8.3.

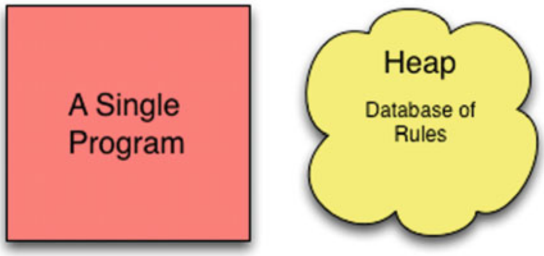


Fig. 1.5 Logic model of computation

1.2.3 The Logic Model

The logic model of computation, pictured in Fig. 1.5, is quite different from either the imperative or functional model. In the logic model the programmer doesn't actually write a program at all. Instead, the programmer provides a database of facts or rules. From this database, a single program tries to answer questions with a yes or no answer. In the case of Prolog, the program acts in a predictable manner allowing the programmer to provide the facts in an order that determines how the program will work. The actual implementation of this conceptual view is accomplished by a virtual machine, a technique for implementing languages that is covered later in this text.

There is still the concept of a heap in Prolog. One can assert new rules and retract rules as the program executes. To dynamically add rules or retract them there must be an underlying heap. In fact, the run-time stack is there too. However, the run-time stack and heap are so hidden in this view of the world that it is debatable whether they should appear in the conceptual model at all.

Practice 1.4 Answer these questions on what you just read.

1. How many programs can you write in a logic programming language like Prolog?
2. What does the programmer do when writing in Prolog?

You can check your answer(s) in Section 1.8.4.

1.3 The Origins of a Few Programming Languages

This book explores language implementation using several small languages and exercises that illustrate each of these models of computation. In addition, exercises within the text will require implementation in four different languages: assembly language, Java (or alternatively C++), Standard ML, and Prolog. But where did these languages come from and why are we interested in learning how to use them?

1.3.1 A Brief History of C and C++

The Unix operating system was conceived of, designed, and written around 1972. Ken Thompson was working on the design of Unix with Dennis Ritchie. It was their project that encouraged Ritchie to create the C language. C was more structured than the assembly language most operating systems were written in at the time and it was portable and could be compiled to efficient machine code. Thompson and Ritchie wanted an operating system that was portable, small, and well organized.

While C was efficient, there were other languages that had either been developed or were being developed that encouraged a more structured approach to programming. For several years there had been ideas floating around about how to write code in object-oriented form. Simula, created by Ole-Johan Dahl and Kristen Nygaard around 1967, was an early example of a language that supported Object-Oriented design. Modula-2, created by Niklaus Wirth around 1978, was also taking advantage of these ideas. Smalltalk, an interpreted language, was object-oriented and was also developed in the mid 1970s and released in 1980.

In 1980 Bjarne Stroustrup, pictured in Fig. 1.6, began working on the design of C++ while working at Bell Labs. He envisioned C++ as a language that would allow C programmers to keep their old code while new code could be written using these Object-Oriented concepts. In 1983 he named this new language C++, as in the next increment of C, and with much anticipation, in 1985 the language was released. About the same time Dr. Stroustrup released a book called *The C++ Programming Language* [19], which described the language. The language is still evolving. For instance, templates, an important part of C++ were first described by Stroustrup in 1988 [17] and it wasn't until 1998 that it was standardized as ANSI C++. Today an ANSI committee oversees the continued development of C++. The latest C++ standard was released in 2014 as of this writing. The previous standard was released

Fig. 1.6 Bjarne Stroustrup [18]

in 2011. C++ is a mature language, but is still growing and evolving. The 2017 standard is currently in the works with comments presently being solicited by the standards committee.

1.3.2 A Brief History of Java

C++ is a very powerful language, but also demands that programmers be very careful when writing code. The biggest problem with C++ programs are memory leaks. When objects are created on the heap in C++, they remain on the heap until they are freed. If a programmer forgets to free an object, then that space cannot be re-used while the program is running. That space is gone until the program is stopped, even if no code has a pointer to that object anymore. This is a memory leak. And, for long-running C++ programs it is the number one problem. Destructors are a feature of C++ that help programmers prevent memory leaks. Depending on the structure of a class in your program, it may need a destructor to take care of cleaning up instances of itself (i.e. objects of the class) when they are freed.

C++ programs can create objects in the run-time stack, on the heap, or within other objects. This is another powerful feature of C++. But, with this power over the creation of objects comes more responsibility for the programmer. This control over object creation leads to the need for extra code to decide how copies of objects are made. In C++ every class may contain a copy constructor so the programmer can control how copies of objects are made.

In 1991 a team called the *Green Team*, was working for a company named *Sun Microsystems*. This group of software engineers wanted to design a programming language and run-time system that could be used in the next generation of personal devices. The group was led by a man named James Gosling. To support their vision, they designed the Java Virtual Machine (i.e. JVM), a program that would interpret byte code files. The JVM was designed as the run-time system for the Java programming language. Java programs, when compiled, are translated into bytecode files that run on the JVM.

The year 1995 brought the birth of the world wide web and with it one of the first web browsers, Netscape Navigator, which later became Mozilla Firefox. In 1995 it was announced that Netscape would include Java technology within the browser. This led to some of the initial interest in the language, but the language has grown way beyond web browsers. In fact, Java is not really a web browser technology anymore. It is used in many web backends, where Java programs wait for connections from web browsers, but it doesn't run programs within web browsers much these days. Another language, Javascript, is now the primary language of web browsers. Javascript is similar to Java in name, but not its technology. Javascript was licensed as a name from Sun Microsystems in its early days because of the popularity of Java [22].

The original intention of Java was to serve as a means for running software for personal devices. Java has become very important in that respect. It now is the basis for the Android operating system that runs on many phones and other personal devices like tablets. So, in a sense, the original goal of the Green Team has been realized, just fifteen or so years later.

When the original Green Team was designing Java they wanted to take the best of C++ while leaving behind some of its complexity. In Java objects can only be created in one location, on the heap. Sticking to one and only one memory model for objects simplifies many aspects of Java. Objects are never copied by the language. So, copy constructors are unnecessary in Java. When an object is passed to a function, a reference to an object is passed without making a copy of the object. When one object wants to contain another object, it keeps a reference to that object. Java objects are never stored inside other objects. Simplifying the memory model for objects means that in Java programs we don't have to worry about copying objects.

Objects can still be copied in Java, but making copies of objects is the responsibility of the programmer. The Java language does not make copies. Programmers make copies by calling a special method called *clone*.

Java also includes garbage collection. This means that the Java Virtual Machine takes care of deciding when the space that an object resides in can be reclaimed. It can be reclaimed when no other objects or code have a reference to it anymore. This means that programmers don't have to write destructors. The JVM manages this for them.

So, while C++ and Java share a lot of syntax, there are many differences as well. Java has a simpler memory model. Garbage collection removes the fear of memory leaks in Java programs. The Java Virtual Machine also provides other advantages to writing Java programs. This does not make C++ a bad language by any means. It's just that Java and C++ have different goals. The JVM and Java manage a lot of the complexity of writing object-oriented programs, freeing the programmer from these duties. C++ on the other hand, gives you the power to manage all the details of a program, right down to the hardware interface. Neither is better than the other, they just serve different purposes while the two languages also share a lot of the same syntax.

Fig. 1.7 Guido van Rossum [25]

1.3.3 A Brief History of Python

Python was designed and implemented by Guido van Rossum, pictured in Fig. 1.7. He started Python as a hobby project during the winter months of 1989. A more complete history of this language is available on the web at http://python-history.blogspot.com. Python is another object-oriented language like C++ and Java. Unlike C++, Python is an interpreted language. Mr. van Rossum designed Python's interpreter as a virtual machine, like the Java Virtual Machine (i.e. JVM). But Python's virtual machine is not accessible separately, unlike the JVM. The Python virtual machine is an internal implementation detail of the Python interpreter. Virtual machines have been around for some time including an operating system for IBM mainframe computers, called VM. Using a virtual machine when implementing a programming language can make the language and its programs more portable across platforms. Python runs on many different platforms like Apple's Mac OS X, Linux, and Microsoft Windows. Virtual machines can also provide services that make language implementation easier.

Programmers world-wide have embraced Python and have developed many libraries for Python and written many programs. Python has gained popularity among developers because of its portability and the ability to provide libraries to others. Guido van Rossum states in his history of Python, "A large complex system should have multiple levels of extensibility. This maximizes the opportunities for users, sophisticated or not, to help themselves." Extensibility refers to the ability to define libraries of classes to solve problems from many different application areas. Python is used in internet programming, server scripting, computer graphics, visualization, Mathematics, Computer Science education, and many, many other application areas.

Mr. van Rossum continues, saying "In many ways, the design philosophy I used when creating Python is probably one of the main reasons for its ultimate success. Rather than striving for perfection, early adopters found that Python worked "well enough" for their purposes. As the user-base grew, suggestions for improvement were gradually incorporated into the language." Growing the user-base has been key to the success of Python. As the number of programmers that know Python has increased so has interest in improving the language. Python now has two major versions, Python 2 and Python 3. Python 3 is not backward compatible with Python 2. This break in compatibility gave the Python developers an opportunity to make improvements in the language. Chapters 3 and 4 cover some of the implementation details of the Python programming language.

1.3.4 A Brief History of Standard ML

Standard ML originated in 1986, but was the follow-on of ML which originated in 1973 [16]. Like many other languages, ML was implemented for a specific purpose. The ML stands for Meta Language. Meta means above or about. So a metalanguage is a language about language. In other words, a language used to describe a language. ML was originally designed for a theorem proving system. The theorem prover was called LCF, which stands for Logic for Computable Functions. The LCF theorem

Fig. 1.8 Robin Milner [15]

prover was developed to check proofs constructed in a particular type of logic first proposed by Dana Scott in 1969 and now called Scott Logic. Robin Milner, pictured in Fig. 1.8, was the principal designer of the LCF system. Milner designed the first version of LCF while at Stanford University. In 1973, Milner moved to Edinburgh University and hired Lockwood Morris and Malcolm Newey, followed by Michael Gordon and Christopher Wadsworth, as research associates to help him build a new and better version called Edinburgh LCF [9].

For the Edinburgh version of LCF, Dr. Milner and his associates created the ML programming language to allow proof commands in the new LCF system to be extended and customized. ML was just one part of the LCF system. However, it quickly became clear that ML could be useful as a general purpose programming language. In 1990 Milner, together with Mads Tofte and Robert Harper, published the first complete formal definition of the language; joined by David MacQueen, they revised this standard to produce the Standard ML that exists today [16].

ML was influenced by Lisp, Algol, and the Pascal programming languages. In fact, ML was originally implemented in Lisp. There are now two main versions of ML: Moscow ML and Standard ML. Today, ML's main use is in academia in the research of programming languages. But, it has been used successfully in several other types of applications including the implementation of the TCP/IP protocol stack [4] and a web server as part of the Fox Project. A goal of the Fox Project was the development of system software using advanced programming languages [10].

ML is a very good language to use in learning to implement other languages. It includes tools for automatically generating parts of a language implementation including components called a scanner and a parser which are introduced in Chap. 6. These tools, along with the polymorphic strong type checking provided by Standard ML, make implementing a compiler or interpreter a much easier task. Much of the work of implementing a program in Standard ML is spent in making sure all the types in the program are correct. This strong type checking often means that once a program is properly typed it will run the first time. This is quite a statement to make, but nonetheless it is often true.

Important Standard ML features include:

- ML is higher-order supporting functions as first-class values. This means functions may be passed as parameters to functions and returned as values from functions.
- Strong type checking (discussed later in this chapter) means it is pretty infrequent that you need to debug your code. What a great thing!
- Pattern-matching is used in the specification of functions in ML. Pattern-matching is convenient for writing recursive functions.
- The exception handling system implemented by Standard ML has been proven type safe, meaning that the type system encompasses all possible paths of execution in an ML program.

1.3.5 A Brief History of Prolog

Prolog was developed in 1972 by Alain Colmerauer, pictured in Fig. 1.9, with Philippe Roussel. Colmerauer and Roussel and their research group had been working on natural language processing for the French language and were studying logic and automated theorem proving [7] to answer simple questions in French. Their research led them to invite Robert Kowalski, pictured in Fig. 1.10, who was working in the area of logic programming and had devised an algorithm called SL-Resolution, to work with them in the summer of 1971 [11,29]. Colmerauer and Kowalski, while working together in 1971, discovered a way formal grammars could be written as clauses in predicate logic. Colmerauer soon devised a way that logic predicates could be used to express grammars that would allow automated theorem provers to parse natural language sentences efficiently. This is covered in some detail in Chap. 7.

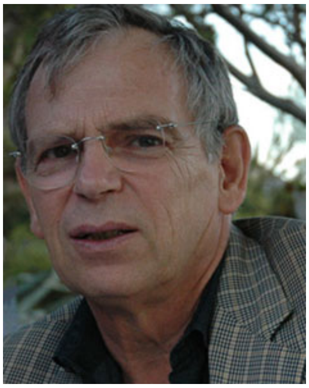


Fig. 1.9 Alain Colmerauer [6]

Fig. 1.10 Robert Kowalski [12]

In the summer of 1972, Kowalski and Colmerauer worked together again and Kowalski was able to describe the procedural interpretation of what are known as Horn Clauses. Much of the debate at the time revolved around whether logic programming should focus on procedural representations or declarative representations. The work of Kowalski showed how logic programs could have a dual meaning, both procedural and declarative.

Colmerauer and Roussel used this idea of logic programs being both declarative and procedural to devise Prolog in the summer and fall of 1972. The first large Prolog program, which implemented a question and answering system in the French language, was written in 1972 as well.

Later, the Prolog language interpreter was rewritten at Edinburgh to compile programs into DEC-10 machine code. This led to an abstract intermediate form that is now known as the Warren Abstract Machine or WAM. WAM is a low-level intermediate representation that is well-suited for representing Prolog programs. The WAM virtual machine can be (and has been) implemented on a wide variety of hardware. This means that Prolog implementations exist for most computing platforms.

Practice 1.5 Answer the following questions.

1. Who invented C++? C? Standard ML? Prolog? Python? Java?
2. What do Standard ML and Prolog's histories have in common?
3. What do Prolog and Python have in common?
4. What language or languages is Standard ML based on?

You can check your answer(s) in Section 1.8.5.

1.4 Language Implementation

There are three ways that languages can be implemented.

- A language can be interpreted.
- A language can be compiled to a machine language.
- A language can be implemented by some combination of the first two methods.

Computers are only capable of executing machine language. Machine language is the language of the Central Processing Unit (CPU) and is very simple. For instance, typical instructions are *fetch this value into the CPU*, *store this value into memory from the CPU*, *add these two values together*, and *compare these two values and if they are equal, jump here next*. The goal of any programming language implementation is to translate a source program into this simpler machine language so it can be executed by the CPU. The overall process is pictured in Fig. 1.11.

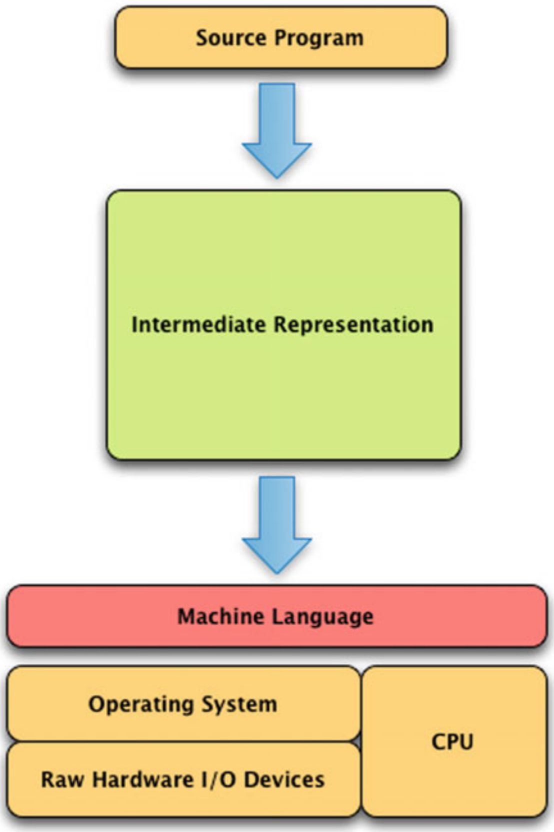


Fig. 1.11 Language implementation

All language implementations translate a source program to some intermediate representation before translating the intermediate representation to machine language. Exactly how these two translations are packaged varies significantly from one programming language to the next, but luckily most language implementations follow one of a few methodologies. The following sections will present some case studies of different languages so you can see how this translation is accomplished and packaged.

1.4.1 Compilation

The most direct method of translating a program to machine language is called compilation. The process is shown in Fig. 1.12. A compiler is a program that internally is composed of several parts. The *parser* reads a source program and translates it into an intermediate form called an abstract syntax tree (*AST*). An *AST* is a tree-like data structure that internally represents the source program. We'll read about abstract syntax trees in later chapters. The *code generator* then traverses the AST and produces another intermediate form called an assembly language program. This program is not machine language, but it is much closer. Finally, an *assembler* and *linker* translate an assembly language program to machine language making the program ready to execute.

This whole process is encapsulated by a tool called a *compiler*. In most instances, the assembler and linker are separate from the compiler, but normally the compiler runs the assembler and linker automatically when a program is compiled so as programmers we tend to think of a compiler compiling our programs and don't necessarily think about the assembly and link phases.

Programming in a compiled language is a three-step process.

- First, you write a source program.
- Then you compile the source program, producing an executable program.
- Then you run the executable program.

When you are done, you have a source program and an executable program that represent the same computation, one in the source language, the other in machine language. If you make further changes to the source program, the source program and the machine language program are not in sync. After making changes to the source program you must remember to recompile before running the executable program again.

Machine language is specific to a CPU architecture and operating system. Compiling a source program on Linux means it will run on most Linux machines with a similar CPU. However, you cannot take a Linux executable and put it on a Microsoft Windows machine and expect it to run, even if the two computers have the same CPU. The Linux and Windows operating systems each have their own format for executable machine language programs. In addition, compiled programs use operating system services for printing, reading input, and doing other Input/Output (I/O) operations. These services are invoked differently between operating systems. Lan-

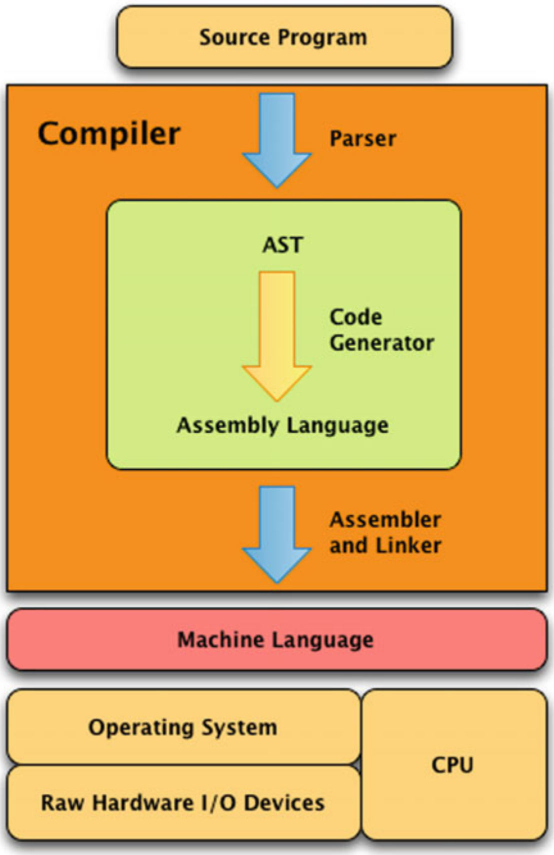


Fig. 1.12 The compilation process

guages like C++ hide these implementation details from you in the code generator, but the end result is that a program compiled for one operating system will not work on another operating system without being recompiled.

C, C++, Pascal, Fortran, COBOL and many others are typically compiled languages. On the Linux operating system the C compiler is called *gcc* and the C++ compiler is called *g++*. The *g* in both names reflects the fact that both compilers come out of the GNU project and the Free Software Foundation. Linux, gcc, and g++ are freely available to anyone who wants to download them. The best way to get these tools is to download a Linux distribution and install it on a computer. The *gcc* and *g++* compilers come standard with Linux.

There are implementations of C and C++ for many other platforms. The web site http://gcc.gnu.org contains links to source code and to prebuilt binaries for the g++ compiler. You can also download C++ compilers from Apple and Microsoft. For Mac OS X computers you can get C++ by downloading the XCode Developer Tools.

You can also install g++ and gcc for Mac OS X computers using a tool called *brew*. If you run Microsoft Windows you can install Visual C++ Express from Microsoft. It is free for educational use.

1.4.2 Interpretation

An interpreter is a program that is written in some other language and compiled into machine language. The interpreter itself is the machine language program. The interpreter itself is written to read source programs from the interpreted language and interpret them. For instance, Python is an interpreted language. The Python interpreter is written in C and is compiled for a particular platform like Linux, Mac OS X, or Microsoft Windows. To run a Python program, you must download and install the Python interpreter that goes with your operating system and CPU.

When you run an interpreted source program, as depicted in Fig. 1.13, you are actually running the interpreter. Your program is not running because your program is never translated to machine language. The interpreter is the machine language program that executes all the programs you write in the interpreted language. The source program you write controls the behavior of the interpreter program.

Programming in an interpreted language is a two step process.

- First you write a source program.
- Then you execute the source program by running the interpreter.

Each time your program is executed it is translated into an AST by a part of the interpreter called the parser. There may be an additional step that translates the AST to some lower-level representation, often called bytecode. In an interpreter, this lower-level representation is still internal to the interpreter program. Then a part of the interpreter, often called a virtual machine, executes the byte code instructions.

Not every interpreter translates the AST to bytecode. Sometimes the interpreter directly interprets the AST but it is often convenient to translate the source program's AST to some simpler representation before executing it.

Eliminating the compile step has a few implications.

- Since you have one less step in development you may be encouraged to run your code more frequently during development. This is a generally a good thing and can shorten the development cycle.
- Secondly, because you don't have an executable version of your code, you don't have to manage the two versions. You only have a source code program to keep track of.
- Finally, because the source code is not platform dependent, you can usually easily move your program between platforms. The interpreter insulates your program from platform dependencies.

Of course, source programs for compiled languages are generally platform independent too. But, they must be recompiled to move the executable program from one

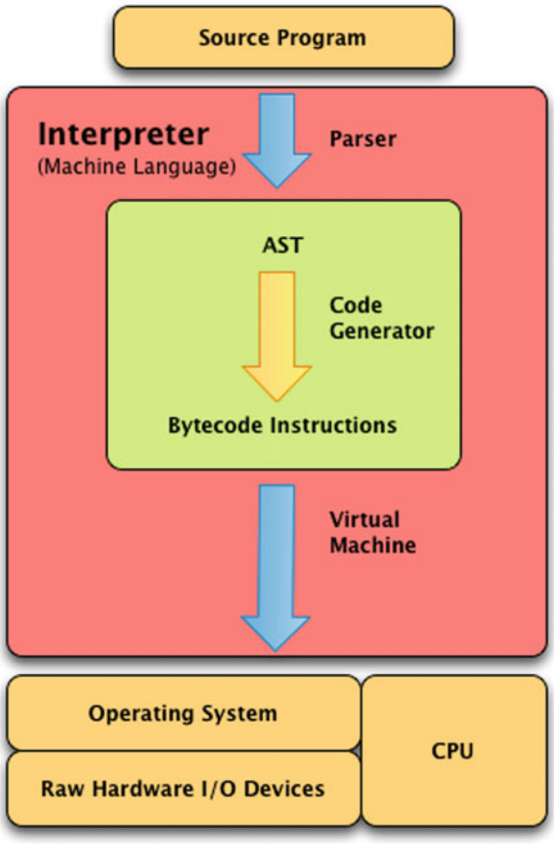


Fig. 1.13 The interpretation process

platform to another. The interpreter itself isn't platform independent. There must be a version of an interpreter for each platform/language combination. So there is a Python interpreter for Linux, another for Microsoft Windows, and yet another for Mac OS X. Thankfully, because the Python interpreter is written in C the same Python interpreter program can be compiled (with some small differences) for each platform.

There are many interpreted languages available including Python, Ruby, Standard ML, Unix scripting languages like Bash and Csh, Prolog, and Lisp. The portability of interpreted languages has made them very popular among programmers, especially when writing code that needs to run across multiple platforms.

One huge problem that has driven research into interpreted languages is that of heap memory management. Recall that the heap is the place where memory is dynamically allocated. As mentioned earlier in the chapter, C and C++ programs are notorious for having memory leaks. Every time a C++ programmer reserves some space on the heap he/she must remember to free that space. If they don't free the space when they are done with it the space will never be available again while the program continues to execute. The heap is a big space, but if a program runs long enough and continues to allocate and not free space, eventually the heap will fill up and the program will terminate abnormally. In addition, even if the program doesn't terminate abnormally, the performance of the system will degrade as more and more time is spent managing the large heap space.

Most, if not all, interpreted languages don't require programmers to free space on the heap. Instead, there is a special task or thread that runs periodically as part of the interpreter to check the heap for space that can be freed. This task is called the *garbage collector*. Programmers can allocate space on the heap but don't have to be worried about freeing that space. For a garbage collector to work correctly, space on the heap has to be allocated and accessed in the right way. Many interpreted languages are designed to insure that a garbage collector will work correctly.

The disadvantage of an interpreted language is in speed of execution. Interpreted programs typically run slower than compiled programs. In a compiled program, parsing and code generation happen once when the program is compiled. When running an interpreted program, parsing and code generation happen each time the program is executed. In addition, if an application has real-time dependencies then having the garbage collector running at more or less random intervals may not be desirable. As you'll read in the next section some steps have been taken to reduce the difference in execution time between compiled and interpreted languages.

1.4.3 Virtual Machines

The advantages of interpretation over compilation are pretty significant. It turns out that one of the biggest advantages is the portability of programs. It's nice to know when you invest the time in writing a program that it will run the same on Linux, Microsoft Windows, Mac OS X, or some other operating system. This portability issue has driven a lot of research into making interpreted programs run as fast as compiled languages.

As discussed earlier in this chapter, the concept of a virtual machine has been around quite a while. A virtual machine is a program that provides insulation from the actual hardware and operating system of a machine while supplying a consistent implementation of a set of low-level instructions, often called bytecode. Figure 1.14 shows how a virtual machine sits on top of the operating system/CPU to act as this insulator.

There is no one specification for bytecode instructions. They are specific to the virtual machine being defined. Python has a virtual machine buried within the interpreter. Prolog is another interpreter that uses a virtual machine as part of its implementation. Some languages, like Java have taken this idea a step further. Java has a virtual machine that executes bytecode instructions as does Python. The creators of Java separated the virtual machine from the compiler. Instead of storing the bytecode instructions internally as in an interpreter, the Java compiler, called *javac*, compiles a Java source code program to a bytecode file. This file is not machine language so it cannot be executed directly on the hardware. It is a Java bytecode file which

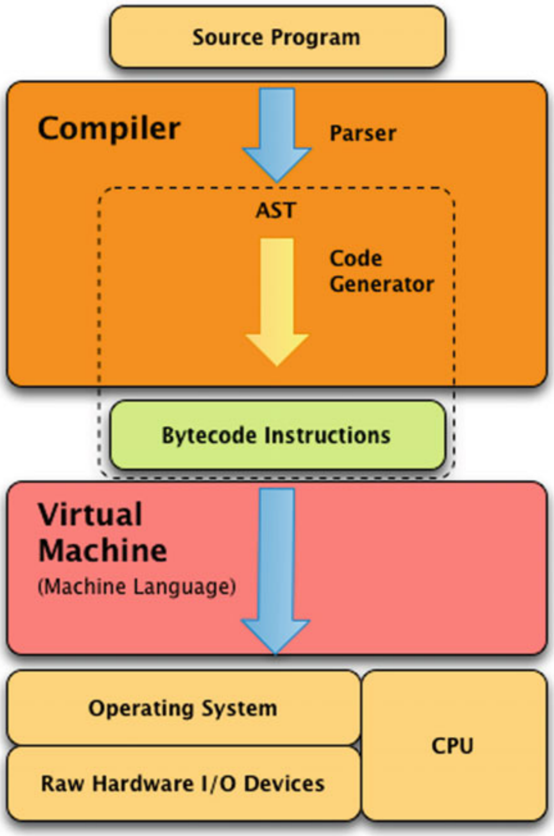


Fig. 1.14 Virtual machine implementation

is interpreted by the Java virtual machine, called *java* in the Java set of tools. Java bytecode files all end with a *.class* extension. You may have noticed these files at some point after compiling a Java program.

Programs written using a hybrid language like Java are compiled. However, the compiled bytecode program is interpreted. Source programs in the language are not interpreted directly. By adding this intermediate step the interpreter can be smaller and faster than traditional interpreters. Very little parsing needs to happen to read the program and executing the program is straightforward because each bytecode instruction usually has a simple implementation.

Languages that fall into this virtual machine category include Java, ML, Python, C#, Visual Basic .NET, JScript, and other .NET platform languages. You might notice that Standard ML and Python were included as examples of interpreted languages. Both ML and Python include interactive interpreters as well as the ability to compile and run low-level bytecode programs. Python bytecode files are named with a *.pyc* extension. Standard ML compiled files are named with a *-platform* as the last part of the compiled file name. In the case of Python and Standard ML the virtual machine is not a separate program. Both interpreters are written to recognize a bytecode file and execute it just like a source program.

Java and the .NET programming environments do not include interactive interpreters. The only way to execute programs with these platforms is to compile the program and then run the compiled program using the virtual machine. Programs written for the .NET platform run under Microsoft Windows and in some cases Linux. Microsoft submitted some of the .NET specifications to the ISO to allow third party software companies to develop support for .NET on other platforms. In theory all .NET programs are portable like Java, but so far implementations of the .NET framework are not as generally available as Java. The Java platform has been implemented and released on all major platforms. In fact, in November 2006 Sun, the company that created Java, announced they were releasing the Java Virtual Machine and related software under the GNU Public License to encourage further development of the language and related tools. Since then the rights to Java have now been purchased by Oracle where it continues to be supported.

Java and .NET language implementations maintain backwards compatibility of their virtual machines. This means that a program compiled for an earlier version of Java or .NET will continue to run on newer implementations of the language's virtual machine. In contrast, Python's virtual machine is regarded as an internal design issue and does not maintain backwards compatibility. A *.pyc* file compiled for one version of Python will not run on a newer version of Python. This distinction makes Python more of an interpreted language, while Java and .NET languages are truly virtual machine implementations.

Maintaining backwards compatibility of the virtual machine means that programmers can distribute application for Java and .NET implementations without releasing their source code. .NET and Java applications can be distributed while maintaining privacy of the source code. Since intellectual property is an important asset of companies, the ability to distribute programs in binary form is important. The development of virtual machines made memory management and program portability much easier in languages like Java, Standard ML, and the various .NET languages while also providing a means for programmers to distribute programs in binary format so source code could be kept private.

1.5 Types and Type Checking

Every programming language defines operations that can be used to transform data. Data transformation is the fundamental operation that is performed by all programming languages. Some programming languages mutate data to new values. Other languages transform data by building new values from old values. However the transformation takes place, these data transformation operations are defined for certain *types* of data. Not every transformation operation makes sense for every type of value. For instance, addition is an operation that makes sense for numbers, but does not make any sense for customers. How would you add two customers together and what would that mean?

Since programming languages define data transformation operations, they similarly define *types* to specify which operations make sense on which types of data. Types in programming languages include integers, booleans, real numbers (i.e. sometimes called floating point numbers), strings, lists, tuples, and user-defined types like customers. Transformation operations are defined operators on these values. The plus sign (e.g. +) often defines addition. String concatenation might also be denoted by the plus sign. Or it might be some other symbol.

One of the jobs of a programming language implementation is to determine which operation is meant when, for instance, the plus sign is written in a program. Does it mean the addition of two numbers, string concatenation, or is it an error because you can't add two customers together? Determining if the plus sign makes sense in the context of its operands (i.e. the two things being added together) is the job of a programming language implementation. More generally, the programming language implementation is responsible for checking that the operations performed on its data types are defined and the programming language is responsible for invoking the correct operation.

There are two different times that this type checking might occur. Some programming languages defer all type checking until the last possible second when the program is actually executing. When the next operation occurs, a programming language implementation may terminate a program and report that the next operation to be executed is not defined. This is called a *dynamically typed programming language*.

Python is a dynamically typed programming language. You don't find out that an operation is undefined until the operation is about to be executed. No earlier warning is given. When the code is executed, if you try to add two customers together, you find out that this has not been defined.

Other programming languages report any operation that is not defined before the program begins execution. In this case the programming language is statically typed. A *statically typed language* goes through a step during before execution where type checking is performed to see if the operations are defined for the given types of operands. This type checking step is performed in languages like Java and C++. These languages are statically typed.

An important facet of Standard ML is the strong type checking provided by the language. The type inference system, commonly called Hindley-Milner type inference, statically checks the types of all expressions and operations in the language. In addition, the type checking system is polymorphic, meaning that it handles types that may contain type variables. The polymorphic type checker is sound. It will never say a program is typed correctly when it is not. Interestingly, the type checker has also been proven complete, which means that all correctly typed programs will indeed pass the type checker. No correctly typed program will be rejected by the type checker. We expect soundness out of type checkers but completeness is much harder to prove and it has been proven for Standard ML.

There are trade-offs between statically and dynamically typed languages. Typically there is more overhead to programming with a statically typed language. Types in C++ and Java must be declared so that static type checking can be performed. But this is not always the case. Standard ML infers the types of most values in the language without requiring the types of its values to be declared.

Dynamically typed languages typically require less overhead in declaring values. In Python you don't declare the value of any object, except through the creation of that object. Writing

```
x = 6
print(x+x)
```

results in x referring to an integer value. Then printing $x + x$ will result in 12 being printed because $+$ is determined to be a valid operation on integers at run-time, as $x + x$ is computed.

The problem with dynamically typed languages comes from the lateness of determining if the operation is defined on the objects. If the operation is only determined to be valid right before it is executed, then every single line of a program must be tested to determine if the program is correct or not. While static typing tells you whether operations are defined or not before the program executes, dynamically typed languages don't help you with that. They only tell you that an operation is not defined if you actually try to execute that line of code.

So, which is better, dynamically or statically typed languages? It depends on the complexity of the program you are writing and its size. Static typing is certainly desirable if all other things are equal. But static typing typically does increase the work of a programmer up front. On the other hand, static typing is likely to decrease the amount of time you spend testing as evidenced by the Fox Project [10] at Carnegie Mellon.

1.6 Chapter Summary

The history of languages is fascinating and a lot more detail is available than was covered in this chapter. There are many great resources on the web where you can get more information. Use Google or Wikipedia and search for "History of your_favorite_language" as a place to begin. However, be careful. You can't believe everything you read on the web and that includes Wikipedia. While the web is a great source, you should always research your topic enough to independently verify the information you find there.

While learning new languages and studying programming language implementation it becomes important to understand models of computation. A compiler translates a high-level programming language into a lower level computation. These low-level computations are usually expressed in terms of machine language but not always. More important than the actual low-level language is the model of computation. Some models are based on register machines. Some models are based on stack machines. Still other models may be based on something entirely different. Chapters 3 and 4 explore stack-based virtual machines in much more detail.

The next chapter provides the foundations for understanding how the syntax of a language is formally defined by a grammar. Then chapter three introduces a Python Virtual Machine implementation called JCoCo. JCoCo is an interpreter of Python bytecode instructions. Chapter three introduces assembly language programming using JCoCo, providing some insight into how programming languages are implemented.

Subsequent chapters in the book will again look at language implementation to better understand the languages you are learning, their strengths and weaknesses. While learning these languages you will also be implementing a compiler for a high level functional language called *Small* which is a robust subset of Standard ML. This will give you even more insight into language implementation and knowledge of how to use these languages to solve problems.

Finally, in the last two chapters of this text, you will learn about type checking and type inference using Prolog, a language that is well-suited to logic problems like type inference. Learning how to use Prolog and implement a type checker is a great way to cap off a text on programming languages and language implementation.

A great way to summarize the rest of this text is to see it moving from very prescriptive approaches to programming to very descriptive approaches to programming. The word *prescriptive* means that you dwell on details, thinking very carefully about the details of what you are writing. For instance, in a prescriptive approach you might ask yourself, how do you set things up to invoke a particular type of instruction? In contrast, *descriptive* programming relies on programmers describing relationships between things. Functional programming languages, to some extent, and logic programming languages employ this descriptive approach to programming. Read on to begin the journey from prescriptive to descriptive programming!

1.7 Review Questions

1. What are the three ways of thinking about programming, often called programming paradigms?
2. Name at least one language for each of the three methods of programming described in the previous question.
3. Name one person who had a great deal to do with the development of the imperative programming model. Name another who contributed to the functional model. Finally, name a person who was responsible for the development of the logic model of programming.
4. What are the primary characteristics of each of the imperative, functional, and logic models?
5. Who are recognized as the founders of each of the languages this text covers: Java, C++, Python, Standard ML, and Prolog?
6. Name a language, other than Python, C++, or Java, that is imperative object-oriented in nature.
7. Name a language besides Standard ML, that is a functional programming language.

8. What other logic programming languages are there other than Prolog? You might have to get creative on this one.
9. Why is compiling a program preferred over interpreting a program?
10. Why is interpreting a program preferred over compiling a program?
11. What benefits do virtual machine languages have over interpreted languages?
12. What is a bytecode program? Name two languages that use bytecode in their implementation.
13. Why are types important in a programming language?
14. What does it mean for a programming language to be dynamically typed?
15. What does it mean for a programming language to be statically typed?

1.8 Solutions to Practice Problems

These are solutions to the practice problems. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

1.8.1 Solution to Practice Problem 1.1

1. The origins of the three models are the Turing Machine, the λ-calculus, and propositional and predicate logic.
2. Alan Turing as a PhD student of Alonzo Church. Alan Turing developed the Turing Machine and Alonzo Church developed the λ-calculus to answer prove there were somethings that are not computable. They later proved the two approaches were equivalent in their power to express computation.
3. Both von Neumann and Turing contributed to the idea of a stored-program computer.
4. Backus developed BNF notation which was used in the development of Algol 60.
5. 1936 was a big year for Computer Science.
6. So was 1946. That was the year ENIAC was unveiled. Eckert and Mauchly designed and built ENIAC.
7. The problems in Mathematics were growing complex enough that many mathematicians were developing models and languages for expressing their algorithms. This was one of the driving factors in the development of computers and Computer Science as a discipline.

1.8.2 Solution to Practice Problem 1.2

1. The run-time stack, global memory, and the heap are the three divisions of data memory.
2. Data on the heap is created at run-time.

3. An activation record holds information like local variables, the program counter, the stack pointer, and other state information necessary for a function invocation.
4. An activation record is created each time a function is called.
5. An activation record is deleted when a function returns.
6. The primary goal of imperative, object-oriented programming is to update memory by updating variables and/or objects as the program executes. The primary operation is memory updates.

1.8.3 Solution to Practice Problem 1.3

1. Functional languages include Standard ML, Lisp, Haskell, and Scheme.
2. In the imperative model the primary operation revolves around updating memory (the assignment statement). In the functional model the primary operation is function application.
3. The functional model emphasizes immutable data. However, some imperative languages have some immutable data as well. For instance, Java strings are immutable.

1.8.4 Solution to Practice Problem 1.4

1. You never write a program in Prolog. You write a database of rules in Prolog that tell the single Prolog program (depth first search) how to proceed.
2. The programmer provides a database of facts and predicates that tell Prolog about a problem. In Prolog the programmer describes the problem instead of programming the solution.

1.8.5 Solution to Practice Problem 1.5

1. C++ was invented by Bjourne Stroustrup. C was created by Dennis Ritchie. Standard ML was primarily designed by Robin Milner. Prolog was designed by Alain Colmerauer and Philippe Roussel with the assistance of Robert Kowalski. Python was created by Guido van Rossum. Java was the work of the Green team and James Gosling.
2. Standard ML and Prolog were both designed as languages for automated theorem proving first. Then they became general purpose programming languages later.
3. Both Python and Prolog run on virtual machine implementations. Python's virtual machine is internal to the interpreter. Prolog's virtual machine is called WAM (Warren Abstract Machine).
4. Standard ML is influenced by Lisp, Pascal, and Algol.

Syntax 2

Once you've learned to program in one language, learning a similar programming language isn't all that hard. But, understanding just how to write in the new language takes looking at examples or reading documentation to learn its details. In other words, you need to know the mechanics of putting a program together in the new language. Are the semicolons in the right places? Do you use *begin...end* or do you use curly braces (i.e. { and })? Learning how a program is put together is called learning the syntax of the language. Syntax refers to the words and symbols of a language and how to write the symbols down in some meaningful order.

Semantics is the word that is used when deriving meaning from what is written. The semantics of a program refers to what the program will do when it is executed. Informally it is much easier to say what a program does than to describe the syntactic structure of the program. However, syntax is a lot easier to formally describe than semantics. In either case, if you are learning a new language, you need to learn something about both the syntax and semantics of the language.

2.1 Terminology

Once again, the *syntax* of a programming language determines the well-formed or grammatically correct programs of the language. *Semantics* describes how or whether such programs will execute.

- *Syntax* is how programs look
- *Semantics* is how programs work

Many questions we might like to ask about a program either relate to the syntax of the language or to its semantics. It is not always clear which questions pertain to syntax and which pertain to semantics. Some questions may concern semantic issues that can be determined statically, meaning before the program is run. Other semantic issues may be dynamic issues, meaning they can only be determined at run-time. The difference between static semantic issues and syntactic issues is sometimes a difficult distinction to make.

The code

```
a=b+c;
```

is correct syntax in many languages. But is it a correct C++ statement?

1. Do *b* and *c* have values?
2. Have *b* and *c* been declared as a type that allows the + operation? Or, do the values of *b* and *c* support the + operation?
3. Is *a* assignment compatible with the result of the expression $b + c$?
4. Does the assignment statement have the proper form?

There are lots of questions that need to be answered about this assignment statement. Some questions could be answered sooner than others. When a C++ program is compiled it is translated from C++ to machine language as described in the previous chapter. Questions 2 and 3 are issues that can be answered when the C++ program is compiled. However, the answer to the first question might not be known until the C++ program executes in some cases. The answers to questions 2 and 3 can be answered at *compile-time* and are called *static* semantic issues. The answer to question 1 is a *dynamic* issue and is probably not determinable until run-time. In some circumstances, the answer to question 1 might also be a static semantic issue. Question 4 is definitely a syntactic issue.

Unlike the dynamic semantic issues, the correct syntax of a program is statically determinable. Said another way, determining a syntactically valid program can be accomplished without running the program. The syntax of a programming language is specified by a grammar. But before discussing grammars, the parts of a grammar must be defined. A *terminal* or *token* is a symbol in the language.

- C++, Java, and Python terminals: *while*, *for*, *(*, *;*, *5*, *b*
- Type names like *int* and *string*

Keywords, types, operators, numbers, identifiers, etc. are all tokens or terminals in a language.

A *syntactic category* or *nonterminal* is a set of phrases, or strings of tokens, that will be defined in terms of symbols in the language (terminal and nonterminal symbols).

- C++, Java, or Python nonterminals: <statement>, <expression>, <if-statement>, etc.
- Syntactic categories define parts of a program like statements, expressions, declarations, and so on.

A *metalanguage* is a higher-level language used to specify, discuss, describe, or analyze another language. English is used as a metalanguage for describing programming languages, but because of the ambiguities in English, more formal metalanguages have been developed. The next section describes a formal metalanguage for describing programming language syntax.

2.2 Backus Naur Form (BNF)

Backus Naur Format (i.e. BNF) is a formal metalanguage for describing language syntax. The word *formal* is used to indicate that BNF is unambiguous. Unlike English, the BNF language is not open to our own interpretations. There is only one way to read a BNF description.

BNF was used by John Backus to describe the syntax of Algol in 1963. In 1960, John Backus and Peter Naur, a computer magazine writer, had just attended a conference on Algol. As they returned from the trip it became apparent that they had very different views of what Algol would look like. As a result of this discussion, John Backus worked on a method for describing the grammar of a language. Peter Naur slightly modified it. The notation is called BNF, or Backus Naur Form or sometimes Backus Normal Form. BNF consists of a set of rules that have this form:

<syntactic category> ::= a string of terminals and nonterminals

The symbol ::= can be read as *is composed of* and means the syntactic category is the set of all items that correspond to the right hand side of the rule.

Multiple rules defining the same syntactic category may be abbreviated using the | character which can be read as "or" and means set union. That is the entire language. It's not a very big metalanguage, but it is powerful.

2.2.1 BNF Examples

Here are a couple BNF examples from Java.

```
<primitive-type> ::= boolean
<primitive-type> ::= char
```

BNF syntax is often abbreviated when there are multiple similar rules like these primitive type rules. Whether abbreviated or not, the meaning is the same.

```
<primitive-type> ::= boolean | char | byte | short | int | long | float | ...
<argument-list> ::= <expression> | <argument-list> , <expression>
<selection-statement> ::=
  if ( <expression> ) <statement> |
  if ( <expression> ) <statement> else <statement> |
  switch ( <expression> ) <block>
<method-declaration> ::=
  <modifiers> <type-specifier> <method declarator> <throws-clause> <method-body> |
  <modifiers> <type-specifier> <method-declarator> <method-body> |
  <type-specifier> <method-declarator> <throws-clause> <method-body> |
  <type-specifier> <method-declarator> <method-body>
```

This description can be described in English: *The set of method declarations is the union of the sets of method declarations that explicitly throw an exception with those that don't explicitly throw an exception with or without modifiers attached to their definitions*. The BNF is much easier to understand and is not ambiguous like this English description.

2.2.2 Extended BNF (EBNF)

Since a BNF description of the syntax of a programming language relies heavily on recursion to provide lists of items, many definitions use these extensions:

1. **item?** or **[item]** means the item is optional.
2. **item*** or **{item}** means zero or more occurrences of an item are allowable.
3. **item+** means one or more occurrences of an item are allowable.
4. Parentheses may be used for grouping

2.3 Context-Free Grammars

A BNF is a way of describing the grammar of a language. Most interesting grammars are context-free, meaning that the contents of any syntactic category in a sentence are not dependent on the context in which it is used. A context-free grammar is defined as a four tuple:

$$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, \mathcal{S})$$

where

- $\mathcal{N}$ is a set of symbols called nonterminals or syntactic categories.
- $\mathcal{T}$ is a set of symbols called terminals or tokens.
- $\mathcal{P}$ is a set of productions of the form $n \rightarrow \alpha$ where $n \in \mathcal{N}$ and $\alpha \in \{\mathcal{N} \cup \mathcal{T}\}^*$.
- $\mathcal{S} \in \mathcal{N}$ is a special nonterminal called the start symbol of the grammar.

Informally, a context-free grammar is a set of nonterminals and terminals. For each nonterminal there are one or more productions with strings of zero or more non-terminals and terminals on the right hand side as described in the BNF description. There is one special nonterminal called the start symbol of the grammar.

2.3.1 The Infix Expression Grammar

A context-free grammar for infix expressions can be specified as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$\begin{aligned}&E \rightarrow E \ + \ T \mid E \ - \ T \mid T\\&T \rightarrow T \ * \ F \mid T \ / \ F \mid F\\&F \rightarrow (\ E \ ) \mid identifier \mid number\end{aligned}$$

2.4 Derivations

A *sentence* of a grammar is a string of tokens from the grammar. A sentence belongs to the language of a grammar if it can be derived from the grammar. This process is called constructing a derivation. A *derivation* is a sequence of sentential forms that starts with the start symbol of the grammar and ends with the sentence you are trying to derive. A *sentential form* is a string of terminals and nonterminals from the grammar. In each step in the derivation, one nonterminal of a sentential form, call it A, is replaced by a string of terminals and nonterminals, β, where $A \rightarrow \beta$ is a production in the grammar. For a grammar, G, the language of G is the set of sentences that can be derived from G and is usually written as *L(G)*.

2.4.1 A Derivation

Here we prove that the expression $(5*x)+y$ is a member of the language defined by the grammar given in Sect. 2.3.1 by constructing a derivation for it. The derivation begins with the start symbol of the grammar and ends with the sentence.

$$\begin{aligned}&\underline{E} \Rightarrow \underline{E} + T \Rightarrow \underline{T} + T \Rightarrow \underline{F} + T \Rightarrow (\underline{E}) + T \Rightarrow (\underline{T}) + T \Rightarrow (\underline{T} * F) + T\\&\Rightarrow (\underline{F} * F) + T \Rightarrow (5 * \underline{F}) + T \Rightarrow (5 * x) + \underline{T} \Rightarrow (5 * x) + \underline{F} \Rightarrow (5 * x) + y\end{aligned}$$

Each step is a sentential form. The underlined nonterminal in each sentential form is replaced by the right hand side of a production for that nonterminal. The derivation proceeds from the start symbol, E, to the sentence $(5 * x) + y$. This proves that $(5 * x) + y$ is in the language *L(G)* as G is defined in Sect. 2.3.1.

Practice 2.1 Construct a derivation for the infix expression $4 + (a - b) * x$.
You can check your answer(s) in Section 2.17.1.

2.4.2 Types of Derivations

A sentence of a grammar is *valid* if there exists at least one derivation for it using the grammar. There are typically many different derivations for a particular sentence of a grammar. However, there are two derivations that are of some interest to us in understanding programming languages.

- Left-most derivation - Always replace the left-most nonterminal when going from one sentential form to the next in a derivation.
- Right-most derivation - Always replace the right-most nonterminal when going from one sentential form to the next in a derivation.

The derivation of the sentence $(5 * x) + y$ in Sect. 2.4.1 is a left-most derivation. A right-most derivation for the same sentence is:

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + y \Rightarrow T + y \Rightarrow F + y \Rightarrow (E) + y \Rightarrow (T) + y$$
$$\Rightarrow (T * F) + y \Rightarrow (T * x) + y \Rightarrow (F * x) + y \Rightarrow (5 * x) + y$$

Practice 2.2 Construct a right-most derivation for the expression $x * y + z$.
You can check your answer(s) in Section 2.17.2.

2.4.3 Prefix Expressions

Infix expressions are expressions where the operator appears between the operands. Another type of expression is called a prefix expression. In prefix expressions the operator appears before the operands. The infix expression $4 + (a - b) * x$ would be written $+4 * -abx$ as a prefix expression. Prefix expressions are in some sense simpler than infix expressions because we don't have to worry about the precedence of operators. The operator precedence is determined by the order of operations in the expression. Because of this, parentheses are not needed in prefix expressions.

2.4.4 The Prefix Expression Grammar

A context-free grammar for prefix expressions can be specified as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E\}$
$\mathcal{T} = \{identifier, number, +, -, *, /\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow +\ E\ E \mid -\ E\ E \mid *\ E\ E \mid /\ E\ E \mid identifier \mid number$$

Practice 2.3 Construct a left-most derivation for the prefix expression $+4 * -abx$.

You can check your answer(s) in Section 2.17.3.

2.5 Parse Trees

A grammar, *G*, can be used to build a tree representing a sentence of *L(G)*, the language of the grammar *G*. This kind of tree is called a *parse tree*. A parse tree is another way of representing a sentence of a given language. A parse tree is constructed with the start symbol of the grammar at the root of the tree. The children of each node in the tree must appear on the right hand side of a production with the parent on the left hand side of the same production. A program is syntactically valid if there is a parse tree for it using the given grammar.

While there are typically many different derivations of a sentence in a language, there is only one parse tree. This is true as long as the grammar is not ambiguous. In fact that's the definition of ambiguity in a grammar. A grammar is *ambiguous* if and only if there is a sentence in the language of the grammar that has more than one parse tree.

The parse tree for the sentence derived in Sect. 2.4.1 is depicted in Fig. 2.1. Notice the similarities between the derivation and the parse tree.

Practice 2.4 What does the parse tree look like for the right-most derivation of $(5 * x) + y$?

You can check your answer(s) in Section 2.17.4.

Practice 2.5 Construct a parse tree for the infix expression $4 + (a - b) * x$.

HINT: What has higher precedence, "+" or "$*$"? The given grammar automatically makes "$*$" have higher precedence. Try it the other way and see why!

You can check your answer(s) in Section 2.17.5.

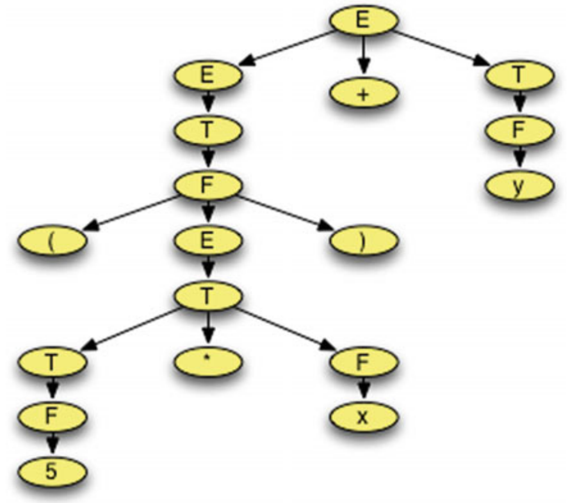


Fig. 2.1 A parse tree

Practice 2.6 Construct a parse tree for the prefix expression $+4 * -abx$.
You can check your answer(s) in Section 2.17.6.

2.6 Abstract Syntax Trees

There is a lot of information in a parse tree that isn't really needed to capture the meaning of the program that it represents. An abstract syntax tree is like a parse tree except that non-essential information is removed. More specifically,

- Nonterminal nodes in the tree are replaced by nodes that reflect the part of the sentence they represent.
- Unit productions in the tree are collapsed.

For example, the parse tree from Fig. 2.1 can be represented by the abstract syntax tree in Fig. 2.2. The abstract syntax tree eliminates all the unnecessary information and leaves just what is essential for evaluating the expression. Abstract syntax trees, often abbreviated ASTs, are used by compilers while generating code and may be used by interpreters when running your program. Abstract syntax trees throw away superfluous information and retain only what is essential to allow a compiler to generate code or an interpreter to execute the program.

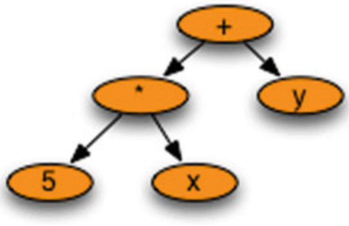


Fig. 2.2 An AST

Practice 2.7 Construct an abstract syntax tree for the expression $4+(a-b)*x$.
You can check your answer(s) in Section 2.17.7.

2.7 Lexical Analysis

The syntax of modern programming languages are defined via grammars. A grammar, because it is a well-defined mathematical structure, can be used to construct a program called a parser. A language implementation, like a compiler or an interpreter, has a parser that reads the program from the source file. The parser reads the tokens, or terminals, of a program and uses the language's grammar to check to see if the stream of tokens form a syntactically valid program.

For a parser to do its job, it must be able to get the stream of tokens from the source file. Forming tokens from the individual characters of a source file is the job of another program often called a tokenizer, or scanner, or lexer. Lex is the Latin word for *word*. The words of a program are its tokens. In programming language implementations a little liberty is taken with the definition of *word*. A *word* is any terminal or token of a language. It turns out that the tokens of a language can be described by another language called the language of regular expressions.

2.7.1 The Language of Regular Expressions

The language of regular expression is defined by a context-free grammar. The context-free grammar for regular expressions is $RE = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, K, F\}$
$\mathcal{T} = \{character, *, +, ., (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T.K \mid K \\
K &\rightarrow F* \mid F \\
F &\rightarrow character \mid (E)
\end{aligned}
$$

The + operator is the *choice* operator, meaning either E or T, but not both. The dot operator means that T is *followed by* K. The $*$ operator, called *Kleene Star* for the mathematician that first defined it, means *zero or more occurrences* of F. The grammar defines the precedence of these operators. Kleene star has the highest precedence followed by the dot operator, followed by the choice operator. At its most primitive level, a regular expression may be just a single character.

Frequently, a choice between many different characters may be abbreviated with some sensible name. For instance, *letter* may be used to abbreviate $A + B + \cdots + Z + a + b + \cdots z$ and digit may abbreviate $0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$. Usually these abbreviations are specified explicitly before the regular expression is given.

The tokens of the infix grammar are identifier, number, $+$, $-$, $*$, /, (, and). For brevities sake, assume that letter and digit have the usual definitions. We'll also put each operator character in single quotes so as not to confuse them with the metalanguage. Then, these tokens might be defined by the regular expression

letter.letter* + digit.digit* + '+' + '−' + '*' + '/' + '(' + ')'

From this regular expression specification a couple of things come to light. Identifiers must be at least one character long, but can be as long as we wish them to be. Numbers are only non-negative integers in the infix expression language. Floating point numbers cannot be specified in the language as the tokens are currently defined.

Practice 2.8 Define a regular expression so that negative and non-negative integers can both be specified as tokens of the infix expression language.
You can check your answer(s) in Section 2.17.8.

2.7.2 Finite State Machines

A finite state machine is a mathematical model that accepts or rejects strings of characters for some regular expression. A finite state machine is often called a finite state automaton. The word *automaton* is just another word for machine. Every regular expression has at least one finite state machine and vice versa, every finite state machine has at least one matching regular expression. In fact, there is an algorithm that given any regular expression can be used to construct a finite state machine for it.

Formally a finite state automata is defined as follows.

$M = (\Sigma, S, F, s_0, \delta)$ where Σ (pronounced sigma) is the input alphabet (the characters understood by the machine), S is a set of states, F is a subset of S usually written as $F \subseteq S$, s_0 is a special state called the start state, and δ (pronounced delta) is a function that takes as input an alphabet symbol and a state and returns a new state. This is usually written as $\delta : \Sigma \times S \rightarrow S$.

A finite state machine has a current state which initially is the start state. The machine starts in the start state and reads characters one at a time. As characters are read, the finite state machine changes state. Each state has transitions to other states based on the last character read. Each time the machine transitions to a new state, another character is read from the stream of characters.

After reading all the characters of a token, if the current state is in the set of final states, F, then the token is accepted by the finite state machine. Otherwise, it is rejected. Finite state machines are typically represented graphically by drawing the states, transitions, start state, and final states. States in a graphical representation are depicted as nodes in a graph. The start state has an arrow going into it with nothing at the back side of the arrow. The transitions are represented as arrows going from one state to another and are labelled with the characters that trigger the given transition. Finally, final or accepting states are denoted with a double circle.

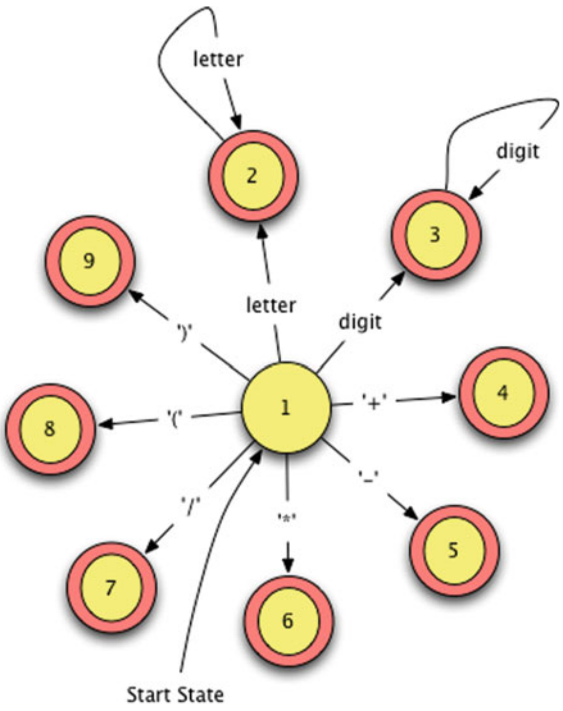


Fig. 2.3 A finite state machine

Figure 2.3 depicts a finite state machine for the language of infix expression tokens. The start state is 1. Each of states 2 through 9 are accepting states, denoted with a double circle. State 2 accepts identifier tokens. State 3 accepts number tokens. States 4 to 9 accept operators and the parenthesis tokens. The finite state machine accepts one token at a time. For each new token, the finite state machine starts over in state 1.

If, while reading a token, an unexpected character is read, then the stream of tokens is rejected by the finite state machine as invalid. Only valid strings of characters are accepted as tokens. Characters like spaces, tabs, and newline characters are not recognized by the finite state machine. The finite state machine only responds with *yes* the string of tokens is in the language accepted by the machine or *no* it is not.

2.7.3 Lexer Generators

It is relatively easy to construct a lexer by writing a regular expression, drawing a finite state machine, and then writing a program that mimics the finite state machine. However, this process is largely the same for all programming languages so there are tools that have been written to do this for us. Typically these tools are called lexer generators. To use a lexer generator you must write regular expressions for the tokens of the language and provide these to the lexer generator.

A lexer generator will generate a lexer program that internally uses a finite state machine like the one pictured in Fig. 2.3, but instead of reporting *yes* or *no*, for each token the lexer will return the string of characters, called the *lexeme* or *word* of the token, along with a classification of the token. So, identifiers are categorized as *identifier* tokens while '+' is categorized as an *add* token.

The *lex* tool is an example of a lexical generator for the C language. If you are writing an interpreter or compiler using C as the implementation language, then you would use *lex* or a similar tool to generate your lexer. *lex* was a tool included with the original *Unix* operating system. The Linux alternative is called *flex*. Java, Python, Standard ML, and most programming languages have equivalent available lexer generators.

2.8 Parsing

Parsing is the process of detecting whether a given string of tokens is a valid sentence of a grammar. Every time you compile a program or run a program in an interpreter the program is first parsed using a parser. When a parser isn't able to parse a program the programmer is told there is a *syntax error* in the program. A *parser* is a program that given a sentence, checks to see if the sentence is a member of the language of the given grammar. A parser usually does more than just answer *yes* or *no*. A parser frequently builds an abstract syntax tree representation of the source program. There are two types of parsers that are commonly constructed.

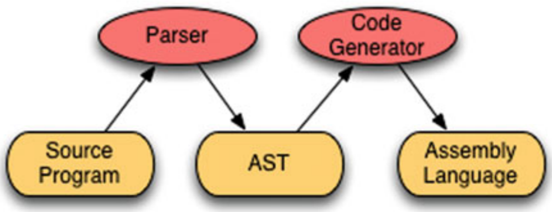


Fig. 2.4 Parser data flow

- A *top-down parser* starts with the root of the parse tree.
- A *bottom-up parser* starts with the leaves of the parse tree.

Top-down and bottom-up parsers check to see if a sentence belongs to a grammar by constructing a derivation for the sentence, using the grammar. A parser either reports success (and possibly returns an abstract syntax tree) or reports failure (hopefully with a nice error message). The flow of data is pictured in Fig. 2.4.

2.9 Top-Down Parsers

Top-down parsers are generally written by hand. They are sometimes called recursive descent parsers because they can be written as a set of mutually recursive functions. A top-down parser performs a left-most derivation of the sentence (i.e. source program).

A top-down parser operates by (possibly) looking at the next token in the source file and deciding what to do based on the token and where it is in the derivation. To operate correctly, a top-down parser must be designed using a special kind of grammar called an LL(1) grammar. An LL(1) grammar is simply a grammar where the next choice in a left-most derivation can be deterministically chosen based on the current sentential form and the next token in the input. The first *L* refers to scanning the input from left to right. The second *L* signifies that while performing a left-most derivation, there is only *1* symbol of lookahead that is needed to make the decision about which production to choose next in the derivation.

2.9.1 An LL(1) Grammar

The grammar for prefix expressions is LL(1). Examine the prefix expression grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E\}$
$\mathcal{T} = \{identifier, number, +, -, *, /\}$
$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow + \ E \ E \mid - \ E \ E \mid * \ E \ E \mid / \ E \ E \mid identifier \mid number$$

While constructing any derivation for a sentence of this language, the next production chosen in a left-most derivation is going to be obvious because the next token of the source file must match the first terminal in the chosen production.

2.9.2 A Non-LL(1) Grammar

Some grammars are not LL(1). The grammar for infix expressions is not LL(1). Examine the infix expression grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$\begin{aligned} &E \rightarrow E \ + \ T \mid E \ - \ T \mid T \\ &T \rightarrow T \ * \ F \mid T \ / \ F \mid F \\ &F \rightarrow (\ E \ ) \mid identifier \mid number \end{aligned}$$

Consider the infix expression $5 * 4$. A left-most derivation of this expression would be

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow 5 * F \Rightarrow 5 * 4$$

Consider looking at only the 5 in the expression. We have to choose whether to use the production $E \rightarrow E \ + \ T$ or $E \rightarrow T$. We are only allowed to look at the 5 (i.e. we can't look beyond the 5 to see the multiplication operator). Which production do we choose? We can't decide based on the 5. Therefore the grammar is not LL(1).

Just because this infix expression grammar is not LL(1) does not mean that infix expressions cannot be parsed using a top-down parser. There are other infix expression grammars that are LL(1). In general, it is possible to transform any context-free grammar into an LL(1) grammar. It is possible, but the resulting grammar is not always easily understandable.

The infix grammar given in Sect. 2.9.2 is left recursive. That is, it contains the production $E \rightarrow E + T$ and another similar production for terms in infix expressions. These rules are left recursive. Left recursive rules are not allowed in LL(1) grammars. A left recursive rule can be eliminated in a grammar through a straightforward transformation of its production.

Common prefixes in the right hand side of two productions for the same nonterminal are also not allowed in an LL(1) grammar. The infix grammar given in Sect. 2.9.2 does not contain any common prefixes. Common prefixes can be eliminated by introducing a new nonterminal to the grammar, replacing all common prefixes with the new nonterminal, and then defining one new production so the new nonterminal is composed of the common prefix.

2.9.3 An LL(1) Infix Expression Grammar

The following grammar is an LL(1) grammar for infix expressions. $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, RestE, T, RestT, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

$$\begin{aligned}
&E \rightarrow T\ RestE\\
&RestE \rightarrow +\ T\ RestE \mid -\ T\ RestE \mid \epsilon\\
&T \rightarrow F\ RestT\\
&RestT \rightarrow *\ F\ RestT \mid /\ F\ RestT \mid \epsilon\\
&F \rightarrow (\ E\ ) \mid identifier \mid number
\end{aligned}$$

In this grammar the ϵ (pronounced epsilon) is a special symbol that denotes an empty production. An empty production is a production that does not consume any tokens. Empty productions are sometimes convenient in recursive rules.

Once common prefixes and left recursive rules are eliminated from a context-free grammar, the grammar will be LL(1). However, this transformation is not usually performed because there are more convenient ways to build a parser, even for non-LL(1) grammars.

Practice 2.9 Construct a left-most derivation for the infix expression $4 + (a - b) * x$ using the grammar in Sect. 2.9.3, proving that this infix expression is in L(G) for the given grammar.

You can check your answer(s) in Section 2.17.9.

2.10 Bottom-Up Parsers

While the original infix expression language is not *LL(1)* it is *LALR(1)*. In fact, most grammars for programming languages are LALR(1). The *LA* stands for *look ahead* with the *1* meaning just one symbol of look ahead. The *LR* refers to scanning the input from left to right while constructing a right-most derivation. A bottom-up parser constructs a right-most derivation of a source program in reverse. So, an LALR(1) parser constructs a reverse right-most derivation of a program.

Building a bottom-up parser is a somewhat complex task involving the computation of item sets, look ahead sets, a finite state machine, and a stack. The finite state machine and stack together are called a *pushdown automaton*. The construction of the pushdown automaton and the look ahead sets are calculated from the grammar. Bottom-up parsers are not usually written by hand. Instead, a parser generator is used

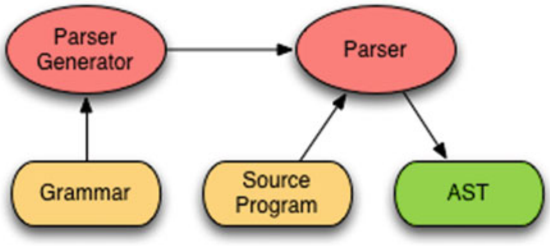


Fig. 2.5 Parser generator data flow

to generate the parser program from the grammar. A parser generator is a program that is given a grammar and builds a parser for the language of the grammar by constructing the pushdown automaton and lookahead sets needed to parse programs in the language of the grammar.

The original parser generator for Unix was called *yacc*, which stood for *yet another compiler compiler* since it was a compiler for grammars that produced a parser for a language. Since a parser is part of a compiler, *yacc* was a compiler compiler. The Linux version of yacc is called *Bison*. Hopefully you see the pun that was used in naming it *Bison*. The *Bison* parser generator generates a parser for compilers implemented in C, C++, or Java. There are versions of yacc for other languages as well. Standard ML has a version called ml-yacc for compilers implemented in Standard ML. ML-yacc is introduced and used in Chap. 6.

Parser generators like Bison produce what is called a bottom-up parser because the right-most derivation is constructed in reverse. In other words, the derivation is done from the bottom up. Usually, a bottom-up parser is going to return an AST representing a successfully parsed source program. Figure 2.5 depicts the dataflow in an interpreter or compiler. The parser generator is given a grammar and runs once to build the parser. The generated parser runs each time a source program is parsed.

A bottom-up parser parses a program by constructing a reverse right-most derivation of the source code. As the reverse derivation proceeds the parser *shifts* tokens from the input onto the stack of the pushdown automaton. Then at various points in time it reduces by deciding, based on the look ahead sets, that a reduction is necessary.

2.10.1 Parsing an Infix Expression

Consider the grammar for infix expressions as $G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E, T, F\}$
$\mathcal{T} = \{identifier, number, +, -, *, /, (,)\}$
$\mathcal{P}$ is defined by the set of productions

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow number$
(6) $F \rightarrow (E)$

Now assume we are parsing the expression $5 * 4 + 3$. A right-most derivation for this expression is as follows.

$$E \Rightarrow E+T \Rightarrow E+F \Rightarrow E+3 \Rightarrow T+3 \Rightarrow T*F+3 \Rightarrow T*4+3 \Rightarrow F*4+3 \Rightarrow 5*4+3$$

A bottom-up parser does a right-most derivation in reverse using a pushdown automaton. It can be useful to look at the stack of the pushdown automaton as it parses the expression as pictured in Fig. 2.6. In step A the parser is beginning. The dot to the left of the 5 indicates the parser has not yet processed any tokens of the source program and is looking at the 5. The stack is empty. From step A to step B one token, the 5 is shifted onto the stack. From step B to C the parser looks at the multiplication operator and realizes that a *reduction* using rule 5 of the grammar must be performed. It is called a reduction because the production is employed in reverse order. The reduction pops the right hand side of rule 5 from the stack and replaces it with the nonterminal F. If you look at this derivation in reverse order, the first step is to replace the number 5 with F.

The rest of the steps of parsing the source program follow the right-most derivation either shifting tokens onto the stack or reducing using rules of the grammar. In step O the entire source has been parsed, the stack is empty, and the source program is accepted as a valid program. The actions taken while parsing include shifting and reducing. These are the two main actions of any bottom-up parser. In fact, bottom-up parsers are often called shift-reduce parsers.

Practice 2.10 For each step in Fig. 2.6, is there a shift or reduce operation being performed? If it is a reduce operation, then what production is being reduced? If it is a shift operation, what token is being shifted onto the stack?

You can check your answer(s) in Section 2.17.10.

Practice 2.11 Consider the expression $(6 + 5) * 4$. What are the contents of the pushdown automaton's stack as the expression is parsed using a bottom-up parser? Show the stack after each shift and each reduce operation.

You can check your answer(s) in Section 2.17.11.

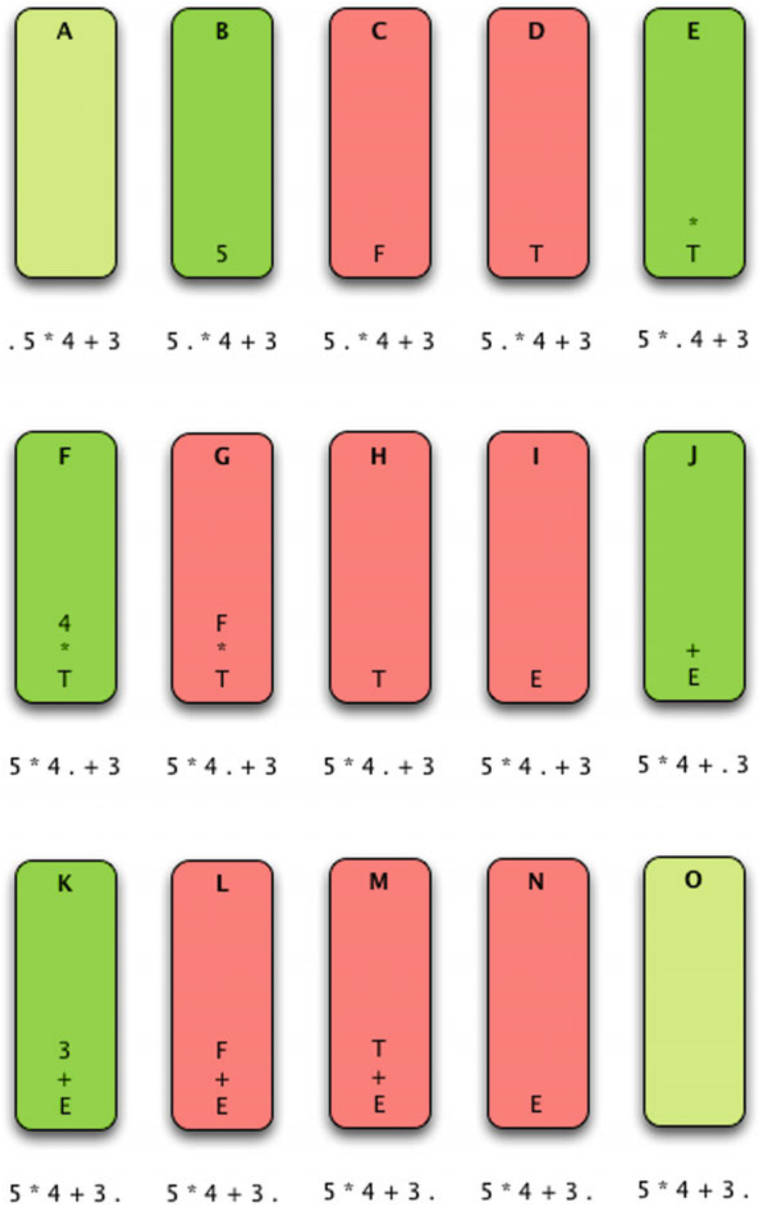


Fig. 2.6 A pushdown automaton stack

2.11 Ambiguity in Grammars

A grammar is ambiguous if there exists more than one parse tree for a given sentence of the language. In general, ambiguity in a grammar is a bad thing. However, some ambiguity may be allowed by parser generators for LALR(1) languages.

A classic example of ambiguity in languages arises from nested if-then-else statements. Consider the following Pascal statement:

```
if a<b then
  if b<c then
    writeln("a<c")
else
  writeln("?")
```

Which *if* statement does the *else* go with? It's not entirely clear. The BNF for an if-then-else statement might look something like this.

```
<statement> ::= if <expression> then <statement> else <statement>
              | if <expression> then <statement>
              | writeln ( <expression> )
```

The recursive nature of this rule means that if-then-else statements can be arbitrarily nested. Because of this recursive definition, the *else* in this code is dangling. That is, it is unclear if it goes with the first or second *if* statement.

When a bottom-up parser is generated using this grammar, the parser generator will detect that there is an ambiguity in the grammar. The problem manifests itself as a conflict between a shift and a reduce operation. The first rule says when looking at an *else* keyword the parser should *shift*. The second rule says when the parser is looking at an *else* it should *reduce*. To resolve this conflict there is generally a way to specify whether the generated parser should shift or reduce. The default action is usually to shift and that is what makes the most sense in this case. By shifting, the *else* would go with the nearest *if* statement. This is the normal behavior of parsers when encountering this if-then-else ambiguity.

2.12 Other Forms of Grammars

As a computer programmer you will likely learn at least one new language and probably a few during your career. New application areas frequently cause new languages to be developed to make programming applications in that area more convenient. Java, JavaScript, and ASP.NET are three languages that were created because of the world wide web. Ruby and Perl are languages that have become popular development languages for database and server side programming. Objective C is another language made popular by the rise of iOS App programming for Apple products. A recent trend in programming languages is to develop domain specific languages for particular embedded platforms.

Programming language manuals contain some kind of reference that describes the constructs of the language. Many of these reference manuals give the grammar of the language using a variation of a context free grammar. Examples include CBL (Cobol-like) grammars, syntax diagrams, and as we have already seen, BNF and EBNF. All these syntax metalanguages share the same features as grammars. They all have some way of defining parts of a program or syntactic categories and they all have a means of defining a language through recursively defined productions. The definitions, concepts, and examples provided in this chapter will help you understand a language reference when the time comes to learn a new language.

2.13 Limitations of Syntactic Definitions

The concrete syntax for a language is almost always an incomplete description. Not all syntactically valid strings of tokens should be regarded as valid programs. For instance, consider the expression $5 + 4/0$. Syntactically, this is a valid expression, but of course cannot be evaluated since division by zero is undefined. This is a semantic issue. The meaning of the expression is undefined because division by zero is undefined. This is a semantic issue and semantics are not described by a syntactic definition. All that a grammar can ensure is that the program is syntactically valid.

In fact, there is no BNF or EBNF grammar that generates only legal programs in any programming language including C++, Java, and Standard ML. A BNF grammar defines a context-free language: the left-hand side of each rules contains only one syntactic category. It is replaced by one of its alternative definitions regardless of the context in which it occurs.

The set of programs in any interesting language is not context-free. For instance, when the expression $a + b$ is evaluated, are a and b type compatible and defined over the $+$ operator? This is a context sensitive issue that can't be specified using a context-free grammar. Context-sensitive features may be formally described as a set of restrictions or context conditions. Context-sensitive issues deal mainly with declarations of identifiers and type compatibility. Sometimes, context-sensitive issues like this are said to be part of the *static semantics* of the language.

While a grammar describes how tokens are put together to form a valid program the grammar does not specify the semantics of the language nor does it describe the static semantics or context-sensitive characteritics of the language. Other means are necessary to describe these language characteristics. Some methods, like type inference rules, are formally defined. Most semantic characteristics are defined informally in some kind of English language description.

These are all context-sensitive issues.

- In an array declaration in C++, the array size must be a nonnegative value.
- Operands for the && operation must be boolean in Java.
- In a method definition, the return value must be compatible with the return type in the method declaration.

- When a method is called, the actual parameters must match the formal parameter types.

2.14 Chapter Summary

This chapter introduced you to programming language syntax and syntactic descriptions. Reading and understanding syntactic descriptions is worthwhile since you will undoubtedly come across new languages in your career as a computer scientist. There is certainly more that can be said about the topic of programming language syntax. Aho, Sethi, and Ullman [2] have written the widely recognized definitive book on compiler implementation which includes material on syntax definition and parser implementation. There are many other good compiler references as well. The Chomsky hierarchy of languages is also closely tied to grammars and regular expressions. Many books on Discrete Structures in Computer Science introduce this topic and a few good books explore the Chomsky hierarchy more deeply including an excellent text by Peter Linz [13].

In the next chapter you put this knowledge of syntax definition to good use learning a new language: the JCoCo assembly language. JCoCo is a virtual machine for interpreting Python bytecode instructions. Learning assembly language helps in having a better understanding of how higher level languages work and Chap. 3 provides many examples of Python programs and their corresponding JCoCo assembly language programs to show you how a higher level language is implemented.

2.15 Review Questions

1. What does the word syntax refer to? How does it differ from semantics?
2. What is a token?
3. What is a nonterminal?
4. What does BNF stand for? What is its purpose?
5. What kind of derivation does a top-down parser construct?
6. What is another name for a top-down parser?
7. What does the abstract syntax tree for $3 * (4 + 5)$ look like for infix expressions?
8. What is the prefix equivalent of the infix expression $3 * (4 + 5)$? What does the prefix expression's abstract syntax tree look like?
9. What is the difference between lex and yacc?
10. Why aren't all context-free grammars good for top-down parsing?
11. What kind of machine is needed to implement a bottom-up parser?
12. What is a context-sensitive issue in a language? Give an example in Java.
13. What do the terms *shift* and *reduce* apply to?

2.16 Exercises

1. Rewrite the BNF in Sect. 2.2.1 using EBNF.
2. Given the grammar in Sect. 2.3.1, derive the sentence $3*(4+5)$ using a right-most derivation.
3. Draw a parse tree for the sentence $3 * (4 + 5)$.
4. Describe how you might evaluate the abstract syntax tree of an expression to get a result? Write out your algorithm in English that describes how this might be done.
5. Write a regular expression to describe identifier tokens which must start with a letter and then can be followed by any number of letters, digits, or underscores.
6. Draw a finite state machine that would accept identifier tokens as specified in the previous exercise.
7. For the expression $3 * (4 + 5)$ show the sequence of shift and reduce operations using the grammar in Sect. 2.10.1. Be sure to say what is shifted and which rule is being used to reduce at each step. See the solution to practice problem 2.1 for the proper way to write the solution to this problem.
8. Construct a left-most derivation of $3 * (4 + 5)$ using the grammar in Sect. 2.9.3.

2.17 Solutions to Practice Problems

These are solutions to the practice problems. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

2.17.1 Solution to Practice Problem 2.1

This is a left-most derivation of the expression. There are other derivations that would be correct as well.

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow F + T \Rightarrow 4 + T \Rightarrow 4 + T * F \Rightarrow 4 + F * F \Rightarrow 4 + (E) * F$$
$$\Rightarrow 4 + (E - T) * F \Rightarrow 4 + (T - T) * F \Rightarrow 4 + (F - T) * F \Rightarrow 4 + (a - T) * F \Rightarrow$$
$$4 + (a - F) * F \Rightarrow 4 + (a - b) * F \Rightarrow 4 + (a - b) * x$$

2.17.2 Solution to Practice Problem 2.2

This is a right-most derivation of the expression $x * y + z$. There is only one correct right-most derivation.

$$E \Rightarrow E+T \Rightarrow E+F \Rightarrow E+z \Rightarrow T+z \Rightarrow T*F+z \Rightarrow T*y+z \Rightarrow F*y+z \Rightarrow x*y+z$$

2.17.3 Solution to Practice Problem 2.3

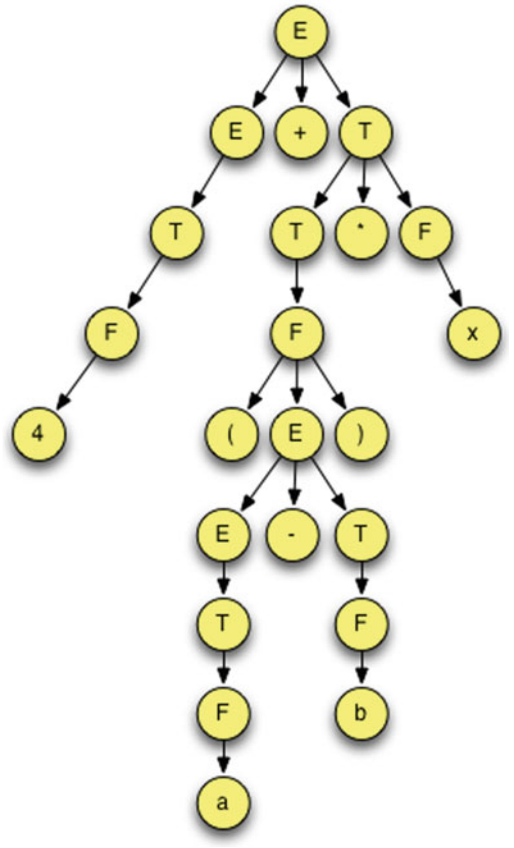
This is a left-most derivation of the expression $+4 * -abx$.

$$E \Rightarrow +EE \Rightarrow +4E \Rightarrow +4 * EE \Rightarrow +4 * -EEE \Rightarrow +4 * -aEE \Rightarrow +4* -abE \Rightarrow +4 * -abx$$

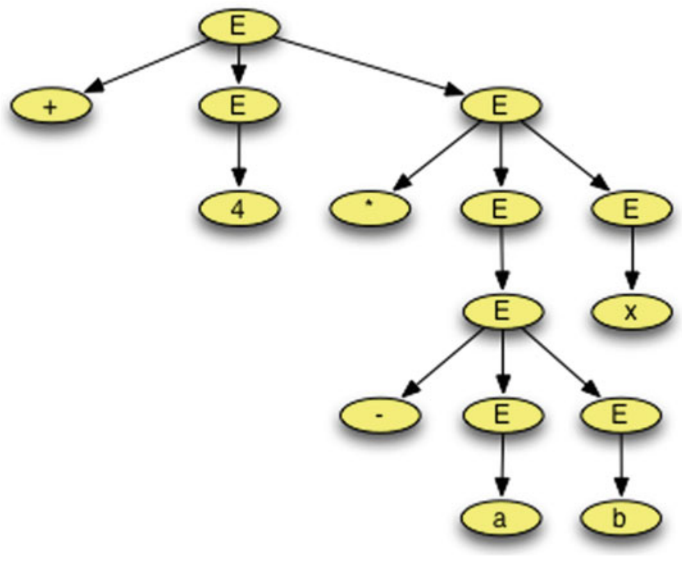
2.17.4 Solution to Practice Problem 2.4

Exactly like the parse tree for any other derivation of $(5 * x) + y$. There is only one parse tree for the expression given this grammar.

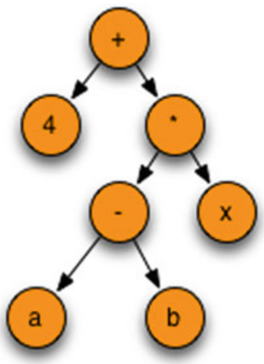
2.17.5 Solution to Practice Problem 2.5



2.17.6 Solution to Practice Problem 2.6



2.17.7 Solution to Practice Problem 2.7



2.17.8 Solution to Practice Problem 2.8

In order to define both negative and positive numbers, we can use the choice operator.

letter.letter* + digit.digit* + '-'.digit.digit* '+' + '-' + '*' + '/' + '(' + ')'

2.17.9 Solution to Practice Problem 2.9

$$E \Rightarrow T\ RestE \Rightarrow F\ RestT\ RestE \Rightarrow 4\ RestT\ RestE \Rightarrow 4\ RestE \Rightarrow$$
$$4 + T\ RestE \Rightarrow 4 + F\ RestT\ RestE \Rightarrow 4 + (E)\ RestT\ RestE \Rightarrow 4 + (T\ RestE) RestT\ RestE$$
$$\Rightarrow 4 + (F\ RestT\ RestE)\ RestT\ RestE \Rightarrow 4 + (a\ RestT\ RestE) RestT\ RestE\ \Rightarrow$$
$$4 + (a\ RestE)\ RestT\ RestE \Rightarrow 4 + (a - T\ RestE)\ RestT\ RestE \Rightarrow$$
$$4 + (a - F\ RestE)\ RestT\ RestE \Rightarrow 4 + (a - b\ RestE) \Rightarrow 4 + (a - b)\ RestT\ RestE$$
$$\Rightarrow 4 + (a - b) * F\ RestT\ RestE \Rightarrow 4 + (a - b) * x\ RestT\ RestE \Rightarrow 4 + (a - b) * x\ RestE$$
$$\Rightarrow 4 + (a - b) * x$$

2.17.10 Solution to Practice Problem 2.10

In the parsing of $5 * 4 + 3$ the following shift and reduce operations: step A initial condition, step B shift, step C reduce by rule 5, step D reduce by rule 4, step E shift, step F shift, step G reduce by rule 5, step H reduce by rule 3, step I reduce by rule 2, step J shift, step K shift, step L reduce by rule 5, step M reduce by rule 4, step N reduce by rule 1, step O finished parsing with dot on right side and E on top of stack so pop and complete with success.

2.17.11 Solution to Practice Problem 2.11

To complete this problem it is best to do a right-most derivation of $(6 + 5) * 4$ first. Once that derivation is complete, you go through the derivation backwards. The difference in each step of the derivation tells you whether you shift or reduce. Here is the result.

$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * 4 \Rightarrow F * 4 \Rightarrow (E) * 4 \Rightarrow (E + T) * 4 \Rightarrow (E + F) * 4$$
$$\Rightarrow (E + 5) * 4 \Rightarrow (T + 5) * 4 \Rightarrow (F + 5) * 4 \Rightarrow (6 + 5) * 4$$

We get the following operations from this. Stack contents have the top on the right up to the dot. Everything after the dot has not been read yet. We shift when we must move through the tokens to get to the next place we are reducing. Each step in the reverse derivation provides the reduce operations. Since there are seven tokens there should be seven shift operations.

1. Initially: . (6 + 5) ∗ 4
2. Shift: (. 6 + 5) ∗ 4
3. Shift: (6 . + 5) ∗ 4
4. Reduce by rule 5: (F . + 5) ∗ 4
5. Reduce by rule 4: (T . + 5) ∗ 4
6. Reduce by rule 2: (E . + 5) ∗ 4
7. Shift: (E + . 5) ∗ 4

8. Shift: $(E + 5 .) * 4$
9. Reduce by rule 5: $(E + F .) * 4$
10. Reduce by rule 4: $(E + T .) * 4$
11. Shift: $(E + T) . * 4$
12. Reduce by rule 1: $(E) . * 4$
13. Reduce by rule 6: $F . * 4$
14. Reduce by rule 4: $T . * 4$
15. Shift: $T * . 4$
16. Shift: $T * 4 .$
17. Reduce by rule 5: $T * F .$
18. Reduce by rule 3: $T .$
19. Reduce by rule 2: $E .$

Assembly Language 3

Python is an object-oriented, interpreted language. Internally to the Python interpreter, a Python program is converted to bytecode and interpreted using a virtual machine. Most modern programming languages have support for high-level abstractions while the instructions of a virtual machine are closer to the machine language instructions supported by hardware architectures, making the interpretation of bytecode easier than interpretation of the original source program. The advantage of virtual machine implementations results from dividing the mapping from high-level abstractions to low-level machine instructions into two parts: high-level abstractions to bytecode and bytecode to machine instructions.

While bytecode is a higher level abstraction than machine language, it is not greatly so. As programmers, if we understand how the underlying machine executes our programs, we better equip ourselves to make good choices about how we program. Just as importantly, having an understanding of how programs are executed can help us diagnose problems when things go wrong.

This chapter introduces assembly language programming in the bytecode language of the Python virtual machine. The Python virtual machine is an internal component of the Python interpreter and is not available to use directly. Instead, a bytecode interpreter called JCoCo has been developed that mimics a subset of the behavior of the Python 3.2 virtual machine. Instead of writing bytecode files directly, JCoCo supports a Python virtual machine assembly language.

While learning assembly language, we'll limit ourselves to a subset of Python. JCoCo supports boolean values, integers, strings, floats, tuples, lists, and dictionaries. It supports class and function definitions and function calls. It also supports most of the instructions of the Python virtual machine including support for conditional execution, iteration, and exception handling. It does not support importing modules or module level code. JCoCo differs from Python by requiring a main function where execution of a JCoCo assembled program begins.

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_3

To run an assembly language program it must first be assembled, then it can be executed. The JCoCo virtual machine includes the assembler so assembly isn't a separate step. An assembly language programmer writes a program in the JCoCo assembly language format, providing it to JCoCo, which then assembles and interprets the program.

The main difference between JCoCo assembly language and bytecode is the presence of labels in the assembly language format. Labels are the targets of instructions that change the normal sequence of execution of instructions. Instructions like branch and jump instructions are much easier to decipher if it says "jump to loop1" rather than "jump to address 63". Of course, bytecode instructions are encoded as numbers themselves, so the assembler translates "jump to loop1" to something like "48 63" which of course would require a manual to decipher.

Learning to program in assembly isn't all that hard once you learn how constructs like while loops, for loops, if-then statements, function definitions, and function calls are implemented in assembly language. String and list manipulation is another skill that helps if you have examples to follow. A *disassembler* is a tool that will take a machine language program and produce an assembly language version of it. Python includes a module called *dis* that includes a disassembler. When you write a Python program it is parsed and converted to bytecode when read by the interpreter. The *dis* module disassembler produces an assembly language program from this bytecode. JCoCo includes its own disassembler which uses the Python *dis* module and produces output suitable for the JCoCo virtual machine.

The existence of the disassembler for JCoCo means that learning assembly language is as easy as writing a Python program and running it through the disassembler to see how it is implemented in assembly language. That means you can discover how Python is implemented while learning assembly language! Because Python's virtual machine is not guaranteed to be backwards compatible, you must use Python 3.2 when disassembling programs so make sure that version 3.2 is installed on your system. To test this you can try typing "python3.2" in a terminal window in your favorite operating system. If it says command not found, you likely don't have Python 3.2 installed. In that case you can download it from http://python.org or directly from the JCoCo website at http://cs.luther.edu/~leekent/JCoCo. The rest of this chapter introduces you to assembly language programming using the JCoCo virtual machine.

You can download the full binary implementation of the JCoCo virtual machine by going to http://cs.luther.edu/~leekent/JCoCo. Download the zip file containing a coco shell script which runs the Java Virtual Machine on the JCoCo jar file. You can also go to github and get the source code for the reduced functionality JCoCo project at http://github.com/kentdlee/JCoCo.

3.1 Overview of the JCoCo VM

JCoCo, like Python, is a virtual machine, or interpreter, for bytecode instructions. JCoCo is written in Java using object-oriented principles and does not store its instructions in actual bytecode format. Instead, it reads an assembly language file and assembles it building an internal representation of the program as a sequence of functions each with their own sequence of bytecode instructions. CoCo is another implementation of this virtual machine, implemented in C++. You can find documentation on the C++ version at http://cs.luther.edu/~leekent/CoCo. JCoCo is backwards compatible with CoCo, but JCoCo does provide some additional functionality including the ability to define classes, create objects, and utilize single inheritance which are not used extensively in this text. Additionally, JCoCo provides an interactive command-line debugger that can be used for debugging JCoCo assembly language programs.

Most of the material presented in this chapter is true of either JCoCo or CoCo. Chap. 6 again revisits JCoCo as a target language for *Small*, but either JCoCo or CoCo will work as the *Small* compiler target.

A JCoCo program, like programs in other programming languages, utilizes a *run-time stack* to store information about each function called while the program is executing. Each function call in a JCoCo program results in a new stack frame object being created and pushed onto the run-time stack. When a function returns, its corresponding stack frame is popped from the run-time stack and discarded. Figure 3.1 depicts four active function calls. Function A called function B, which called function C, which called function D before any of the functions returned. The top of the stack is at the top of Fig. 3.1. Each stack frame contains all local variables that are defined in the function. Each stack frame also contains two additional stacks, an *operand stack* and a *block stack*.

JCoCo, like the Python virtual machine, is a stack based architecture. This means that operands for instructions are pushed onto an *operand stack*. Virtual machine instructions pop their operands from the operand stack, do their intended operation, and push their results onto the operand stack. Most CPUs are not stack based. Instead they have general purpose registers for holding intermediate results. Stack based architectures manage the set of intermediate results as a stack rather than forcing the programmer to keep track of which registers hold which results. The stack abstraction makes the life of an assembly language programmer a little easier. The operand stack is used by the virtual machine to store all intermediate results of instruction execution. This style of computation has been in use a long time, from Hewlett Packard mainframe computers of the 1960's through the 1980's to calculators still made by Hewlett Packard today. The Java Virtual Machine, or JVM, is another example of a stack machine.

The other stack utilized by JCoCo is a *block stack*. The block stack keeps track of exit points for blocks of code within a JCoCo function. When a loop is entered, the exit address of the loop is pushed onto the block stack. The instructions of each function are at zero-based offsets from the beginning of the function, so we can think of each function having its own instruction address space starting at 0. By storing each loop's exit point address on the block stack, if a break instruction is executed inside a loop, the exit point of the loop can be found and the execution of the break instruction will jump to that address. Exception handlers also push the address of the handler onto the block stack. If an exception occurs, execution jumps to the exception

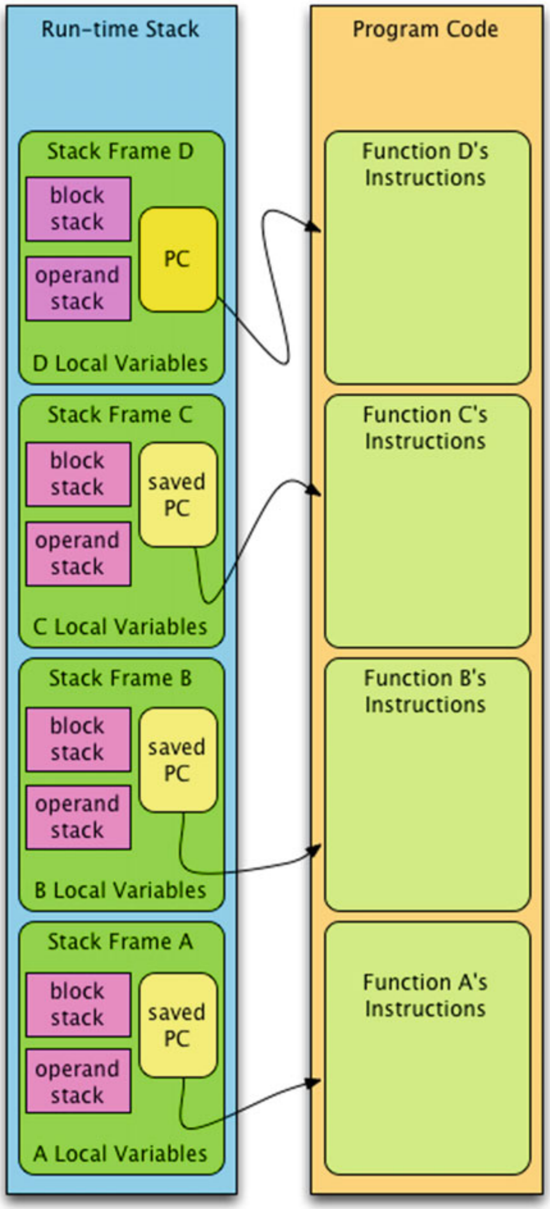


Fig. 3.1 The JCoCo virtual machine

handler by popping the address from the block stack. When a loop or try block is exited, the corresponding block stack address is popped from the block stack.

A *program counter*, or PC, is responsible for holding the address of the next instruction to be executed. The machine proceeds by fetching an instruction from the code, incrementing the PC, and executing the fetched instruction. Execution proceeds this way until a RETURN_VALUE instruction is executed or an exception occurs. When a function call is executed, the current program counter is stored in the stack frame until the called function returns, when the PC is restored to the next instruction in the current stack frame. This is depicted in Fig. 3.1 with the arrows from the stack frames to the code of their corresponding functions.

When an exception occurs, if no matching exception handler is found, execution of the function terminates and control is passed to the previously called function where the exception continues to propagate back until a matching exception handler is found. If no matching handler is found, the complete traceback of the exception is printed. If no exception occurs during the running of a program, execution terminates when the main function executes the RETURN_VALUE instruction.

The specification for JCoCo, including all instructions, global functions, and the complete assembly language BNF supported by JCoCo can be found in *Appendix A*. The rest of this chapter examines various Python language constructs and the corresponding assembly language that implement these constructs. JCoCo assembly language can be learned by examining Python code and learning how it is implemented in assembly language. The rest of this chapter proceeds in this fashion.

3.2 Getting Started

JCoCo includes a disassembler that works with Python 3.2 to disassemble Python programs into JCoCo assembly language programs, providing a great way to learn assembly language programming using the JCoCo virtual machine. Consider the following Python program that adds 5 and 6 together and prints the sum to the screen.

```
from disassembler import *
import sys

def main():
    x=5
    y=6
    z=x+y
    print(z)

if len(sys.argv) == 1:
    main()
else:
    disassemble(main)
```

Running this with python 3.2 as follows produces this output. Note that the 1 argument is required to get assembly output because of the code on lines 10-13 of the Python program.

```
MyComputer> python3.2 addtwo.py 1
Function: main/0
Constants: None, 5, 6
Locals: x, y, z
Globals: print
BEGIN
                LOAD_CONST 1
                STORE_FAST 0
                LOAD_CONST 2
                STORE_FAST 1
                LOAD_FAST 0
                LOAD_FAST 1
                BINARY_ADD
                STORE_FAST 2
                LOAD_GLOBAL 0
                LOAD_FAST 2
                CALL_FUNCTION 1
                POP_TOP
                LOAD_CONST 0
                RETURN_VALUE
END
MyComputer> python3.2 addtwo.py  1 > addtwo.casm
```

The disassembler prints the assembly language program to standard output, which is usually the screen. The second run of the addtwo.py program redirects the standard output to a file called addtwo.casm. The *casm* is the extension chosen for JCoCo assembly language files and stands for CoCo Assembly. This CASM file holds all the lines between the two MyComputer prompts above. To run this program you can invoke the JCoCo virtual machine as shown here.

```
MyComputer> coco -v addtwo.casm
Function: main/0
Constants: None, 5, 6
Locals: x, y, z
Globals: print
BEGIN
          LOAD_CONST                    1
          STORE_FAST                    0
          LOAD_CONST                    2
          STORE_FAST                    1
          LOAD_FAST                     0
          LOAD_FAST                     1
          BINARY_ADD
          STORE_FAST                    2
          LOAD_GLOBAL                   0
          LOAD_FAST                     2
          CALL_FUNCTION                 1
          POP_TOP
          LOAD_CONST                    0
          RETURN_VALUE
END

11
MyComputer> coco addtwo.casm
11
MyComputer>
```

The first run invokes *coco* which assembles the program producing the assembled output and then runs the program producing the 11 that appears below the assembled output. The assembled output is shown because the *-v* option was used when invoking *coco*. The assembled output is printed to a stream called *standard error* which is separate from the *standard output* stream where the 11 is printed. To only print the exact output of the program, the *-v* option can be omitted.

In this JCoCo program there is one function called main. The assembly indicates main has 0 formal parameters. Constants that are used in the code include None, 5, and 6. There are three local variables in the function: *x*, *y*, and *z*. The global *print* function is called and so is in the list of globals. Every function in JCoCo has these categories of identifiers and values within each defined function. Sometimes one or more of these categories may be empty and can be omitted in that case.

The instructions follow the *begin* keyword and preceed the *end* keyword. LOAD_CONST loads the constant value at its index (zero based and 1 in this case) into the constants onto the operand stack. JCoCo is a stack machine and therefore all operations are performed with operands pushed and popped from the operand stack.

The STORE_FAST instruction stores a value in the locals list, in this case at offset 0, the location of x. LOAD_FAST does the opposite of STORE_FAST, pushing a value on the operand stack from the locals list of variables. BINARY_ADD pops two operands from the stack and adds them together, pushing the result. CALL_FUNCTION pops the number of arguments specified in the instruction (1 in this case) and then pops the function from the stack. Finally, it calls the popped function with the popped arguments. The result of the function call is left on the top of the operand stack. In the case of the print function, *None* is returned and left on the stack. The POP_TOP instruction pops the *None* from the stack and discards it only to have the main function push a None on the stack just before returning. RETURN_VALUE pops the top argument from the operand stack and returns that value to the calling function. Since main was the only function called, returning from it ends the coco interpretation of the program.

To run this code, you must have the coco executable somewhere in your path. Then you can execute the following code to try it out.

```
MyComputer> python3.2 addtwo.py 1 > addtwo.casm
MyComputer> coco addtwo.casm
```

3.3 Input/Output

JCoCo provides one built-in function to read input from the keyboard and several functions for writing output to this screen or standard output. The following program demonstrates getting input from the keyboard and printing to standard output.

```
import disassembler
def main():
    name = input("Enter your name: ")
    age = int(input("Enter your age: "))
     print(name + ", a year from now you will be", age+1, "years old.")
#main()
disassembler.disassemble(main)
```

In the Python code in Sect. 3.3, the *input* function is called. Calling input requires a string prompt and returns a string of the input that was entered. Calling the *int* function on a string, as is done in the line that gets the *age* from the user, returns the integer representation of the string's value. Finally, the *print* function takes a random number of arguments, converts each to a string using the __str__ magic method, and prints each string separated by spaces. The first argument to print in the code of Sect. 3.3 is the result of concatenating *name* and the string", a year from now you will be". String concatenation was used because there shouldn't be a space between the *name* value and the comma.

The assembly language that implements the program in Sect. 3.3 is given in Fig. 3.2. Notice that built-in functions like *input*, *int*, and *print* are declared under the *Globals* list. The *name* and *age* variables are the locals.

Line 9 pushes the *input* function onto the operand stack. Line 10 pushes the string prompt for *input*. Line 11 calls the *input* function with the one allowed argument given to it. The *1* in line 11 is the number of arguments. When the *input* function returns it leaves string entered by the user on the operand stack. Line 12 stores that string in the *name* location in the locals.

Line 13 prepares to convert the next input to an integer by first pushing the *int* function on the operand stack. Then line 14 loads the *input* function. Line 15 loads the prompt like line 10 did previously. Line 16 calls the input function. The result is immediately passed to the *int* function by calling it on line 17. The *int* function leaves an integer on the top of the operand stack and line 18 stores that in the *age* variable location.

The next part of the program prints the output. To prepare for calling the *print* function, the arguments must be evaluated first, then *print* can be called. Line 19 pushes the *print* function onto the stack but does not call *print*. There are three arguments to the *print* function. The first argument is the result of concatenating two strings together. Line 20 pushes the *name* variable's value on the stack. Line 21 pushes the string", a year from now you will be" onto the stack. Line 22 calls the __add__ magic method to concatenate the two strings. The BINARY_ADD instruction pops two operands from the stack, calls the __add__ method on the first object popped with the second object as the argument which is described in more detail in *Appendix A*.

Lines 23–25 add together *age* and *1* to get the correct age value to pass to *print*. Line 26 pushes the last string constant on the operand stack and line 27 finally calls

```
1  Function: main/0
2  Constants: None, "Enter your name: ",
3      "Enter your age: ",
4      ", a year from now you will be",
5      1, "years old."
6  Locals: name, age
7  Globals: input, int, print
8  BEGIN
9           LOAD_GLOBAL         0
10          LOAD_CONST          1
11          CALL_FUNCTION       1
12          STORE_FAST          0
13          LOAD_GLOBAL         1
14          LOAD_GLOBAL         0
15          LOAD_CONST          2
16          CALL_FUNCTION       1
17          CALL_FUNCTION       1
18          STORE_FAST          1
19          LOAD_GLOBAL         2
20          LOAD_FAST           0
21          LOAD_CONST          3
22          BINARY_ADD
23          LOAD_FAST           1
24          LOAD_CONST          4
25          BINARY_ADD
26          LOAD_CONST          5
27          CALL_FUNCTION       3
28          POP_TOP
29          LOAD_CONST          0
30          RETURN_VALUE
31 END
```

Fig. 3.2 JCoCo I/O

the *print* function leaving *None* on the operand stack afterwards. Line 28 pops the *None* value and immediately *None* is pushed back on the stack in line 29 because the *main* function returns *None* in this case, which is returned in line 30, ending the iotest.casm program's execution.

A few important things to learn from this section:

- Getting input and producing output rely on the built-in functions *input* and *print*.
- Before a function can be called, it must be pushed on the operand stack. All required arguments to the function must also be pushed onto the stack on top of the function to be called.
- Finally, when a function returns, it leaves its return value on the operand stack.

Practice 3.1 The code in Fig. 3.2 is a bit wasteful which often happens when compiling a program written in a higher level language. Optimize the code in Fig. 3.2 so it contains fewer instructions.

You can check your answer(s) in Section 3.17.1.

3.4 If-Then-Else Statements

Programming languages must be able to execute code based on conditions, either externally provided via input or computed from other values as the program executes. If-then statements are one means of executing code conditionally. The code provided here isolates just an if-then statement to show how it is implemented in JCoCo assembly.

```
import disassembler
def main():
    x=5
    y=6
    if x > y:
        z=x
    else:
        z=y
    print(z)
disassembler.disassemble(main)
```

Disassembling this Python code results in the code in Fig. 3.3. There are new instructions in Fig. 3.3 that haven't been encountered until now, but just as importantly, there are labels in this code. A label provides a symbolic target to jump to in the code. Labels, like *label00* and *label01*, are defined by writing them before an instruction and are terminated with a colon. A label to the right of an instruction is a target for that instruction. Labels are a convenience in all assembly languages. They let assembly language programmers think of jumping to a target in a program, rather than changing the contents of the PC register, which is what actually happens. When a program is executed using JCoCo the labels disappear because JCoCo assembles the code, replacing the labels with the actual PC target addresses. The JCoCo code in Fig. 3.4 shows the JCoCo code after it has been assembled. The assembled code is printed by *coco* when the program is executed.

The first instruction, the *LOAD_CONST*, is at offset 0 in the code. The instructions of each function are at zero-based offsets from the beginning of the function, so we can think of each function as having its own address space starting at zero. In the code in Figs. 3.3 and 3.4 the line number of the first instruction is 6, so 6 can be subtracted from the line numbers to determine any instruction's address within the function and 6 can be added to any target to determine the line number of the target

```
 1  Function: main/0
 2  Constants: None, 5, 6
 3  Locals: x, y, z
 4  Globals: print
 5  BEGIN
 6            LOAD_CONST        1
 7            STORE_FAST        0
 8            LOAD_CONST        2
 9            STORE_FAST        1
10            LOAD_FAST         0
11            LOAD_FAST         1
12            COMPARE_OP        4
13            POP_JUMP_IF_FALSE label00
14            LOAD_FAST         0
15            STORE_FAST        2
16            JUMP_FORWARD label01
17  label00:  LOAD_FAST         1
18            STORE_FAST        2
19  label01:  LOAD_GLOBAL       0
20            LOAD_FAST         2
21            CALL_FUNCTION     1
22            POP_TOP
23            LOAD_CONST        0
24            RETURN_VALUE
25  END
```

Fig. 3.3 If-Then-Else assembly

location. In Fig. 3.4 the target of line 13 is 11 which corresponds to line 17. Looking at Fig. 3.3 this corresponds to the line where *label00* is defined. Likewise, the target of the *JUMP_FORWARD* instruction in Fig. 3.4 is *label01* which is defined on line 19. Subtracting 6, we expect to see 13 as the target PC address in the assembled code of Fig. 3.4.

Consulting the JCoCo BNF in *Appendix A*, there can be multiple labels on one instruction. In addition, instruction addresses have nothing to do with which line they are on. That only appears to be the case in Fig. 3.4 because the instructions are on consecutive lines. But, adding blank lines to the program would do nothing to change the instruction addresses. So, we could have a program like this where one instruction has two labels. These three instructions would be at three addresses within the program even though there are four lines in the code.

```
  onelabel:  LOAD_FAST     1
             STORE_FAST    2
  twolabel:
threelabel: LOAD_GLOBAL    0
```

```
Function: main/0
Constants: None, 5, 6
Locals: x, y, z
Globals: print
BEGIN
          LOAD_CONST          1
          STORE_FAST          0
          LOAD_CONST          2
          STORE_FAST          1
          LOAD_FAST           0
          LOAD_FAST           1
          COMPARE_OP          4
          POP_JUMP_IF_FALSE   11
          LOAD_FAST           0
          STORE_FAST          2
          JUMP_FORWARD        13
          LOAD_FAST           1
          STORE_FAST          2
          LOAD_GLOBAL         0
          LOAD_FAST           2
          CALL_FUNCTION       1
          POP_TOP
          LOAD_CONST          0
          RETURN_VALUE
END
```

Fig. 3.4 Assembled code

Labels can be composed of any sequence of letters, digits, underscores, or the @ character, but must start with a letter, underscore, or the @ character. They can be any number of characters long.

In Fig. 3.3, lines 6–11 load the two values to be compared on the stack. The *COMPARE_OP* instruction on line 12 has an argument of 4. Consulting the *COMPARE_OP* instruction in *Appendix A* reveals that a 4 corresponds to a *greater than* comparison. The comparison is done by calling the *__gt__* magic method on the second item from the top of the operand stack and passing it the top of the operand stack. The two operands are popped by the *COMPARE_OP* instruction and a boolean value, either *True* or *False*, is pushed on the operand stack as the result.

The next instruction jumps to the target location if the value left on the operand stack was *False*. Either way, the *POP_JUMP_IF_FALSE* instruction pops the top value from the operand stack.

Take note of line 16 in Fig. 3.3. In assembly there is nothing like an if-then-else statement. Line 15 is the end of the code that implements the *then* part of the statement. Without line 16, JCoCo would continue executing and would go right into the *else* part of the statement. The *JUMP_FORWARD* instruction is necessary to jump past the *else* part of the code if the *then* part was executed. Line 17 begins the *else* code and line 18 is the last instruction of the if-then-else statement. The label definition for *label01* is still part of the if-then-else statement, but labels the instruction immediately following the if-then-else statement.

Practice 3.2 Without touching the code that compares the two values, the assembly in Fig. 3.4 can be optimized to remove at least three instructions. Rewrite the code to remove at least three instructions from this code. With a little more work, five instructions could be removed.

You can check your answer(s) in Section 3.17.2.

3.4.1 If-Then Statements

Frequently if-then statements are written without an else clause. For instance, this program prints *x* if *x* is greater than *y*. In either case *y* is printed.

```python
import disassembler

def main():
    x = 5
    y = 6
    if x > y:
        print(x)

    print(y)

disassembler.disassemble(main)
```

Disassembling this code produces the program in Fig. 3.5. The code is very similar to the code presented in Fig. 3.3. Line 13 once again jumps past the *then* part of the program. Lines 14–17 contain the *then* code. Interestingly, line 18 jumps forward to line 19. Comparing this to the code in Fig. 3.3 where the jump forward jumps past the *else* part, the same happens in Fig. 3.5 except that there is no *else* part of the statement.

Some assembly languages do not have an equivalent to *POP_JUMP_IF_FALSE*. Instead, only an equivalent to *POP_JUMP_IF_TRUE* is available. In that case, the opposite of the condition can be tested and the jump will be executed if the opposite is true, skipping over the *then* part. For instance, if testing for *greater than* is the intent of the code, *less than or equal to* can be tested to jump around the *then* part of an if-then-else statement.

```
Function: main/0
Constants: None, 5, 6
Locals: x, y
Globals: print
BEGIN
          LOAD_CONST         1
          STORE_FAST         0
          LOAD_CONST         2
          STORE_FAST         1
          LOAD_FAST          0
          LOAD_FAST          1
          COMPARE_OP         4
          POP_JUMP_IF_FALSE  label00
          LOAD_GLOBAL        0
          LOAD_FAST          0
          CALL_FUNCTION      1
          POP_TOP
          JUMP_FORWARD       label00
label00:  LOAD_GLOBAL        0
          LOAD_FAST          1
          CALL_FUNCTION      1
          POP_TOP
          LOAD_CONST         0
          RETURN_VALUE
END
```

Fig. 3.5 If-Then assembly

Whether testing the original condition or the opposite, clearly the *JUMP_FORWARD* is not needed in the code in Fig. 3.5. As was seen in practice 3.1, the Python compiler generated a wasteful instruction. It isn't wrong to jump forward, it's just not needed. The convenience of writing in a language like Python far outweighs the inconvenience of writing in a language like JCoCo assembly language, so an extra instruction now and then is not that big a deal. In this case though, the Python compiler could be written in such a way as to recognize when the extra instruction is not needed.

Practice 3.3 Rewrite the code in Fig. 3.5 so it executes with the same result using *POP_JUMP_IF_TRUE* instead of the jump if false instruction. Be sure to optimize your code when you write it so there are no unnecessary instructions.

You can check your answer(s) in Section 3.17.3.

3.5 While Loops

Consider this code which computes the Fibonacci number for the value stored in the variable *f*. The sequence of Fibonacci numbers are computed by adding the previous two numbers in the sequence together to get the next number. The sequence consists of 1, 1, 2, 3, 5, 8, 13, 21, and so on, the eighth element of the sequence being 21.

```
import disassembler
def main():
  f=8
  i=1
  j=1
  n=1
  while n < f:
    n=n+1
    tmp = j
    j=j+i
    i = tmp
  print("Fib("+str(n)+") is",i)
disassembler.disassemble(main)
```

The JCoCo assembly for this program implements the while loop of the Python program using *JUMP_ABSOLUTE* and *POP_JUMP_IF_FALSE* instructions. Prior to the loop, the *SETUP_LOOP* instruction's purpose is not readily apparent. In Python a loop may be exited using a *break* instruction. Using *break* inside a loop is not a recommended programming style. A *break* is never needed. It is sometimes used as a convenience. To handle the *break* instruction when it is executed there must be some knowledge about where the loop ends. In the code in Fig. 3.6 the first instruction after the loop is on line 33, where *label02* is defined. The *SETUP_LOOP* instruction pushes the address of that instruction on the block stack. If a break instruction is executed, the block stack is popped and the *PC* is set to the popped instruction address.

Lines 15–18 of Fig. 3.6 implement the comparison of $n < f$ similarly to the way if-then-else comparisons are performed. The first line of this code is labeled with *label00* because the end of the loop jumps back there to see if another iteration should be performed. A while loop continues executing until the condition evaluates to *False* so the *POP_JUMP_IF_FALSE* instruction jumps to *label01* when the loop terminates.

The instruction at *label01* labels the *POP_BLOCK* instruction. This instruction is needed if the loop exits normally, not as the result of a break statement. The block stack is popped, removing the loop exit point from it. When exiting as a result of a break, execution jumps to the instruction at line 33, skipping the *POP_BLOCK* instruction since the break statement already popped the block stack.

An important thing to notice is that a while loop and an if-then-else statement are implemented using the same instructions. There is no special *loop* instruction in assembly language. The overall flow of a while loop is a test before the body of the loop corresponding to the while loop condition. If the loop condition is not

```
Function: main/0
Constants: None,8,1,"Fib(",") is"
Locals: f, i, j, n, tmp
Globals: print, str
BEGIN
          LOAD_CONST         1
          STORE_FAST         0
          LOAD_CONST         2
          STORE_FAST         1
          LOAD_CONST         2
          STORE_FAST         2
          LOAD_CONST         2
          STORE_FAST         3
          SETUP_LOOP label02
label00:  LOAD_FAST          3
          LOAD_FAST          0
          COMPARE_OP         0
          POP_JUMP_IF_FALSE label01
          LOAD_FAST          3
          LOAD_CONST         2
          BINARY_ADD
          STORE_FAST         3
          LOAD_FAST          2
          STORE_FAST         4
          LOAD_FAST          2
          LOAD_FAST          1
          BINARY_ADD
          STORE_FAST         2
          LOAD_FAST          4
          STORE_FAST         1
          JUMP_ABSOLUTE label00
label01:  POP_BLOCK
label02:  LOAD_GLOBAL        0
          LOAD_CONST         3
          LOAD_GLOBAL        1
          LOAD_FAST          3
          CALL_FUNCTION      1
          BINARY_ADD
          LOAD_CONST         4
          BINARY_ADD
          LOAD_FAST          1
          CALL_FUNCTION      2
          POP_TOP
          LOAD_CONST         0
          RETURN_VALUE
END
```

Fig. 3.6 While loop assembly

met, execution jumps to the next instruction after the loop. After the body of the loop a jump returns execution to the while loop condition code to check if another iteration of the body will be performed. This idiom, or pattern of instructions, is used to implement loops and similar patterns are used for loops in other assembly languages as well.

Practice 3.4 Write a short program that tests the use of the *BREAK_LOOP* instruction. You don't have to write a while loop to test this. Simply write some code that uses a *BREAK_LOOP* and prints something to the screen to verify that it worked.

You can check your answer(s) in Section 3.17.4.

3.6 Exception Handling

Exception handling occurs in Python within a try-except statement. Statements within the *try block* are executed and if an exception occurs execution jumps to the *except block* of statements. If main were called on the Python program given here, any error condition would send it to the except block which simply prints the exception in this case. The except block is only executed if there is an error in the try block. Errors that could occur in this program would be a conversion error for either of the two floating point number conversions or a division by zero error. The code catches an exception if a zero is entered for the second value.

```
import disassembler
def main():
    try:
        x = float(input("Enter a number: "))
        y = float(input("Enter a number: "))
        z=x/y
        print(x,"/",y,"=",z)
    except Exception as ex:
        print(ex)
disassembler.disassemble(main)
```

Implementing exception handling in JCoCo is similar in some ways to implementing the *BREAK_LOOP* instruction. The difference is that the exception causes the program to jump from one place to the next instead of the *BREAK_LOOP* instruction. Both exception handling and the break instruction make use of the block stack. When a loop is entered, the *SETUP_LOOP* instruction pushes the exit point of the loop onto the block stack; the exit point being an integer referring to the address of the first instruction after the loop.

To distinguish between loop exit points and exception handling, the *SETUP_EXCEPT* instruction pushes the negative of the except handler's address (i.e. –1*address). So a negative number on the block stack refers to an exception handler while a positive value refers to a loop exit point. In the code in Fig. 3.7 the exception handler's code begins at *label00*.

The *try* block code begins on line 7 with the *SETUP_EXCEPT*. This pushes the handler's address for *label00* on the block stack which corresponds to a –27. Execution proceeds by getting input from the user, converting the input to floats, doing the division, and printing the result. The *print* completes on line 24 where *None*, which is returned by *print*, is popped from the operand stack.

If execution makes it to the end of the *try* block, then no exception occurred and line 25 pops the –27 from the block stack, ending the *try* block. Line 26 jumps past the end of the *except* block.

If an exception occurs, three things are pushed onto the operand stack before any handling of the exception occurs. The *traceback* is pushed first. The traceback is a copy of the run-time stack containing each function call and the stored PC of all pending functions including the current function's stack frame and PC. Above the traceback there are two copies of the exception object pushed on the operand stack when an exception occurs.

If an exception occurs in the *try* block, JCoCo consults the block stack and pops values until a negative address is found corresponding to some *except* block. Multiple try-except statements may be nested, so it is possible that the block stack will contain more than one negative address. When a negative address is found, the *PC* is set to its positive value causing execution to jump to the *except* block. In Fig. 3.7, that's line 27. The traceback and two copies of the exception are pushed onto the stack prior to line 27 being executed.

Why are three objects pushed on the operand stack when an exception occurs? Python's *RAISE_VARARGS* instruction describes the contents of the operand stack as TOS2 containing the traceback, TOS1 the parameter, and TOS the exception object. In the JCoCo implementation the parameter to an exception can be retrieved by converting the exception to a string, so the object at TOS1 is simply the exception again. For the sake of compatibility with the Python disassembler JCoCo pushes three operands pushed onto the operand stack when an exception is raised.

Exception handlers in Python may be written to match only certain types of exceptions. For instance, in Python a division by zero exception is different than a float conversion error. The JCoCo virtual machine currently only has one type of exception, called *Exception*. It is possible to extend JCoCo to support other types of exceptions, but currently there is only one type of exception object that can be created. The argument to the exception object can be anything that is desired. The program in Fig. 3.7 is written to catch any type of exception, but it could be written to catch only a certain type of exception. Line 27 duplicates the exception object on the top of the operand stack. Line 35 loads a global *Exception* object onto the stack. The *COMPARE_OP 10* instruction compares the exception using the exception match comparison which calls the *__excmatch__* magic method to see if there is a match between the thrown exception and the specified pattern. If there is not a match, line 30 jumps to the end of the except block. The *END_FINALLY* instruction on line 47 detects if the exception was handled and if not, it re-throws the exception for some outer exception handling block.

```
Function: main/0
Constants: None,
    "Enter a number: ", "/", "="
Locals: x, y, z, ex
Globals: float,input,print,Exception
BEGIN
          SETUP_EXCEPT label00
          LOAD_GLOBAL          0
          LOAD_GLOBAL          1
          LOAD_CONST           1
          CALL_FUNCTION        1
          CALL_FUNCTION        1
          STORE_FAST           0
          ...
          BINARY_TRUE_DIVIDE
          STORE_FAST           2
          LOAD_GLOBAL          2
          LOAD_FAST            0
          LOAD_CONST           2
          LOAD_FAST            1
          LOAD_CONST           3
          LOAD_FAST            2
          CALL_FUNCTION        5
          POP_TOP
          POP_BLOCK
          JUMP_FORWARD label03
label00:  DUP_TOP
          LOAD_GLOBAL          3
          COMPARE_OP          10
          POP_JUMP_IF_FALSE label02
          POP_TOP
          STORE_FAST           3
          POP_TOP
          SETUP_FINALLY label01
          LOAD_GLOBAL          2
          LOAD_FAST            3
          CALL_FUNCTION        1
          POP_TOP
          POP_BLOCK
          POP_EXCEPT
          LOAD_CONST           0
label01:  LOAD_CONST           0
          STORE_FAST           3
          DELETE_FAST          3
          END_FINALLY
          JUMP_FORWARD label03
label02:  END_FINALLY
label03:  LOAD_CONST           0
          RETURN_VALUE
END
```

Fig. 3.7 Exception handling assembly

If the exception was a match, execution of the handler code commences as it does on line 31 of the program. The top of the operand stack contains the extra exception object so it is thrown away by line 31. Line 32 takes the remaining exception object and makes the *ex* reference point to it. Line 33 pops the traceback from the operand stack.

Should an exception occur while executing an exception handler, then JCoCo must clean up from the exception. Line 34 executes the *SETUP_FINALLY* instruction to push another block stack record to keep track of the end of the exception handler. Lines 35–38 print the exception named *ex* in the code.

Line 39 pops the exit address that was pushed by the *SETUP_FINALLY* instruction. The *POP_EXCEPT* instruction on line 40 then pops the block stack address for the exception handler exit address. Line 41 pushes a *None* on the operand stack.

Line 42 is either the next instruction executed or it is jumped to as a result of an exception while executing the handler code for the previous exception. Either way, the *ex* variable is made to refer to *None*. The *DELETE_FAST* instruction doesn't appear to do much in this code. It is generated by the disassembler, but appears to delete *None* which doesn't seem to need to be done.

The last instruction of the handler code, the *END_FINALLY* instruction checks to see if the exception was handled. In this case, it was handled and the instruction does nothing. If execution jumps to line 47 then the exception handler did not match the raised exception and therefore the exception is re-raised. Line 48 wraps up by setting up to return *None* from the *main* function.

Practice 3.5 Write a short program that tests creating an exception, raising it, and printing the handled exception. Write this as a JCoCo program without using the disassembler.

You can check your answer(s) in Section 3.17.5.

3.7 List Constants

Building a compound value like a list is not too hard. To build a list constant using JCoCo you push the elements of the list on the operand stack in the order you want them to appear in the list. Then you call the *BUILD_LIST* instruction. The argument to the instruction specifies the length of the list. This code builds a list and prints it to the screen.

```
import disassembler
def main():
    lst = ["hello","world"]
    print(lst)
disassembler.disassemble(main)
```

```
 1  Function: main/0
 2  Constants: None, "hello", "world"
 3  Locals: lst
 4  Globals: print
 5  BEGIN
 6            LOAD_CONST         1
 7            LOAD_CONST         2
 8            BUILD_LIST         2
 9            STORE_FAST         0
10            LOAD_GLOBAL        0
11            LOAD_FAST          0
12            CALL_FUNCTION      1
13            POP_TOP
14            LOAD_CONST         0
15            RETURN_VALUE
16  END
```

Fig. 3.8 Assembly for building a list

The assembly language program in Fig. 3.8 builds a list with two elements: [‘hello’, ‘world’]. Lines 6 and 7 push the two strings on the operand stack. Line 8 pops the two operands from the stack, builds the list object, and pushes the resulting list on the operand stack. Python defines the *__str__* magic method for built-in type of value, which is called on the list on line 12.

If you run this program using the JCoCo interpreter you will notice that [‘hello’, ‘world’] is not printed to the screen. Instead, [hello, world] is printed. This is because currently the *__str__* method is called on each element of the list to convert it to a string for printing. This is not the correct method to call. Instead, the *__repr__* magic method should be called which returns a printable representation of the value retaining any type information. In the next chapter there will be an opportunity to fix this.

3.8 Calling a Method

Calling functions like *print* and *input* was relatively simple. Push the function name followed by the arguments to the function on the operand stack. Then, call the function with the *CALL_FUNCTION* instruction. But, how about methods? How does a method like *split* get called on a string? Here is a program that demonstrates how to call *split* in Python.

```
1  Function: main/0
2  Constants: None,"Enter integers:"
3  Locals: s, lst
4  Globals: input, split, print
5  BEGIN
6            LOAD_GLOBAL          0
7            LOAD_CONST           1
8            CALL_FUNCTION        1
9            STORE_FAST           0
10           LOAD_FAST            0
11           LOAD_ATTR            1
12           CALL_FUNCTION        0
13           STORE_FAST           1
14           LOAD_GLOBAL          2
15           LOAD_FAST            1
16           CALL_FUNCTION        1
17           POP_TOP
18           LOAD_CONST           0
19           RETURN_VALUE
20 END
```

Fig. 3.9 Assembly for Calling a method

```python
import disassembler
def main():
    s = input("Enter integers:")
    lst = s.split()
    print(lst)
disassembler.disassemble(main)
```

Line 6 of the assembly language code in Fig. 3.9 prepares to call the *input* function by loading the name *input* onto the operand stack. Line 7 loads the argument to *input*, the prompt string. Line 8 calls the input function leaving the entered text on the operand stack. Calling *split* is done similarly.

In this Python code the syntax of calling *input* and *split* is quite different. Python sees the difference and uses the *LOAD_ATTR* instruction in the assembly language instructions to get the *split* attribute of the object referred to by *s*. Line 10 loads the object referred to by *s* on the stack. Then line 11 finds the *split* attribute of that object. Each object in JCoCo and Python contains a dictionary of all the object's attributes. This *LOAD_ATTR* instruction examines the dictionary and with the key found in the globals list at the operands index. It then loads that attribute onto the operand stack. The *CALL_FUNCTION* instruction then calls the method that was located with the *LOAD_ATTR* instruction.

The *STORE_ATTR* instruction stores an attribute in an object in much the same way that an attribute is loaded. JCoCo does not presently support the *STORE_ATTR* instruction but could with relatively little effort. The ability to load and store object attributes means that JCoCo could be used to implement an object-oriented language. This makes sense since Python is an object-oriented language.

Practice 3.6 Normally, if you want to add to numbers together in Python, like 5 and 6, you write 5+6. This corresponds to using the *BINARY_ADD* instruction in JCoCo which in turn calls the magic method *__add__* with the method call 5.__add__(6). Write a short JCoCo program where you add two integers together without using the *BINARY_ADD* instruction. Print the result to the screen.

You can check your answer(s) in Section 3.17.6.

3.9 Iterating Over a List

Iterating through a sequence of any sort in JCoCo requires an iterator. There are iterator objects for every type of sequence: lists, tuples, strings, and other types of sequences that have yet to be introduced. Here is a Python program that splits a string into a list of strings and iterates over the list.

```
from disassembler import *
def main():
    x = input("Enter a list: ")
    lst = x.split()
    for b in lst:
        print(b)
disassemble(main)
```

Lines 6–8 of the assembly code in Fig. 3.10 gets an input string from the user, leaving it on the operand stack. Line 9 stores this in the variable *x*. Lines 10–12 call the *split* method on this string, leaving a list object on the top of the operand stack. The list contains the list of space separated strings from the original string in *x*. Line 13 stores this list in the variable *lst*.

Line 14 sets up the exit point of a loop as was covered earlier in this chapter. Line 15 loads the *lst* variable onto the operand stack. The *GET_ITER* instruction creates an iterator with the top of the operand stack. The *lst* is popped from the operand stack during this instruction and the resulting iterator is pushed onto the stack.

An iterator has a *__next__* magic method that is called by the *FOR_ITER* instruction. When *FOR_ITER* executes the iterator is popped from the stack, *__next__* is called on it, and the iterator and the next value from the sequence are pushed onto the operand stack. The iterator is left below the next value in the sequence at TOS1. When *__next__* is called on the iterator and there are no more elements left in the sequence, the PC is set to the label of the *FOR_ITER* instruction, ending the loop.

When the loop is finished the block stack is popped to clean up from the loop. Line 25 loads the *None* on the stack before returning from the *main* function.

```
Function: main/0
Constants: None, "Enter a list: "
Locals: x, lst, b
Globals: input, split, print
BEGIN
          LOAD_GLOBAL      0
          LOAD_CONST       1
          CALL_FUNCTION    1
          STORE_FAST       0
          LOAD_FAST        0
          LOAD_ATTR        1
          CALL_FUNCTION    0
          STORE_FAST       1
          SETUP_LOOP  label02
          LOAD_FAST        1
          GET_ITER
label00:  FOR_ITER  label01
          STORE_FAST       2
          LOAD_GLOBAL      2
          LOAD_FAST        2
          CALL_FUNCTION    1
          POP_TOP
          JUMP_ABSOLUTE label00
label01:  POP_BLOCK
label02:  LOAD_CONST       0
          RETURN_VALUE
END
```

Fig. 3.10 List iteration assembly

Practice 3.7 Write a JCoCo program that gets a string from the user and iterates over the characters of the string, printing them to the screen.

You can check your answer(s) in Section 3.17.7.

3.10 Range Objects and Lazy Evaluation

Indexing into a sequence is another way to iterate in a program. When you index into a list, you use a subscript to retrieve an element of the list. Generally, indices are zero-based. So the first element of a sequence is at index 0, the second at index 1, and so on.

There are two versions of Python in use today. Version 2, while older is still widely used because there are many Python programs that were written using it and there is a cost to converting them to use Python 3. Python 3 was created so new features could be added that might be incompatible with the older version. One difference was in the range function. In Python 2, the range function generated a list of integers of the specified size and values. This is inefficient because some ranges might consist of millions of integers. A million integers takes up a lot of space in memory and takes some time to generate. In addition, depending on how code is written, not all the integers in a range may be needed. These problems are a result of *eager evaluation* of the range function. Eager evaluation is when an entire sequence is generated before any element of the sequence will actually be used. In Python 2 the entire list of integers is created as soon as the range function is called even though the code can only use one integer at a time.

Python 3 has dealt with the *eager evaluation* of the range function by defining a range object that is *lazily evaluated*. This means that when you call the range function to generate a million integers, you don't get any of them right away. Instead, you get a *range object*. From the range object you can access an iterator. When *__next__* is called on an iterator you get the next item in the sequence. When *__next__* is called on a range object iterator you get the next integer in the range's sequence. *Lazy evaluation* is when the next value in a sequence is generated only when it is ready to be used and not before. This code creates a range object. The range object is designed to provide lazy evaluation of integer sequences.

```python
from disassembler import *
def main():
    x = input("Enter list: ")
    lst = x.split()
    for i in range(len(lst)-1,-1,-1):
        print(lst[i])
disassemble(main)
```

This Python code uses indices to iterate backwards through a list. In this case an iterator over the range object yields a descending list of integers which are the indices into the list of values entered by the user. If the use enters four space separated values, then the range object will yield the sequence [3, 2, 1, 0]. The first argument to range is the start value, the second is one past the stop value, and the third argument is the increment. So the sequence in the Python code in Sect. 3.10 is a descending sequence that goes down one integer at a time from the length of the list minus one to zero.

The JCoCo assembly code in Fig. 3.11 implements this same program. Lines 15–23 set up for calling the range function with the three integer values. Lines 15–20 call the *len* function to get the length of the list and subtract one. Lines 21 and 22 put

```
 1  Function: main/0
 2  Constants: None,"Enter list: ",1,-1,-1
 3  Locals: x, lst, i
 4  Globals: input,split,range,len,print
 5  BEGIN
 6            LOAD_GLOBAL       0
 7            LOAD_CONST        1
 8            CALL_FUNCTION     1
 9            STORE_FAST        0
10            LOAD_FAST         0
11            LOAD_ATTR         1
12            CALL_FUNCTION     0
13            STORE_FAST        1
14            SETUP_LOOP label02
15            LOAD_GLOBAL       2
16            LOAD_GLOBAL       3
17            LOAD_FAST         1
18            CALL_FUNCTION     1
19            LOAD_CONST        2
20            BINARY_SUBTRACT
21            LOAD_CONST        3
22            LOAD_CONST        4
23            CALL_FUNCTION     3
24            GET_ITER
25  label00:  FOR_ITER label01
26            STORE_FAST        2
27            LOAD_GLOBAL       4
28            LOAD_FAST         1
29            LOAD_FAST         2
30            BINARY_SUBSCR
31            CALL_FUNCTION     1
32            POP_TOP
33            JUMP_ABSOLUTE label00
34  label01:  POP_BLOCK
35  label02:  LOAD_CONST        0
36            RETURN_VALUE
37  END
```

Fig. 3.11 Range assembly

two –1 values on the operand stack. Line 23 calls the range function which creates and pushes a range object onto the operand stack as its result.

Line 24 creates an iterator for the range object. As described in the last section, the *FOR_ITER* instruction calls the *__next__* magic method on the iterator to get the next integer in the range's sequence. The lazy evaluation occurs because an iterator keeps track of which integer is the next value in the sequence. Line 26 stores the next integer in the variable *i*.

The *BINARY_SUBSCR* instruction is an instruction that has not been encountered yet in this chapter. Line 28 loads the list called *lst* onto the operand stack. Line 29 loads the value of *i* onto the operand stack. The *BINARY_SUBSCR* instruction indexes into *lst* at position *i* and pushes the value found at that position onto the operand stack. That value is printed by the print function call on line 31 of the program.

Lazy evaluation is an important programming language concept. If you ever find yourself writing code that must generate a predictable sequence of values you probably want to generate that sequence lazily. Iterators, like range iterators, are the means by which we can lazily access a sequence of values and range objects define a sequence of integers without eagerly generating all of them.

3.11 Functions and Closures

Up to this point in the chapter all the example programs have been defined in a *main* function. JCoCo supports the definition of multiple functions and even nested functions. Here is a Python program that demonstrates how to write nested functions in the Python programming language. The *main* function calls the function named *f* which returns the function *g* nested inside the *f* function. The *g* function returns *x*. This program demonstrates nested functions in JCoCo along with how to build a closure.

```
import disassembler
def main():
  x = 10
  def f(x):
    def g():
      return x
    return g
  print(f(3)())
disassembler.disassemble(main)
```

Notice the Python code in section 3.11 calls the disassembler on the top-level function *main*. It is not called on *f* or *g* because they are nested inside *main* and the disassembler automatically disassembles any nested functions of a disassembled function.

The format of the corresponding JCoCo program in Fig. 3.12 is worth noting as well. The top level *main* function is defined along the left hand side. Indentation has no effect on JCoCo but visually you see that *f* is nested inside *main*. The function *g* is nested inside *f* because it appears immediately after the first line of the definition of *f* on line 3. The rest of the definition of *f* starts again on line 10 and extends to line 21. The definition of *g* starts on line 3 and extends to line 9.

The number of arguments for each function is given by the integer after the slash. The *f/1* indicates that *f* expects one argument. The *main* and *g* functions expect zero arguments. These values are used during a function call to verify that the function is called with the required number of arguments.

```
Function: main/0
    Function: f/1
        Function: g/0
        Constants: None
        FreeVars: x
        BEGIN
                  LOAD_DEREF    0
                  RETURN_VALUE
        END
    Constants: None, code(g)
    Locals: x, g
    CellVars: x
    BEGIN
              LOAD_CLOSURE      0
              BUILD_TUPLE       1
              LOAD_CONST        1
              MAKE_CLOSURE      0
              STORE_FAST        1
              LOAD_FAST         1
              RETURN_VALUE
    END
Constants: None, 10, code(f), 3
Locals: x, f
Globals: print
BEGIN
          LOAD_CONST            1
          STORE_FAST            0
          LOAD_CONST            2
          MAKE_FUNCTION         0
          STORE_FAST            1
          LOAD_GLOBAL           0
          LOAD_FAST             1
          LOAD_CONST            3
          CALL_FUNCTION         1
          CALL_FUNCTION         0
          CALL_FUNCTION         1
          POP_TOP
          LOAD_CONST            0
          RETURN_VALUE
END
```

Fig. 3.12 Nested functions assembly

Examine the Python code in section 3.11 carefully. The *main* function calls the function *f* which returns the function *g*. Notice that *f returns g*, it does not *call g*. In the print statement of *main* the function *f* is called, passing 3 to the function that returns *g*. The extra set of parens after the function call *f(3)* calls *g*. This is a valid Python program, but not a common one. The question is: What does the program print? There are two possible choices it seems: either 10 or 3. Which seems more likely?

On the one hand, *g* is being called from the *main* function where *x* is equal to 10. If the program printed 10, we would say that Python is a dynamically scoped language, meaning that the function executes in the environment in which it is called. Since *g* is called from *main* the value of *x* is 10 and in a dynamically scoped language 10 would be printed. The word *dynamic* is used because if *g* were called in another environment it may return something completely different. We can only determine what *g* will return by tracing the execution of the program to the point where *g* is called.

On the other hand, *g* was defined in the scope of an *x* whose value was 3. In that case, the environment in which *g* executes is the environment provided by *f*. If 3 is printed then Python is a statically scoped language meaning that we need only understand what the environment contained when *g* was defined, not when it was called. In a statically scoped language this specific instance of *g* will return the same value each and every time it is called, not matter where it is called in the program. The value of *x* is determined when *g* is defined.

Dynamically scoped languages are rare. Lisp, when it was first defined, was dynamically scoped. McCarthy quickly corrected that and made Lisp a statically scoped language. It is interesting to note that Emacs Lisp is dynamically scoped. Python is statically scoped as are most modern programming languages.

To execute functions in a statically scoped language, two pieces are needed when a function may return another function. To execute *g* not only is the code for *g* required, but so also is the environment in which this instance of *g* was defined. A *closure* is formed. A closure is the environment in which a function is defined and the code for the function itself. This closure is what is called when the function *g* is finally called in *main*.

Take a look at the JCoCo code for this program in Fig. 3.12. Line 14 begins creating a new closure object in the body of function *f* by loading the cell variable named *x* onto the stack. A cell variable is an indirect reference to a value. Figure 3.13 depicts what is happening in the program just before the *x* is returned in the function *g*. A variable in Python, like Java and many other languages, is actually a *reference* that points to a *value*. Values exist on the heap and are created dynamically as the program executes. When a variable is assigned to a new value, the variables reference is made to point to a new value on the heap. The space for values on the heap that are no longer needed is reclaimed by a *garbage collector* that frees space on the heap so it can be re-used. In Fig. 3.13 there are three values on the heap, a 10, a 3, and one other value called a cell in JCoCo and the Python virtual machine.

Because the function *g* needs access to the variable *x* outside the function *f*, the *3* is indirectly referenced through a cell variable. The *LOAD_CLOSURE* instruction

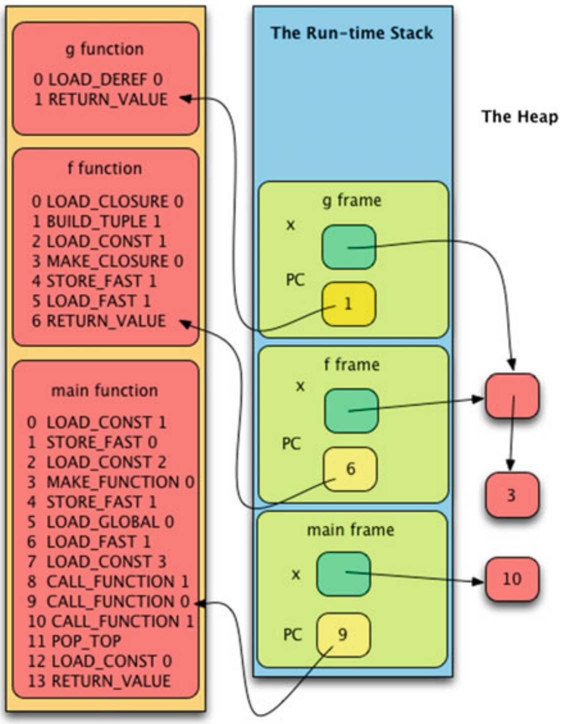


Fig. 3.13 Execution of nested.casm

pushes that cell variable onto the stack to be used in the closure. Since only one value is needed from the environment, the next instruction on line 15 builds a tuple of all the values needed from the environment. Line 16 loads the code for *g* onto the stack. Line 17 forms the closure by popping the function and the environment from the stack and building a *closure* object.

The variable *x* is a local variable for the function *f*. But, because *x* is referenced in *g* and *g* is nested inside *f*, the variable *x* is also listed as a cell variable in *f*. A cell variable is an indirect reference to a value. This means there is one extra step to finding the value that *x* refers to. We must go through the cell to get to the 3.

The *LOAD_DEREF* instruction on line 7 is new. A *LOAD_DEREF* loads the value that is referenced by the reference pointed to in the list of cellvars. So, this instructions pushes the 3 onto the operand stack. Finally, line 35 calls the closure consisting of the function and its data.

In the function *g* the freevars refer to the tuple of references in the closure that was just called, so the first instruction, the *LOAD_DEREF*, loads the 3 onto the operand stack. Figure 3.13 depicts this state right before the *RETURN_VALUE* instruction is executed.

To finish up the execution of this program a 3 is returned from the call to *g* and its frame is popped from the run-time stack. Control returns to *main* where the 3 is printed. After returning from *main* its frame is also popped from the run-time stack which ends the program.

Practice 3.8 The program in Fig. 3.12 would work just fine without the cell. The variable *x* could refer directly to the 3 in both the *f* and *g* functions without any ramifications. Yet, a cell variable is needed in some circumstances. Can you come up with an example where a cell variable is absolutely needed?

You can check your answer(s) in Section 3.17.8.

3.12 Recursion

Functions in JCoCo can call themselves. A function that calls itself is a recursive function. Recursive functions are studied in some detail in Chap. 5 of this text. Learning to write recursive functions well is not hard if you follow some basic rules. The mechanics of writing a recursive function include providing a base case that comes first in the function. Then, the solution to the problem you are solving must be solved by calling the same function on some smaller piece of data while using that result to construct a solution to the bigger problem.

Consider the factorial definition. Factorial of zero, written 0!, is defined to be 1. This is the base case. For integer *n* greater than 0, *n! = n*(n–1)!*. This is a recursive definition because factorial is defined in terms of itself. It is called on something smaller, meaning *n–1* which is closer to the base case, and the result is used in computing *n!*. Here is a Python program that computes *5!*.

```python
import disassembler
def factorial(n):
    if n==0:
        return 1
    return n*factorial(n-1)
def main():
    print(factorial(5))

disassembler.disassemble(factorial)
disassembler.disassemble(main)
```

The JCoCo implementation of this program is given in Fig. 3.14. The program begins in main by loading 5 on the operand stack and calling the *factorial* function. The result is printed to the screen with the print function.

Calling *factorial* jumps to the first instruction of the function where *n* is loaded onto the operand stack, which in this case is 5. Lines 7–8 compare *n* to 0 and if the two values are equal, 1 is returned. Notice that the *RETURN_VALUE* instruction appears in the middle of the factorial function in this case. A return instruction doesn't have

```
Function: factorial/1
Constants: None, 0, 1
Locals: n
Globals: factorial
BEGIN
          LOAD_FAST          0
          LOAD_CONST         1
          COMPARE_OP         2
          POP_JUMP_IF_FALSE label00
          LOAD_CONST         2
          RETURN_VALUE
label00:  LOAD_FAST          0
          LOAD_GLOBAL        0
          LOAD_FAST          0
          LOAD_CONST         2
          BINARY_SUBTRACT
          CALL_FUNCTION      1
          BINARY_MULTIPLY
          RETURN_VALUE
END
Function: main/0
Constants: None, 5
Globals: print, factorial
BEGIN
          LOAD_GLOBAL        0
          LOAD_GLOBAL        1
          LOAD_CONST         1
          CALL_FUNCTION      1
          CALL_FUNCTION      1
          POP_TOP
          LOAD_CONST         0
          RETURN_VALUE
END
```

Fig. 3.14 Recursion assembly

to appear at the end of a function. It can appear anywhere it makes sense and in this case, it makes sense to return from the base case as soon as soon as possible.

The code from *label00* forward is the recursive case since otherwise we would have returned already. The code subtracts one from n and calls factorial with that new, smaller value. Notice that the recursive function call is identical to any other function call. Finally, after the function call the result of the call is on the operand stack and it is multiplied by n to get $n!$ which is returned.

Because this is a recursive function, the preceding two paragraphs are repeated 5 more times, each time reducing n by 1. The program continues to count down until 1 is returned for the factorial of 0. At its deepest, there are 7 stack frames on the run-time stack for this program: one for the *main* function, and six more for the recursive *factorial* function calls. The run-time stack grows to 7 stack frames deep when the base case is executed and then shrinks again as the recursion unwinds. Finally, when the program returns to *main*, 120 is printed to the screen.

Practice 3.9 Draw a picture of the run-time stack just before the instruction on line 11 of Fig. 3.14 is executed. Use Fig. 3.13 as a guide to how you draw this picture. Be sure to include the code, the values of *n*, and the *PC* values.

You can check your answer(s) in Section 3.17.9.

3.13 Support for Classes and Objects

In Python a class definition consists of a class declaration. The class declaration contains a magic method named *__init__* which is responsible for initializing any created object. On line 14 of this Python program an object is instantiated. Python automatically calls the constructor to initialize the space allocated by Python for the object.

```
1  import disassembler
2  class Dog:
3      def __init__(self):
4          self.food = 0
5      def eat(self):
6          self.food = self.food + 1
7      def speak(self):
8          if self.food > 2:
9              print("I am happy!")
10         else:
11             print("I am hungry!!!")
12         self.food=self.food - 1
13 def main():
14     mesa  = Dog()
15     mesa.eat()
16     mesa.speak()
17     mesa.eat()
18     mesa.eat()
19     mesa.speak()
20 disassembler.disassemble(Dog)
21 disassembler.disassemble(main)
```

A class is a collection of functions that all operate on some given grouping of data. For instance, the class *Dog* contains a function called *eat* and another called *speak*. Both the *eat* and *speak* functions operate on objects of type *Dog*.

The functions of a class definition are often called methods to differentiate them from stand-alone functions. Methods are provided a reference to the *current object* which is the collection of data on which a method operates. The reference *self* is used to reference the current object and is always the first parameter to a method in Python. This description of methods isn't entirely accurate when considering the Python virtual machine.

In the Python virtual machine *methods* are created from the class' *functions* when an object is instantiated. Consider the assembly program in Fig. 3.15 which demonstrates the instantiation of a class called *Dog*. In line 36 of the main function the class called *Dog* is *called*. Calling a class in Python means executing the code that allocates a *Dog* object in memory. This work is handled by the virtual machine. The assembly language program does not directly allocate the space for the object through any instruction. It is accomplished by calling the class.

All objects in Python consist of a dictionary that stores the attributes of the object. When the *Dog* class is called, the dictionary in the *Dog* object is initialized with the methods *Dog* object. The methods of the *Dog* object are essentially the functions of the *Dog* class. The difference between a method and a function is the *self* parameter. A method provides the *self* argument to its function by providing a reference to the current object as the first parameter. The *method* is a wrapper for the *function*. After the methods are stored in the object's dictionary, the *__init__* method is called to perform any further initialization of the object.

To get a better understanding of the difference between methods and functions, consider the code that calls the *eat* method. When the *eat* method is called on the *Dog* object, the method provides a reference to the current *Dog* object *mesa* as the first parameter before calling the *Dog* class' *eat* function. This is appearent starting on line 38 in Fig. 3.15. Line 38 loads the the reference for *mesa* onto the operand stack. Then line 39 looks up the *eat* method in the object's dictionary, leaving the method on top of the operand stack, but not the reference to the Dog object. Also, notice that no arguments are loaded on top of the method. Yet, the *eat* function has one parameter, self. When the *eat* method is called on line 40 the virtual machine loads the self parameter before calling the *eat* function. The distinction between methods and functions is revisted again in the next chapter.

Practice 3.10 In this section it was stated that every object consists of a dictionary which holds the attributes of the object. What is stored in the dictionary of the object that *mesa* refers to in this section?

You can check your answer(s) in Section 3.17.10.

```
Class: Dog
BEGIN
   Function: eat/1
   Constants: None, 1
   Locals: self
   Globals: food
   BEGIN
             LOAD_FAST     0
             LOAD_ATTR     0
             LOAD_CONST    1
             BINARY_ADD
             LOAD_FAST     0
             STORE_ATTR    0
             LOAD_CONST    0
             RETURN_VALUE
   END
   Function: __init__/1
   Constants: None, 0
   Locals: self
   Globals: food
   BEGIN
             LOAD_CONST  1
             LOAD_FAST   0
             STORE_ATTR  0
             LOAD_CONST  0
             RETURN_VALUE
   END
   # speak function omitted
END
Function: main/0
Constants: None
Locals: mesa
Globals: Dog, eat, speak
BEGIN
          LOAD_GLOBAL   0
          CALL_FUNCTION 0
          STORE_FAST    0
          LOAD_FAST     0
          LOAD_ATTR     1
          CALL_FUNCTION 0
          POP_TOP
          ...
          RETURN_VALUE
END
```

Fig. 3.15 The dog class

3.13.1 Inheritance

In object-oriented programming, inheritance comes into play when one class inherits from another. Inheritance is useful for polymorphism and code re-use. When programming using Python, polymorphism happens without inheritance because Python is a dynamically typed language, meaning that all method calls are looked up at run-time as was seen in the last section when the *LOAD_ATTR* instruction was executed. The *LOAD_ATTR* instruction looks up a method by name in the object's dictionary. The run-time look up of methods by name creates the polymorphic behavior of Python. The only purpose of inheritance in Python is code re-use. The next chapter will have more on polymorphism and how it applies to Java programming where inheritance is needed to implement polymorphism.

Consider the Python program in this section. Again there is a *Dog* class that this time inherits from an *Animal* class. The *Animal* class defines an *eat* method which is re-used by the *Dog* class. The *Animal* constructor also contains code that is re-used by the *Dog* class. But, the *Dog* class defines its own *speak* method, overriding the *speak* method in the *Animal* class.

The inheritance of *Animal* contributing to the *Dog* class is indicated by writing *Dog(Animal)* on line 12. The call to *super* on line 14 returns an instance of a super class object which can be used to reference the super class, in this case the *Animal* class. Python programs can use multiple inheritance. This is not true in JCoCo. Only single inheritance is currently supported.

```
1  import disassembler
2
3  class Animal:
4      def __init__(self,name):
5          self.name = name
6          self.food = 0
7      def eat(self):
8          self.food = self.food + 1
9      def speak(self):
10         print(self.name, "is an animal")
11
12 class Dog(Animal):
13     def __init__(self,name):
14         super().__init__(name)
15     def speak(self):
16         print(self.name, "says woof!")
17
18 def main():
19     mesa = Dog("Mesa")
20     mesa.eat()
21     mesa.speak()
22
23 disassembler.disassemble(Animal)
24 disassembler.disassemble(Dog)
25 disassembler.disassemble(main)
```

```
Class: Animal
BEGIN
    Function: __init__/2
    Constants: None, 0
    Locals: self, name
    Globals: name, food
    BEGIN
              LOAD_FAST    1
              LOAD_FAST    0
              STORE_ATTR   0
              LOAD_CONST   1
              LOAD_FAST    0
              STORE_ATTR   1
              LOAD_CONST   0
              RETURN_VALUE
    END
    Function: eat/1
    Constants: None, 1
    Locals: self
    Globals: food
    BEGIN
              LOAD_FAST   0
              ...
              RETURN_VALUE
    END
    Function: speak/1
    Constants: None, "is an animal"
    Locals: self
    Globals: print, name
    BEGIN
              LOAD_GLOBAL   0
              ...
              RETURN_VALUE
    END
END
```

Fig. 3.16 Inheritance in JCoCo - part 1

Practice 3.11 Code was omitted in Figs. 3.16 and 3.17 for brevity in the chapter. Pick a method and complete the assembly code according to the original Python code from which it is derived.

You can check your answer(s) in Section 3.17.11.

```
Class: Dog(Animal)
BEGIN
    Function: __init__/2
    Constants: None
    Locals: self, name
    FreeVars: __class__
    Globals: super, __init__
    BEGIN
              LOAD_GLOBAL     0
              CALL_FUNCTION  0
              LOAD_ATTR  1
              LOAD_FAST    1
              CALL_FUNCTION    1
              POP_TOP
              LOAD_CONST 0
              RETURN_VALUE
    END
    Function: speak/1
    Constants: None, "says woof!"
    Locals: self
    Globals: print, name
    BEGIN
              LOAD_GLOBAL   0
              ...
              RETURN_VALUE
    END
END
Function: main/0
Constants: None, "Mesa"
Locals: mesa
Globals: Dog, eat, speak
BEGIN
          LOAD_GLOBAL  0
          LOAD_CONST   1
          CALL_FUNCTION  1
          STORE_FAST   0
          ...
          RETURN_VALUE
END
```

Fig. 3.17 Inheritance in JCoCo - part 2

3.13.2 Dynamically Created Classes

The previous section demonstrates declaring a class and creating objects in the JCoCo assembly language. It is also possible to create a class dynamically, at run-time. This also makes it possible to define class variables if desired. A class variable is a variable assigned to the class instead of instances of the class (i.e. objects). Consider this Python program where dogNumber is a class variable that can be used to count the number of instances of an object that have been created. Building this class requires that extra code be executed when building the class. The class variable must be initialized to 0.

```
import disassembler

def main():

    class Dog:
        dogNumber = 0

        def __init__(self,name):
            self.name = name
            self.id = Dog.dogNumber
            Dog.dogNumber += 1

        def speak(self):
            print("Dog number: ", self.id)

    x = Dog("Mesa")
    y = Dog("Sequoia")

    x.speak()
    y.speak()

disassembler.disassemble(main)
```

The assembly code for this program looks a bit different than the previous section. Instead of seeing a *Dog* class, there is a *Dog* function. The *Dog* function becomes the *Dog* class as a result of its execution. To explain what happens, let's start with Fig. 3.19 which contains the code for the main function. Line 55 of this code contains the *LOAD_BUILD_CLASS* instruction. This instruction loads a built-in function which builds a class from two arguments. A closure is one argument. As was stated in Sect. 3.11, a closure is code and an environment. We will visit this again in chapters 4 and 6. A closure is both code and the environment (i.e. the collection of variables) in which the code should be executed. The closure in this example is responsible for building the class' contents including its class variable and the methods of the class, which if you recall from earlier in this section are really functions until an object is instantiated.

The other argument to the built-in class builder function is the name of the class. Line 56 creates a closure. Line 57 buildes a tuple from the closure. Line 58 loads the code for the class initialization (i.e. the code for the *Dog* function). Line 59 builds the closure with the tuple and the code. Line 60 loads the name of the class onto the operand stack. Line 61 then calls the built-in class builder function passing to it the closure and the name of the class.

The built-in class builder function then does some housekeeping by creating a class instance, naming it *Dog* since that was passed as the name of the class, and calls the *Dog* function to complete the class instantiation. This takes us to the code on line 33 in Fig. 3.18.

```
Function: main/0
    Function: Dog/1
        Function: __init__/2
        Constants: None, 1
        Locals: self, name
        FreeVars: Dog
        Globals: name, dogNumber, id
        BEGIN
                  LOAD_FAST                      1
                  LOAD_FAST                      0
                  STORE_ATTR                     0
                  LOAD_DEREF                     0
                  LOAD_ATTR                      1
                  LOAD_FAST                      0
                  STORE_ATTR                     2
                  ...
                  RETURN_VALUE
        END
        Function: speak/1
        Constants: None, "Dog number: "
        Locals: self
        Globals: print, id
        BEGIN
                  LOAD_GLOBAL                    0
                  ...
                  RETURN_VALUE
        END
    Constants: 0, code(__init__), code(speak), None
    Locals: __locals__
    FreeVars: Dog
    Globals: __name__, __module__, dogNumber, __init__, speak
    BEGIN
              LOAD_FAST                      0
              STORE_LOCALS
              LOAD_NAME                      0
              STORE_NAME                     1
              LOAD_CONST                     0
              STORE_NAME                     2
              LOAD_CLOSURE                   0
              BUILD_TUPLE                    1
              LOAD_CONST                     1
              MAKE_CLOSURE                   0
              STORE_NAME                     3
              LOAD_CONST                     2
              MAKE_FUNCTION                  0
              STORE_NAME                     4
              LOAD_CONST                     3
              RETURN_VALUE
    END
```

Fig. 3.18 Dynamically created class - part 1

```
Constants: None, code(Dog), "Dog", "Mesa", "Sequoia"
Locals: x, y
CellVars: Dog
Globals: speak
BEGIN
          LOAD_BUILD_CLASS
          LOAD_CLOSURE                  0
          BUILD_TUPLE                   1
          LOAD_CONST                    1
          MAKE_CLOSURE                  0
          LOAD_CONST                    2
          CALL_FUNCTION                 2
          STORE_DEREF                   0
          LOAD_DEREF                    0
          LOAD_CONST                    3
          CALL_FUNCTION                 1
          STORE_FAST                    0
          ...
          RETURN_VALUE
END
```

Fig. 3.19 Dynamically created class - part 2

The *STORE_LOCALS* instruction on line 34 deserves some explanation. The local variables of the function *Dog* are a dictionary or map from strings (i.e. the names of the variables) to their values. When the built-in class builder function calls the *Dog* function to complete the construction of the class, it passes the dictionary for the class into the function. This dictionary is the dictionary of the class *Dog* and by executing the *STORE_LOCALS* instruction the dictionary also becomes the dictionary of locals for the *Dog* function. So, anything that is stored in the local variables will then be stored in the class instance. This sharing of the local variables dictionary and the class dictionary simplifies the class construction by making any variables stored in the *Dog* function also named variables, including named methods, in the class instance.

Line 35 of Fig. 3.18 stores *Dog* as the module name. The *__name__* attribute is already set to *Dog* by the built-in class builder function. So line 36 gives the same name to the *__module__* attribute.

Lines 37 and 38 initialize the class variable *dogNumber* to 0. Line 39 begins the work of adding the functions into the class instance, which will be instantiated to methods when a *Dog* instance is created. The first function to be stored is the *__init__* constructor which happens on lines 39–43. Lines 44–46 store the *speak* function in the class. The reason the constructor takes a bit more work is because it references and increments the class variable *dogNumber* and because of this the constructor needs both the code and the environment to executed correctly. The *speak* method does not reference any class variables so no environment is needed. The *MAKE_FUNCTION* instruction builds a closure with an empty environment.

While classes can be built either dynamically (i.e. at run-time) or statically using the assembly language syntax, the disassembler will use dynamic allocation when the environment is used in one of the instance methods of the class. Using the environment requires a closure and closures can be constructed during dynamic allocation of the class.

Practice 3.12 In some detail, describe the contents of the operand stack before and after the built-in class builder function is called to create a class instance.
You can check your answer(s) in Section 3.17.12.

3.14 Chapter Summary

An understanding of assembly language is key to learning how higher level programming languages work. This chapter introduced assembly language programming through a series of examples, drawing parallels between Python and Python virtual machine or JCoCo instructions. The use of a disassembler was key to gaining this insight and is a great tool to be able to use with any platform.

Most of the key constructs of programming languages were presented as both Python programs and JCoCo programs. The chapter concluded by covering classes, inheritance, and dynamic class creation.

The assembly language covered in this chapter comes up again in Chaps. 4 and 6. Chapter 4 covers the implementation of the JCoCo virtual machine and Chap. 6 implements a high-level functional language compiler that produces JCoCo assembly language programs.

JCoCo is an assembler/virtual machine for Python virtual machine instructions. Of course, there are other assembly languages. MIPS is a CPU architecture that has wide support for writing assembly language programs including a MIPS simulator called SPIM. In fact, assemblers are available for pretty much any hardware/operating system combination in use today. Intel/Linux, Intel/Windows, Intel/Mac OS X all support assembly language programming. The Java Virtual Machine can be programmed with the instructions of the JVM using a java assembler called Jasmin. Assembly language is the fundamental language that all higher level programming languages use in their implementations.

3.15 Review Questions

1. How do the Python virtual machine and JCoCo differ? Name three differences between the two implementations.

2. What is a disassembler?
3. What is an assembler?
4. What is a stack frame? Where are they stored? What goes inside a stack frame?
5. What is the purpose of the block stack and where is it stored?
6. What is the purpose of the Program Counter?
7. Name an instruction that is responsible for creating a list object and describe how it works.
8. Describe the execution of the *STORE_FAST* and *LOAD_FAST* instructions.
9. How can JCoCo read a line of input from the keyboard?
10. What is the difference between a disassembled Python program and an assembled JCoCo program? Provide a short example and point out the differences.
11. When a Python while loop is implemented in JCoCo, what is the last instruction of the loop and what is its purpose?
12. What do exception handling and loops have in common in the JCoCo implementation?
13. What is lazy evaluation and why is it important to Python and JCoCo?
14. What is a closure and why are closures needed?
15. How do you create an instance of a class in JCoCo? What instructions must be executed to create objects?
16. Write a class, using JCoCo, and create some instances of the class.

3.16 Exercises

1. Consulting the JCoCo assembly language program in *the solution to exercise 3.2*, provide the contents of the operand stack after each instruction is executed.
2. Write a JCoCo program which reads an integer from the user and then creates a list of all the even numbers from 0 up to and including that integer. The program should conclude printing the list to the screen. Test your program with JCoCo to be sure it works. Do this with as few instructions as possible.
3. With as few instructions as possible add some exception handling to the previous exercise to print "You didn't enter an integer!" if the user fails to enter an integer in their program.
4. In as few instructions as possible write a JCoCo program that computes the sum of the first n integers where the non-negative n is read from the user.
5. Write a recursive JCoCo program that adds up the first n integers where n is read from the user. Remember, there must be a base case that comes first in this function and the recursive case must be called on something smaller which is used in computing the solution to the whole problem.
6. Write a Rational class that can be used to add and multiply fractions together. A Rational number has an integer numerator and denominator. The *__add__* method is needed to add together Rationals. The *__mul__* method is for multiplication. To get fractions in reduced format you may want to find the greatest common divisor of the numerator and the denominator when creating a Rational

number. Write this code in Python first, then disassemble it to get started with this assignment.

You may wish to write the greatest common divisor function *gcd* as part of the class although no self parameter is needed for this function. The greatest common divisor of two integers, x and y, can be defined recursively. If y is zero then x is the greatest common divisor. Otherwise, the greatest common divisor of x and y is equal to the greatest common divisor of y and the remainder x divided by y. Write a recursive function called *gcd* to determine the greatest common divisor of x and y.

Disassemble the program and then look for ways of shortening up the JCoCo assembly language program. Use the following main program in your code.

```
import disassembler

def main():
    x = Rational(1,2)
    y = Rational(2,3)
    print(x+y)
    print(x*y)
disassembler.disassemble(Rational)
disassembler.disassemble(main)
```

From this code you should get the following output which matches the output you should get had this been a Python program. Remember to use Python 3.2 when disassembling your program. And, remember to turn in as short a program as possible while getting this output below from the main program given above.

```
7/6
1/3
```

3.17 Solutions to Practice Problems

These are solutions to the practice problems. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

3.17.1 Solution to Practice Problem 3.1

The assembly code in Fig. 3.2 blindly pops the *None* at the end and then pushes *None* again before returning from main. This can be eliminated resulting in two fewer instructions. This would also mean that *None* is not needed in the constants, but this was not eliminated below.

```
Function: main/0
Constants: None,
    "Enter your name: ", "Enter your age: ",
    ", a year from now you will be",
    1, "years old."
Locals: name, age
Globals: input, int, print
BEGIN
          LOAD_GLOBAL                    0
          LOAD_CONST                     1
          CALL_FUNCTION                  1
          STORE_FAST                     0
          LOAD_GLOBAL                    1
          LOAD_GLOBAL                    0
          LOAD_CONST                     2
          CALL_FUNCTION                  1
          CALL_FUNCTION                  1
          STORE_FAST                     1
          LOAD_GLOBAL                    2
          LOAD_FAST                      0
          LOAD_CONST                     3
          BINARY_ADD
          LOAD_FAST                      1
          LOAD_CONST                     4
          BINARY_ADD
          LOAD_CONST                     5
          CALL_FUNCTION                  3
          RETURN_VALUE
END
```

3.17.2 Solution to Practice Problem 3.2

As in practice 3.1 the *POP_TOP* and *LOAD_CONST* from the end can be eliminated. In the if-then-else code both the *then* part and the *else* part execute exactly the same *STORE_FAST* instruction. That can be moved after the if-then-else code and written just once, resulting in one less instruction and three less overall. Furthermore, if we move the *LOAD_GLOBAL* for the call to *print* before the if-then-else statement, we can avoid storing the maximum value in *z* at all and just leave the result on the top of the operand stack: either *x* or *y*. By leaving the bigger of *x* or *y* on the top of the stack, the call to *print* will print the correct value. This eliminates five instructions from the original code.

```
Function: main/0
Constants: None, 5, 6
Locals: x, y
Globals: print
BEGIN
          LOAD_CONST                     1
          STORE_FAST                     0
          LOAD_CONST                     2
```

```
            STORE_FAST              1
            LOAD_GLOBAL             0
            LOAD_FAST               0
            LOAD_FAST               1
            COMPARE_OP              4
            POP_JUMP_IF_FALSE label00
            LOAD_FAST               0
            JUMP_FORWARD      label01
label00:    LOAD_FAST               1
label01:    CALL_FUNCTION           1
            RETURN_VALUE
END
```

It is worth noting that the code above is exactly the disassembled code from this Python program.

```
import disassembler
def main():
    x=5
    y=6
    print(x if x > y else y)

disassembler.disassemble(main)
```

When main is called, this code prints the result of a *conditional expression*. The if-then-else expression inside the print statement is different than an if-then-else statement. An if-then-else statement updates a variable or has some other side-effect. An if-then-else expression, or *conditional expression* as it is called in Python documentation, yields a value: either the *then* value or the *else* value. In the assembly language code we see that the yielded value is passed to the print function as its argument.

3.17.3 Solution to Practice Problem 3.3

```
Function: main/0
Constants: None, 5, 6
Locals: x, y
Globals: print
BEGIN
            LOAD_CONST              1
            STORE_FAST              0
            LOAD_CONST              2
            STORE_FAST              1
            LOAD_FAST               0
            LOAD_FAST               1
            COMPARE_OP              1
            POP_JUMP_IF_TRUE  label00
            LOAD_GLOBAL             0
            LOAD_FAST               0
            CALL_FUNCTION           1
```

```
             POP_TOP
label00:     LOAD_GLOBAL                  0
             LOAD_FAST                    1
             CALL_FUNCTION                1
             RETURN_VALUE
END
```

3.17.4 Solution to Practice Problem 3.4

The following code behaves differently if the *BREAK_LOOP* instruction is removed from the program.

```
Function: main/0
Constants: None, 7, 6
Locals: x, y
Globals: print
BEGIN
             SETUP_LOOP             label01
             LOAD_CONST                   1
             STORE_FAST                   0
             LOAD_CONST                   2
             STORE_FAST                   1
             LOAD_FAST                    0
             LOAD_FAST                    1
             COMPARE_OP                   1
             POP_JUMP_IF_TRUE       label00
             BREAK_LOOP
             LOAD_GLOBAL                  0
             LOAD_FAST                    0
             CALL_FUNCTION                1
             POP_TOP
label00:     POP_BLOCK
label01:     LOAD_GLOBAL                  0
             LOAD_FAST                    1
             CALL_FUNCTION                1
             RETURN_VALUE
END
```

3.17.5 Solution to Practice Problem 3.5

This is the hello world program with exception handling used to raise and catch an exception. This solution does not include code for *finally* handling in case an exception happened while handling the exception. It also assumes the exception will match when thrown since JCoCo only supports one type of exception.

```
Function: main/0
Constants: None, "Hello World!"
Locals: ex
```

```
Globals: Exception, print
BEGIN
          SETUP_EXCEPT            label00
          LOAD_GLOBAL                   0
          LOAD_CONST                    1
          CALL_FUNCTION                 1
          RAISE_VARARGS                 1
          POP_BLOCK
          JUMP_FORWARD            label01
label00:  LOAD_GLOBAL                   1
          ROT_TWO
          CALL_FUNCTION                 1
          POP_TOP
          POP_EXCEPT
label01:  LOAD_CONST                    0
          RETURN_VALUE
END
```

3.17.6 Solution to Practice Problem 3.6

This program adds 5 and 6 together using the *__add__* magic method associated with integer objects. First 5 is loaded onto the operand stack. Then *LOAD_ATTR* is used to load the *__add__* of the 5 object onto the stack. This is the function. The argument to *__add__* is loaded next which is the 6. The 6 is loaded by the *LOAD_CONST* instruction. Then *__add__* is called with one argument. The 11 is left on the operand stack after the function call. It is stored in *x*, the *print* is loaded, *x* is loaded onto the operand stack, and *print* is called to print the value. Since *print* leaves *None* on the stack, that value is returned from the main function.

```
Function: main/0
Constants: None, 5, 6
Locals: x
Globals: __add__, print
BEGIN

          LOAD_CONST                    1
          LOAD_ATTR                     0
          LOAD_CONST                    2
          CALL_FUNCTION                 1
          STORE_FAST                    0
          LOAD_GLOBAL                   1
          LOAD_FAST                     0
          CALL_FUNCTION                 1
          RETURN_VALUE
END
```

3.17.7 Solution to Practice Problem 3.7

```
Function: main/0
Constants: None, "Enter a string: "
Locals: x, a
Globals: input, print
BEGIN
          LOAD_GLOBAL                 0
          LOAD_CONST                  1
          CALL_FUNCTION               1
          STORE_FAST                  0
          SETUP_LOOP            label02
          LOAD_FAST                   0
          GET_ITER
label00:  FOR_ITER              label01
          STORE_FAST                  1
          LOAD_GLOBAL                 1
          LOAD_FAST                   1
          CALL_FUNCTION               1
          POP_TOP
          JUMP_ABSOLUTE         label00
label01:  POP_BLOCK
label02:  LOAD_CONST                  0
          RETURN_VALUE
END
```

3.17.8 Solution to Practice Problem 3.8

A cell variable is needed if an inner function makes a modification to a variable that is located in the outer function. Consider the JCoCo program below. Without the cell the program below would print 10 to the screen and with the cell it prints 11. Why is that? Draw the run-time stack both ways to see what happens with and without the cell variable.

```
Function: f/1
    Function: g/1
    Constants: None, 1
    Locals: y
    FreeVars: x
    BEGIN
              LOAD_DEREF              0
              LOAD_CONST              1
              BINARY_ADD
              STORE_DEREF             0
              LOAD_DEREF              0
              LOAD_FAST               0
              BINARY_ADD
              RETURN_VALUE
    END
```

```
Constants: None, code(g)
Locals: x, g
CellVars: x
BEGIN
          LOAD_CLOSURE                    0
          BUILD_TUPLE                     1
          LOAD_CONST                      1
          MAKE_CLOSURE                    0
          STORE_FAST                      1
          LOAD_FAST                       1
          LOAD_DEREF                      0
          CALL_FUNCTION                   1
          LOAD_DEREF                      0
          BINARY_ADD
          RETURN_VALUE
END
Function: main/0
Constants: None, 3
Globals: print, f
BEGIN
          LOAD_GLOBAL                     0
          LOAD_GLOBAL                     1
          LOAD_CONST                      1
          CALL_FUNCTION                   1
          CALL_FUNCTION                   1
          POP_TOP
          LOAD_CONST                      0
          RETURN_VALUE
END
```

Interestingly, this program cannot be written in Python. The closest Python equivalent of this program is given below. However, it is not the equivalent of the program written above. In fact, the program below won't even execute. There is an error on the line *x = x + 1*. The problem is that as soon as Python sees *x* = in the function *g*, it decides there is another *x* that is a local variable in *g*. But, then *x = x + 1* results in an error because *x* in *g* has not yet been assigned a value.

```
def f(x):
    def g(y):
        x=x+1
        return x + y
    return g(x) + x
def main():
    print(f(3))
main()
```

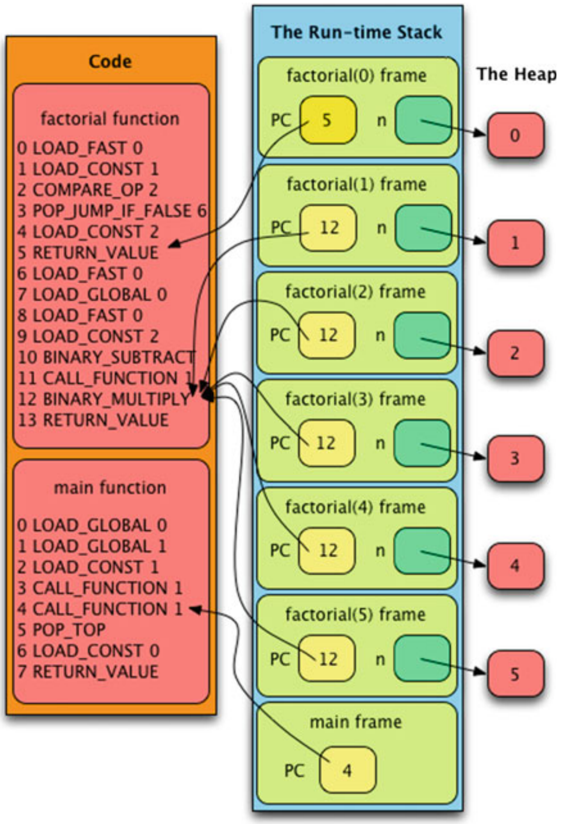


Fig. 3.20 Execution of fact.casm

3.17.9 Solution to Practice Problem 3.9

A couple things to notice in Fig. 3.20. The run-time stack contains one stack frame for every function call to factorial. Each of the stack frames, except the one for the *main* function, point at the *factorial* code. While there is only one copy of each function's code, there may be multiple stack frames executing the code. This happens when a function is recursive. There also multiple *n* values, one for each stack frame. Again this is expected in a recursive function.

3.17.10 Solution to Practice Problem 3.10

Python is a very transparent language. It turns out there is function called *dir* that can be used to print the attributes of an object which are the keys of its dictionary.

The dictionary maps names (i.e. strings) to the attributes of the object. The following strings map to their indicated values.

- *__init__* is mapped to the constructor code.
- *eat* is mapped to the eat method.
- *speak* is mapped to the speak method.
- *food* is mapped to an integer.
- This is all that is mapped by JCoCo. However, if you try this in Python you will discover that a number of other methods are mapped to default implementations of magic methods in Python including a hash method, comparison methods like equality (i.e. *__eq__*), a repr method, a str method, and a number of others.

3.17.11 Solution to Practice Problem 3.11

```
Class: Animal
BEGIN
    Function: eat/1
    Constants: None, 1
    Locals: self
    Globals: food
    BEGIN
              LOAD_FAST                    0
              LOAD_ATTR                    0
              LOAD_CONST                   1
              BINARY_ADD
              LOAD_FAST                    0
              STORE_ATTR                   0
              LOAD_CONST                   0
              RETURN_VALUE
    END
    Function: __init__/2
    Constants: None, 0
    Locals: self, name
    Globals: name, food
    BEGIN
              LOAD_FAST                    1
              LOAD_FAST                    0
              STORE_ATTR                   0
              LOAD_CONST                   1
              LOAD_FAST                    0
              STORE_ATTR                   1
              LOAD_CONST                   0
              RETURN_VALUE
    END
    Function: speak/1
    Constants: None, "is an animal"
    Locals: self
    Globals: print, name
    BEGIN
              LOAD_GLOBAL                          0
              LOAD_FAST                            0
              LOAD_ATTR                            1
              LOAD_CONST                           1
              CALL_FUNCTION                        2
              POP_TOP
              LOAD_CONST                           0
              RETURN_VALUE
    END
END
Class: Dog(Animal)
BEGIN
    Function: __init__/2
    Constants: None
    Locals: self, name
    FreeVars: __class__
    Globals: super, __init__
    BEGIN
              LOAD_GLOBAL                          0
              CALL_FUNCTION                        0
              LOAD_ATTR                            1
              LOAD_FAST                            1
              CALL_FUNCTION                        1
              POP_TOP
              LOAD_CONST                           0
              RETURN_VALUE
    END
    Function: speak/1
    Constants: None, "says woof!"
    Locals: self
    Globals: print, name
    BEGIN
              LOAD_GLOBAL                          0
              LOAD_FAST                            0
              LOAD_ATTR                            1
              LOAD_CONST                           1
              CALL_FUNCTION                        2
              POP_TOP
              LOAD_CONST                           0
              RETURN_VALUE
    END
END
Function: main/0
Constants: None, "Mesa"
Locals: mesa
Globals: Dog, eat, speak
BEGIN
          LOAD_GLOBAL                          0
          LOAD_CONST                           1
          CALL_FUNCTION                        1
```

```
                 STORE_FAST                          0
                 LOAD_FAST                           0
                 LOAD_ATTR                           1
                 CALL_FUNCTION                       0
                 POP_TOP
                 LOAD_FAST                           0
                 LOAD_ATTR                           2
                 CALL_FUNCTION                       0
                 POP_TOP
                 LOAD_CONST                          0
                 RETURN_VALUE
END
```

3.17.12 Solution to Practice Problem 3.12

To get ready to execute the built-in class builder function the stack must contain the following in order from the top of the stack down: The name of the class is on the top of the operand stack. Below the name is the closure of the class initializing function and its environment. Below that is the built-in class builder function itself. The *CALL_FUNCTION* instruction is executed with two arguments indicated to call the class builder.

Upon its return, the two arguments and the class builder function have been popped from the stack and the instance of the class is left on the operand stack. This class instance may then be stored as a reference from some known location, likely by a *STORE_FAST* instruction.

Object-Oriented Programming 4

In this chapter you'll learn about the implementation of the JCoCo virtual machine while at the same time you'll be introduced to the Java and C++, two statically typed object-oriented programming languages. The primary focus of the chapter is on learning advanced object-oriented programming using Java and C++. Statically typed languages, like C++ and Java, differ from dynamically typed languages like Python in the way that type errors are caught. When running a Python program a type error can occur in any branch of code. One of the big problems with Python programming is that these errors may exist until every possible path in a Python program is executed. Testing Python code takes considerable effort to ensure every possible path is executed. While thorough testing is always a good idea, these type errors may not be discovered until much later in the development cycle.

Java and C++ programs are statically typed. This means that type errors are found at compile time, when the program is translated into executable format, without executing the program at all. Programmers must declare the types of all values or the compiler must be able to infer their type from the context of expressions in the program. With the declaration of value types in C++ and Java programs, the programmer is notified if any operation is not allowed without executing a single line of code. Run-time errors are still possible, but those run-time errors are due to logic problems and not due to type errors.

The JCoCo implementation will serve as nice examples while learning Java and C++. We'll compare Java and C++ when appropriate to show you the differences and similarities between the two languages. In the interest of seeing the big picture, we'll start with an overview of the JCoCo implementation as pictured in Fig. 4.1. JCoCo reads an input file which must be formatted according to the grammar specified in Sect. A.1. Two parts of the JCoCo implementation, the *scanner* and the *parser* are responsible for reading the input file. The scanner is implemented as a finite state machine. The parser is written as a top-down parser. The parser produces a list of *function* and *class* definitions which make up the *abstract syntax tree* definition of

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics
in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_4

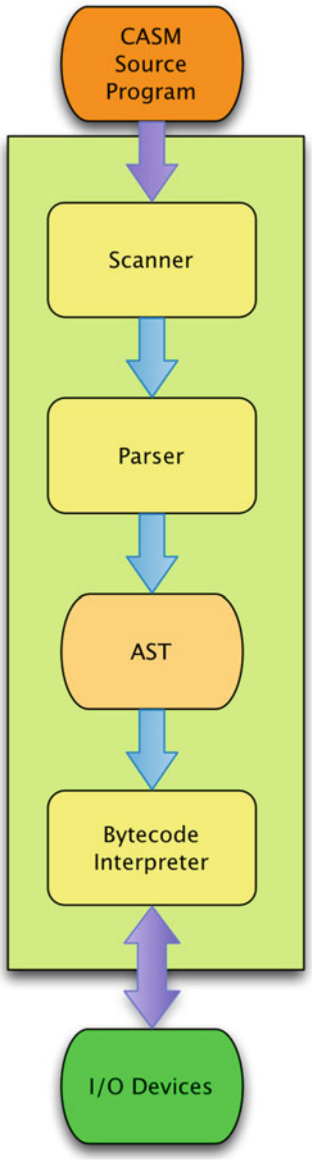


Fig. 4.1 JCoCo

the program. The remainder of the JCoCo implementation makes up the bytecode interpreter.

The bytecode interpreter evaluates the abstract syntax tree (i.e. AST) which consists of *function* and *class* objects. The AST is interpreted within the context of *frame* objects. A frame is one activation record on the run-time stack of the executing program. When the program begins, the interpreter starts execution at the *main* function, creating a frame for it and starting execution at the first instruction.

The examples of Java and C++ used within this chapter will come from the implementation of the scanner, parser, function, class, and frame classes along with classes for types and instances of types like integers, floats, strings, and lists.

JCoCo is written in Java. A similar project, CoCo, was earlier written in C++. JCoCo is an improved, more fully developed version of CoCo, rewritten in Java. JCoCo is a large project consisting of 56 source files and around 8,900 lines of code. Don't be intimidated, a lot of the code is repetitive. With such a large program, structuring it correctly is of the utmost importance. The JCoCo virtual machine is an interpreter of bytecode instructions somewhat like the Java Virtual Machine (i.e. JVM). The JCoCo interpreter reads assembly language source files called *CASM* files as you learned about in the last chapter.

Much of the design of CoCo and JCoCo is similar. JCoCo includes support for programmer-defined classes. CoCo does not support classes definition. Otherwise, the two implementations are very similar. The early part of this chapter will present

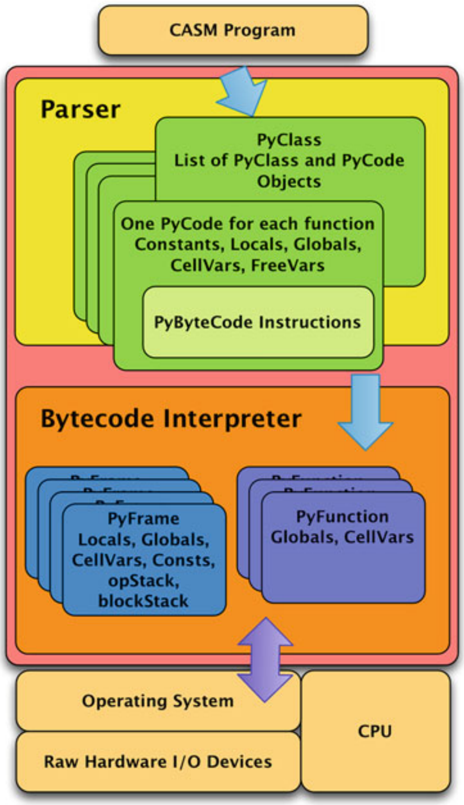

CASM Program
Parser
PyClass
List of PyClass and PyCode Objects
One PyCode for each function
Constants, Locals, Globals, CellVars, FreeVars
PyByteCode Instructions
Bytecode Interpreter
PyFrame
Locals, Globals, CellVars, Consts, opStack, blockStack
PyFunction
Globals, CellVars
Operating System
Raw Hardware I/O Devices
CPU

Fig. 4.2 The JCoCo virtual machine

examples of both the Java and C++ implementations when appropriate to compare and contrast the two languages. Later sections will explore the details of the Java implementation of JCoCo.

Like other interpreters, the JCoCo/CoCo implementation is divided into some logical components: the scanner, parser, and bytecode interpreter. The *scanner* reads characters from the CASM source file and creates objects called tokens. The last chapter had many examples of CASM files. Tokens from a CASM file like the one in Sect. 3.2 and pictured in Fig. 4.2 include a *Function* keyword, a colon, a *main* identifier, a slash, an integer 0, another keyword *Constants*, another colon, a *None* keyword, and so on. These tokens are returned one at a time to the parser when the parser requests another token.

The *parser* reads the tokens one at a time from the scanner and uses them while parsing the source file according to the grammar for JCoCo given in *Appendix A*. The grammar given there is LL(1) so the parser is implemented as a recursive descent parser. Each non-terminal of the grammar is a function in the parser. The right hand sides of rules for each nonterminal defines the body of each function in the parser. There will be more on this later in the chapter. The result of parsing the source file is an Abstract Syntax Tree, or AST. This AST is an internal representation of the program to be interpreted.

The JCoCo bytecode interpreter is the part of the program, given the AST, that interprets the byte code instructions of each function. As the instructions are executed, the virtual machine interacts with I/O devices like the keyboard and the screen. Bytecode interpretation is the responsibility of several parts of the JCoCo implementation as you will read later. The last part of this chapter has a detailed explanation of the implementation of the JCoCo virtual machine.

4.1 The Java Environment

In Chap. 1 we learned that Java originated as part of the Green project. Today, Java is a robust language that contains many features that make it convenient for programmers while also being efficient and powerful. Java actually consists of two important tools: a Java Virtual Machine (i.e. JVM) and a Java compiler that compiles from Java source code to Java bytecode (which is what the JVM executes).

Let's consider the hello world example, written in Java. The Java program is given in Fig. 4.3. This program must be saved in a file called *HelloWorld.java*. The program is compiled and run using commands like this.

```
My Mac> javac HelloWorld.java
My Mac> java HelloWorld
Hello World
My Mac>
```

The *javac* program is the Java compiler which translates a Java *source program*, ending in an extension of *.java*, to a *bytecode file*, ending in *.class*. The bytecode file

```
public class HelloWorld {
 public static
  void main(String args[]) {
  System.out.println("Hello World");
 }
}
```

Fig. 4.3 Hello world

is a binary file that is machine readable, but not human readable. The bytecode file is read by the *Java Virtual Machine* or *JVM* which is actually the program called *java* or *java.exe* if you are running on a Windows machine. The JVM interacts with the operating system to execute the bytecode file.

Typically, a Java program consists of many source code files which are compiled into many bytecode files. When programming in Java, each source code file must be named the same as the public class within the source code file. For instance, the code in Fig. 4.3 must be in a file called *HelloWorld.java* because the public class is called *HelloWorld*. Every Java source code file can have exactly one public class.

Writing a non-trivial Java program involves creating many classes and therefore many source files. When a Java project is compiled, the Java compiler looks at the dates of all source files and all bytecode files. Every time a file is created or modified its last modification date is changed. This is information that is maintained by every operating system including Mac OS X, Microsoft Windows, and Linux. If any bytecode file is found to be older than its corresponding source file, then the source file is recompiled to produce a new bytecode file with a newer date than its source code file. This mechanism of using timestamps to determine what needs to be recompiled is called a *make facility* for historical reasons which we'll revisit in the next section.

Practice 4.1 Another program is written and compiled. Here is the error message from the compiler. What can you discern from the compile message? Why would this be important to Java?

```
test.java:1: error: class Test is public,
   should be declared in a file named Test.java
public class Test {
       ^
1 error
My Mac>
```

You can check your answer(s) in Section 4.34.1.

4.2 The C++ Environment

C++ and Java share a lot of syntax. C++ was designed first with Java starting development about 10 years later. As mentioned in chapter one, Bjarne Stroustrup was developing C++ during the early eighties. He designed the language to be backward compatible with C so there were some decisions already made for him like the need for *separate compilation* and the presence of a *macro processor*. C++ is one of the most widely used object-oriented languages today and continues to evolve. A standards committee now oversees C++ with regular revisions to the language like the C++11 revision which came out in 2011 and the 2014 version which contained small changes over the 2011 version. A new version was formally accepted late in 2017. Development of the C++ language is ongoing.

Using C/C++ for a programming project does not come without some risks. A significant problem, perhaps the most persistent problem over time, with C/C++ programs are memory leaks. C/C++ programmers must be disciplined in their allocation and deallocation of memory. It is common that programs that run for a long time will have a memory leak that has to be tracked down, which is a difficult task. In many languages a garbage collector takes care of freeing memory that is no longer needed by a program. A garbage collector cannot safely be included as part of C and C++ programs. Both C and C++ are designed to give the programmer maximum control. This means that more responsibility is left to the programmer and as a result programmers need to be very disciplined when using C/C++. There is more on this in Sect. 4.7.

C and C++ have many uses including operating system development, timing critical software, and detailed hardware access. Learning to program in C++ well will take you a long ways towards being a great programmer in any language. This chapter won't teach you everything you need to know to become a C++ programmer. That could be and is the topic of many books. But this chapter will introduce you to many of the important concepts and skills you'll need to become a good C++ programmer.

Like Java, C++ programs must be compiled before you can run the them. Java programs are compiled to Java bytecode and the bytecode is run on the JVM. C++ programs are compiled into the machine language of the CPU that will execute them. The operating system of the computer where a C++ program runs is responsible for loading the executable program and getting it ready to run but otherwise a compiled C++ program runs directly on the CPU of the machine for which it was compiled.

Figure 4.5 depicts the compilation process for C++ programs. Examine Fig. 4.4 to contrast that with the execution of Java programs. The C++ environment looks more complicated. But everything in the green box is actually accomplished using one compile command. Figure 4.6 contains the classic *hello world* program written in C++.

To run the hello world program it must first be compiled. Figure 4.5 shows the process of compiling a C++ program like the one that appears in Fig. 4.6. First, the macro processor reads the *iostream* header file and combines it with the rest of the source file. The *iostream* header file does not include the code for streams. It just declares the streams and the operators used to write to the *out* stream. The combined

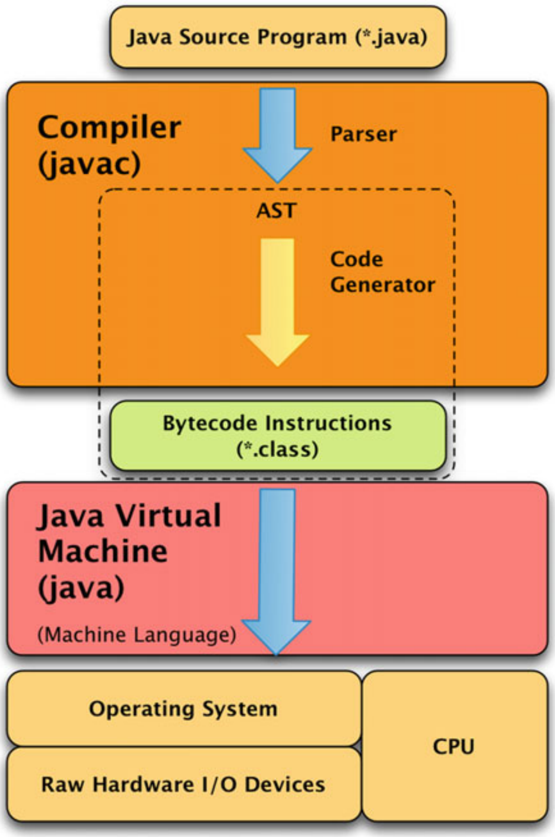


Fig. 4.4 Java compiler and virtual machine

program text is sent to the compiler which parses the program and generates machine language code using an assembler. The machine language code is then linked with the *iostream* library to produce the executable code. Thankfully, this whole process is encapsulated in one command. There will be more about both the macro processor and I/O streams in the next sections.

Executing the g++ command compiles the program as shown in Fig. 4.7. By default g++ produces a program called a.out. To execute the program you type *a.out* and the operating system will load and run it. The default *a.out* can be renamed or a different name can be provided on the compile command.

The $-g$ option in Fig. 4.8 tells g++ to include debugging information in the program. The *-o* tells g++ to name the executable program *hello* instead of *a.out*.

To compile a C++ program you must have a C++ compiler installed on your system. The g++ compiler used in Fig. 4.7 is the GNU C++ compiler. This compiler is available for Mac OS X, Linux, and Microsoft Windows.

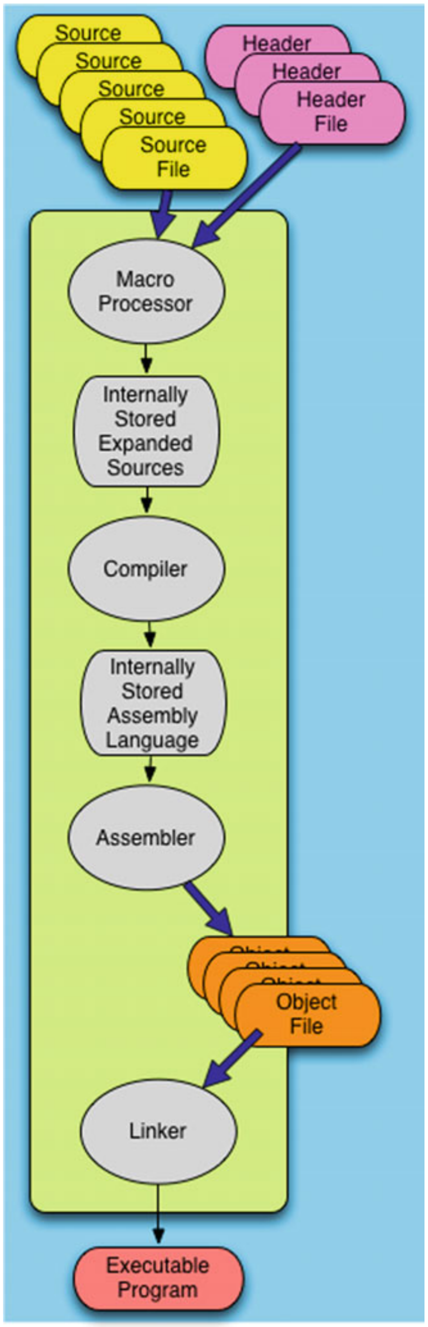


Fig. 4.5 C++ compile

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
  cout << "Hello World!"<< endl;
}
```

Fig. 4.6 hello.cpp

```
My Computer> g++ hello.cpp
My Computer> a.out
Hello World!
My Computer>
```

Fig. 4.7 Compiling hello.cpp

```
My Computer> g++ -g -o hello hello.cpp
My Computer> hello
Hello World!
My Computer>
```

Fig. 4.8 Include debug and name

4.2.1 The Macro Processor

The first line of the program in Fig. 4.6 is called a *macro processor directive*. The macro processor is a part of the C++ compiler that is responsible for pulling other files into the source program and sometimes for some simple editing of a source file to get it ready to be compiled. In this program the macro processor *includes* another file or library called *iostream*. The iostream file is called a header file because it defines functions and variables that exist in some other library or code on the system where it is compiled. Header files define the interfaces to these other libraries or code. When a header file is enclosed in angle braces, a less than/greater than pair, it is a system provided header file. More information on the macro processor and header files can be found in Sect. 4.9.1.

4.2.2 The Make Tool

A file system is the software and format that controls how files are stored on the hard disk of a computer. All operating systems have their own file systems and sometimes support multiple file systems. Microsoft Windows supports NTFS and Fat32 among others. Linux support ext2, ext3, reiserfs, and others. Mac OS X supports several file systems including HFS+. Every one of these file systems store attributes of every file including the date and time the file was last modified.

Make is a program that can be used to compile programs that are composed of modules and utilize separate compilation. C and C++ programs utilize separate compilation and typically you write a make file to compile programs written in these languages, or you use a tool to automatically create a make file. Java programs are not compiled via the make program because the make program is built into the Java compiler as was mentioned in the previous section.

The idea is simple. Every time a module is compiled by the C++ compiler it produces an object file. For instance, when *PyObject.cpp* is compiled, the C++ compiler writes a file called *PyObject.o*. For each of these files the date and time when it was last modified or created is stored with the file. After a compile the date on *PyObject.cpp* is older than the date on *PyObject.o*. When a programmer changes *PyObject.cpp*, its date will be newer than *PyObject.o*'s date.

Make uses this simple observation along with make rules to execute the compile commands necessary to make *PyObject.o*'s date newer than *PyObject.cpp*'s date. Here is a make rule for PyObject.cpp.

```
PyObject.o: PyObject.cpp PyObect.h
  g++ -g -c -std=c++0x PyObject.cpp
```

This rule says that *PyObject.o* must be newer than *PyObject.cpp* and *PyObject.h*. If either of these two files are newer then make will execute the command on the next line, which must be indented under the first line. The result of executing this compile command is to produce a new *PyObject.o* file with a newer date than either of the two source files.

To make the *coco* executable, all the object files must be linked together. To link everything together the first rule is written like this.

```
coco: main.o PyObject.o PyInt.o PyType.o ....
  g++ -o coco -std=c++0x main.o PyObject.o PyInt.o PyType.o ....
```

All 38 object files must be listed here. This says that the date on *coco*, the executable program, must be newer than the date on all its object files.

All these rules are placed in a file called *Makefile* in the same directory as the C++ source files. When *make* is invoked it will look for a file named *Makefile*. By keeping track of the dates, only the source files that have been updated will get recompiled and the *coco* executable will get recreated by linking together all the object files.

Writing a good *Makefile* is sometimes difficult and almost always error prone, so often there is a rule in the makefile called *clean*. Executing *make clean* will erase all object files so you can get a fresh compile. There are also tools like *autoreconf* that will generate a *Makefile* automatically with just a few inputs. Take a look at the *rebuild* script in the CoCo distribution to see how this might be used. To use *autoreconf* you must have the automake tools installed on your system. But if you do, you can execute

```
./rebuild
./configure
make
```

to build the entire CoCo Virtual Machine. Without the automake tools you should still be able to execute the *configure* and *make* commands to build CoCo.

Separate compilation in C++ programs means that each module in the program is compiled separately. Each object file, generated by the compilation of a module, is produced independently of the other source files. This is important because large C++ projects often contain hundreds of C++ source files. Separate compilation means that only the small piece a programmer changes needs to be recompiled if the interface (i.e. the header file) to other modules does not change. After compiling the source files to object files, the object files can be linked together to form an executable program. Linking is a very fast operation compared to compiling.

Practice 4.2 Using C++ there are no naming requirements for modules and classes like Java. So, when class *A* uses class *Test* both class *A* and class *Test* can be put into a file by any name. Why is this OK for C++ programs, but not for Java programs?
You can check your answer(s) in Section 4.34.2.

4.3 Namespaces

Line 2 of the program in Fig. 4.6 opens up the *std*, short for standard, namespace in the program. The first two lines of this C++ program are like importing a module in Python or a package in Java. When importing a module in Python the programmer writes an import statement like one of these two lines.

```
from iostream import * # merges the namespace with the current module
import iostream # preserves the namespace while importing the module
```

In Java the programmer would write an import statement somewhat like this, although not exactly.

```
  import java.iostream.cout
```

The Python equivalent of a namespace is a module. Python modules can be imported in one of two ways, preserving the namespace or merging it with the existing namespace. In Java packages are the equivalent of a namespace and selected classes and objects can be imported from a package. Namespaces are important in C++, Python, and Java, because without them there would be many potential name conflicts between header files and modules that would create compile errors and prevent

```
#include <iostream>
int main(int argc, char* argv[]) {
  std::cout <<
     "Hello World!"<< std::endl;
}
```

Fig. 4.9 Namespace std

programs from compiling, or in the case of Python - from running, that were otherwise correct programs. In C++ programs, if we didn't want to open the *std* namespace we could rewrite the program as shown in Fig. 4.9.

The safest way to program is to not open up namespaces or merge them together. But, that is also inconvenient since the whole name must be written each time. What is correct for your program depends on the program being written.

4.4 Dynamic Linking

Dynamic linking is related to namespaces, modules, and packages. Modern programming languages like C++ and Java are reliant on many libraries so programmers can solve problems instead of rewriting code that is common to more than one program. Libraries containing commonly used code are generally available to be used by programs written in a high level language, including C++ and Java programs. These libraries must be linked into your program to be able to use them. Figure 4.5 shows object files (i.e. modules) being linked together into a C++ program.

There is a problem with the picture in Fig. 4.5. Early C programs could be self-contained programs that relied on only a small number of system calls from the Unix operating system. However, modern C, C++, Java, and almost any other modern programming language are reliant on so many libraries that linking all of them together would be a problem in several ways.

- The size of the linked executable program would be huge taking up a lot of space in memory as it was executing.
- Any change in any library would require each program that uses it to be re-linked to get the new version of the library.
- There is no reason to have multiple copies of libraries, one for each program that uses it. This wastes space in addition to the overhead of having to manage multiple copies of libraries.

Modern languages don't *statically* link all the libraries that are required by a program. They *dynamically* link them. When you hear the word *dynamic* you should think *run-time*. Libraries are generally linked at run-time. Software, often part of the operating system, detects when a library is going to be used by a program and loads it into memory and links it to the program that requests its services as the program is executing. Microsoft Windows calls these dynamically linked libraries *DLL's*. Windows includes services that let libraries be written so they can be dynamically linked to programs as they execute. Mac OS X and Linux also have the ability to dynamically link libraries. C++ programs often dynamically link to libraries that are provided by the C++ run-time libraries and other libraries that may be required by a program but have been supplied with the program.

Java programs also use dynamic linking. In fact, dynamic linking is built into the very foundations of Java. The JVM loads bytecode files (i.e. modules) as needed in your Java program. Java programs consist of *.class* files, called bytecode files, which must be in the current working directory or in a directory on the class path. The class path is a list of directories, or folders, where the JVM looks for bytecode files. The class path is recorded in an environment variable called *CLASSPATH*. Here is one example of a class path.

```
export CLASSPATH=./DBBrowser/lib/mysql-connector-
java-5.1.17-bin.jar:.:$CLASSPATH
```

The class path is a list of folders, or directories, where these dynamically loaded *.class* files can be found. Sometimes a whole group of classes are written to implement some library. For instance, this class path includes *mysql-connector-java-5.1.17-bin.jar*. This is actually what is called a *JAR* file. A JAR file stands for Java Archive, and is a zipped up set of *.class* files that are stored in compressed format. Dynamically linked libraries are so common to Java programs that JAR files were added as a means to conveniently group and redistribute collections of classes for Java programs. The bytecode files found in a JAR file are organized into packages. Importing something like

```
import java.io.BufferedInputStream
```

in a Java program would cause the *BufferedInputStream* class to be dynamically linked from the *java.io* package, which is a library provided by the Java run-time environment.

4.5 Defining the Main Function

Lines 3–5 of the hello world program in Fig. 4.6 define the *main* function. Every C++ program must have one main function, and only one. The main function should return an integer and it is given an integer and an array of character arrays which are the command-line arguments. The command-line arguments are elaborated on in more detail in the section on arrays and pointers later in this chapter.

Every Java program must also have a main function. However, when the program is run the class whose main function you want to run must be specified. In this way, each class could potentially have a main function. The main of the specified class will be the first to run. The main function of the Java hello world program can be found in Fig. 4.3.

Practice 4.3 Command-line arguments are typed in after the name of the program. For instance if a program is called *grep* then you might provide command-line arguments like this.

```
grep def *.py
```

Both C++ and Java programs can receive command-line arguments through the main function. With C++ the number of command-line arguments is passed as *argc* and the actual arguments are passed in the array of strings (i.e. character arrays) in the parameter named *argv* as declared in Fig. 4.9. The variable *argc* is always at least one for C++ programs but the length of the command-line arguments String array in Java may be zero if no command-line arguments are passed. Do you know why?
You can check your answer(s) in Section 4.34.3.

4.6 I/O Streams

Line 4 of the program in Fig. 4.6 prints *Hello World* to the screen. To be a little more precise, *cout* represents what is called a stream in C++. You can think of a C++ *stream* like a real stream with water in it. You can place things in the stream and they will be carried downstream. To place something in a C++ stream you use the << operator. Writing

```
cout << "Hello World";
```

places the string "Hello World" into the cout stream. This expression returns the cout stream. This means that multiple << operators can be chained together. Line 4 is like writing

```
(cout << "Hello World") << endl;
```

The parentheses are not needed in this example since the << is already left-associative. But they were included so you can see that the function call to << returns a stream which can be used in the next << operator to the right.

There are three streams automatically associated with programs. These three streams are associated with any program, whether C++, Python, Java, or other language. In C++, the first stream is called *cout* and by default it writes to the screen. The *cerr* stream also writes to the screen by default. The *cin* stream reads from the keyboard by default. The operator for reading from a stream is the right-shift operator, written cin>>*variable* where the variable will hold the value of its type which was read from the stream. In each of these cases these streams can be redirected to read or write to different locations. Redirecting input and output is an operating system feature and not really associated with a specific programming language. You can search on the web for information about redirecting standard output, standard error, or standard input if you are interested in learning more about redirection.

Java programs have the same three streams. *System.out* is the name of the standard output stream. *System.err* is the standard error stream. *System.in* is the input stream. Figure 4.3 demonstrates writing to standard output in a Java program. The right-shift and left-shift operators are not used to read from and write to streams in Java. Instead the more traditional function call syntax is used.

4.7 Garbage Collection

Garbage collection occurs when dynamically allocated space needs to be returned to the pool of available space in memory. This space that is available for run-time allocation is called the *heap*. Every time an object is created a little of the heap memory of the computer must be reserved to hold that object's state information. When the object is no longer needed, the space on the heap has to be freed so it can be used by another object later.

Languages like Java and Python provide garbage collection as part of the underlying model of computation. They can do this because these languages are careful about how pointers are exposed to the programmer. In fact pointers are called references in these languages to distinguish them from pointers in languages like C and C++. The trade-off is that these languages take some control away from the programmer. Java, Python, and many languages that provide garbage collection require a virtual machine to execute their programs and the virtual machine takes care of managing and freeing unused memory.

Garbage collection can impact the run-time performance of a system. Languages like Java and Python aren't as well-suited to real-time applications where timing is critical. In these languages garbage collection can occur at any time. Usually, running of a program is not time critical and the time taken for garbage collection is negligible. The advantages of garbage collection typically far outweigh the possibility of memory leaks, but not in timing critical applications.

The existence of a run-time system that supports garbage collection, like the Java and Python virtual machines, means that those programs have less access to the underlying hardware of the machine. To safely free unused memory any garbage collection system must restrict the use of pointers in programs and as a result programs written in languages like Java and Python have less access to the details of the hardware platform. Again, this is not usually a problem for most programs, but there are instances where direct hardware access is important. Programs like operating systems are typically not written in Python or Java. To avoid any misconceptions, Android applications are written in Java, but the Android operating system itself is based on the Linux kernel which is implemented in C.

C++ programs must manage the allocation and freeing of heap space. But, it's not always clear when an object will no longer be used. A memory leak occurs when memory never gets freed even though the C++ program is done using it. There is extra work involved in writing C++ classes to insure that objects get freed when they are no longer needed. In the case of the CoCo virtual machine it is not safe to free objects once created because there is no reference counting in CoCo to decide when an object is no longer in use. Because objects are created and often referenced from multiple parts of a CASM program it is safe to simply free objects in CoCo. True garbage collection is needed in the C++ implementation of CoCo to make it a really useful virtual machine. As it stands, CoCo works for running short programs, but would not be suitable for long running applications.

For Java programs the garbage collector is a thread that runs once in a while and checks to see how many references are still referring to an object. If there are no parts of the program using an object, then it can be freed. Once in a while you might have a group of objects that are not being used, but all appear to be using each other. In this case the garbage collector can form a dependency graph and figure out that the objects involved form a cycle and no other objects outside the cycle are depending on the group of objects in the cycle. In this case, all objects in the cycle can be freed. The existence of a garbage collector greatly simplifies writing Java programs including JCoCo.

4.8 Threading

In the previous section it was mentioned that the JVM garbage collector runs in a different thread. A *thread* is a running sequence of instructions that shares access to objects with other threads. Each thread runs largely independently from other threads in the same program. You can think of each thread as essentially an independent program that is working with other threads in the same larger program to accomplish some work.

Java was built from the ground up to be a multi-threaded programming language. Every object in Java contains a lock that can be used to synchronize the behavior of multiple threads. When more than one thread is running, its work should not undo or change the work another thread is doing. When more than one thread is running there are two issues that need to be dealt with: synchronization of the threads, and communication between the threads.

Locks on objects make it possible for Java threads to both synchronize their work, and communicate with each other in structured ways. Every object in Java has a lock associated with it. There are also keywords in Java like *synchronized* that insure that only one method at a time may run on an object. This text won't teach you about Java threading, but it is an important topic and should be studied at some point.

C++ also has support for threads through the thread class in the standard namespace. However, C++ support of threads is quite a bit different than the Java support. For instance, C++ does not have keywords that allow for synchronized methods like Java. Threading in C++ requires a little more thought and work. C++ and Java are equally powerful in their support of multi-threaded programs, but given a choice, Java is the language to use for multi-threaded applications.

4.9 The PyToken Class

Object-Oriented programming is all about creating objects. Objects have *state information*, sometimes just called *state*, and methods that operate on that state, sometimes altering the state. If we alter the state of an object we call it a *mutable* object. If we cannot alter the object's state once it is created, the object is called *immutable*.

A class defines the state information maintained by an object and the methods that operate on that state. We'll start by examining the *PyToken* class in Fig. 4.10 since

```java
package jcoco;
public class PyToken {

    public enum TokenType {
        PYIDENTIFIERTOKEN,
        PYINTEGERTOKEN,
        PYFLOATTOKEN,
        PYSTRINGTOKEN,
        PYKEYWORDTOKEN,
        PYCOLONTOKEN,
        PYCOMMATOKEN,
        PYSLASHTOKEN,
        PYLEFTPARENTOKEN,
        PYRIGHTPARENTOKEN,
        PYEOFTOKEN,
        PYBADTOKEN
    }

    private String lexeme;
    private TokenType type;
    private int line;
    private int col;

    public PyToken(TokenType type, String lex, int line, int col) {
        this.lexeme = lex;
        this.type = type;
        this.line = line;
        this.col = col;
    }

    public TokenType getType() {
        return this.type;
    }

    public String getLex() {
        return this.lexeme;
    }

   // See getCol and getLine in the full source code.
}
```

Fig. 4.10 The PyToken class

it is a simple immutable class. The PyToken class defines the token objects that are returned by the scanner to the parser during parsing of a JCoCo program. All the JCoCo code is in a package called *jcoco*. Line 1 declares that this class belongs in this *jcoco* package.

Lines 4–17 of Fig. 4.10 define the *TokenType* enum, which is short for *enumeration*. The *TokenType* enumeration defines the types of the tokens returned by the scanner. If you recall from Chap. 3, the syntax of CASM programs is pretty simple and these constant values are all the possible token types. Each constant serves as a name for each token type. With this enumeration it is possible to use these constant names in the Java code where the type of token is needed. For instance, here is one snippet of code from the JCoCo interpreter. Using descriptive constant names is useful in writing self-documenting code which you should always strive to do.

```
if (tok.getType() != TokenType.PYEOFTOKEN)
{
    badToken(tok, "Excpected End Of File (EOF)");
}
```

Lines 19–22 define the *instance variables* of the object (i.e. the object state). Each of these variables is declared private so that only the class' methods may access the state information directly. Lines 24–29 define the PyToken constructor which is called when a PyToken object is created. Here is how a PyToken object gets created.

```
PyToken t;
t = new PyToken(type, lex, line, column);
```

Of course, the variables *type*, *lex*, *line*, and *column* would all have to have values already and be of the proper types. For instance, the *lex* variable must be declared as a *String*. Java is a statically typed language, so all variables must be declared before they can be used. In this code the variable *t* was declared to have PyToken type.

There are a couple of things you can't see in the textbook. Because this class is defined in a package called *jcoco*, the result of compiling this class will be placed in a subdirectory named *jcoco*. Packages and subdirectories go together in Java organization of files. In addition, this file must be named *PyToken.java* since it contains the public class *PyToken*. This is also part of Java's organization of files.

4.9.1 The C++ PyToken Class

The implementation of *PyToken* in C++ looks a bit different than the Java version. Java and C++ both support separate compilation of code. Using Java each class is written in a separate file. Each file is compiled separately by the Java compiler. When to re-compile a Java class is decided based on dates of both the *.java* and the *.class* files.

In C++ there is no such make mechanism built into the compiler. Instead, the separate *make* tool provides this functionality as described in Sect. 4.2.2. In addition, the interface to the class (i.e. the declaration of the class' instance variables and methods) is separated from the actual code that implements the methods. So, the

```
 1  #include <string>
 2  using namespace std;
 3  class PyToken {
 4  public:
 5     PyToken(int tokenType, string lex, int line, int col);
 6     virtual ~PyToken();
 7     string getLex() const;
 8     int getType() const;
 9     int getCol() const;
10     int getLine() const;
11  private:
12     string lexeme;
13     int tokenType;
14     int line;
15     int column;
16  };
17  const int PYIDENTIFIERTOKEN = 1;
18  const int PYINTEGERTOKEN = 2;
19  const int PYFLOATTOKEN = 3;
20  const int PYSTRINGTOKEN = 4;
21  const int PYKEYWORDTOKEN = 5;
22  const int PYCOLONTOKEN = 6;
23  const int PYCOMMATOKEN = 7;
24  const int PYSLASHTOKEN = 8;
25  const int PYLEFTPARENTOKEN = 9;
26  const int PYRIGHTPARENTOKEN = 10;
27  const int PYEOFTOKEN = 98;
28  const int PYBADTOKEN = 99;
```

Fig. 4.11 The C++ PyToken class declaration - PyToken.h

PyToken class definition is separated into two files: the PyToken header file, named *PyToken.h*, and the method implementations located in *PyToken.cpp*. Figure 4.11 shows the contents of the header file declaration of the class. Only the *.cpp* source files are compiled using the compiler. The header files are included in the source files for use during compilation of the source files.

Much of the class declaration in the header file looks like the Java version. The *enum* defined in Fig. 4.10 is implemented as integer constants in Fig. 4.11 for no good reason. *Enums* are supported in C++ as well. Line 6 provides the declaration of a *destructor* which in general is used by C++ because C++ programs must free up their space since there is no garbage collector as there is in Java. However, the destructor in this case doesn't really have a purpose since these tokens don't have pointers to other objects. A destructor is needed precisely when an object contains pointers to other resources that must be freed. PyToken objects do not contain any pointers to other objects and hence the destructor has no purpose in this class.

The other difference worth noting is the use of *const* after the four methods that return values. This declares that these methods don't mutate the *PyToken* object. They only return values from the *PyToken* object. The use of *const* exists in C++ because C++ is very flexible in the way parameters are passed and values are returned. Declaring that a method is *const* helps C++ know where the method can be safely called and can optimize the performance of C++ programs.

```
 1 #include "PyToken.h"
 2 PyToken::PyToken(int tokenType, string lex, int line, int col) {
 3     this->lexeme = lex;
 4     this->tokenType = tokenType;
 5     this->line = line;
 6     this->column = col;
 7 }
 8 PyToken::~PyToken() {}
 9 int PyToken::getType() const {
10     return tokenType;
11 }
12 string PyToken::getLex() const {
13     return lexeme;
14 }
15 // getLine and getCol omitted.
```

Fig. 4.12 The C++ PyToken class implementation - PyToken.cpp

The C++ implementation of PyToken is given in Fig. 4.12. The first line includes the declaration of the *PyToken.h* header file. This is a macro processor directive to bring that source code into this file. By doing this, the *PyToken* class is declared for this source code.

The *PyToken::* that you see in Fig. 4.12 is a scope qualifier. It indicates that while none of this code is physically written inside the *PyToken* class definition, it is meant to be a part of the *PyToken* class.

Lines 3–6 of Fig. 4.12 use an arrow operator, written ->. In Java this is written as a dot. The arrow operator follows a pointer. In C++, *this* is a pointer that points to the current object. In Java, *this* is a reference and we use the dot notation to dereference the *this* reference. Pointers are the address of data in the memory of the computer. Pointers can be used in expressions to create new pointers using pointer arithmetic. In a programming language a pointer can point anywhere. A *reference* is much more controlled. References are somewhat like pointers except that they cannot be used in arithmetic expressions. They also don't directly point to locations in memory. When a reference is dereferenced using a dot, the run-time system does the lookup in a reference table.

This difference between references and pointers means that we can safely rely on every reference pointing to a real object where we don't necessarily know if a pointer is pointing to space that might be safely freed or not since the pointer might be the result of some pointer arithmetic. References are safe for garbage collection. Pointers are not.

4.10 Inheritance and Polymorphism

Object-oriented programming languages help us organize our code. One great advantage of organizing our code around objects occurs when we are able to re-use code.

```
1  #ifndef PYOBJECT_H_
2  #define PYOBJECT_H_
3
4  #include <string>
5  #include <unordered_map>
6  #include <vector>
7  #include <iostream>
8  using namespace std;
9
10 class PyType;
11
12 class PyObject {
13 public:
14    PyObject();
15    virtual ~PyObject();
16    virtual PyType* getType();
17    virtual string toString();
18    PyObject* callMethod(string name, vector<PyObject*>* args);
19
20 protected:
21     unordered_map<string, PyObject* (PyObject::*)(vector<PyObject*>*)> dict;
22     virtual PyObject* __str__(vector<PyObject*>* args);
23     virtual PyObject* __type__(vector<PyObject*>* args);
24     virtual PyObject* __hash__(vector<PyObject*>* args);
25     virtual PyObject* __repr__(vector<PyObject*>* args);
26 };
27
28 ostream& operator << (ostream& os, PyObject& t);
29 extern bool verbose;
30 #endif /* PYOBJECT_H_ */
```

Fig. 4.13 The C++ PyObject header file - PyObject.h

Code re-use is important so that we can write something once and forget it while we solve other problems. But for code re-use to work we need a way of customizing this code for our purposes.

Inheritance is the mechanism we employ to re-use code in software we are currently writing. *Polymorphism* is the mechanism we employ to customize the behavior of code we have already written. In this section we'll look at some C++ code to see how inheritance and polymorphism are specified. In the next section we'll revisit the same code as implemented in Java.

Consider the header file for PyObject in Fig. 4.13. The CoCo and JCoCo virtual machines work on Python objects. Every data value in Python is an object, so this idea of objects is very pervasive in the JCoCo/CoCo implementations. In fact, it is so pervasive there are certain things that every object in Python must be able to do. Certain methods can be called on every Python object. To be able to re-use as much code as possible it makes sense to write that common code in one place. One such place is the *PyObject* class in the C++ CoCo implementation.

Every object in Python can be converted to a string. While the string representations vary, the mechanism to convert an object to a string is to call the *__str__* method on the object. This is declared on line 22 of Fig. 4.13. Since all objects should respond to this method, the *PyObject* class defines a toString method and a *__str__* which calls the toString method. The *__repr__* method is similar to the *__str__* method. In some case the two methods return exactly the same string.

But, the *__repr__* returns a string that if evaluated using the eval function, would construct the same object. For instance, consider this code.

```
x = [1,2,3]
y = eval(repr(x))
```

After evaluating this code, both *x* and *y* refer to lists of integers, where *y* is a complete copy of the contents of the list referred to by *x*.

Every object in Python responds to a number of basic method calls. CoCo does not attempt to implement all of them. But another that it does implement is the *__hash__* method which returns a hash value for all hashable objects in Python. This is used when the object is used as a key in a dictionary (i.e. hash table). Only immutable objects may be used as keys in dictionaries.

The *__type__* method returns the type of any Python object. Every Python object has a type. The type is returned via this method which is also an object.

There are a few things to note in Fig. 4.13 related to programming in C++. The first seven lines are called macro processor directives. Any line starting with a pound sign (i.e. #) is a macro processor directive. The first line is an *if-not-defined* directive and the second line is a *define* directive. The last line of Fig. 4.13 is an *endif* that goes with the first line. The pattern of *ifndef*, *define*, *endif* macro processor directives is needed because *include* directives often end up with circular references where include *A* includes *B* which may in turn includes *C* which includes *A* again. This kind of circular reference is avoided by defining *PYOBJECT_H_* in Fig. 4.13. Once the *PyObject.h* include is included, the *PYOBJECT_H_* is defined and if *PyObject.h* is included again through a circular reference, or even through another include including it without a circular reference, it will not get included twice. This pattern of *ifndef-define-endif* is used for every header file in C and C++ programming.

Line 10 declares a class (i.e. type) called *PyType*. This is called a forward declaration. The *PyType* class is used in this header file, but *PyType.h* also includes *PyObject.h* so the forward declaration was necessary because of the circular reference. Line 14 is a constructor for the PyObject class. Line 15 is the destructor declaration which again doesn't really do anything for this class.

The use of the keyword *virtual* is important. Virtual methods are methods that are included in the virtual function table of a C++ class. This virtual function table is how C++ implements *polymorphism*. When a virtual function is called, there is an extra lookup of the function's address because classes that inherit from this class may override any of the virtual functions. For instance, it can't be known at compile-time which version of *toString* should be called, the one in PyObject or one of the *toString* methods defined in a subclass of PyObject.

Examine the *PyObject::__str__* method shown in Fig. 4.14. This method calls *toString* and returns a new *PyStr* object as the result of converting the object to a string. The subtly here is that which toString will be called is unknown until this code actually executes. For instance, if the current object is a *PyInt* then the code would be executed from PyInt.cpp as shown in Fig. 4.15. But if a *PyList* were the current object, then the *toString* method would be executed from Fig. 4.16.

```
PyObject* PyObject::__str__(vector<PyObject*>* args) {
    ostringstream msg;

    if (args->size() != 0) {
        msg << "TypeError: expected 0 arguments, got " << args->size();
        throw new PyException(PYWRONGARGCOUNTEXCEPTION,msg.str());
    }

    return new PyStr(toString());
}
```

Fig. 4.14 The CoCo str magic method

```
string PyInt::toString() {
    stringstream ss;
    ss << val;
    return ss.str();
}
```

Fig. 4.15 The PyInt toString method

```
string PyList::toString() {
    ostringstream s;
    vector<PyObject*> args;
    s << "[";
    for (int i=0;i<data.size();i++) {
        s << *(data[i]->callMethod("__repr__",&args));
        if (i < data.size()-1)
            s << ", ";
    }
    s << "]";
    return s.str();
}
```

Fig. 4.16 The PyList toString method

Both PyInt and PyList inherit from PyObject in the C++ implementation. So, polymorphism works because *toString* is declared virtual and therefore the determination of which *toString* to call is made through an extra lookup of the actual pointer to the function, in the virtual function table, at run-time.

Line 28 declares a function, which in this case is an overloaded left-shift operator (i.e. <<) that can be used to print objects. The implementation of this overloaded left-shift operator, from the file *PyObject.cpp*, relies on polymorphism to customize its behavior, like the *__str__* method also implemented in that module. The *toString* to get called will depend on which type of object this is called.

```
ostream& operator <<(ostream &os, PyObject &t) {
    return os << t.toString();
}
```

4.11 Interfaces and Adapters

In early Object-Oriented programming languages the specification of an interface was tied directly to a class. For instance, in the previous section we learned that *toString* was tied directly to the *PyObject* class and any classes that inherited from *PyObject*. This works great until you have some class that inherits from something other than *PyObject* and would like to use the polymorphism defined by the *toString* method. Then you have a situation where you would like to inherit from two different classes at the same time. C++ solves this problem with multiple inheritance. C++ classes can inherit from more than one class.

Java does a few things a little different than C++. First, unlike C++, every class in Java inherits from the *Object* class either directly or indirectly. There is one class hierarchy in Java of which every class participates. C++ has no built-in inheritance hierarchy. Using C++, a class that does not explicitly inherit from something does not inherit from anything. In Java a class that does not explicitly inherit from anything inherits from *Object*.

Secondly, multiple inheritance is not supported using Java, which simplifies inheritance and its implementation. But, Java solves the whole problem of interfaces being tied to class declarations by separating the two concepts. An interface is a promise to support certain methods in a class. Classes can implement as many interfaces as they wish, which is the Java way of achieving multiple inheritance. But, interfaces are in no way tied to the class hierarchy. What's more, you can declare a parameter to be of the type of an interface. Consider the code in Fig. 4.17.

Like the C++ version, the Java PyObject interface declares a *str* method, similar in purpose to the C++ *toString* method. Declaring the *str* method in the interface means that all classes that implement this interface must implement the *str* method.

While the interface declaration separates the interface from the class hierarchy, it also does not implement any of the code for the interface. It's often the case that many classes which implement an interface will have at least some common code. Either each class must implement the same code or the programmer may choose to

```
package jcoco;

import java.util.ArrayList;
import java.util.HashMap;

public interface PyObject {
    public String str() ;
    public PyType getType();

    public void set(String key, PyObject value) ;
    public PyObject get(String key) ;
    public PyObject callMethod(String name, ArrayList<PyObject> args) ;
}
```

Fig. 4.17 The PyObject interface

```
 1 package jcoco;
 2 
 3 public class PyObjectAdapter implements PyObject {
 4     protected HashMap<String, PyObject> dict = new HashMap<String, PyObject>();
 5     protected HashMap<String, PyObject> attrs = new HashMap<String, PyObject>();
 6     protected String name;
 7     protected PyType.PyTypeId type;
 8 
 9     public PyObjectAdapter(String name, PyType.PyTypeId type) {
10         this();
11         this.name = name;
12         this.type = type;
13     }
14 
15     public PyObjectAdapter() {
16         name = "PyObject()";
17         type = PyType.PyTypeId.PyClassType;
18         PyObjectAdapter self = this;
19         this.dict.put("__str__", new PyBaseCallable() {
20             @Override
21             public PyObject __call__(ArrayList<PyObject> args) {
22                 if (args.size() != 0) {
23                  throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
24                         "TypeError: expected 0 argument, got " + args.size());
25                 }
26 
27                 return new PyStr(self.str());
28             }
29         });
30       ...
31     }
32 
33     @Override
34     public PyType getType() {
35         return JCoCo.PyTypes.get(type);
36     }
37 
38     @Override
39     public String str() {
40         return name;
41     }
42 
43     @Override
44     public String toString() {
45         PyStr s = (PyStr)callMethod("__repr__",new ArrayList<PyObject>());
46          return s.str();
47     }
48 
49     @Override
50     public void set(String key, PyObject value) {
51         this.dict.put(key, value);
52     }
53    ...
54 }
```

Fig. 4.18 The PyObjectAdapter

use inheritance to write an adapter class that implements the interface and provides common code to several subclasses. This is the case in Fig. 4.18. A significant portion of the code is omitted for brevity here.

There are several things to note in this code. Lines 4–7 define protected variables. Protected variables are hidden (i.e. not accessible) from any code that is not in the same package, in this case the *jcoco* package. Line 10 of the code calls *this()* which calls the default constructor on line 15 so that code common to both constructors need not be duplicated.

Line 33 uses a decorator called *@Override*. This decorator tells the compiler that the method that follows overrides or implements a method from a base class or interface. This is useful in case you make a spelling error or incorrectly specify a type of parameter of a method. Spelling a name wrong, or changing the type of a parameter will result in a method that will never get called because polymorphism only works when the name and the types of arguments all match. Otherwise you are just defining a different method since Java supports parametrically overloading names, meaning that two methods may have the same name if they have different types of parameters. So, *@Override* can be useful in catching mistakes that might otherwise be difficult to debug.

Practice 4.4 We have seen how polymorphism is provided by the C++ and Java programming languages. Polymorphism is also provided by Python. Yet with Python we don't declare methods virtual, like C++, and we don't have an built-in class hierarchy like Java. How does polymorphism happen in Python?
You can check your answer(s) in Section 4.34.4.

4.12 Functions as Values

Python is a dynamically typed language. As such, methods are looked up at run-time, not compile time. All methods and values in Python are objects that are stored in dictionaries within other objects so they can be looked up at run-time. The keys in these dictionaries are strings: the names of the values, methods, or functions.

To implement a virtual machine that works like Python's virtual machine, it is necessary to treat functions as values and store them in dictionaries, or hash tables, like Python's virtual machine implementation. C++ supports treating functions as values. Using C++ we can write the following.

```
dict["__str__"]=(PyObject* (PyObject::*)(vector<
PyObject*>*)) (&PyObject::__str__);
```

This code comes from the *PyObject* class in the C++ implementation of CoCo, in the file *PyObject.cpp*. There is a subtle nuance to this code. Once the *__str__* method is set in the object's dictionary any and all subclasses that inherit from *PyObject* and override the *__str__* method will automatically get the overridden method definition. There is no need to set *"__str__"* to point to the new, overridden *__str__* method. Because the *__str__* method is declared virtual, polymorphism means it only has to be set in the dictionary once.

SImilar code is not possible in Java because Java does not treat functions and methods as values. But there is a solution. A *class* can simulate a function or method.

```
public interface PyCallable extends PyObject {
    public PyObject __call__(ArrayList<PyObject> args) ;
}
```

Fig. 4.19 The PyCallable interface

In fact, Java contains support for doing just this. JCoCo implements its own version of run-time lookup of a method or function by using objects to simulate the functions and methods.

JCoCo defines an interface called *PyCallable*, shown in Fig. 4.19. This interface defines one method called *__call__*. This method takes a list of *PyObject* references and returns a *PyObject* as its result. This reflects the calling mechanism within Python. All Python functions are given a list of Python objects and return a Python object. This uniformity means that any class that implements the *PyCallable* interface can be called by calling its *__call__* method. This means that functions can be treated as values in JCoCo by encoding the functions as objects.

4.13 Anonymous Inner Classes

Interfaces in Java specify the methods that must be supported by classes that implement them. The *PyCallable* interface, described in the last section, specifies a *__call__* method. Like other interfaces, there are adapter classes that provide some common code for classes that choose to implement the *PyCallable* interface. The simplest of these is the *PyBaseCallable* class which is used in the functions implemented in the *PyObjectAdapter* class and the *PyCallableAdapter* class to avoid a circular reference problem within the object hierarchy. Another class also implements the *PyCallable* interface, named *PyCallableAdapter*, which is used by all other implementations of *PyCallable* other than those created in the *PyObjectAdapter* and *PyCallableAdapter* classes.

Figure 4.18 contains code on lines 19–29 that creates an instance of the PyBaseCallable adapter. This is an example of an anonymous inner class. Line 19 creates a class that has no name, but inherits from PyBaseCallable and overrides the *__call__* method. There is one instance of this *PyBaseCallable* class created, the instance for this *__str__* method. When an object wishes to call the *__str__* method on an object it calls the *callMethod* method of the *PyObjectAdapter* class.

The *callMethod* code in Fig. 4.20 looks up the method in the object's dictionary and if it finds it, it calls the *__call__* method on it, which in the case of the code on lines 19–29 of Fig. 4.18 is overridden to call the *str()* method on the object and return a *PyStr* object with the contents returned by the *str()* method.

Anonymous inner classes are very important in Java. An inner class is any class defined within another class. Inner classes are important because they provide a means to implement call-backs. When an event occurs in a Java program, like an event from a GUI program (i.e. a mouse-click) or a message being received from the

```java
@Override
public PyObject callMethod(String name, ArrayList<PyObject> args) {
    PyCallable mbr = null;
    if (this.dict.containsKey(name)) {
        mbr = (PyCallable) this.dict.get(name);
        return mbr.__call__(args);
    }
    throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
        "TypeError: '" + this.getType().str() +
        "' object has no attribute '" + name + "'");
}
```

Fig. 4.20 The *callMethod* code

internet, if a call-back has been registered to handle that event, then the call-back is called. Inner classes are the perfect way to implement a call-back because the inner class automatically has access to all the variables and methods of the outer object.

In the case of the code in Fig. 4.18, the inner class is anonymous, it does not have a name. This is okay because typically a call-back has one and only one instance created for it, for a particular outer object. Earlier versions of Java did not have anonymous classes leaving Java programs littered with inner classes that only ever had one instance created. Java programmers needed a more compact, precise syntax to manage call-backs and anonymous classes were introduced.

The advantage of the inner class defined on lines 19–29 of Fig. 4.18 is on line 27 where the *str()* method is called which is a member of *PyObjectAdapter* which is the class of the outer object in this instance. Since this is an inner class, we can directly call the *str()* method in this call-back. Anonymous inner classes are used extensively throughout the JCoCo implementation.

4.14 Type Casting and Generics

Line 21 of the C++ code in Fig. 4.13 is an example of declaring a variable *dict* whose type is defined by a template. A template is how C++ programmers write generic classes. A generic class is usually a container of some type, in this case a hash table. This hash table maps strings to functions that are given a *vector* of *PyObject* pointers and return a *PyObject* pointer. The *vector* class is again a template. The vector passed to these functions is a sequence of *PyObject* pointers.

Generics are an important part of object-oriented languages. Generics let programmers re-use classes, especially classes that are designed as data structures like maps and vectors. A *map* is a data structure mapping *keys* to *values*. The type of the keys and values can be practically anything. So, a generic *map* class provides the ability to map any type of *keys* to any type of *values*. In Java a *map* is called a *HashMap*. In C++ there are several kinds of map classes. Please note that in C++ the

```
private ArrayList BodyPart() {
        ArrayList instructions = new ArrayList();
        PyToken tok = this.in.getToken();
        this.target.clear();
        this.index = 0;
        if (!tok.getLex().equals("BEGIN")) {
            badToken(tok, "Expected a BEGIN keyword.");
        }
        ....
```

Fig. 4.21 An ArrayList example

standard *map* class is not implemented as a hash table. It guarantees O(log n) insert and lookup time. A hash table guarantees an amortized complexity of O(1) insert and lookup. The *unordered_map* of C++11 is implemented as a hash table. Before C++11 this class was not included with C++.

Java has one type hierarchy. Everything inherits from Object either directly or indirectly. By having one type hierarchy the creators of Java could provide container classes for many of the data structures we need in our programs including *HashMap* and *ArrayList* which provides a means for storing a list of objects. For instance, if you needed a list of objects to be returned from a function you might code it as shown in Fig. 4.21.

The *BodyPart* function, a part of the *PyParser* module, returns a list of *PyByteCode* objects. When one of these *PyByteCode* objects is needed, we would be forced to write code like this to access the first PyByteCode object in the list.

```
ArrayList bp = BodyPart();
PyByteCode byteCode = (PyByteCode) bp.get(0);
```

The *(PyByteCode)*, with the parens, is called a *type cast* or just a *cast*. Casting is necessary when moving down in the inheritance hierarchy. A cast is a way to tell the Java compiler that you know the actual type of the value while the compiler does not. There is a run-time check that is inserted into your code. If the cast is invalid, the Java program will throw an exception so casting is safe. It's just not convenient and the extra run-time check is less than desirable, although arguably better than not checking at all.

Casting is the same in C++ and Java. The same issue occurs in C++. When moving down the inheritance hierarchy, a cast would be required. C++ has a datatype similar to Java's *ArrayList*, called *vector*. However, C++ does not have one super class of all other classes. So the *vector* datatype would be a little harder to write without something called generics.

Moving down the inheritance hierarchy in either Java or C++ requires the programmer to write more code. If we could avoid moving up the hierarchy in the first place, then moving down again would become unnecessary. This is the aim of generics. Generics were added to Java to make moving up and down the inheritance hierarchy, and therefore casting, unnecessary in many circumstances. Consider the real version of the *BodyPart* function in Fig. 4.22.

```
    private ArrayList<PyByteCode> BodyPart() {
        ArrayList<PyByteCode> instructions = new ArrayList<PyByteCode>();
        PyToken tok = this.in.getToken();
        this.target.clear();
        this.index = 0;
        if (!tok.getLex().equals("BEGIN")) {
            badToken(tok, "Expected a BEGIN keyword.");
        }
        ...
```

Fig. 4.22 An ArrayList example using generics

```
template < class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Alloc = allocator< pair<const Key,T> >
           > class unordered_map {
           ...
}
```

Fig. 4.23 The unordered_map template

In this code the angle brackets (i.e. < and >) delimit the type of the *ArrayList*. The *ArrayList* is a list of *PyByteCode* elements. This declares the specific type contained in the *ArrayList* making the declaration of the *ArrayList* generic, so that it can be a container of any type, not just *Object* values. To declare the *ArrayList* class to be generic the Java creators would have changed its definition to look like this.

```
class ArrayList<T> {
    private T data[] = new T[10];
    ...
```

The type, T, becomes a parameter to the class declaration. A version of the class is created for each declared version of the ArrayList. So, when the *ArrayList <PyByteCode>* class is specified, a *PyByteCode ArrayList* object is created.

Python, since it is dynamically typed, does not need generics. Generics are only needed for statically typed languages like Java and C++. In C++ generics are called templates. A template is a parameterized class. Like Java, the parameter to the class is a type or types. Standard template containers in C++ include *unordered_map*, *map*, *vector*, *list*, *queue*, *stack*, *deque*, *set*, and *array* among others. Consider the declaration of the *unordered_map* template in C++ in Fig. 4.23. This template definition shows us that more than one type parameter can be used for a template or a generic in C++ or Java. In the case of the *unordered_map* there are five type parameters passed to the declaration of the map.

Diamond notation is one topic related to generics in Java. To save even more writing, Java programmers may use diamond notation when writing a generic object declaration. The declaration of the variable *instructions* earlier in this section could have be written as follows if we would have used diamond notation.

```
ArrayList<PyByteCode> instructions = new ArrayList<>();
```

You can probably see the diamond in the code. Since *PyByteCode* is already written once on this line, using Java you don't have to write it again. The compiler can infer the type of the ArrayList as it is created from the type of the reference *instructions* pointing to it.

4.15 Auto-Boxing and Unboxing

In C++ you can declare a *vector* of *int* if you need to by writing

```
vector<int> intVec;
```

It is not possible to declare an *ArrayList* of *int* in Java. Templates in C++ can take any type as an argument to the class, even primitive types. In Java, only classes can serve as arguments to generics. The type *int* is a primitive type. That means it is not a class. However, the creators of Java understood this problem and provided wrapper classes for each of the primitive types so we could declare collections of *ints* for instance by wrapping each *int* as an *Integer*. So, while we can't declare an *ArrayList* of *int*, we can declare an *ArrayList* of *Integer* as follows.

```
ArrayList<Integer> intList = new ArrayList<>();
```

A wrapped *int* is created and added to our list as follows.

```
int x = 6;
Integer y = new Integer(x);
intList.add(y);
```

And getting it back out again involves writing some code like this.

```
x = intList.get(0).intValue();
```

This is the old way of wrapping and unwrapping integers, and other primitive types, in Java. Once again, programmers do this often enough they wanted a more compact way of wrapping and unwrapping primitive types. Java programmers refer to this as *boxing* and *unboxing*. Recent versions of Java support auto-boxing and unboxing. So, now in Java you can write the following.

```
int x = 6;
intList.add(x);
...
x = intList.get(0);
```

The variable *x's* value is auto-boxed when it is added to the list and auto-unboxed when it is extracted. Java determines when to box and unbox primitive values based on the type of value and the method being called. The syntax is much more compact and it is easier to read.

C++ does not support autoboxing and unboxing, but since you can declare template containers of primitive types it isn't as necessary to wrap and unwrap primitive values when using C++.

Practice 4.5 The Java *ArrayList* contains two overloaded methods called *remove*. One takes an *int* parameter and removes an object at the specified index in the *ArrayList*. The other takes an *Object* as a parameter and removes the first instance of the object from the *ArrayList*. Why might this pose a problem?
You can check your answer(s) in Section 4.34.5.

4.16 Exception Handling in Java and C++

Java and C++ can throw exceptions and catch them as in many languages. Sometimes exceptions are thrown in code not written by us but code we use. For instance, indexing beyond the end of a vector. Other times we may wish to throw an exception. In C++ literally any type of value can be thrown.

Figure 4.24 shows how an object called a PyException is thrown using C++. This code was taken from the PyRange.cpp module. When *indexOf* is called beyond the end of a range object, CoCo throws a PyException object with a value of *stop iteration* as shown in Fig. 4.24.

Using Java, you throw values that inherit from the class Exception. Additionally, in some cases you must declare that a function throws an exception. For instance, in Fig. 4.25 the *indexOf* method declares that it throws *PyException*. It turns out in this case that declaring that the method throws this exception is optional because PyException inherits from *RuntimeException*. Exceptions that inherit from *Runtime-Exception* don't have to be declared to be thrown in Java.

Exceptions that are thrown can be caught and the C++ version of the exception is caught in PyFrame.cpp in the *FOR_ITER* instruction. The code for this appears

```
PyObject* PyRange::indexOf(int index) {
    int val = start + index * increment;
    if (increment > 0 && val >= stop) {
        throw new PyException(PYSTOPITERATIONEXCEPTION,"Stop Iteration");
    }
    if (increment < 0 && val <= stop) {
        throw new PyException(PYSTOPITERATIONEXCEPTION,"Stop Iteration");
    }
    return new PyInt(start + increment*index);
}
```

Fig. 4.24 Throwing an exception in C++

```
    public PyObject indexOf(int index) throws PyException {
        int val = start + index * increment;
        if (increment > 0 && val >= stop) {
            throw new PyException(
            PyException.ExceptionType.PYSTOPITERATIONEXCEPTION,
            "Stop Iteration");
        }
        if (increment < 0 && val <= stop) {
            throw new PyException(
            PyException.ExceptionType.PYSTOPITERATIONEXCEPTION,
            "Stop Iteration");
        }
        return new PyInt(start + increment * index);
    }
```

Fig. 4.25 Throwing an exception in Java

```
case FOR_ITER:
    u = safetyPop();
    args = new vector<PyObject*>();
    try {
        v = u->callMethod("__next__", args);
        opStack->push(u);
        opStack->push(v);
    } catch (PyException* ex) {
        if (ex->getExceptionType() == PYSTOPITERATIONEXCEPTION) {
            PC = operand;
        } else
            throw ex;
    }
    try {
        delete args;
    } catch (...) {
        cerr <<
         "Delete of FOR_ITER args caused an exception for some reason." <<
         endl;
    }
    break;
```

Fig. 4.26 Catching an exception in C++

```
case FOR_ITER:
    u = this.safetyPop();
    args = new ArrayList<PyObject>();
    try {
        v = u.callMethod("__next__", args);
        this.opStack.push(u);
        this.opStack.push(v);
    } catch (PyException ex) {
        if (ex.getExceptionType() == ExceptionType.PYSTOPITERATIONEXCEPTION) {
            this.PC = operand;
        } else {
            throw ex;
        }
    }
    break;
```

Fig. 4.27 Catching an exception in Java

in Fig. 4.26. To catch an exception it must be thrown in a *try block* or in some code called from a try block. Then the type of value caught in the *catch* must match the type of value thrown. Figure 4.27 provides the Java version of catching an exception. There is not much difference between C++ and Java in handling exceptions. The additional code in Fig. 4.26 on lines 14–20 are needed because C++ does not have garbage collection while Java does.

Figures 4.24 and 4.26 demonstrate how exception handling can be used to implement iteration within the CoCo interpreter, while Figs. 4.25 and 4.27 provide the Java version for JCoCo. When the end of an iteration is reached, a stop iteration exception is thrown and when caught it signals the end of the iteration.

```
void sigHandler(int signum) {
  cerr << "\n\n";
  cerr << "*********************************************************" << endl;
  cerr << "            An Uncaught Exception Occurred" << endl;
  cerr << "*********************************************************" << endl;
  cerr << "Signal: ";
  switch (signum) {
      case SIGABRT:
          cerr << "Program Execution Aborted" << endl;
          break;
      case SIGFPE:
          cerr << "Arithmetic or Overflow Error" << endl;
          break;
      case SIGILL:
          cerr << "Illegal Instruction in Virtual Machine" << endl;
          break;
      case SIGINT:
          cerr << "Execution Interrupted" << endl;
          break;
      case SIGSEGV:
          cerr << "Illegal Memory Access" << endl;
          break;
      case SIGTERM:
          cerr << "Termination Requested" << endl;
          break;
  }
  cerr << "---------------------------------------------------------" << endl;
  cerr << "              The Exception's Traceback" << endl;
  cerr << "---------------------------------------------------------" << endl;
  for (int k=callStack.size()-1;k>=0;k--) {
      cerr << "==========> At PC=" << (callStack[k]->getPC()-1) <<
      " in this function. " << endl;
      cerr << callStack[k]->getCode().prettyString("",true);
  }
  exit(0);
}

int main(int argc, char* argv[]) {
  char* filename;

  signal(SIGABRT,sigHandler);
  signal(SIGFPE,sigHandler);
  signal(SIGILL,sigHandler);
  signal(SIGINT,sigHandler);
  signal(SIGSEGV,sigHandler);
  signal(SIGTERM,sigHandler);
  ...
```

Fig. 4.28 Signal handling

Exception handling is a means of handling conditions within a program, whether planned or unplanned. C++ and Java programs can throw and/or catch exceptions as needed. However some problems in C++, like division by zero errors, do not surface as exceptions. They are signaled instead which is the topic of the next section.

4.17 Signals

The C version of exception handling is signal handling. C programs can generate signals, but it is more common to put a signal handler in place to handle signals generated by the operating system. Figure 4.28 contains an excerpt of the code from main.cpp where a signal handler is implemented and is installed in main.

There are several types of signals and the code in Fig. 4.28 is written to catch all of the signals defined in the C standard. The constant signal types are defined in an include called *signal.h*. When a signal is generated the program immediately jumps to the signal handler passing it the signal value that was generated. The signal handler usually is written to report some type of error and then terminates. The signal handler presented in Fig. 4.28 does that. It prints a traceback of the program and then terminates.

4.18 JCoCo in Depth

The rest of the chapter will cover only the Java implementation of JCoCo. Many similarities exist to the C++ CoCo implementation. JCoCo is mostly a superset of the CoCo implementation. When there are differences between the two implementations they will be noted. Primarily JCoCo adds support for the creation of programmer-defined classes.

4.19 The Scanner

The JCoCo virtual machine reads a CASM file as depicted in Fig. 4.1. The virtual machine starts by using a scanner to return the tokens of the CASM file. It is common to implement a scanner as a finite state machine. The finite state machine consists of states and transitions between states depending on the characters read from the input file. The finite state machine accepts tokens of the CASM file. The finite state machine employed by JCoCo is depicted in Fig. 4.29.

When the parser is constructed, it first creates a scanner to read the tokens of the CASM program. The PyScanner's getToken method is written as a finite state machine to get the tokens of the CASM program. Figure 4.30 contains the *getToken*

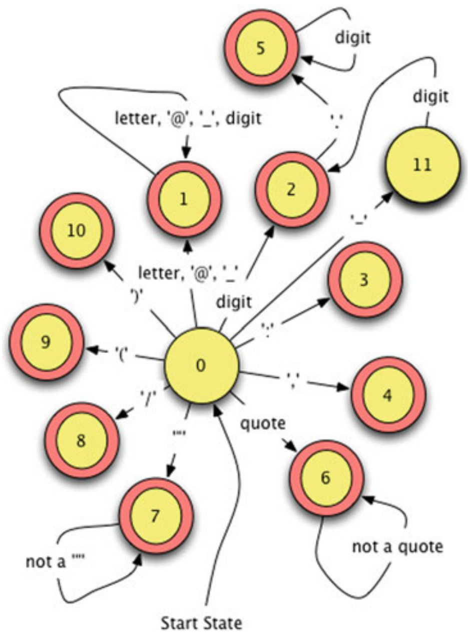


Fig. 4.29 The JCoCo scanner FSM

and *putBackToken* methods for the scanner. The *putBackToken* method is capable of putting back one token which is used by the parser when it has to look ahead one token to determine its next action.

The scanner reads from a stream, which in the case of JCoCo is a *PushbackInput-Stream* so that a character can be *unread*. The scanner also keeps track of its position within the file so each token can carry along the position where it was found in the input file.

The start state is 0 as shown in the Fig. 4.29. There are several things to take note of in the finite state machine. First, identifiers are accepted by state 1 and are limited to letters and digits where the first character is a letter. Underscore characters and @ characters are considered letters by the scanner so tokens like @x_1 are recognized as identifiers by JCoCo even though that is an illegal identifier in Python. State 2 accepts integers. State 5 accepts floating point numbers which must have a decimal point. Scientific floating point notation is not accepted by JCoCo.

States 6 and 7 are responsible for recognizing strings. These states keep reading until a single or double quote is found to end the string. However, strings cannot have a quote or double quote in them as they are defined. For instance, the string 'how's it going?' is not allowed because there is no escape character implemented in JCoCo and the second quote would end the string. The entire implementation of the finite state machine can be found in PyScanner.java with only an excerpt of the code appearing in Fig. 4.30.

```java
public PyToken getToken() {
    if (!this.needToken) {
        this.needToken = true;
        return this.lastToken;
    }

    try {
        next = this.in.read();
        if (next == -1) {
            type = TokenType.PYBADTOKEN;
            foundOne = true;
        }

        while (!foundOne) {
            this.colCount++;
            c = (char) next;
            switch (state) {
                case 0:
                    lex = "";
                    column = this.colCount;
                    line = this.lineCount;
                    if (isLetter(c)) state = 1;
                    else if (Character.isDigit(c)) state = 2;
                    else if (c == '-') state = 11;
                    ...
                    break;
                ...
            }
            if (!foundOne) {
                lex += c;
                next = this.in.read();
                if (next == -1) {
                    type = TokenType.PYEOFTOKEN;
                    foundOne = true;
                }
            }
        }
        this.in.unread((char)next);
        this.colCount--;
    } catch(IOException e) {
        System.err.println(e.getMessage());
    }
    PyToken t = new PyToken(type, lex, line, column);
    this.lastToken = t;
    return t;
}

public void putBackToken() {
    needToken = false;
}
```

Fig. 4.30 PyScanner getToken and putBackToken methods

The input stream contains a method to put back the last character which is used by the scanner code on line 38 of the finite state machine loop in Fig. 4.30. The last character must be put back when a token is returned because the last character is not a part of that token. Consider state 1 for example. The finite state machine remains in state 1 as long as the character is still a letter or a digit. When it is neither, the *foundOne* variable is set to true, terminating the loop. But, that last character may be part of the next token and so is put back before returning.

Just before *getToken* returns a token, the token to be returned is saved. This is used by the *putBackToken* method. If the last token is put back then *needToken* is simply set to false. When *getToken* is called, lines 2–5 check to see if *needToken* is false and if so return the token that was put back by the *putBackToken* method. By saving the token before it is returned the last token is always remembered in case it needs to be returned again.

4.20 The Parser

The tokens of a CASM file are read by the parser and parsed according to the grammar rules in *Appendix A*. Each BNF non-terminal corresponds to one function in the parser. The parser returns an abstract syntax tree representing the CASM program. In this implementation, the abstract syntax tree is an *ArrayList* of PyCode and PyClass objects, which means the *ArrayList* is declared as a list of *PyObjects*. Figure 4.31 contains an outline of the *parse* method and some of the code called by *parse* which can be found in *PyParser.java*.

Each method of the parser corresponds to a non-terminal of the grammar. The implementation of each method method is determined by the right hand sides of its rules. The entire parser implementation is in *PyParser.java*. Figures 4.31 and 4.32 contain two excerpts of this code. Examining the rules for *ClassFunctionList*, *FunDef*, and *ConstPart* will shed some light on the implementations of the methods in Figs. 4.31 and 4.32.

```
CoCoAssemblyProg ::= ClassFunctionListPart EOF
ClassFunctionListPart ::= ClassFunDef ClassFunctionList
ClassFunctionList ::= ClassFunDef ClassFunctionList | <null>
ClassFunDef ::= ClassDef | FunDef
FunDef ::= Function colon Identifier slash Integer
    ClassFunctionList ConstPart LocalsPart FreeVarsPart
    CellVarsPart GlobalsPart BodyPart
ClassDef ::= Class colon Identifier [ ( Identifier ) ]
    BEGIN ClassFunctionList END
ConstPart ::= <null> | Constants colon ValueList
```

Starting with the *ClassFunctionList* non-terminal, its rules say that either it is empty (i.e. <null>) or it is a *ClassFunDef* followed by a *ClassFunctionList*. How do we know which rule to follow? The answer can be found by looking ahead one token. If we examine the *FunDef* rule, it must start with the keyword *Function* and that should be the next token to be read in the *ClassFunctionList* implementation unless

```
1  public ArrayList<PyObject> parse() {
2      try {
3          return PyAssemblyProg();
4      } catch (PyException e) {
5          this.in.putBackToken();
6          PyToken tok = this.in.getToken();
7          // print error message
8          System.exit(0);
9      }
10     // unreachable
11     return null;
12 }
13 private ArrayList<PyObject> PyAssemblyProg() {
14     ArrayList<PyObject> code = ClassFunctionListPart();
15     PyToken tok = this.in.getToken();
16     if (tok.getType() != TokenType.PYEOFTOKEN) {
17         badToken(tok, "Excpected End Of File (EOF)");
18     }
19     return code;
20 }
21 private ArrayList<PyObject> ClassFunctionListPart() {
22     PyObject obj = ClassFunDef();
23     ArrayList<PyObject> codeList = new ArrayList<PyObject>();
24     codeList.add(obj);
25     codeList = ClassFunctionList(codeList);
26     return codeList;
27 }
28 private ArrayList<PyObject> ClassFunctionList(ArrayList<PyObject> codeList) {
29     PyToken tok = this.in.getToken();
30     this.in.putBackToken();
31 
32     PyObject obj = null;
33     String lexeme = tok.getLex();
34     if (lexeme.equals("Function") || lexeme.equals("Class")) {
35         obj = ClassFunDef();
36         codeList.add(obj);
37         codeList = ClassFunctionList(codeList);
38     }
39     return codeList;
40 }
41 private PyObject ClassFunDef() {
42     PyToken tok = this.in.getToken();
43     this.in.putBackToken();
44     PyObject obj = null;
45     if (tok.getLex().equals("Function")) {
46         obj = FunDef();
47     } else if (tok.getLex().equals("Class")) {
48         obj = ClassDef();
49     } else {
50         // throw  exception
51     }
52     return obj;
53 }
```

Fig. 4.31 *PyParser.java* excerpt 1

the next part of the program is a class definition. In that case the *ClassDef* non-terminal requires a *Class* keyword. To determine what to do we get the next token in line 29 of Fig. 4.31, put it back right away, and check to see if it was a *Function* or *Class* keyword. If it was either of these, then the first rule is executed by calling *ClassFunDef* followed by *ClassFunctionList*. If *Function* or *Class* is not the next token, then we return the *ArrayList* passed to the method since we follow the <*null*> rule.

```java
private PyCode FunDef() {
    PyToken tok = this.in.getToken();
    if (!tok.getLex().equals("Function")) {
        badToken(tok, "Expected Function keyword.");
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals(":")) {
        badToken(tok, "Excpected a ':'.");
    }
    PyToken funName = this.in.getToken();
    if (funName.getType() != TokenType.PYIDENTIFIERTOKEN) {
        badToken(funName, "Expected an identifier");
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals("/")) {
        badToken(tok, "Expected a '/'");
    }
    PyToken numArgsToken = this.in.getToken();
    int numArgs = 0;
    if (numArgsToken.getType() != TokenType.PYINTEGERTOKEN) {
        badToken(numArgsToken, "Expected an integer token");
    }
    try {
        numArgs = Integer.parseInt(numArgsToken.getLex());
    } catch (NumberFormatException e) {
        System.err.println(e.getMessage());
        System.exit(0);
    }
    ArrayList<PyObject> nestedClassFunctionList = new ArrayList<PyObject>();
    nestedClassFunctionList = ClassFunctionList(nestedClassFunctionList);
    ArrayList<PyObject> constants = ConstPart(nestedClassFunctionList);
    ArrayList<String> locals = LocalsPart();
    ArrayList<String> freevars = FreeVarsPart();
    ArrayList<String> cellvars = CellVarsPart();
    ArrayList<String> globals = GlobalsPart();
    ArrayList<PyByteCode> instructions = BodyPart();
    return new PyCode(funName.getLex(), nestedClassFunctionList, constants,
            locals, freevars, cellvars, globals, instructions, numArgs);
}
private ArrayList<PyObject> ConstPart(ArrayList<PyObject> nestedCFList) {
    ArrayList<PyObject> constants = new ArrayList<PyObject>();
    PyToken tok = this.in.getToken();
    if (!tok.getLex().equals("Constants")) {
        this.in.putBackToken();
        return constants;
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals(":")) {
        badToken(tok, "Expected a ':'.");
    }
    constants = ValueList(constants, nestedCFList);
    return constants;
}
```

Fig. 4.32 *PyParser.java* excerpt 2

Why is an *ArrayList* of *PyObjects* passed to the *ClassFunctionList* method? This ArrayList is the abstract syntax tree “so far”, as it has been read up to this point in the parser. The *ClassFunctionList* method adds to that ArrayList if it finds another function or class definition.

The *FunDef* method has only one rule to follow. It is responsible for building a PyCode object to return to the *ClassFunDef* method. When *FunDef* is called, we have already checked that the first token is the keyword *Function* so lines 2–5 could be omitted. The rest of the method gets tokens, checks them to see if they are the expected tokens, and calls other methods of the parser to read the rest of the function definition.

The *ConstPart* method has two rules to follow, like the *ClassFunctionList* method. Again, it must get a token to determine which rule to follow. If the next token is not *Constants*, then the empty rule is used and the *ConstPart* method returns an empty *ArrayList*. Otherwise, it returns an *ArrayList* of the constants used in the function. Each constant string is used to build a *PyObject* value for that constant. The ArrayList *nestedCFList* is passed to the *ConstPart* method because a nested class or function is itself a constant value stored in a *PyClass* or *PyCode* object respectively. When a constant like *code(g)* appears in the list of constants it tells the parser to look up the code for it in the list of nested classes or functions passed to the *ConstPart* method.

The code excerpts in Figs. 4.31 and 4.32 demonstrate that the functions of the parser are straightforward implementations of the rules in the grammar. Once in a while a lookahead token is needed to determine which rule to follow, but otherwise the parser gets tokens when required and calls other nonterminal methods when indicated by the rule. The trickiest part of writing the parser is probably determining what should be returned. This is dictated by the information that is required in the abstract syntax tree which is determined by the intended use of the information in the source file.

4.21 The Assembler

Before CoCo can execute the code in a function, all labels must be converted to target addresses in the instructions. Labels make no sense to the bytecode interpreter. Labels are convenient for programmers but are not for code execution. The assembly phase looks for labels and replaces any instruction jump label with the address to which it corresponds. For instance, consider the CASM program in Fig. 4.33. The *label00* identifies the instruction at offset 11 in the main function. The *label01* maps to offset 18 and *label02* maps to offset 19. The instructions on line 14, 17, and 23 need to get the offset, not the label, of their intended targets. This is the job of the assembler.

The assembler is simple enough to include in the parser code when the body part of a function is parsed. There are two parts to it utilizing a *HashMap* to remember and then update the target addresses in the code.

The code for the assembler is contained in two of the parser methods, the *LabeledInstruction* method and the *BodyPart* method. The grammar rules surrounding this code are provided here.

```
<BodyPart> ::= BEGIN <InstructionList> END
<InstructionList> ::= <null> | <LabeledInstruction> <InstructionList>
<LabeledInstruction> ::= Identifier colon <LabeledInstruction> |
                         <Instruction> | <OpInstruction>
<Instruction> ::= STOP_CODE | NOP | POP_TOP | ROT_TWO | ROT_THREE | ...
```

The code for *LabeledInstruction* adds each discovered label to a map from labels to integer offsets. Lines 32–39 of Fig. 4.34 do this when they discover an instruction contains a label. If the code finds a label, then line 34 adds the label to the map making it point to the offset, called *index* in the code.

Target locations are updated in the body of the function on lines 11–19 of Fig. 4.34. If an instruction is found that uses a label as its target, the instruction is deleted and a new instruction with identical opcode is created with the actual target address of the instruction.

4.22 ByteCode

A PyByteCode object is created for each instruction found in a CASM program. The class definition, partially defined in Fig. 4.35, shows an *enum* being declared with all possible opcodes. This *enum* construct in Java is convenient and powerful. An enum is actually a class definition that declares both the name of each enumerated value and any attributes associated with it. In the case of the *PyOpCode* enum each instruction

```
Function: main/0
Constants: None, "Enter a list: "
Locals: x, lst, b
Globals: input, split, print
BEGIN
          LOAD_GLOBAL          0
          LOAD_CONST           1
          CALL_FUNCTION        1
          STORE_FAST           0
          LOAD_FAST            0
          LOAD_ATTR            1
          CALL_FUNCTION        0
          STORE_FAST           1
          SETUP_LOOP     label02
          LOAD_FAST            1
          GET_ITER
label00:  FOR_ITER       label01
          STORE_FAST           2
          LOAD_GLOBAL          2
          LOAD_FAST            2
          CALL_FUNCTION        1
          POP_TOP
          JUMP_ABSOLUTE  label00
label01:  POP_BLOCK
label02:  LOAD_CONST           0
          RETURN_VALUE
END
```

Fig. 4.33 listiter.casm

```
private ArrayList<PyByteCode> BodyPart() {
    ArrayList<PyByteCode> instructions = new ArrayList<PyByteCode>();
    PyToken tok = this.in.getToken();
    this.target.clear();
    this.index = 0;
    if (!tok.getLex().equals("BEGIN")) {
        badToken(tok, "Expected a BEGIN keyword.");
    }
    instructions = InstructionList(instructions);
    //find the target of any labels in the byte code instructions
    for (int i = 0; i < instructions.size(); i++) {
        PyByteCode inst = instructions.get(i);
        String label = inst.getLabel();
        if (!label.equals("")) {
            String op = inst.getOpCodeName();
            instructions.remove(instructions.get(i));
            instructions.add(i, new PyByteCode(op, target.get(label)));
        }
    }
    tok = this.in.getToken();
    if (!tok.getLex().equals("END")) {
        badToken(tok, "Expected a END keyword.");
    }
    return instructions;
}

private PyByteCode LabeledInstruction() {
    PyToken tok1 = this.in.getToken();
    String tok1Lex = tok1.getLex();
    PyToken tok2 = this.in.getToken();
    String tok2Lex = tok2.getLex();
    if (tok2Lex.equals(":")) {
        if (!this.target.containsKey(tok1Lex)) {
            this.target.put(tok1Lex, this.index);
        } else {
            badToken(tok1, "Duplicate label found.");
        }
        return LabeledInstruction();
    }
  // code omitted here.
}
```

Fig. 4.34 Assembling a program

name has associated with it the number of arguments that will be supplied with the instruction. Each instruction either has zero or one arguments, which appear immediately following the instruction in a CASM file. Referring back to Fig. 4.33 most instructions in that example have one argument with the exception of the *GET_ITER*, *POP_TOP*, and *POP_BLOCK* instructions which have zero arguments.

Enumerated values are convenient because they aid in writing self-documenting code. The enumerated values in the program are constructed as objects, one for each value enumerated in the declaration. They can be referred to by their enumerated value in code. For instance, in *PyFrame.java* a switch statement chooses between possible instruction values.

```
class PyByteCode {
    enum PyOpCode {
        BINARY_ADD (0),
        LOAD_CONST (1),
        COMPARE_OP (1),
        CALL_FUNCTION (1),
        ...;
        private int args;
        PyOpCode(int args) {
            this.args = args;
        }
        public int args() {
            return this.args;
        }
    };

    private static HashMap<String, PyOpCode> OpCodeMap = createOpCodeMap();
    private static HashMap<String, Integer> ArgMap = createArgMap();

    private static HashMap<String, PyOpCode> createOpCodeMap() {
        HashMap<String, PyOpCode> map = new HashMap<String, PyOpCode>();
        for (PyOpCode opcode : PyOpCode.values()) {
            map.put(opcode.name(), opcode);
        }
        return map;
    }

    private static HashMap<String, Integer> createArgMap() {
        HashMap<String, Integer> map = new HashMap<String, Integer>();
        for (PyOpCode opcode : PyOpCode.values()) {
            map.put(opcode.name(), opcode.args());
        }
        return map;
    }
    // code omitted here.
    public PyByteCode(String opcode, int operand)  {
        if (!OpCodeMap.containsKey(opcode)) {
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
                                        "Unknown opcode "+opcode);
        }

        this.opcode = OpCodeMap.get(opcode);
        this.operand = operand;
        this.label = "";
    }
    // code omitted here.
}
```

Fig. 4.35 Static initialization

```
inst = this.code.getInstructions().get(this.PC);
switch (inst.getOpCode()) {
    case LOAD_FAST:
        u = this.locals.get(this.code.getLocals().get(operand));
        if (u == null) {
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
                    "NameError: name '" + this.code.getLocals().get(operand) ...
        }
        this.opStack.push(u);
        break;
    ...
```

This code gets the opcode from the next instruction. The switch statement is written with each instruction opcode enumerated in the case statements. This makes the code very clear when examining it. The behavior of the instructions is associated with the name of each instruction.

To build the PyByteCode objects from the CASM file it is necessary to translate the string opcodes, like *"LOAD_FAST"* into their actual opcodes, like *LOAD_FAST*. To accomplish this there has to be a way to look up a string and find its corresponding opcode. This lookup is done in O(1) time by using a *HashMap*. The hash map is created once, when the program begins. When code is executed once and only once at program initialization, it is called *static initialization*. Java and C++ both support static initialization of values.

Internally to the PyByteCode class there are two statically allocated maps that help in translating each instruction that is read by the parser into a PyByteCode object. The code in Fig. 4.35 appears in the *PyByteCode* module. The two variables *OpCodeMap* and *ArgMap* are statically initialized and available to all code implemented in the class. *OpCodeMap* is used when an opcode name is found in a CASM file. It serves to verify it is a valid instruction and to provide a translation to its enumerated value. *ArgMap* provides a count of the number of operands, either 0 or 1, allowed for the instruction. For example, looking up *"BINARY_ADD"* as *OpCodeMap["BINARY_ADD"]* would yield the enumerated value *BINARY_ADD*.

Static initialization of variables can be helpful when you have one-time code that needs to be run during program initialization. This section demonstrates how to do this using Java. Similar code exists for C++. See the module *PyByteCode.cpp* and *PyByteCode.h* in the CoCo implementation for an example of doing this using C++ for further details. In this Java version of it the two functions that create the maps are executed when called by the static initialization on lines 17 and 18 of the code in Fig. 4.35.

4.23 JCoCo's Class and Interface Type Hierarchy

The JCoCo implementation consists of approximately fifty six classes and interfaces. Approximately fifty six classes because JCoCo continues to grow and evolve. Class inheritance is used for code re-use and polymorphism throughout the implementation. Figure 4.36 provides a look at the hierarchy of classes and interfaces. The PyBuiltIns represent a collection of classes for all the built-in functions provided with JCoCo which include concat, print, fprint, tprint, iter, len, open, and repr. The fprint and tprint built-in functions are not part of Python. The fprint function is a functional version of print that takes one value and returns the fprint instance. The tprint built-in function takes a tuple and prints the elements of the tuple with spaces separating values in the tuple. PyBuildClass is similar to a built-in function, but is only accessible via the *LOAD_BUILD_CLASS* instruction.

PyIterators represents the collection of all iterator classes which include all the iterable values supported by JCoCo including dictionaries, lists, files, funlists (which are functional lists implemented as head/tail links). The key shows that the dark grey classes are used internally in the JCoCo implementation and are not available to CASM programs. These classes are part of the internal implementation of JCoCo and are not accessible to the programmer. Some of them have been described earlier

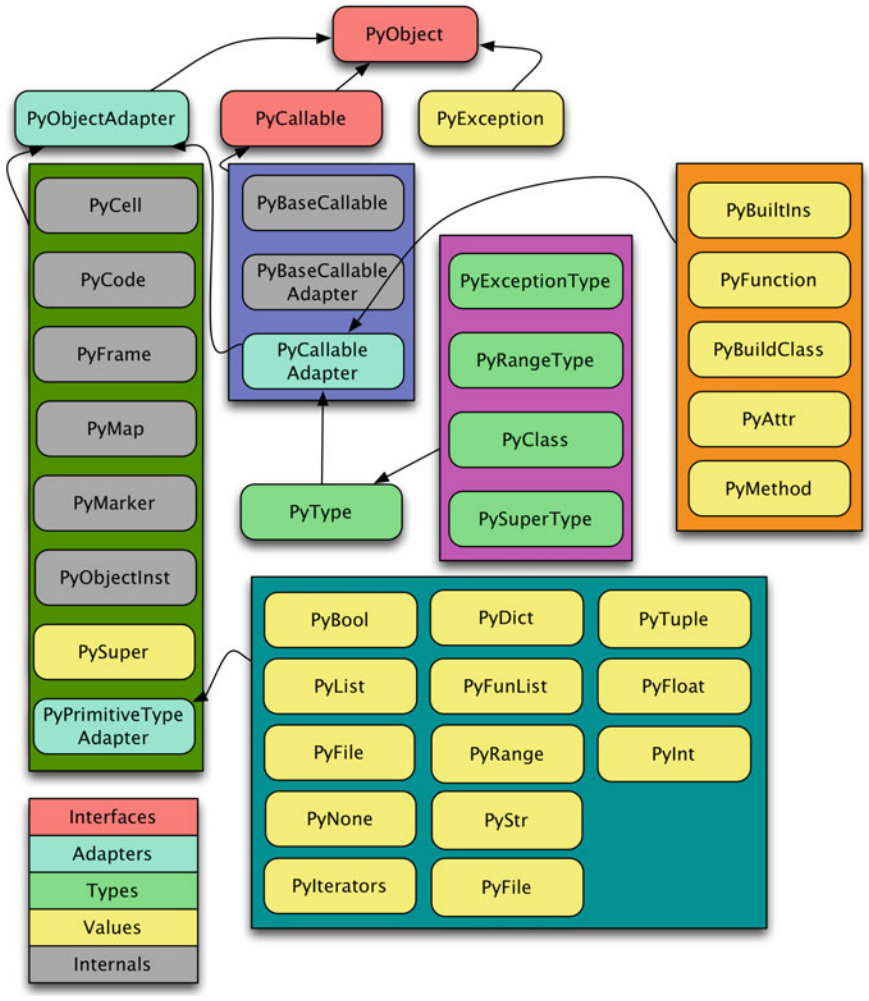


Fig. 4.36 JCoCo type hierarchy

in this chapter. The PyType class is used by all but two of the JCoCo types of values in the construction of their type objects. The type of exception and range objects had to inherit from the PyType class so the behavior of calling the type could be overridden since these two types can be called to build either an exception object or range object, respectively.

There are two interfaces and three adapter classes. When possible, adapters were written to allow code to be shared between multiple classes. Consider the PyPrimitiveTypeAdapter class. This class defines the magic methods *__repr__*, *__str__*, *__hash__*, *__iter__*, and *__type__*. These methods are called by the *repr*, *str*, *hash*, *iter*, and *type* methods.

The PyException class is one interesting example of the need for multiple inheritance in Java. PyException inherits from RuntimeException. By inheriting from RuntimeException, JCoCo exceptions don't have to be declared to be thrown in each and every method that either throws or calls something that could throw an exception. Without inheriting from RuntimeException pretty much every method in JCoCo would have to be declared as possibly throwing a PyException which would have made quite a mess of the code.

Since PyException inherits from RuntimeException it cannot inherit from PyObjectAdapter. Instead, PyException implements the PyObject interface and therefore must re-implement the methods common to the PyObjectAdapter class. With multiple inheritance this could have been avoided. But, this is the only case where multiple inheritance would have been useful in this collection of classes.

JCoCo supports a number of different types of values, including integers, floats, dictionaries, lists, strings, booleans, and a few others. Each of these values has a type associated with it. Each JCoCo object has a method called *getType* that returns its type. It turns out even *types* are objects and they too have a type. Calling *getType* on a type returns the type named *type*. This has to end somewhere and it does with the type object named *type*. The type of *type* is *type*. This comes from the first two lines of the *initTypes* function in the JCoCo.java source file. The type object is created on the first line and is created with itself as its own type identifier.

```
PyType typeType = new PyType("type", PyTypeId.PyTypeType);
PyTypes.put(PyTypeId.PyTypeType, typeType);
```

The next few sections will dive deeper into a few of the JCoCo classes and interfaces to explore their purpose and how they fit in to the larger implementation of JCoCo.

Practice 4.6 The existence of a class named *PySuper* suggests that classes can be built dynamically (i.e. at run-time). The need for PySuper stems from needing to look up the superclass dynamically, while the program is running, because in general it cannot be known before its use. What instruction in appendix A is responsible for getting JCoCo ready to dynamically create a class?
You can check your answer(s) in Section 4.34.6.

4.24 Code

A CASM function consists of more than just a sequence of PyByteCode objects. There is the name of the function, the number of arguments passed to the function, the list of constants used by the function, the local variables, the global variable references, and any enclosed functions or classes declared within this CASM function. All this information and more is encapsulated within a PyCode object.

```java
package jcoco;

import java.util.ArrayList;
import jcoco.PyException.ExceptionType;
import jcoco.PyType.PyTypeId;

class PyCode  extends PyObjectAdapter {
    private String name;
    private ArrayList<PyObject> nestedClassFunctions;
    private ArrayList<String> locals;
    private ArrayList<String> freevars;
    private ArrayList<String> cellvars;
    private ArrayList<String> globals;
    private ArrayList<PyObject> consts;
    private ArrayList<PyByteCode> instructions;
    private int argCount;
    ... // methods and constructors omitted.
}
```

Fig. 4.37 The PyCode class instance variables

From a Java programming perspective, this code not that unique. It is a container for all the information that goes along with each function. The class declaration of instance variables is given in Fig. 4.37.

PyCode objects cannot be executed. There is no way to *run* a PyCode object. To run code you need two things: the code and the environment in which it should be run. The environment is the variables, functions, and other values that are already defined and initialized before the function executes. The environment and code are both provided to PyFunction objects when they are executed, which is described in more detail in Sect. 4.25.

When a Python function is encoded as a PyCode object there are two lists that are necessary. They reflect the eventual contents of the environment. The free variables are variables that are referred to that exist in the environment and not in the code of the function. The other list, the cellvars, are a list of variables that come from the environment or are part of an inner function's environment that may be modified and therefore have to be indirectly referenced. This means that we go through an extra step to reference cellvars so we can update their values while accessing them from another environment. See Sect. 4.25 for an example.

4.25 Functions

Each function in a CASM file is scanned by the parser and a PyCode object is created in the abstract syntax tree to represent the code, its name and number of arguments along with its declaration of constants, locals, freevars, cellvars, and globals

```
 1 public PyFunction(PyCode theCode, HashMap<String, PyObject> theGlobals,
 2         PyObject env) {
 3     PyTuple tuple = (PyTuple)env;
 4     this.cellvars = new HashMap<String, PyCell>();
 5     this.code = theCode;
 6     this.globals = theGlobals;
 7 
 8     for (int i = 0; i < theCode.getFreeVars().size(); i++) {
 9         this.cellvars.put(theCode.getFreeVars().get(i),
10                           (PyCell)tuple.getVal(i));
11     }
12 
13     PyFunction self = this;
14     this.dict.put("__call__", new PyCallableAdapter() {
15         @Override
16         public PyObject __call__(ArrayList<PyObject> args)  {
17             return self.__call__(args);
18         }
19     });
20 }
21 @Override
22 public PyObject __call__(ArrayList<PyObject> args)  {
23     if (args.size() != this.code.getArgCount()) {
24         throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
25                         "Type Error: expected "+this.code.getArgCount() +
26                         " arguments, got "+args.size());
27     }
28     PyFrame frame = new PyFrame(this.code, args, this.globals,
29                            this.code.getConsts(), this.cellvars);
30     PyObject result = frame.execute();
31     return result;
32 }
```

Fig. 4.38 The PyFunction constructor and __call__ method

as described in Sect. 4.24. But, PyCode objects are not callable as shown in Fig. 4.36. To be callable you need both the code and an environment in which to execute the code. The environment fills in the gaps so to speak. The freevars are not defined within the function's code. The freevars come from the environment.

The code in Fig. 4.38 provides the constructor and the call method for the PyFunction class. The constructor builds what is called a closure from the environment and the code. The closure is initialized on line 8–10 where we iterate over the free variables in the code mapping the cell variables in the closure from their free variable names to their cells variable values. All variables that are accessed from the closure are accessed indirectly, through cell variables.

When a function gets called, the *__call__* method gets called. When this occurs a new PyFrame object is created. The PyFrame contains the program counter and space for local variables to be stored. A PyFrame object is executed by calling the execute method.

Practice 4.7 What are the free variables and bound variables in this Python function?

```
def f(x):
    y = x
    return aVal + lstInts[0] + y
```

You can check your answer(s) in Section 4.34.7.

4.26 Classes

User-defined classes in JCoCo are collections of PyFunction objects and nested classes. The class contains the name of the class, (i.e. its type) its super class, and a list of PyFunction or PyClass objects as shown in Fig. 4.39. This reflects the implementation in Python. For instance, while it's not normally written this way, to add two integers together it is possible to write this.

```
z = int.__add__(x,y)
```

This code looks up the addition magic method in the int class and calls it passing the two arguments to the function. While addition is a method called on an integer object, it can also be called on the class by providing both integers.

```
> python3.2
Python 3.2.5 (v3.2.5:cef745775b65, May 13 2013, 13:37:00)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> int.__add__(4,6)
10
>>>
```

When a PyClass is constructed it is passed a list of inner classes and one PyCode object for each of its methods. The PyCode objects are used to build PyFunction objects on lines 22–23. The PyFunction objects are placed in the *attr* dictionary. The variable *attr* of the class holds attributes that are to be passed on to any instance of this class. The class contains the functions that will become methods of any instance of the class. For example, the *__add__* function in the int class will become a method in an instance of the int class.

4.27 Methods

Instances of a class are created by calling their class. For instance, writing *Dog("Mesa")* creates an instance of the *Dog* class. The code that is executed when a class is called is shown in Fig. 4.39 on lines 48–54, which calls the *initInstance* method on lines 36–47. In this code all PyFunctions found in the *attrs* variable are

```
1  public PyClass(String name, ArrayList<PyObject> nestedclassesandfuns,
2              String baseClass, HashMap<String, PyObject> globals) {
3      super(name, PyTypeId.PyClassType);
4      this.baseClass = baseClass;
5      this.name = name;
6      this.classesandfuns = nestedclassesandfuns;
7      this.globals = (HashMap<String, PyObject>)globals;
8      this.attrs.put("__name__", new PyStr(name));
9      for (int i = 0; i < classesandfuns.size(); i++) {
10         if (classesandfuns.get(i).getType().typeId() == PyTypeId.PyCodeType) {
11             PyCode code = (PyCode) classesandfuns.get(i);
12             ArrayList<PyObject> env = new ArrayList<PyObject>();
13             for (int j = 0; j<code.getFreeVars().size(); j++) {
14                 String freeVar = code.getFreeVars().get(j);
15                 if (freeVar.equals("__class__")) {
16                     env.add(new PyCell(this));
17                 } else {
18                     throw new PyException(ExceptionType.PYMATCHEXCEPTION,
19                      "Error: Found unexpected freevar in class declaration.");
20                 }
21             }
22             PyFunction fun = new PyFunction((PyCode) classesandfuns.get(i),
23                 globals, new PyTuple(env));
24             this.attrs.put(fun.callName(), fun);
25         } else if (classesandfuns.get(i).getType().typeId() ==
26                   PyTypeId.PyTypeType) {
27             PyClass cls = (PyClass) classesandfuns.get(i);
28             this.attrs.put(cls.getName(), cls);
29         } else {
30             throw new PyException(ExceptionType.PYMATCHEXCEPTION,
31                         "TypeError: expected a Function or Class, got "+
32                         classesandfuns.get(i).getType().str());
33         }
34     }
35 }
36 public void initInstance(PyObjectAdapter obj) {
37     if (!baseClass.equals("")) {
38         ((PyClass)globals.get(baseClass)).initInstance(obj);
39     }
40     for (String name : this.attrs.keySet()) {
41         if (this.attrs.get(name).getType().typeId() ==
42               PyTypeId.PyFunctionType) {
43             obj.dict.put(name, new PyMethod(name, obj,
44                 (PyCallable)this.attrs.get(name)));
45         }
46     }
47 }
48 @Override
49 public PyObject __call__(ArrayList<PyObject> args) {
50     PyObjectAdapter obj = new PyObjectInst(this);
51     initInstance(obj);
52     ((PyMethod) obj.dict.get("__init__")).__call__(args);
53     return obj;
54 }
```

Fig. 4.39 The PyClass constructor and __call__ method

passed on to the instance of the class (i.e. the object instance) as PyMethod objects. Then, on line 52 the object's constructor is called to perform any initialization of the object. Continuing our example from the previous section, this means that like in Python, the following code can be disassembled and executed in JCoCo.

```
1  @Override
2  public PyObject __call__(ArrayList<PyObject> args)  {
3      args.add(this.self);
4      PyObject result = fun.__call__(args);
5      //take self back out of args because when a method
6      //is called no one would suspect that a side-effect
7      //was that args was mutated.
8      args.remove(args.size()-1);
9      return result;
10 }
```

Fig. 4.40 The PyMethod __call__ method

```
> python3.2
Python 3.2.5 (v3.2.5:cef745775b65, May 13 2013, 13:37:00)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> int(4).__add__(6)
10
>>> 4 + 6
10
>>>
```

The integer 4 had to be written *int(4)* because otherwise it would look like there was a decimal point in the number. Syntactically it would not be an integer in that case. Of course, the more usual way to call this method is by using the overloaded + operator.

The PyMethod class is a wrapper class for a PyFunction. A PyFunction of a class is passed on to any instance of that class as a PyMethod as shown in Fig. 4.39. Calling a method on an object is usually written as *object.method(arg1, arg2, ...)*. It is useful to know that JCoCo passes the list of args in reverse order. So the last argument is first, followed by the second to last, and so on.

When a method is called as in *object.method(args)* the first argument passed to the method is always *self* in Python. This *self* variable is the reference to the *object*. Since the arguments are passed in reverse order in JCoCo, the *self* argument can be added to the end of the arguments passed to the function that the method encapsulates. This is shown in Fig. 4.40 on line 3 where the reference to the current object is added to the end of the ArrayList of arguments. Note that line 8 removes self from the *args* ArrayList before returning so the calling code won't see a modified list of arguments.

Practice 4.8 Consider the code in the example below. When a Dog object is created its __*init*__ method implicitly gets called. We never explicitly call the constructor on a Python object. Where does __*init*__ get called in the JCoCo virtual machine?

```
mydog = Dog("Mesa")
```

You can check your answer(s) in Section 4.34.8.

```
 1 public PyObject execute() {
 2   this.PC = 0;
 3   boolean handled = false;
 4   JCoCo.pushFrame(this);
 5   while (true) {
 6     try {
 7       inst = this.code.getInstructions().get(this.PC);
 8       this.PC++;
 9       opcode = inst.getOpCode();
10       operand = inst.getOperand();
11       switch (opcode) {
12         // instructions omitted
13         case SETUP_EXCEPT:
14           this.blockStack.push(-1 * operand);
15           opStack.push(new PyMarker());
16           break;
17       }
18     } catch (PyException ex) {
19       int exitAddress;
20       boolean found = false;
21       while (!found && !this.blockStack.isEmpty()) {
22         exitAddress = this.blockStack.pop();
23         if (exitAddress < 0) {
24           found = true;
25           if (!opStack.isEmpty()) {
26             PyObject obj = opStack.pop();
27             while (!obj.str().equals("Marker") && !opStack.isEmpty()) {
28               obj = opStack.pop();
29             }
30           }
31           this.opStack.push(ex.getTraceBack()); //The traceback at TOS2
32           this.opStack.push(ex); //The exception at TOS1
33           this.opStack.push(ex); //the exception at TOS
34           this.PC = -1 * exitAddress;
35           this.blockStack.push(0);
36         }
37       }
38       if (!found) {
39         ex.tracebackAppend(this);
40         throw ex;
41       }
42     } catch (Exception e) {
43       PyException ex =
44          new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
45          e.getMessage()+" while executing instruction "+inst.getOpCodeName());
46       ex.tracebackAppend(this);
47       throw ex;
48     }
49   }
50 }
```

Fig. 4.41 A synopsis of exception handling in JCoCo

4.28 JCoCo Exceptions and Tracebacks

The *execute* method for *PyFrame* exits in one of two ways. Either the *RETURN_VALUE* instruction is executed, or an exception occurs that is not handled within this function. If an exception occurs, execution jumps to line 18 of the code in Fig. 4.41. All intentionally thrown exceptions thrown by JCoCo are *PyException* objects, so the catch block on line 18 will catch it.

JCoCo exceptions use the exception handling mechanism of Java to jump to the code starting on line 18 anytime a PyException is *thrown* in a JCoCo program, which would be any exception intentionally thrown by a CASM program. Upon entering the catch block on line 19 the code looks for an exception handler that may have been put in place to handle the exception in this PyFrame object. There may be an exception handler and there might not. JCoCo includes a block stack used for iteration and exception handling. The block stack records the exit points of loops and the addresses of exception handlers. For loops, the exit point is pushed on the block stack in case a *BREAK_LOOP* instruction is executed to break out of a loop. When an exception handler is put in place the location of the handler is indicated by a negative value on the block stack to differentiate it from loop exit points. Lines 13–16 show how an exception handler is set up within a frame. The *SETUP_EXCEPT* instruction pushes the exception handler address onto the block stack.

When an exception occurs the block stack is popped until the address of an exception handling block is found (i.e. a negative value is popped). The address of the exception handler is the negation of this negative value.

When an exception occurs it can occur anywhere within the code. In particular there could be operands left on the operand stack because the exception occurred in the middle of some other work. For instance, an exception might occur while preparing for a function call and arguments may be left on the operand stack. The PyMarker class serves to help clean up the operand stack after an exception is caught. When an exception handler is installed with the *SETUP_EXCEPT* instruction, a PyMarker object is pushed onto the operand stack. If a PyMarker object is popped during normal instruction execution, it is just thrown away. If an exception occurs the exception handling code on line 25–29 pops arguments from the operand stack until it is emptied or until a PyMarker object is found, thus cleaning up the operand stack.

If an exception handler was found in the JCoCo function's code, the exception is pushed onto the stack to get ready to jump to the exception handler code. Lines 31–33 seem a little strange until you remember that JCoCo maintains compatibility with Python 3.2 and disassembled Python code expects there to be three operands pushed onto the stack when an exception handler begins executing. Line 34 causes execution to jump to the first instruction of the exception handler. The Python virtual machine also assumes there is another exception handling block pushed on the blockStack. This is not needed by JCoCo, but to maintain compatibility line 35 pushes an entry onto the block stack.

If no exception handler is found then line 38 adds the current frame to the exception's traceback. The traceback is a list of all the PyFrame objects that are popped until an exception handler is found. If no exception handler is found, control returns to the main function where the traceback is printed.

Non-JCoCo Java exceptions may also be thrown by JCoCo code. This can happen in one of two scenarios. JCoCo may have a bug and if so, an exception may be thrown due to some unforeseen circumstance. The other reason a standard Java exception may be thrown would be due to the programmer attempting an illegal operation, like an arithmetic operation causing integer overflow for instance. If either of these scenarios occur, then lines 42–49 will handle those exceptions. When a non-JCoCo Java exception occurs a PyException is created, the current frame is added to its traceback and the PyException is thrown. If this PyException is not caught anywhere within the CASM program then control will return to the main function in JCoCo.java and the exception's traceback will be printed as the source of the error.

4.29 Magic Methods

The JCoCo implementation of Python's virtual machine makes use of inheritance to reuse code in the implementation and to create *is-a* relationships between the objects manipulated by JCoCo. That type hierarchy is provided in Fig. 4.36. All of the JCoCo datatypes implement the PyObject interface which sits at the root of this type hierarchy.

Through inheritance every descendent of PyObjectAdapter contains a dictionary called *dict* that maps method names to methods as shown in Fig. 4.18. In Python, when a method is called, the method name is looked up in this dictionary to locate the code that corresponds to the method name. JCoCo mirrors this implementation to emulate the dynamic run-time typing behavior of Python. The dynamic lookup of methods occurs in the *callMethod* method of the PyObjectAdapter class as described in the related Sect. 4.13.

Figure 4.42 contains four methods that are defined on every type of value in JCoCo. For instance, any object can be converted to a string and all objects have a type within the hierarchy that can be retrieved. Notice that all these methods have the same signature. Each function or method in JCoCo (and Python) takes a list of objects as arguments and returns an object. Every JCoCo object implements methods with this signature and only this signature. The *__str__* and *__type__* methods are called magic methods by Python developers because they get called automatically by certain operators in Python. For instance, converting a PyObject to a string calls the *__str__* magic method to get a string representation of the object. Calling *type* on an object in a Python program results in calling the *__type__* method to get an object's type.

Calling *repr* on an object in a Python program calls the *__repr__* method. The *repr* function returns a string representation of an object that, when evaluated, would yield a copy of that same object. The *hash* function may also be called on any object, but only hashable objects implement the *__hash__* magic method with something other than throwing an exception.

Methods are the operations that can be performed on objects within the JCoCo type hierarchy. Magic methods are methods that get called as a result of either a built-in function call or some operator overloading in Python. The *__str__*, *__type__*, *__repr__*, and *__hash__* magic methods are added to the dictionary of methods for all objects by the PyObjectAdapter constructor. Figure 4.42 shows the methods that are supported by the PyObjectAdapter class for all subclasses. But, subclasses of PyObjectAdapter may add to the map of supported operations. For instance, the

```
public PyObjectAdapter() {
    name = "PyObject()";
    type = PyType.PyTypeId.PyClassType;
    PyObjectAdapter self = this;
    this.dict.put("__str__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            return new PyStr(self.str());
        }
    });
    this.dict.put("__hash__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
            "TypeError: unhashable type: '" + self.getType().str() + "'");
        }
    });
    this.dict.put("__repr__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            return self.callMethod("__str__", args);
        }
    });
    this.dict.put("__iter__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            throw new PyException(ExceptionType.PYILLEGALOPERATIONEXCEPTION,
            "TypeError: '" + self.getType().str() + "' object is not iterable");
        }
    });
    this.dict.put("__type__", new PyBaseCallable() {
        @Override
        public PyObject __call__(ArrayList<PyObject> args) {
            if (args.size() != 0) {
                throw new PyException(ExceptionType.PYWRONGARGCOUNTEXCEPTION,
                "TypeError: expected 0 argument, got " + args.size());
            }
            return (PyObject) self.getType();
        }
    });
}
```

Fig. 4.42 PyObjectAdapter's constructor

PyInt object's constructor calls the *funs* method to add a whole host of additional supported methods to integer objects as shown in Fig. 4.43.

The method name, called just *name* in the code in Fig. 4.20, is searched for in the dictionary. If it is not found, an exception is thrown. Otherwise, *mbr* is made to point at the code of the method (i.e. a PyCallable object). Line 6 calls the *__call__* method for this member function or method on the current object, returning whatever is returned from the call to the caller. The use of the dictionary maps names (i.e. strings) provided to JCoCo, to the methods of objects, which are implemented as PyCallable objects, within JCoCo. If the object does not have a method defined, the *callMethod* code gracefully handles this by throwing an exception which will result in a traceback being printed of the offending *CALL_FUNCTION* instruction.

Within JCoCo, magic methods get called as the result of many instructions. For instance, the *__add__* magic method gets called on an object as the result of executing the *BINARY_ADD* instruction. The *COMPARE_OP* instruction calls several different magic methods depending on the comparison operand of the instruction. Precisely which magic method gets called for a given instruction is detailed in the documentation provided in Appendix A.

Practice 4.9 How can an object or class override the default behavior of a magic method like the *__str__* method without changing the JCoCo virtual machine itself?
You can check your answer(s) in Section 4.34.9.

```
public static HashMap<String, PyCallable> funs() {
    HashMap<String, PyCallable> funs = new HashMap<String, PyCallable>();
    funs.put("__hash__", new PyCallableAdapter() {...});
    funs.put("__add__", new PyCallableAdapter() {...});
    funs.put("__sub__", new PyCallableAdapter() {...});
    funs.put("__mul__", new PyCallableAdapter() {...});
    funs.put("__pow__", new PyCallableAdapter() {...});
    funs.put("__truediv__", new PyCallableAdapter() {...});
    funs.put("__floordiv__", new PyCallableAdapter() {...});
    funs.put("__mod__", new PyCallableAdapter() {...});
    funs.put("__eq__", new PyCallableAdapter() {...});
    funs.put("__ne__", new PyCallableAdapter() {...});
    funs.put("__lt__", new PyCallableAdapter() {...});
    funs.put("__le__", new PyCallableAdapter() {...});
    funs.put("__gt__", new PyCallableAdapter() {...});
    funs.put("__ge__", new PyCallableAdapter() {...});
    funs.put("__float__", new PyCallableAdapter() {...});
    funs.put("__int__", new PyCallableAdapter() {...});
    funs.put("__bool__", new PyCallableAdapter() {...});
    funs.put("__str__", new PyCallableAdapter() {...});
    return funs;
}
```

Fig. 4.43 PyInt's additional magic methods

```
 1 def main():
 2   d = {}
 3   d["hello"] = "goodbye"
 4   d["dog!"] = "cat!"
 5   d["young"] = "old"
 6   s = "hello young dog!"
 7   t = s.split()
 8   for x in t:
 9     print(x)
10   for x in t:
11     print(d[x])
12   for x in d.keys():
13     print(x, d[x])
14   for y in d.values():
15     print(y)
16   for key in d:
17     print(key, d[key])
18   print(type(d))
19   print(type(type(d)))
20 main()
```

Fig. 4.44 dicttest.py

4.30 Dictionaries

Python implements a type called *dict*, short for dictionary, which is a map from *keys* to *values*. Dictionary objects are implemented as a hash table with O(1) get and set methods. A dictionary is created by using the braces around an optional list of key/value pairs. Line 2 of Fig. 4.44 shows an empty dictionary being created. Items are put in the dictionary using subscript notation. The key is the subscript and the value is the assigned value at the key's location. Lines 3–5 provide an example of storing key/value pairs in a dictionary. Line 11 uses subscript notation to look for a value corresponding to a key.

Dictionaries differ from lists because the subscript can be almost any type of value. Dictionaries are not limited to integer subscripts like lists. There are three requirements of a dictionary key. The key must be *hashable*. Hashing refers to deriving an integer from a value, as close to unique as possible. Keys in a dictionary must support an equality test. There must be a way of determining if two keys are equal. Finally, keys should not be mutable. Python lists are not suitable for keys because they are mutable. In JCoCo strings, integers, floats, tuples, and funlists are not mutable and therefore are suitable as keys in a dictionary. Funlists are a JCoCo specific functional programming list that is not a part of standard Python, but is supplied with JCoCo. Floats are not usually used as keys due to their being approximations of real numbers and the chance for round-off error in calculations. In general floats should

```java
package jcoco;

public class PyDict extends PyPrimitiveTypeAdapter {
    private HashMap<PyObject, PyObject> map = new HashMap<PyObject, PyObject>();
    public PyDict() {
        super();
        initMethods(funs());
    }
    public void setVal(PyObject key, PyObject val) {...}
    @Override
    public PyType getType() {
        return JCoCo.PyTypes.get(PyTypeId.PyDictType);
    }
    @Override
    public String str() {...}
    public static HashMap<String, PyCallable> funs() {
        HashMap<String, PyCallable> funs = new HashMap<String, PyCallable>();
        funs.put("__getitem__", new PyCallableAdapter() {...});
        funs.put("__setitem__", new PyCallableAdapter() {...});
        funs.put("__len__", new PyCallableAdapter() {...});
        funs.put("__iter__", new PyCallableAdapter() {...});
        funs.put("keys", new PyCallableAdapter() {...});
        funs.put("values", new PyCallableAdapter() {...});
        return funs;
    }
}
```

Fig. 4.45 Outline of PyDict.java

not be compared for equality and therefore, while immutable, they are not usually appropriate as keys in a dictionary.

The *dict* datatype is not included in the JCoCo implementation available to you on Github. This section describes the steps to add dictionaries as a case study of extending the JCoCo virtual machine. You must have the Java development environment installed on your computer to complete the steps described here. When you have completed the steps in this section the disassembled code from Fig. 4.44 will execute on JCoCo producing output similar to that of Python.

4.30.1 Two New Classes

Two new classes are required to support dictionaries; the PyDict class and the PyDictIterarator class. The PyDict class resembles the PyList class in some ways. The PyDict class must be added to the Java source code in a file called PyDict.java. An excerpt of the PyDict.java header file is given in Fig. 4.45.

There are several methods to be implemented to support dictionaries. The *__getitem__* method is given a key in the args vector and returns the correponding value. The *__setitem__* method maps a key to a value. The key is at index 0 in *args* and the value is at *args[1]*. The *__len__* method returns the size of the map. All of these methods use the HashMap called *map* and the methods of a HashMap are used in the implementation of the PyDict class.

The *map* instance variable is central to the PyDict implementation. A HashMap is part of the utility library of Java. There is a subtle implementation detail with the use of HashMap here. Java provides built-in support for hashing items. The *Object* class of Java includes a method called *hashCode* that is used to get a hash value for any hashable Java value. But, in JCoCo we wish to call the *__hash__* magic method to give the JCoCo assembly language programmer control over how an object is hashed. The *PyObjectAdapter* class comes to the rescue here by defining the *hashCode* method to call the *__hash__* magic method.

```
@Override
public int hashCode() {
    ArrayList<PyObject> args = new ArrayList<PyObject>();
    PyInt val = (PyInt) this.callMethod("__hash__", args);
    return val.getVal();
}
```

This means that the JCoCo PyDict implementation can just use the HashMap as you would in any Java program and the *__hash__* magic method will automatically get called when needed.

Many of the classes provided with Java have hashing functions defined for them already. The PyStr class can use the string hashing function provided by the *hashCode* method of PyObject. That *hashCode* function can be called in the *__hash__* magic method of PyStr to hash the string. Every type of object that could be used for a key in a dictionary must implement the *__hash__* method.

The HashMap needs to determine if two keys are equal as part of the hash table implementation. The HashMap needs to know if the key it is looking up matches one that it finds in the hash table. Again, PyObjectAdapter comes to the rescue. The HashMap automatically calls the *equals* method of Java Object to determine if the two keys are equal or not. The PyObjectAdapter overrides the *equals* method to call the *__eq__* magic method. So, the HashMap automatically makes calls to both the hash and equals magic methods. This means that the equals magic method must be implemented in any class that will be used as a key in a dictionary. Of course, this is already done for the built-in types of JCoCo.

The other class to be written is a PyDictIterator class that implements iteration over the keys of the dictionary. The PyListIterator can be used as an example in how to write this iterator. Remember that to terminate the iteration, the *PYSTOPITERATION* exception must be thrown once the iterator is exhausted. Here is how iteration is achieved over a Java HashMap object.

```
import java.util.Iterator;
import java.util.HashMap;
Iterator<HashMap.Entry<PyObject, PyObject>> it;
it = map.entrySet().iterator();
while (it.hasNext()) {
  HashMap.Entry<PyObject, PyObject> pair = it.next();
  System.out.println(pair.getKey());
}
```

```
def main():
  d = { "Kent":"Denise",
    "Sophus":"Addie"}
  print(d)
```

Fig. 4.46 Initializing a dictionary

This code must be divided up into the PyDictIterator implementation in a manner similar to the PyListIterator code.

4.30.2 Two New Types

In addition to the new classes, two new types must also be defined. The main module, JCoCo.java, contains a function called *initTypes*. The *dict* and *dict_keyiterator* types should be added as two new types to this function. To do this, two new values for the PyTypeID enum in PyType.java must also be defined; the *PyDictType* and *PyDictKeyIteratorType* values. This is a relatively simple addition to the code, but must be tied together with the implementations of the PyDict and PyDictIterator classes. Once these types are created, don't forget to set the instance functions in the type objects.

4.30.3 Two New Instructions

Finally, after disassembling the code in Fig. 4.44 a new instruction appears, the *BUILD_MAP* instruction. This instruction creates an empty dictionary and pushes it onto the operand stack.

Disassembling the code in Fig. 4.46 yields one other instruction. The *STORE_MAP* instruction expects three operands on the stack. The TOS element is a *key*, the TOS1 element is a *value* and the TOS2 element is a dictionary. The STORE_MAP instruction stores the *key/value* pair in the dictionary and leaves the dictionary on top of the operand stack when it completes. These two new instructions are implemented in PyFrame.java in the *execute* method. You can look at other examples of instructions to see how these two instructions should be implemented.

4.31 Chapter Summary

This chapter covered object-oriented, imperative programming in Java with C++ covered to a lesser degree. Advanced techniques including inheritance, polymorphism, interfaces, generics, autoboxing and unboxing, inner classes and a few other important topics were covered with examples coming from the JCoCo virtual machine implementation.

Java and C++ are statically typed languages as compared to Python, which is a dynamically typed language. Static typing requires more work by the programmer when writing code, but also provides some level of assurance that code is type-correct. Python program type errors are not found until run-time. The C++ and Java compilers catch most type errors at compile-time.

C++ and Java share a lot of syntax, but they are distinct languages in many ways. C++ is especially suited to low-level and real-time implementations where performance is critical and you need access to the underlying hardware. C++ gives you complete control of when dynamically allocated data is freed. But with that responsibility comes the age old problem of memory leaks. C++ programs are prone to memory leaks and the C++ CoCo implementation is full of them because it is difficult to determine exactly when space can be freed in the virtual machine without implementing some form of garbage collection. C++ has many great programming features like templates, a large standard library, and compiler support for many hardware platforms.

Java programs benefit from garbage collection which is built into the JVM. Java also provides a unified type hierarchy for classes with Object at the root of the tree. C++ has no built-in class hierarchy. Java has built into it several nice programming features like auto-boxing and unboxing, threading support including synchronization support for all Java objects, inner class support, and support for separating interfaces from implementation.

Both C++ and Java serve as good examples of statically typed, object-oriented, imperative programming languages. They each have their benefits and shortcomings, but for the CoCo and JCoCo virtual machines, Java is the better-suited language providing garbage collection and a more unified approach to handling exceptions within the virtual machine. In the next chapter we introduce another programming paradigm while studying another statically typed language.

4.32 Review Questions

1. What does static type checking mean? Does C++ have it? Does Python have it? Does Java have it?
2. What are the names and purposes of the two programs that make up the Java environment for executing programs?
3. What is the number one problem that C/C++ programs must deal with? Why is this not a problem for Java and Python programs?
4. What does the *make* tool do and how does it work for C++ programs?
5. Is there an equivalent to the *make* tool for Java programs?
6. How does the C++ compiler distinguish between macro processor directives and C/C++ statements?

7. What is a namespace in C++? What is comparable to a namespace in Java? In Python?
8. What is the default executable name for a compiled C++ program?
9. What is separate compilation and why is it important?
10. What is dynamic linking? Does it happen in C++ or in Java? Why is it important?
11. Which environment has garbage collection built in, C++ or Java?
12. What are the advantages of garbage collection?
13. Are there any drawbacks to garbage collection?
14. What is a destructor and when is it needed?
15. What do you have to write to get a polymorphic method in C++?
16. What is the purpose of polymorphism?
17. What is the purpose of inheritance?
18. How do interfaces and classes differ in Java? How are they similar? How are they different?
19. What is an adapter class? Why are they useful?
20. What is a callback and how are they usually implemented in Java?
21. What are generics? Why are they convenient?
22. What is a template? How do you declare a vector in C++?
23. What is auto-boxing and unboxing?
24. How is a function represented as a value in Java?
25. What is an anonymous class?
26. What is the *type(6)* in JCoCo and Python? How about the *type(type(6))*? How about the *type(type(type(6)))*? Why isn't it interesting to go any further?
27. The JCoCo scanner is based on a finite state machine. How is the finite state machine implemented? What are the major constructs used by a finite state machine?
28. Does the JCoCo parser run bottom-up or top-down?
29. In JCoCo how are a PyCode object and a PyFunction object related?
30. What is a traceback and why is it important?
31. What is the purpose of a PyMethod class?
32. Arriving at hash values for hashable objects in Java is trivial. Describe how JCoCo determines hash values for objects in the implementation of PyDict objects.

4.33 Exercises

1. Alter the finite state machine of PyScanner.java to allow strings to include the escape character. Any character following the backslash, or escape character, in a string should be allowed. This project can be implemented by altering the PyScanner.java class to allow the escape character to appear within a string. Hint: Two extra states may be needed to implement this code. Note that JCoCo will already allow pretty much any character, including tabs and newline characters, to be included in a string constant. The only characters that pose problems are

single and double quotes. The escape character should not be included in the constant string, only the character that follows the escape character.

2. Implement true division and floor division for floats in JCoCo. Write a test program to thoroughly test these new operations supported by floats. The test program and the source code are both required for the solution to this problem. You may use the disassembler to help generate your test program.
3. Alter the JCoCo grammar to allow each line of a function's code to be either a JCoCo instruction or a source code line. Any source code line should be preceded by a pound sign, a line number, and a colon followed by the text of the source code line. A source code line would reflect a line from a source language other than JCoCo which was compiled to the JCoCo assembly language. Then, when an uncaught exception occurs in the JCoCo program, the traceback should be printed along with the source code line that caused the exception. This is a challenging exercise and requires changes to the scanner, parser, internal storage of PyCode objects, and traceback handling.
4. Add a dictionary object type to JCoCo by following the description at the end of this chapter. This project requires significant programming and there are pieces in the last part of the chapter that are left out. However, the provided code samples along with other similar code in the JCoCo project provides enough details to be able to complete it. When done, the successful project will be able to run the disassembled code from Figs. 4.44 and 4.46. The output should appear to be identical to the output produced by running the Python programs. However, the order of keys may be different since dictionaries are implemented with an unordered_map datatype.
5. Empty type calls produce *empty* results in Python but not in JCoCo. For instance, when *int()* is called in Python, the object 0 is created. In JCoCo this produces an error. Use Python to determine what should happen for all the empty type calls that JCoCo supports. Then modify CoCo so it will behave in a similar fashion.
6. Add a *set* datatype to JCoCo. Lookup the *set* datatype in Python documentation. Include support in your *set* datatype to support contructing a set, union, intersection, mutating union, and mutating set difference along with set cardinality, membership, and addition of an element to the set. Write a Python test program and disassemble it. Then run your test program to test your set datatype.
7. Modify JCoCo to allow instructions like *LOAD_FAST x* in addition to *LOAD_FAST 0*. Currently, the LOAD_FAST and STORE_FAST instructions insist on an integer operand. If an identifier operand were provided then the identifier must exist in the sequence of *LOCALS*. If it does not, the parser should signal an error. Internally, the LOAD_FAST and STORE_FAST instructions should not change. The conversion from identifier to integer should happen in the parser. Convert the *LOAD_GLOBAL*, and *LOAD_ATTR* instructions to allow either an identifier or integer operand in the same manner. Do not try to modify the *LOAD_CONST* instruction since it would be impossible to distinguish between indices and values for constants.

   This project is not too hard to implement. Labels are already converted to offsets in the parser in the *BodyPart* method. That code has to be modified slightly to handle identifiers for things other than labels. The identifiers for the load and store instructions can be converted to integer operands in the *FunDef* function.

8. Currently the assembler has three different load instructions including *LOAD_FAST*, *LOAD_GLOBAL*, and *LOAD_DEREF* that all use indices into different lists as operands. Define a new pseudo *LOAD* instruction that lets you specify an identifier for a value to load. For instance *LOAD x* would result in scanning the *LOCALS* list for *x*. If *x* were found in the first position of the locals list, then the *LOAD x* would be changed to a *LOAD_FAST 0* instruction. Otherwise, if *x* was not in the list of locals, then the *GLOBALS* would be scanned next and if *x* were found there a *LOAD_GLOBAL* instruction would replace the *LOAD* pseudo instruction. If *x* was not found in the globals, then the cellvars could be scanned and finally the freevars. Create a *STORE* pseudo instruction as well for the *STORE_FAST* and *STORE_DEREF* instructions.
   Do not try to implement the pseudo instructions for any of the other load or store instructions. For instance, it would be impossible to know whether a *LOAD* referred to a *LOAD_DEREF* or a *LOAD_CLOSURE* if you tried to include *LOAD_CLOSURE* in your pseudo instruction.

4.34 Solutions to Practice Problems

4.34.1 Solution to Practice Problem 4.1

The Java compiler insists that if the class is called *Test* then the file must be *Test.java*. This is necessary because when class *A* is compiled and uses the *Test* class, the Java compiler must find the *Test* class. There is nothing included or imported into class *A* to tell the compiler where to look. Instead, Java looks for a file called *Test.java* if class *Test* is used in class *A*.

4.34.2 Solution to Practice Problem 4.2

The C++ compiler uses macro processor directives to explicitly include header files which declare classes and other entities. The header file names are explicitly provided in the *include* macro processor directive. The C++ compiler does not need to infer the name of the module from the name of the class like Java does.

4.34.3 Solution to Practice Problem 4.3

In C++ programs the name of the executable for the program is passed in *argv[0]*. So the value of *argc* is always at least one. In a Java program the program consists of a collection of .class files. The main function must be defined inside the public class for one of these .class files, so the name of the main module is always known by the programmer and therefore is not passed as an argument.

4.34.4 Solution to Practice Problem 4.4

Polymorphism occurs in Python because methods are looked up by name at run-time. This leads to run-time type checking, not compile-time as is supported by C++ and Java. C++ and Java are statically typed languages. Python is dynamically typed. Further, Python supports inheritance, but the purpose of inheritance in Python is only for code re-use. Polymorphism is not related to inheritance in Python programs since polymorphism occurs because of the late, just in time, lookup of methods in an object's attribute dictionary.

4.34.5 Solution to Practice Problem 4.5

The confusion may occur when an *ArrayList* of *Integer* was declared. When calling *a.remove(1)* will autoboxing occur? The answer is no, in this case because the *ArrayList* class contains a method with *int* as a parameter Java will choose the method with the closest argument types when there are multiple to choose from. Nevertheless, the programmer must be aware of this to correctly choose the right *remove* method. If the *Object* argument version is really the one to call, then it must called as *a.remove(new Integer(1))* to get the correct *remove* called. Boxing must be explicitly called in this case. No autoboxing will occur.

4.34.6 Solution to Practice Problem 4.6

It's the LOAD_BUILD_CLASS instruction. This instruction loads the built-in function onto the operand stack so it can be called to dynamically build a class.

4.34.7 Solution to Practice Problem 4.7

The bound variables are *f*, *x*, and *y*. They are bound because the name of the function is always defined and the parameter name is given the value of any argument passed to the function. The variable *y* is bound to the same value as *x* because *y* appears on the left side of an assignment statement.

The free variables are *aVal* and *lstInts*. These values have to be supplied in the environment by forming a closure before the function can be executed.

4.34.8 Solution to Practice Problem 4.8

The __init__ gets called during object instantiation on line 52 of Fig. 4.39.

4.34.9 Solution to Practice Problem 4.9

Since magic methods are looked up at run-time (i.e. dynamically) then at any time before the magic method, the object can have its magic method implementation replaced by an alternative implementation. There is no modification necessary to the JCoCo Java code.

5 Functional Programming

Chapter 3 introduced assembly language which was a very prescriptive language. Certain operands had to be on the operand stack before an instruction could be executed. These details had to be dealt with even though the programmer was trying to solve a bigger problem than how to execute the next instruction. This was solved by learning some patterns of assembly language instructions that could be used to solve bigger problems like implementing a loop. Of course, even writing a loop is more prescriptive than trying to compute the sum of some list of integers.

Chapter 4 moved on to Java and C++ where programming was less prescriptive. Most programmers learn to program imperatively first. Object-oriented languages are imperative languages where objects are created and the states of objects are updated as program execution proceeds. Thinking about maintaining and updating the states of objects is a lot less prescriptive than thinking about which instruction to execute next.

This chapter introduces *functional programming*. Functional languages, like Standard ML, obviously concentrate more heavily on writing and calling functions. However, the term *functional programming* doesn't say what functional programming languages lack. Specifically, pure functional languages lack assignment statements and iteration. Iteration relates to the ability to iterate or repeat code as in a loop of some sort. It is impossible in a pure functional language to declare a variable that gets updated as your program executes! If you think about it, if there are no variables, then there isn't any reason for a looping construct in the language. Iteration and variables go hand in hand. But, how do you get any work done without variables? The primary mode of programming in a functional language is through recursion.

Functional languages also contain a feature that other languages don't. They allow functions to be passed to functions as parameters. We say that these functions are higher-order. Higher-order functions take other functions as parameters and use them. There are many useful higher order functions that are derived from common patterns of computation. Particular instances of these patterns commonly have one small

```
program P;
  var b : integer;
  function a() : integer;
  begin
    b:=b+2;
    return 5
   end;
begin
  b:=10;
  write(a()+b)
  (* or write(b+a()) *)
end.
```

Fig. 5.1 Commutativity

difference between them. If that small difference is left as a function to be defined later, we have one function that requires another function to complete its implementation. Higher-order functions may be customized by providing some of their functionality later. In some ways this is the functional equivalent of what inheritance or interfaces provide us in object-oriented languages.

These two features, lack of variables and higher-order functions, drastically change the way in which you think about programming. Programming recursively takes some time to get used to, but in the end it is a very nice way to program. Programming recursively is more declarative than prescriptive. Writing imperative programs is prescriptive. When programming declaratively we can focus on *what* we want to say about a problem instead of exactly *how* to solve a problem.

But why would we want to get rid of variables in a programming language? The problem is that variables often make it hard to reason about our programs. Functional languages are more mathematical in nature and have certain rules like commutativity and associativity that they follow. Rules like associativity and commutativity can make it easier to reason about our programs.

Practice 5.1 Is addition commutative in C++, Pascal, Java, or Python? Will *write(a+b)* always produce the same value as *write(b+a)*? Consider the Pascal program in Fig. 5.1. What does this program produce? What would it produce if the statement were *write(b+a())*?
You can check your answer(s) in Section 5.26.1.

5.1 Imperative Versus Functional Programming

You are probably familiar with at least one imperative language. Languages like C, C++, Java, Python, and Ruby are considered imperative languages because the fundamental construct is the assignment statement. In each of these languages we declare variables and assign them values, updating those variables as a program's execution progresses.

Imperative languages are heavily influenced by the von Neumann architecture of computers that includes a store and an program counter; the computation model has control structures that iterate over instructions that make incremental modifications of memory. Assignment of values to variables, for loops, and while loops are all part of imperative languages. The principal operation is the assignment of values to variables. Programs are statement oriented, and they carry out algorithms with statement level sequential control. In other words, computing is done by side-effects.

Sometimes problems with imperative programs stem from these side-effects. It is difficult to reason about a program that relies on side-effects. If we wish to reuse the code of an imperative program then we must be sure that the same conditions are true before the reused code executes since imperative code relies on a certain machine state. As programmers we sometimes forget which preconditions are required and what postconditions result from executing a segment of code. That can lead to bugs in our programs.

Functional languages are based on the mathematical concept of a function and do not reflect the underlying von Neumann architecture. These languages are concerned with data objects and values instead of variables. The principal operation is function application.

Functions are treated as first-class objects that may be stored in data structures, passed as parameters, and returned as function results. Primitive functions are generally supplied with the language implementation. Functional languages allow new functions to be defined by the programmer. Functional program execution consists of the evaluation of an expression, and sequential control is replaced by recursion.

There is no assignment statement. Values are communicated primarily through the use of parameters and return values. Without variables, loop statements don't have a purpose and so they also don't exist in pure functional languages.

Pure functional languages have no side-effects other than possibly reading some input from the user. Scheme is a pure functional language. In general, functional languages avoid or at least isolate code with side-effects. Even input and output operations in functional languages do not update the state of variables within a program.

What is amazing is that it has been proven that exactly the same things can be computed with functional languages as can be computed with imperative languages. This is known because a Turing machine, the theoretical basis for imperative programming and the design of the computer, have been proven equivalent in power to the Lambda Calculus, the basis for all functional programming languages.

You might be surprised by the number and types of languages that support functional programming. Of course, Standard ML was designed as a functional language from the ground up, but languages like C++, Java, and Python also support functional programming. While C++, Java, and Python are also object-oriented imperative languages, they all support functional programming as well. Functional programming does not depend so much on the language, but how you use the language. The rest of this chapter will introduce the functional style of programming. It all started with the lambda calculus, which is briefly considered next.

5.2 The Lambda Calculus

All functional programming languages are derived either directly or indirectly from the work of Alonzo Church and Stephen Kleene. The lambda calculus was defined by Church and Kleene in the 1930s, before computers existed. At the time, mathematicians were interested in formally expressing computation in some written form other than English or other informal language. The lambda calculus was designed as a way of expressing those things that can be computed. It is a very small, functional programming language. In the lambda calculus, a function is a mapping from the elements of a domain to the elements of a codomain given by a rule. Consider the function $cube(x) = x^3$. What is the value of the identifier *cube* in the definition $cube(x) = x^3$? Can this function be defined without giving it a name?

$\lambda x.x^3$ defines the function that maps each x in the domain to x^3. We can say that this definition or *lambda abstraction*, $\lambda x.x^3$, is the value bound to the identifier *cube*. We say that x^3 is the *body* of the lambda abstraction. Every lambda *abstraction* in lambda notation is a function of one identifier. However, lambda *expressions* may contain more than one identifier.

The expression $y^2 + x$ can be expressed as a lambda abstraction in one of two ways:

$$\lambda x.\lambda y.y^2 + x$$
$$\lambda y.\lambda x.y^2 + x$$

In the first lambda abstraction the x is the first parameter to be supplied to the expression. In the second lambda abstraction the y is the parameter to get a value first. In either case, the abstraction is often abbreviated by throwing out the extra λ. In abbreviated form the two abstractions would become $\lambda xy.y^2 + x$ and $\lambda yx.y^2 + x$.

5.2.1 Normal Form

To say the lambda calculus, or any language, has a normal form means that each expression that can be reduced has a simplest form. It means that we can reduce more complex expressions to simpler expressions in some mechanical way. The lambda calculus exhibits a property called *confluence*.

$$\begin{aligned}
&\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda xy.x)\\
&\Rightarrow \underline{(\lambda yz.(\lambda x.x)z(yz))(\lambda xy.x)}\\
&\Rightarrow \lambda z.\underline{(\lambda x.x)z}((\lambda xy.x)z)\\
&\Rightarrow \lambda z.z(\underline{(\lambda xy.x)z})\\
&\Rightarrow \lambda z.z(\lambda y.z)\Box
\end{aligned}$$

Fig. 5.2 Normal Order Reduction

Confluence means that one or more reduction strategies (or intermixing them) always leads to the same normal form of an expression, assuming the expression can be reduced by the reduction strategy. This property of confluence was proven in the Church–Rosser theorem.

Function application (i.e. calling a function) in lambda notation is written with a lambda abstraction followed by the value to call the abstraction with. Such a combination is called a *redex*.

To call $\lambda x.x^3$ with the value 2 for x we would write

$$(\lambda x.x^3)2$$

This combination of lambda abstraction and value is called a *redex*.

A redex is a lambda expression that may be reduced. Typically a lambda expression contains several redexes that may be chosen to be reduced. Function application is left-associative meaning that if more than one redex is available at the same level of parenthetical nesting, the left-most redex must be reduced first. If the left-most outer-most redex is always chosen for reduction first, the order of reduction is called normal order reduction. When a redex is reduced by applying the lambda calculus equivalent of function application it is called a β-reduction (pronounced beta-reduction).

The normal order reduction of $(\lambda xyz.xz(yz))(\lambda x.x)(\lambda xy.x)$ is given in Fig. 5.2. The redex to be β-reduced at each step is underlined.

Practice 5.2 Another reduction strategy is called applicative order reduction. Using this strategy, the left-most inner-most redex is always reduced first. Use this strategy to reduce the expression in Fig. 5.2. Be sure to parenthesize your expression first so you are sure that you left-associate redexes.
You can check your answer(s) in Section 5.26.2.

In practice problem 5.2 you should have reduced the lambda expression to the same reduced lambda expression derived from the normal order reduction in Fig. 5.2. If you didn't, you did something wrong. If you want more experience with reducing lambda expressions you may wish to consult a lambda expression interpreter. One excellent interpreter was written by Peter Sestoft and is available on the web. It is located at http://www.itu.dk/people/sestoft/lamreduce/. Be sure to read his help page to get familiar with the syntax required for entering lambda expressions in his interpreter. Also be aware that his interpreter does not understand math symbols like $+$. Instead, you can use a p to represent addition if needed. Sestoft's lambda calculus interpreter is for the pure lambda calculus without knowledge of Mathematics or any other language.

5.2.2 Problems with Applicative Order Reduction

Sometimes, applicative order reduction can lead to problems. For instance, consider the expression $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$.

Practice 5.3 Reduce the expression $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$ with both normal order and applicative order reduction. Don't spend too much time on this! *You can check your answer(s) in Section* 5.26.3.

This practice problem shows why the definition of confluence includes the phrase, *assuming the expression can be reduced by the reduction strategy*. Applicative order may not always result in the expression being reduced. No fear, if that happens we are free to use normal order reduction for a while since intermixing reduction strategies will not affect whether we arrive at the normal form for the expression or not.

5.3 Getting Started with Standard ML

Standard ML (or just SML) is a functional language based on Lisp which in turn is based on the lambda calculus. Important ML features are listed below.

- SML is higher-order supporting functions as first-class values.
- It is strongly typed like Pascal, but more powerful since it supports polymorphic type checking. With this strong type checking it is pretty infrequent that you need to debug your code!! What a great thing!!!
- Exception handling is built into Standard ML. It provides a safe environment for code development and execution. This means there are no traditional pointers in ML. Pointers are handled like references in Java.
- Since there are no traditional pointers, garbage collection is implemented in the ML system.
- Pattern-matching is provided for conveniently writing recursive functions.
- There are built-in advanced data structures like lists and recursive data structures.
- A library of commonly used functions and data structures is available called the *Basis Library*.

There are several implementations of Standard ML. Standard ML of New Jersey and Moscow ML are the most complete and certainly the most popular. There is also a SML.NET implementation that targets the Microsoft .NET run-time library and can be integrated with other .NET languages. There is an MLj implementation that targets the Java Virtual Machine. Poly/ML is another implementation that includes support for Windows programming. While many implementations exist, they all support the same definition of SML. If you write a Standard ML program that runs in one environment, it'll run on any other implementation as long as you are not using platform specific functions.

SML has been successfully used on a variety of large programming projects. It was used to implement the entire TCP protocol on the FOX Project at Carnegie Mellon. It has been used to implement server side scripting on web servers. It was originally designed as a language to write theorem provers and has been used extensively in this area. It has been used in hardware design and verification. It has also been used in programming languages research.

The rest of this chapter introduces SML. By the end of the chapter you should understand and be able to use many of the important features of the language. This text is based on the Standard ML of New Jersey implementation. You can download SML of New Jersey from smlnj.org. SML of New Jersey is available for most platforms so you should be able to find an implementation for your needs.

Once you've installed SML you can open a terminal window and start the interpreter. Typing *sml* at the command-line will start the interactive mode of the interpreter. Typing *ctl-d* will terminate the interpreter. You can type expressions and programs directly in at the interpreter's prompt or you can type them in a file and use that file within SML. To do this you type the word *use* as follows:

```
Standard ML of New Jersey v110.59
- use "myfile.txt";
```

SML will take whatever you have typed in the file and evaluate it just as if you had typed it directly into the interpreter.

The examples and practice problems in this chapter introduce SML. The following sections introduce important aspects of SML and ready the reader to write more complicated programs in the next chapter.

5.4 Expressions, Types, Structures, and Functions

Functional programming focuses on the evaluation of expressions. In SML you can evaluate expressions right in the intepreter. When evaluating an expression you will notice that type information is displayed along with the result of the expression evaluation. The dialog below contains some interactive expression evaluations in the SML interpreter.

In SML the identifier *it* is bound to the result of the last successfully evaluated expression. This is convenient if you want to use the result in a subsequent expression.

```
- 6;
val it = 6 : int
- 5*3;
val it = 15 : int
- ~1;
val it = ~1 : int
- 5.0 * 3.0;
val it = 15.0 : real
- true;
val it = true : bool
- 5 * 3.0;
Error: operator and operand don't agree
  operator domain: int * int
  operand:         int * real
  in expression:
5 * 3.0
-
```

Fig. 5.3 Interpreter Interaction

The last expression result can be referred to as *it* in the subsequent, interactively entered expression.

The interaction presented in Fig. 5.3 contains a negative one written as ˜ 1 in SML. While a little unconventional, ˜ is the unary negation operator in SML, distinguishing it from the binary subtraction operator.

SML has a very rigorous type system. In fact, the type system for SML has been proved sound. That means that any correctly typed program is guaranteed to be free of type errors. SML is statically typed like C++ and Java. That means that all type errors are detected at compile-time and not at run-time. Robin Milner proved this for Standard ML. ML is the only widely distributed language whose type system has been formally defined and proven type correct.

While being formally defined and rigorous, the type system of ML is remarkably flexible. It is polymorphic. We'll see what this means for us soon. Many of the types in ML are also implicitly expressed. In C++ and Java the type of every variable and function must be declared. You may notice in Fig. 5.3 that the programmer never entered any types for the expressions given there. In most cases Standard ML's type system frees the programmer from having to specify types in a program since they are mostly determined automatically.

You may have also noticed that there is a type error in Fig. 5.3. ML is polymorphic but it is also strongly typed. Since 5 is an integer in SML and 3.0 is a real, the two cannot be multiplied together. If you should have the need to multiply an integer and a real it can be done, but you must explicitly convert one of the types. The intepreter interaction below show some code to multiply an integer and a real, producing a real number.

```
- Real.fromInt(5) * 3.0;
val it = 15.0 : real
-
```

The integer 5 is converted to 5.0 by calling a function called fromInt in the structure called Real. A Structure in SML is a grouping of functions and types. A structure is like a module in Python or an include in C++. There are several structures that make up the *Basis Library* for Standard ML. The basis library is available in SML when the interpreter is started. The structures in the basis library include Bool, Int, Real, Char, String, and List. *Appendix B* or the website http://standardml.org/Basis contain descriptions of many of these structures.

A function in SML takes one or more arguments and returns a value. The signature of a function is the type of the function. In other words, a function's type is its signature. The signature of the function *fromInt* in the Real structure is

```
val fromInt : int -> real
```

This signature indicates that *fromInt* takes an *int* as an argument and returns a *real*. From the name of the function, and the fact that it is part of the Real structure, we can ascertain that it creates a *real* number from an *int*.

The type on the left side of the arrow (i.e. the ->) is the type of the arguments given to the function. The type on the right side of the arrow is the type of the value returned by the function. The *fromInt* function takes an *int* as an argument and returns a *real*.

Practice 5.4 Write expressions that compute the values described below. Consult the basis library in *Appendix B* as needed.

1. Divide the integer bound to x by 6.
2. Multiply the integer x and the real number y giving the closest integer as the result.
3. Divide the real number 6.3 into the real number bound to x.
4. Compute the remainder of dividing integer x by integer y.

You can check your answer(s) in Section 5.26.4.

5.5 Recursive Functions

Recursion is the way to get things done in a functional language. Recursion happens when a function calls itself. Because of the principle of referential transparency a function must never call itself with the same arguments. If it were to do that, then the function would do exactly what it did the last time, call itself with the same arguments, which would then.... Well, you get the picture!

To spare ourselves from this problem we insist on two things happening. First, every recursive function must have a base case. A base case is a simple subproblem that we are trying to solve that doesn't require recursion. We must write some code that checks for the simple problem and simply returns the answer in that case.

The second rule of recursive functions requires them to call themselves on some simpler or smaller subproblem. In some way each recursive call should take a step toward the base case of the problem. If each recursive call advances toward the base case then by the mathematical principle of induction we can conclude the function will work for all values on which the function is defined! The trick is not to think about this too hard. The recursive case is often referred to as the inductive case.

Writing functional programs is much more declarative than the prescriptive programming of assembly and imperative programming in languages like C++, Python, and Java. What this statement is really saying is that when writing recursive functions we think much less about how it works and more about the structure of the data. This leads to a few simple steps that can be applied to writing any recursive function. Memorize these steps and practice them and you can write any recursive function.

1. Decide what the function is named, what arguments are passed to it, and what the function should return.
2. At least one of the arguments must get smaller each time. Most of the time it is only one argument getting smaller. Decide which one that will be.
3. Write the function declaration, declaring the name, arguments types, and return type if necessary.
4. Write a base case for the argument that you decided will get smaller. Pick the smallest, simplest value that could be passed to the function and just return the result for that base case.
5. The next step is the crucial step. You don't write the next statement from left to right. You write from the inside out at this point.
6. Make a recursive call to the function with a smaller value. For instance, if it is a list you decided will get smaller, call the function with the tail of the list. If an integer is the argument getting smaller, call the function with the integer argument minus 1. Call the function with the required arguments and in particular with a smaller value for the argument you decided would get smaller at each step.
7. Now, here's a leap of faith. That call you made in the last step worked! It returned the result that you expected for the arguments it was given. Use that result in building the result for the original arguments passed to the function. At this step it may be helpful to try a concrete example. Assume the recursive call worked on the concrete example. What do you have to do with that result to get the result you wanted for the initial call? Write code that uses the result in building the final result for your concrete example. By considering a concrete example it will help you see what computation is required to get your final result.
8. That's it! Your function is complete and it will work if you stuck to these guidelines.

```
fun babsqrt(x,guess) =
  if Real.abs(x-guess*guess) <
     x/1000000.0 then
    guess
  else
    babsqrt(x,(guess + x/guess)/2.0);
```

Fig. 5.4 Square Root

To define a function in SML we write the keyword *fun* followed by a function name, parameters, an equal sign, and the body of the function. The syntax is quite similar to defining functions in other languages. The main difference is the body of the function. Instead of being a sequence of statements with variable assignment, the body of the function will be an expression.

One important expression in SML is the *if-then-else* expression. This is not an *if-then-else* statement. Instead, it's an *if-then-else* expression. An *if-then-else* expression gives one of two values and those values must be type compatible. The easiest way to understand *if-then-else* expressions is to see one in practice.

The Babylonian method of computing square root of a number, x, is to start with an arbitrary number as a *guess*. If $guess^2 = x$ we are done. If not, then let the next guess be $(guess + x/guess)/2.0$. To write this as a recursive function we must find a base case and be certain that our successive guesses will approach the base case. Since the Babylonian method of finding a square root is a well-known algorithm, we can be assured it will converge on the square root. The base case has to be written so that when we get close enough, we will be done. Let's let the *close enough* factor be one millionth of the original number.

The SML code in Fig. 5.4 implements this function. Looking at the code there are two things to observe. The base case comes first. If the guess is within one millionth of the right value then the function returns the guess as the square root. The other observation is the recursive call brings us closer to the solution.

Practice 5.5 $n!$ is called the factorial of n. It is defined recursively as $0! = 1$ and $n! = n * (n - 1)!$. Write this as a recursive function in SML.
You can check your answer(s) in Section 5.26.5.

Practice 5.6 The Fibonacci sequence is a sequence of numbers 0, 1, 1, 2, 3, 5, 8, 13, ... Subsequent numbers in the sequence are derived by adding the previous two numbers in the sequence together. This leads to a recursive definition of the Fibonacci sequence. What is the recursive definition of Fibonacci's sequence? HINT: The first number in the sequence can be thought of as the zeroeth element, then the first element is next and so on. So, *fib(0) = 0*. After arriving at the definition, write a recursive SML function to find the nth element of the sequence.
You can check your answer(s) in Section 5.26.6.

5.6 Characters, Strings, and Lists

SML has separate types for characters and strings. A character literal begins with a pound sign (i.e. #). The character is then surrounded by double quotes. So, the first character in the alphabet is represented as #"a" in SML. There are several functions available in the Char structure for testing and converting characters. The signature of the functions in the Char structure is given in *Appendix B*.

Strings in SML are not simply sequences of characters as they are in some languages. A string in SML is its own primitive type. There are functions for converting between strings and characters of course. You can consult *Appendix B* for a list of those functions. A string literal is text surrounded by double quotes. The backslash character (i.e. \) is an escape character in strings. This means to include a double quote in a string you can write " as part of the string. A `\n` is the newline character in a string and `\t` is the tab character as they are in many languages.

Perhaps the most powerful data structure in SML is the list. A list is polymorphic meaning that there are many list types in SML. However, the list functions all work on any type of list. Since it is impossible to determine all the types in SML (because programmers may define their own types), a list's type is parameterized by a type variable. A list's type is written as *'a list*. When the type of the list is known, the type variable *'a* is replaced by the type it represents. So, a list of integers has type *int list*. You may have figured this out already, but lists in SML must be homogeneous. This means all the elements of a list must have the same type. This is not like some languages, but there is a good reason for this restriction. Requiring lists to be homogeneous makes static checking of the types in SML possible and the type checker sound and complete.

A list is constructed in one of several ways. First, an empty list is represented as *nil* or by the empty list (i.e. *[]*). A list may be represented as a literal by putting a left bracket and a right bracket around the list contents, as in *[1,4,9,16]*. A list may also be constructed using the list constructor which is written *::*, and pronounced *cons*. In the functional language Lisp the same list construction operator is written *cons*

```
:: : 'a * 'a list -> 'a list
@ : 'a list * 'a list -> 'a list
hd : 'a list -> 'a
tl : 'a list -> 'a list
```

Fig. 5.5 Function Signatures

so it is called the *cons* operator by many functional programmers. The *cons* operator takes an element on the left side of it and a list on the right side and constructs a new list of its two arguments. A list may be constructed by concatenating two lists together. List concatenation is represented with the @ symbol. The following are all valid list constructions in SML.

- [1,4,9,16]
- 1 ::[4,9,16,25]
- #"a" ::#"b" ::[#"c"]
- 1 ::2 ::3 ::nil
- ["hello","how"]@["are","you"]

The third example works because the *::* constructor is right-associative. So the right-most constructor is applied first, then the one to its left, and so on. The signatures of the list constructor and some list functions are given in Fig. 5.5.

Practice 5.7 The following are NOT valid list constructions in SML. Why not? Can you fix them?

- #"a" ::["beautiful day"]
- "hi" ::"there"
- ["how","are"] ::"you"
- [1,2.0,3.5,4.2]
- 2@[3,4]
- [] ::3

You can check your answer(s) in Section 5.26.7.

You can select elements from a list using the hd and tl functions. The *hd* (pronounced *head*) of a list is the first element of the list. The *tl* is the tail or all the rest of the elements of the list. Calling the *hd* or *tl* functions on the empty list will result in an error. Using these two functions and recursion it is possible to access each element of a list. The code in Fig. 5.6 illustrates a function called implode that takes a list

```
fun implode(lst) =
  if lst = [] then ""
  else str(hd(lst))^implode(tl(lst))
```

Fig. 5.6 The Implode Function

```
fun length(x) =
    if null x then 0
    else 1+length(tl(x))
fun append(L1, L2) =
    if null L1 then L2
    else hd(L1)::append(tl(L1),L2)
```

Fig. 5.7 Two List Functions

of characters as an argument and returns a string comprised of those characters. So, *implode([#"H",#"e",#"l",#"l",#"o"])* would yield *"Hello"*.

When writing a recursive function the trick is to not think too hard about how it works. Think of the base case or cases and the recursive cases separately. So, in Fig. 5.6 the base case is when the list is empty (since a list is the parameter). When the list is empty, the string the function should return should also be empty.

The recursive case is when when the list is not empty. In that case, there is at least one element in the list. If that is true then we can call *hd* to get the first element and *tl* to get the rest of the list. The head of the list is a character and must be converted to a string. The rest of the list is converted to a string by calling some function that will convert a list to a string. This function is called *implode*! We can just assume it will work. That is the nature of recursion. The trick, if there is one, is to trust that recursion will work. Later, we will explore exactly why we can trust recursion.

Practice 5.8 Write a function called *explode* that will take a string as an argument and return a list of characters in the string. So, *explode("hi")* would yield *[#"h",#"i"]*. HINT: How do you get the first character of a string?
You can check your answer(s) in Section 5.26.8.

The code in Fig. 5.7 contains a couple more examples of list functions. The length function counts the number of elements in a list. It must be a list because the *tl* function is used. The append function appends two lists by taking each element from the first list and consing it onto the result of appending the rest of the first list to the second list.

Practice 5.9 Use the append function to write reverse. The reverse function reverses the elements of a list. Its signature is

```
reverse = fn: 'a list -> 'a list
```

You can check your answer(s) in Section 5.26.9.

5.7 Pattern Matching

Frequently, recursive functions rely on several recursive and several base cases. SML includes a nice facility for handling these different cases in a recursive definition by allowing pattern matching of the arguments to a function. Pattern matching works with literal values like 0, the empty string, and the empty list. Generally, you can use pattern matching if you would normally use equality to compare values. Real numbers are not equality types. The *real* type only approximates real numbers. The code in Fig. 5.4 shows how two real numbers are compared for equality.

You can also use constructors in patterns. So the list constructor *::* works in patterns as well. Functions like the append function (i.e. the infix @) and string concatenation (i.e. ^) don't work in patterns. These functions are not constructors of values and cannot be efficiently or deterministically matched to patterns of arguments.

Append can be written using pattern-matching as shown in Fig. 5.8. The extra parens around the recursive call to append are needed because the *::* constructor has higher precedence than function application.

Practice 5.10 Rewrite reverse using pattern-matching.
You can check your answer(s) in Section 5.26.10.

```
fun append(nil,L2) = L2
  | append(h::t,L2) = h::(append(t,L2))
```

Fig. 5.8 Pattern Matching

5.8 Tuples

A tuple type is a cross product of types. A two-tuple is a cross product of two types, a three-tuple is a cross product of three types, and so on. (5,6) is a two-tuple of *int* * *int*. The three tuple *(5,6,"hi")* is of type *int* * *int* * *string*.

You might have noticed the signature of some of the functions in this chapter. For instance, consider the signature of the append function. Its signature is

```
val append : 'a list * 'a list -> 'a list
```

This indicates it's a function that takes as its argument an *'a list* * *'a list* tuple. In fact, every function takes a single argument and returns a single value. The sole argument might be a tuple of one or more values, but every function takes a single argument as a parameter. The return value of a function may also be a tuple.

In many other languages we think of writing function application as the function followed by a left paren, followed by comma separated arguments, followed by a right paren. In Standard ML (and most functional languages) function application is written as a function name followed by the value to which the function is applied. This is just like function application in the lambda calculus. So, we can think of calling a function with zero or more values, but in reality every function in ML is passed on argument, which may be a tuple. In Standard ML rather than writing

```
append([1,2],[3])
```

it is more appropriate to write

```
append ([1,2],[3])
```

because function application is a function name followed by the value to which it will be applied. In this case append is applied to a tuple of *'a list* * *'a list*.

5.9 Let Expressions and Scope

Let expressions are simply syntax for binding a value to an identifier to later be used in an expression. They are useful when you want to document your code by assigning a meaningful name to a value. They can also be useful when you need the same value more than once in a function definition. Rather than calling a function twice to get the same value, you can call it once and bind the value to an identifier. Then the identifier can be used as many times as the value is needed. This is more efficient than calling a function multiple times with the same arguments.

Consider a function that computes the sum of the first *n* integers as shown in Fig. 5.9. Let expressions define identifiers that are local to functions. The identifier called *sum* in Fig. 5.9 is not visible outside the *sumupto* function definition. We say the scope of *sum* is the body of the let expression (i.e. the expression given between the *in* and *end* keywords). Let expressions allow us to declare identifiers with limited scope.

```
fun sumupto(0) = 0
  | sumupto(n) =
    let val sum = sumupto(n-1)
    in
      n + sum
    end
```

Fig. 5.9 Let Expression

Limiting scope is an important aspect of any language. Function definitions also limit scope in SML and most languages. The formal parameters of a function definition are not visible beyond the body of the function.

Binding values to identifiers should not be confused with variable assignment. A binding of a value to an identifier is a one time operation. The identifier's value cannot be updated like a variable. A practice problem will help to illustrate this.

Practice 5.11 What is the value of x at the various numbered points within the following expression? Be careful, it's not what you think it might be if you are relying on your imperative understanding of code.

```
let val x = 10 in
    (* 1. Value of x here? *)
    let val x = x+1
    in
        (* 2. Value of x here? *)
        x
    end;
    (* 3. Value of x here? *)
    x
end
```

You can check your answer(s) in Section 5.26.11.

Bindings are not the same as variables. Bindings are made once and only once and cannot be updated. Variables are meant to be updated as code progresses. Bindings are an association between a value and an identifier that is not updated.

SML and many modern languages use static or lexical scope rules. This means you can determine the scope of a variable by looking at the structure of the program without considering its execution. The word lexical refers to the written word and lexical or static scope refers to determining scope by looking at how the code is written and not the execution of the code. Originally, LISP used dynamic scope rules. To determine dynamic scope you must look at the bindings that were active when the code being executed was called. The difference between dynamic and

```
let fun a() =
        let val x = 1
            fun b() = x
        in
          b
        end
    val x = 2
    val c = a()
in
  c()
end
```

Fig. 5.10 Scope

static scope can be seen when functions may be nested in a language and may also be passed as parameters or returned as function results.

The difference between dynamic and static scope can be observed in the program in Fig. 5.10. In this program the function *a*, when called, declares a local binding of *x* to 1 and returns the function *b*. When *c*, the result of calling *a*, is called it returns a 1, the value of *x* in the environment where *b* was defined, not a 2. This result is what most people expect to happen. It is static or lexical scope. The correct value of *x* does not depend on the value of *x* when it was called, but the value where the function *b* was written.

While static scope is used by many programming languages including Standard ML, Python, Lisp, and Scheme, it is not used by all languages. The Emacs version of Lisp uses dynamic scope and if the equivalent Lisp program for the code in Fig. 5.10 is evaluated in Emacs Lisp it will return a value of 2.

It is actually harder to implement static scope than dynamic scope. In dynamically scoped languages when a function is returned as a value the return value can include a pointer to the code of the function. When the function *b* from Fig. 5.10 is executed in a dynamically scoped language, it simply looks in the current environment for the value of *x*. To implement static scope, more than a pointer to the code is needed. A pointer to the current environment is needed which contains the binding of *x* to the value at the time the function was defined. This is needed so when the function *b* is evaluated, the right *x* binding can be found. The combination of a pointer to a function's code and its environment is called a *closure*. Closures are used to represent function values in statically scoped languages where functions may be returned as results and nested functions may be defined. Chapters 3 and 4 introduced closures and Fig. 5.10 provides an example in Standard ML showing why they are necesssary for statically scoped languages.

5.10 Datatypes

The word datatype is often loosely used in computer science. In ML, a datatype is a special kind of type. A datatype is a tagged structure that can be recursively defined. This type is powerful in that you can define enumerated types with it and you can define recursive data structures like lists and trees.

Datatypes are user-defined types and are generally recursively defined. There are infinitely many datatypes in Standard ML. Defining a datatype is like creating a class in C++ without any methods and only public data. In C/C++ we can create an enumerated type by writing the declaration found in Fig. 5.11. This defines a type called TokenType of eleven values: *identifier* is 0, *keyword* is 1, *number* is 2, etc. You can declare a variable of this type as follows.

```
TokenType t = keyword;
```

However, until C++11 there was nothing preventing you from executing the statement

```
t = 1; //this is the keyword value.
```

In this example, even though *t* is of type *TokenType*, it could be assigned an integer with compilers prior to C++11. This is because the TokenType type was just another name for the integer type in C++ prior the C++11. Assigning *t* to 1 didn't bother C++ in the least. In fact, assigning *t* to 99 wouldn't bother C++ either prior to C++11. In Standard ML, and now in C++, we can't use integers and datatypes (or enums) interchangeably.

```
- datatype TokenType = Identifier | Keyword | Number |
    Add | Sub | Times | Divide | LParen | RParen | EOF |
    Unrecognized;
datatype TokenType = Identifier | Keyword | Number | ...
- val x = Keyword;
x = Keyword : TokenType
```

Datatypes allow programmers to define their own types. Normally, a datatype includes other information. Datatypes are used to represent structured data of some sort. By adding the keyword *of*, a datatype value can include a tuple of other types as

```
enum TokenType {
   identifier,keyword,
   number,add,sub,times,
   divide,lparen,
   rparen,eof,unrecognized
};
```

Fig. 5.11 C++ Enum Type

```
datatype
  AST = add' of AST * AST
      | sub' of AST * AST
      | prod' of AST * AST
      | div' of AST * AST
      | negate' of AST
      | integer' of int
      | store' of AST
      | recall';
```

Fig. 5.12 An AST Datatype

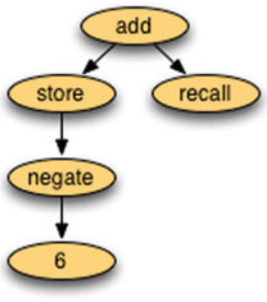


Fig. 5.13 An AST in SML

part of its definition. A datatype can represent any kind of recursive data structure. That includes lists, trees, and other structures that are related to lists and trees. In Fig. 5.12 we have a tree definition with a mix of unary and binary nodes.

Datatypes allow a programmer to write a recursive function that can traverse the data given to it. Functions can use pattern matching to handle each case in a datatype with a pattern match in the function.

In the datatype given in Fig. 5.12 the *add'* value can be thought of as a node in an *AST* that has two children, each of which are *ASTs*. The datatype is recursive because it is defined in terms of itself. The code in Fig. 5.12 is the entire definition of abstract syntax trees for expressions in a calculator language. Store nodes in the tree store their value in the one memory location of the calculator. Recall nodes recall the memory location of the calculator. The *negate'* node represents unary negation of the value we get when evaluating its child. So ~6 is a valid expression if we let the tilde sign represent unary negation as it does in Standard ML.

The abstract syntax tree for ~6S+R is drawn graphically in Fig. 5.13. The value add'(store'(negate'(integer'(6))), recall') is the SML way of representing the AST shown in Fig. 5.13. A function can be written to evaluate such an abstract syntax tree based on the patterns in a value like this and this is done later in the chapter.

```
fun evaluate(add'(e1,e2),min) =
    let val (r1,mout1)= evaluate(e1,min)
        val (r2,mout) = evaluate(e2,mout1)
    in
      (r1+r2,mout)
    end

  | evaluate(sub'(e1,e2),min) =
    let val (r1,mout1)= evaluate(e1,min)
        val (r2,mout) = evaluate(e2,mout1)
    in
      (r1-r2,mout)
    end
```

Fig. 5.14 Pattern Matching Function Results

You can use pattern matching on datatypes. For instance, to evaluate an expression tree you can write a recursive function using pattern-matching. Each pattern that is matched in such a function corresponds to processing one node in the tree. Each subtree can be processed by a recursive call to the same function. In Fig. 5.14, the parameter *min* is the value of the memory before evaluating the given node in the abstract syntax tree. The value *mout* is the value of memory after evaluating the node in the abstract syntax tree.

This example code in Fig. 5.14 illustrates how to use pattern-matching with datatypes and patterns in a *let* construct. This is one way to write the evaluate function to evaluate the abstract syntax trees defined in Fig. 5.12. *mout1* is the value of memory after evaluating *e1*. This is passed to evaluating *e2* as the value of the memory before evaluating *e2*. The value of memory after evaluating *e2* is the value of memory after evaluating the sum/difference of the two expressions. This pattern of passing the memory through the evaluation of the tree is called *single-threading* the memory in the computation.

Practice 5.12 Define a datatype for integer lists. A list is constructed of a head and a tail. Sometimes this constructor is called *cons*. The empty list is also a list and is usually called *nil*. However, in this practice problem, to distinguish from the built-in *nil* you could call it *nil'*.
You can check your answer(s) in Section 5.26.12.

Practice 5.13 Write a function called *maxIntList* that returns the maximum integer found in one of the lists you just defined in practice problem 5.12. You can consult *Appendix B* for help with finding the max of two integers.
You can check your answer(s) in Section 5.26.13.

5.11 Parameter Passing in Standard ML

The types of data in Standard ML include integers, reals, characters, strings, tuples, lists, and the user-defined datatypes presented in the last section. If you look at these types in this chapter and in *Appendix B* you may notice that there are no functions that modify the existing data. The substring function defined on strings returns a new string. In fact most functions on the types of data available in Standard ML return a new value without mutating the arguments passed to them. Not all data in Standard ML is immutable, but most of it is.

There is one type of data that is mutable in Standard ML. A reference is a reference to a value of a determined type. References may be mutated to enable the programmer to program using the imperative style of programming. References are discussed in more detail later in this chapter. The array type in Standard ML is a list of references so by arrays are generally considered mutable data types as well, but only because arrays are lists of references.

The absence of mutable data, except for references, has some impact on the implementation of the language. Values are passed by reference in Standard ML. However, the only time that matters is when a reference is passed as a parameter or one of the few mutable types of objects is passed to a function. Otherwise, the immutability of all data means that how data is passed to a function is irrelevant. This is nice for programmers as they don't have to be concerned about which functions mutate data and which construct new data values. For most practical purposes, there is only one operation that mutates data, the assignment operator (i.e. :=) and the only data it can mutate is a reference. In addition, because most data is immutable and passed by reference, parameters are passed efficiently in ML.

5.12 Efficiency of Recursion

Once you get used to it, writing recursive functions isn't too hard. In fact, it can be easier than writing iterative solutions. But, just because you find a recursive solution to a problem, doesn't mean it's an effficient solution to a problem. Consider the Fibonacci numbers. The recursive definition leads to a very straightforward recursive solution. However, as it turns out, the simple recursive solution is anything but efficient. In fact, given the definition in Fig. 5.15, fib(42) took six seconds to compute

```
fun fib(0) = 0
  | fib(1) = 1
  | fib(n) = fib(n-1) + fib(n-2)
```

Fig. 5.15 The Fib Function

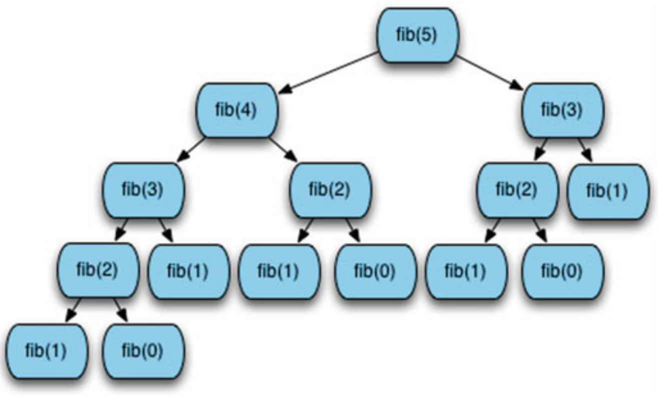


Fig. 5.16 Calls to Calculate fib(5)

on a 2.66 GHz MacBook Pro with 8 GB of RAM. Fib(43) took a third longer, jumping to nine seconds.

The Fibonacci numbers can be computed with the function definition given in Fig. 5.15. This is a very inefficient way of calculating the Fibonacci numbers. The number of calls to fib increases exponentially with the size of n. This can be seen by looking at a tree of the calls to fib as in Fig. 5.16. The number of calls required to calculate *fib(5)* is 15. If we were to enumerate the calls required to calculate *fib(6)* it would be everything in the *fib(5)* call tree plus the number of nodes in the *fib(4)* call tree, $15+9=25$. The number of calls grows exponentially.

Practice 5.14 One way of proving that the *fib* function in Fig. 5.15 is exponential is to show that the number of calls for *fib(n)* is bounded by two exponential functions. In other words, there is an exponential function of n that will always return less than the number of calls required to compute *fib(n)* and there is another exponential function that always returns greater than the number of required calls to compute *fib(n)* for some choice of starting n and all values greater than it. If the number of calls to compute *fib(n)* lies in between then the *fib* function must have exponential complexity. Find two exponential functions of the form c^m that bound the number of calls required to compute *fib(n)*.
You can check your answer(s) in Section 5.26.14.

```
fun fib(n) =
  let fun fibhelper(count,current,previous) =
      if count = n then previous
      else fibhelper(count+1,previous+current,current)
  in
      fibhelper(0,1,0)
  end
```

Fig. 5.17 An Efficient Fib Function

From this analysis you have probably noticed that there is a lot of the same work being done over and over again. It may be possible to eliminate a lot of this work if we are smarter about the way we write the Fibonacci function. In fact it is. The key to this efficient version of *fib* is to recognize that we can get the next value in the sequence by adding together the previous two values. If we just carry along two values, the current and the next value in the sequence, we can compute each Fibonacci number with just one call. The code in Fig. 5.17 demonstrates how to do this. With the new function, computation of fib(43) is instantaneous.

Using a helper function may lead to a better implementation in some situations. In the case of the *fib* function, the *fibhelper* function turns an exponentially complex function into a linear time function. The code in Fig. 5.17 uses a helper function that is private to the *fib* function because we don't want other programmers to call the *fibhelper* function directly. It is meant to be used by the *fib* function. We also wouldn't want to have to remember how to call the *fibhelper* function each time we called it. By hiding it in the *fib* function we can expose the same interface we had with the original implementation, but implement a much more efficient function.

The helper function uses a pattern called an accumulator pattern. The helper function makes use of an accumulator to reduce the amount of work that is done. The work is reduced because the function keeps track of the last two values computed by the helper function to aid in computing the next number.

Practice 5.15 Consider the reverse function from practice problem 5.10. The *append* function is called *n* times, where *n* is the length of the list. How many cons operations happen each time append is called? What is the overall complexity of the reverse function?
You can check your answer(s) in Section 5.26.15.

5.13 Tail Recursion

One criticism of functional programming centers on the heavy use of recursion that is seen by some critics as overly inefficient. The problem is related to the use of caches in modern processors. Depending on the block size of an instruction cache, the code surrounding the currently executing code may be readily available in the cache. However, when the instruction stream is interrupted by a call to a function, even the same function, the cache may not contain the correct instructions. Retrieving instructions from memory is much slower than finding them in the cache. However, cache sizes continue to increase and even imperative languages like C++ and Java encourage many calls to small functions or methods given their object-oriented nature. So, the argument in favor of fewer function calls has certainly diminished in recent years.

It is still the case that a function call takes longer than executing a simple loop. When a function call is made, extra instructions are executed to create a new activation record. In addition, in pipelined processors the pipeline is disrupted by function calls. Standard ML of New Jersey, Scheme, and some other functional languages have a mechanism where they optimize certain recursive functions by reducing the storage on the run-time stack and eliminating calls. In certain cases, recursive calls can be automatically transformed to code that can be executed using jump or branch instructions. For this optimization to be possible, the recursive function must be tail recursive. A tail recursive function is a function where the very last operation of the function is the recursive call to itself.

The *factorial* function is presented in Fig. 5.18. Is factorial tail recursive? The answer is no. Tail recursion happens when the very last thing done in a recursive function is a call to itself. The last thing done in Fig. 5.18 is the multiplication.

When factorial 6 is invoked, activation records are needed for seven invocations of the function, namely factorial 6 through factorial 0. Without each of these stack frames, the local values of n, n=6 through n=0, will be lost so that the multiplication at the end can not be carried out correctly.

At its deepest level of recursion all the information in the expression,

$$(6 * (5 * (4 * (3 * (2 * (1 * (factorial 0)))))))$$

is stored in the run-time execution stack.

```
fun factorial 0 = 1
  | factorial n = n * factorial (n-1);
```

Fig. 5.18 Factorial

```
fun factorial n =
    let fun tailfac(0,prod) = prod
          | tailfac(n,prod) = tailfac(n-1,prod*n)
    in
      tailfac(n,1)
    end
```

Fig. 5.19 Tail Recursive Factorial

Practice 5.16 Show the run-time execution stack at the point that factorial 0 is executing when the original call was factorial 6.
You can check your answer(s) in Section 5.26.16.

The *factorial* function can be written to be tail recursive. The solution is to use a technique similar to the *fib* function improvement made in Fig. 5.17. An accumulator is added to the function definition. An accumulator is an extra parameter that can be used to accumulate a value, much the way you would accumulate a value in a loop. The accumulator value is initially given the identity of the operation used to accumulate the value. In Fig. 5.19 the operation is multiplication. The identity provided as the initial value is 1.

The function presented in Fig. 5.19 is the tail recursive version of the *factorial* function. The tail recursive function is the *tailfac* helper function. Note that although *tailfac* is recursive, there is no need to save it's local environment when it calls itself since no computation remains after the call. The result of the recursive call is simply passed on as the result of the current function call. A function is tail recursive if its recursive call is the last action that occurs during any particular invocation of the function.

Practice 5.17 Use the accumulator pattern to devise a more efficient reverse function. The append function is not used in the efficient reverse function. HINT: What are we trying to accumulate? What is the identity of that operation?
You can check your answer(s) in Section 5.26.17.

5.14 Currying

A binary function, for example, + or @, takes both of its arguments at the same time. a+b will evaluate both *a* and *b* so that values can be passed to the addition operation. There can be an advantage in having a binary function take its arguments one at a time. Such a function is called *curried* after Haskell Curry. ML functions take their parameters one at a time because all functions take exactly one argument. A curried function takes one argument as well. However, that function of one parameter may in turn return a function that takes a single argument. This is probably best illustrated with an example. Here is a function that takes a pair of arguments as its input via a single tuple.

```
- fun plus(a:int,b) = a+b;
val plus = fn : int * int -> int
```

The function *plus* takes one argument that just happens to be a tuple. Calling the function means providing it a single tuple.

```
- plus (5,8);
val it = 13 : int
```

ML functions can be defined with what looks like more than one parameter:

```
- fun cplus (a:int) b = a+b;
val cplus = fn : int -> (int -> int )
```

Observe the signature of the function *cplus*. It appears to take two arguments, but takes them one at a time. Actually, *cplus* takes only one argument. The *cplus* function returns a function that takes the second argument. The second function has no name.

```
- cplus 5 8;
val it = 13 : int
```

Function application is left associative. The parens below show the order of operations.

```
- (cplus 5) 8;
val it = 13 : int
```

The result of *(cplus 5)* is a function that adds *5* to its argument.

```
- cplus 5;
val it = fn : int -> int
```

We can give this function a name.

```
- val add5 = cplus 5;
val add5 = fn : int -> int
- add5 8;
val it = 13 : int
```

The *add5* function adds 5 to whatever might be passed to it.

Practice 5.18 Write a function that given an uncurried function of two arguments will return a curried form of the function so that it takes its arguments one at a time.

Write a function that given a curried function that takes two arguments one at a time will return an uncurried version of the given function.
You can check your answer(s) in Section 5.26.18.

Curried functions allow partial evaluation, a very interesting topic in functional languages, but beyond the scope of this text. It should be noted that Standard ML of New Jersey uses curried functions extensively in its implementation. *Appendix B* contains many functions whose signatures reflect that they are curried.

5.15 Anonymous Functions

The beginning of this chapter describes the lambda calculus. In that section we learned that functions can be characterized as first class objects. Functions can be represented by a lambda abstraction and don't have to be assigned a name. This is also true in SML. Functions in SML don't need names. The anonymous function $\lambda xy.y^2{+}x$ can be represented in ML as

```
fn x => fn y => y*y + x;
```

The anonymous function can be applied to a value in the same way a named function is applied to a value. Function application is always the function first, followed by the value.

```
- (fn x => fn y => y*y + x) 3 4;
val it = 19 : int
```

We can define a function by binding a lambda abstraction to an identifier:

```
- val f = fn x => fn y => y*y + x;
val f = fn: int -> int -> int
- f 3 4;
val it = 19 : int
```

This mechanism provides an alternative form for defining functions as long as they are not recursive; in a *val* declaration, the identifier being defined is not visible in the expression on the right side of the arrow. For recursive definitions a *val rec* expression is required. To define a recursive function using the anonymous function form you must use *val rec* to declare it.

```
- val rec fac = fn n => if n=0 then 1 else n*fac(n-1);
val fac = fn: int -> int
- fac 7;
val it = 5040:int
```

This *val rec* definition of a function is the way all functions are defined in SML. The functional form used when the keyword *fun* is used to define a function is translated into *val rec* form. The *fun* form of function definition is called *syntactic sugar*. Syntactic sugar refers to another way of writing something that gets treated the same way in either case. Usually *sugared* forms are the *nicer* way to write something.

5.16 Higher-Order Functions

The unique feature of functional languages is that functions are treated as first-class objects with the same rights as other objects, namely to be stored in data structures, to be passed as a parameter, and to be returned as function results. Functions can be bound to identifiers using the keywords fun, *val*, and *val rec* and may also be stored in structures. These are examples of functions being treated as values.

```
- val fnlist = [fn (n) => 2*n, abs, ~, fn (n) => n*n];
val fnlist = [fn,fn,fn,fn] : (int -> int) list
```

Notice each of these functions takes an int and returns an int. An ML function can be defined to apply each of these functions to a number. The *construction* function applies a list of functions to a value.

```
- fun construction  nil n = nil
    | construction (h::t) n = (h n)::(construction t n);
val construction = fn : ('a -> 'b) list -> 'a -> 'b list
- construction [op +, op *, fn (x,y) => x - y] (4,5);
val it = [9,20,~1] : int list
```

Construction is based on a functional form found in FP, an early functional programming language developed by John Backus. It illustrates the possibility of passing functions as arguments. Since functions are first-class objects in ML, they may be stored in any sort of structure. It is possible to imagine an application for a stack of functions or even a tree of functions.

A function is called higher-order if it takes a function as a parameter or returns a function as its result. Higher-order functions are sometimes called functional forms since they allow the construction of new functions from already defined functions.

The usefulness of functional programming comes from the use of functional forms that allow the development of complex functions from simple functions using abstract patterns. The *construction* function is one of these abstract patterns of computation. These functional forms, or patterns of computation, appear over and over again in programs. Programmers have recognized these patterns and have abstracted out the details to arrive at several commonly used higher-order functions. The next sections introduce several of these higher-order functions.

5.16.1 Composition

Composing two functions is a naturally higher-order operation that you have probably used in algrebra. Have you ever written something like f(g(x))? This operation can be expressed in ML. In fact, ML has a built-in operator called *o* which represents composition. This example code demonstrates how composition can be written and used.

```
- fun compose f g x = f (g x);
val compose = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
- fun add1 n = n+1;
val add1 = fn : int -> int
- fun sqr n:int = n*n;
val sqr = fn : int -> int
- val incsqr = compose add1 sqr;
val incsqr = fn : int -> int
- val sqrinc = compose sqr add1;
val sqrinc = fn : int -> int
```

Observe that these two functions, *incsqr* and *sqrinc*, are defined without the use of parameters.

```
- incsqr 5;
val it = 26 : int
- sqrinc 5;
val it = 36 : int
```

ML has a predefined infix function *o* that composes functions. Note that *o* is uncurried.

```
- op o;
val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
- val incsqr = add1 o sqr;
val incsqr = fn : int -> int
- incsqr 5;
val it = 26 : int
- val sqrinc = op o(sqr,add1);
val sqrinc = fn : int -> int
- sqrinc 5;
val it = 36 : int
```

5.16.2 Map

In SML, applying a function to every element in a list is called *map* and is predefined. It takes a unary function and a list as arguments and applies the function to each element of the list returning the list of results.

```
- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
- map add1 [1,2,3];
val it = [2,3,4] : int list
- map (fn n => n*n - 1) [1,2,3,4,5];
val it = [0,3,8,15,24] : int list
- map (fn ls => "a"::ls) [["a","b"],["c"],["d","e","f"]];
val it = [["a","a","b"],["a","c"],["a","d","e","f"]] :
            string list list
- map real [1,2,3,4,5];
val it = [1.0,2.0,3.0,4.0,5.0] : real list
```

```
fun map f nil = nil
  | map f (h::t) = (f h)::(map f t);
```

Fig. 5.20 The Map Function

The map function is predefined in the List structure, but is provided in Fig. 5.20 for your reference.

Practice 5.19 Describe the behavior (signatures and output) of these functions:

- map (map add1)
- (map map)

Invoking *(map map)* causes the type inference system of SML to report

```
language
stdIn:12.27-13.7 Warning: type vars not generalized
    because of value restriction are instantiated to
    dummy types (X1,X2,...)
```

This warning message is OK. It is telling you that to complete the type inference for this expression, SML had to instantiate a type variable to a dummy variable. When more type information is available, SML would not need to do this. The warning message only applies to the specific case where you created a function by invoking *(map map)*. In the presence of more information the type inference system will interpret the type correctly without any dummy variables.
You can check your answer(s) in Section 5.26.19.

5.16.3 Reduce or Foldright

Higher-order functions are developed by abstracting common patterns from programs. For example, consider the functions that find the sum or the product of a list of integers. In this pattern the results of the previous invocation of the function are used in a binary operation with the next value to be used in the computation.

In other words, to add up a list of values you start with either the first or last element of the list and then add it together with the value next to it. Then you add the result of that computation to the next value in the list and so on. When we start with the end of the list and work our way backwards through the list the operation is sometimes called foldr (i.e. foldright) or reduce.

```
- fun sum nil = 0
    | sum ((h:int)::t) = h + sum t;

val sum = fn : int list -> int
- sum [1,2,3,4,5];
val it = 15 : int

- fun product nil = 1
    | product ((h:int)::t) = h * product t;

val product = fn : int list -> int
- product [1,2,3,4,5];
val it = 120 : int
```

Each of these functions has the same pattern. If we abstract the common pattern as a higher-order function we arrive at a common higher-order function called *foldr*. *foldr* is an abbreviation for foldright. The *foldr* function keeps applying its function to the result and the next item in the list.

```
- fun foldr f init nil = init
    | foldr f init (h::t) = f(h, foldr f init t);

val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- foldr op + 0 [1,2,3,4,5];
val it = 15 : int
- foldr op * 1 [1,2,3,4,5];
val it = 120 : int
```

Now *sum* and *product* can be defined in terms of *reduce*.

```
- val sumlist = List.foldr (op +) 0;
val sumlist = fn : int list -> int
- val mullist = List.foldr op * 1;
val mullist = fn : int list -> int
- sumlist [1,2,3,4,5];
val it = 15 : int
- mullist [1,2,3,4,5];
val it = 120 : int
```

SML includes two predefined functions that reduce a list, *foldr* and *foldl* which stands for foldleft. They behave slightly differently.

```
- List.foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- List.foldl;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
- fun abdiff (m,n:int) = abs(m-n);
val abdiff = fn : int * int -> int
- foldr abdiff 0 [1,2,3,4,5];
val it = 1 : int
```

```
- foldl abdiff 0 [1,2,3,4,5];
val it = 3 : int
```

Practice 5.20 How does *foldl* differ from *foldr*? Determine the difference by looking at the example code in this section. Then, describe the result of these functions invocations.

- foldr op :: nil ls
- foldr op @ nil ls

You can check your answer(s) in Section 5.26.20.

5.16.4 Filter

A predicate function is a function that takes a value and returns true or false depending on the value. By passing a predicate function, it is possible to filter in only those elements from a list that satisfy the predicate. This is a commonly used higher-order function called *filter*. If we had to write filter ourselves, this is how it would be written. This example also shows how it might be used.

```
- fun filter bfun nil = nil
    | filter bfun (h::t) = if bfun h then h::filter bfun t
                           else filter bfun t;

val it = fn : ('a -> bool) -> 'a list -> 'a list
- even;
val it = fn : int -> bool
- filter even [1,2,3,4,5,6];
val it = [2,4,6] : int list
- filter (fn n => n > 3) [1,2,3,4,5,6];
val it = [4,5,6] : int list
```

Practice 5.21 Use filter to select numbers from a list that are

- divisible by 7
- greater than 10 or equal to zero

You can check your answer(s) in Section 5.26.21.

5.17 Continuation Passing Style

Continuation Passing Style (or CPS) is a way of writing functional programs where control is made explicit. In other words, the continuation represents the remaining work to be done. This style of writing code is interesting because the style is used in the SML compiler. To understand cps it's best to look at an example. Let's consider the len function for computing the length of a list.

```
- fun len nil = 0
    | len (h::t) = 1+(len t);
val len = fn : 'a list -> int
```

To transform this to cps form we represent the rest of the computation explicitly as a parameter called k. In this way, whenever we need the continuation of the calculation, we can just write the identifier k. Here's the cps form of len and an example of calling it.

```
- fun cpslen nil k = k 0
    | cpslen (h::t) k = cpslen t (fn v => (k (1 + v)));
val cpslen = fn : 'a list -> (int -> 'b) -> 'b
- cpslen [1,2,3] (fn v => v);
val it = 3 : int
```

Practice 5.22 Trace the execution of cpslen to see how it works and how the continuation is used.
You can check your answer(s) in Section 5.26.22.

Notice that the recursive call to cpslen is the last thing that is done. This function is tail recursive. However, tail recursion elimination cannot be applied because the function returns a function and recursively calls itself with a function as a parameter. CPS is still important because it can be optimized by a compiler. In addition, since control flow is explicit (passed around as k), function calls can be implemented with jumps and many of the jumps can be eliminated if the code is organized in the right way.

Eliminating calls and jumps is important since calls have the effect of interrupting pipelines in RISC processors. Since functional languages make lots of calls, one of the criticisms of functional languages is that they were inefficient. With the optimization of CPS functions, functional languages get closer to being as efficient as imperative languages. In addition, as cache sizes and processor speeds increase the performance difference becomes less and less of an issue.

Practice 5.23 Write a function called *depth* that prints the longest path in a binary tree. First create the datatype for a binary tree. You can use the *Int.max* function in your solution, which returns the maximum of two integers.
First write a non-cps *depth* function, then write a cps *cpsdepth* function.
You can check your answer(s) in Section 5.26.23.

5.18 Input and Output

SML contains a TextIO structure as part of the basis library. The signature of the functions in the TextIO structure is given in *Appendix B*. It is possible to read and write strings to streams using this library of functions. The usual standard input, standard output, and standard error streams are predefined. Here is an example of reading a string from the keyboard. Explode is used on the string to show the vector type is really the string type. It also shows how to print something to a stream.

```
- val s = TextIO.input(TextIO.stdIn);
hi there
val s = "hi there\n" : vector
- explode(s);
val it = [#"h",#"i",#" ",#"t",#"h",#"e",
            #"r",#"e",#"\n"] : char list
- TextIO.output(TextIO.stdOut,s^"How are you!\n");
hi there
How are you!
- val it = () : unit
```

Since streams can be directed to files, the screen, or across the network, there really isn't much more to input and output in SML. Of course if you are opening your own stream it should be closed when you are done with it. Program termination will also close any open streams.

There are some TextIO functions that may or may not return a value. In these cases an *option* is returned. An *option* is a value that is either *NONE* or *SOME value*. An option is SML's way of dealing with functions that may or may not succeed. Functions must always return a value or end with an exception. To prevent the exception handling mechanism from being used for input operations that may or may not succeed, this idea of an option was created. Options fit nicely into the strong typing that SML provides. The *input1* function of the TextIO structure reads exactly one character from the input and returns an *option* as a result. The reason it returns an *option* and not the character directly is because the stream might not be ready for reading. The *valOf* function can be used to get the value of an *option* that is not *NONE*.

```
- val u = TextIO.input1(TextIO.stdIn);
```

```
hi there
val u = SOME #"h" : elem option
=
= ^C
Interrupt
- u;
val it = SOME #"h" : elem option
- val v = valOf(u);
val v = #"h" : elem
```

5.19 Programming with Side-effects

Standard ML is not a pure functional language. It is possible to write programs with side effects, such as reading from and writing to streams. To write imperative programs the language should support sequential execution, variables, and possibly loops. All three of these features are available in SML. The following sections show you how to use each of these features.

5.19.1 Variables in Standard ML

There is only one kind of variable in Standard ML. Variables are called references. It is interesting to note that you cannot update an integer, real, string, or many other types of values in SML. All these values are immutable. They cannot be changed once created. That is a nice feature of a language because then you don't have to worry about the distinction between a reference to a value and the value itself. Array objects are mutable because they contain a list of references.

A reference in Standard ML is typed. It is either a reference to an *int*, or a *string*, or some other type of data. References can be mutated. So a reference can be updated to point to a new value as your program executes. Declaring and using a reference variable is shown in this example code. In SML a variable is declared by creating a reference to a value of a particular type.

```
- val x = ref 0;
val x = ref 0 : int ref
```

The exclamation point is used to refer to the value to which a reference points. This is called the dereference operator. It is the similar to the star (i.e. *) in C++ which dereferences a pointer.

```
- !x;
val it = 0 : int
- x := !x + 1;
val it = () : unit
- !x;
val it = 1 : int
```

```
let val x = ref 0
in
  x:= !x + 1;
  TextIO.output(TextIO.stdOut,"The new value of x is "^
                 Int.toString(!x)^"\n");
  !x
end
```

Fig. 5.21 Sequential Execution

The assignment operator (i.e. *:=*) operator updates the reference variable to point to a new value. The result of assignment is the empty tuple which has a special type called *unit*. Imperative programming in SML will often result in the unit type. Unlike ordinary identifiers you can bind to values using a *let val id = Expr in Expr end*, a reference can truly be updated to point to a new value.

It should be noted that references in Standard ML are typed. When a reference is created it can only point to a value of the same type it was originally created to refer to. This is unlike references in Python, but is similar to references in Java. A reference refers to a particular type of data.

5.19.2 Sequential Execution

If a program is going to assign variables new values or read from and write to streams it must be able to execute statements or expressions sequentially. There are two ways to write a sequence of expressions in SML. When you write a *let val id = Expr in Expr end* expression, the *Expr* in between the *in* and *end* may be a sequence of expressions. A sequence of expressions is semicolon separated. The code in Fig. 5.21 demonstrates how to write a sequence of expressions.

Evaluating this expression produces the following output.

```
The new value of x is 1
val it = 1 : int
```

In Fig. 5.21 semicolons separate the expressions in the sequence. Notice that semicolons don't terminate each line as in C++ or Java. Semicolons in SML are expression separators, not statement terminators. The last expression in a sequence of expressions is the value of the expression. All previously computed values in the sequential expression are thrown away. The *!x* is the last expression in the sequence in Fig. 5.21 so *1* is yielded as the value of the expression.

There are times when you may wish to evaluate a sequence of expressions in the absence of a *let* expression. In that case the sequence of expressions may be surrounded by parens. A left paren can start a sequence of expressions terminated by a right paren. The sequence of expressions is semicolon separated in either case. Here is some code that prints the value of x to the screen and then returns x + 1.

```
(TextIO.output(TextIO.stdOut,"The value of x is" ^
 Int.toString(x);
 x+1)
```

5.19.3 Iteration

Strictly speaking, variables and iteration are not needed in a functional language. Parameters can be passed in place of variable declarations. Recursion can be used in place of iteration. However, there are times when an iterative function might make more sense. For instance, when reading from a stream it might be more efficient to read the stream in a loop, especially when the stream might be large. A recursive function could overflow the stack in that case unless the recursive function were tail recursive and could be optimized to remove the recursive call.

A while loop in SML is written as *while Expr do Expr*. As is usual with while loops, the first *Expr* must evaluate to a boolean value. If it evaluates to *true* then the second *Expr* is evaluated. This process is repeated until the first *Expr* returns *false*.

5.20 Exception Handling

An exception occurs in SML when a condition occurs that requires special handling. If no special handling is defined for the condition the program terminates. As with most modern languages, SML has facilities for handling these exceptions and for raising user-defined exceptions. Consider the maxIntList function you wrote in practice problem 5.13. You probably had to figure out what to do if an empty list was passed to the function. One way to handle this is to raise an exception.

```
exception emptyList;

fun maxIntList [] = raise emptyList
  | maxIntList (h::t) = Int.max(h,maxIntList t) handle
                              emptyList => h
```

Invoking the maxIntList on an empty list can be handled using an exception handling expression. The handle clause uses pattern matching to match the right exception handler. To handle any exception the pattern _ can be used. The underscore matches anything. Multiple exceptions can be handled by using the vertical bar (i.e. |) between the handlers.

5.21 Encapsulation in ML

ML provides two language constructs that enable programmers to define new datatypes and hide their implementation details. The first of these language constructs we'll look at is the signature. The other construct is the structure.

5.21.1 Signatures

A signature is a means for specifying a set of related functions and types without providing any implementation details. This is analogous to an interface in Java or a template in C++. Consider the datatype consisting of a set of elements. A set is a group of elements with no duplicate values. Sets are very important in many areas of Computer Science and Mathematics. Set theory is an entire branch of mathematics. If we wanted to define a set in ML we could write a signature for it as follows.

The signature of a group of set functions and a set datatype is provided in Fig. 5.22. Notice this datatype is parameterized by a type variable so this could be a signature for a set of anything. You'll also notice that while the type parameter is $'a$ there are type variables named $''a$ within the signature. This is because some of these functions rely on the equals operator. In ML the equals operator is polymorphic and cannot be instantiated to a type. When this signature is used in practice the $'a$ and $''a$ types will be correctly instantiated to the same type.

Before a signature can be used, each of these functions must be implemented in a structure that implements the signature. This encapsulation allows a programmer to write code that uses these set functions without regards to their implementation.

```
signature SetSig =
sig
   exception Choiceset
   exception Restset
   datatype 'a set = Set of 'a list
   val emptyset   : 'a set
   val singleton  : 'a -> 'a set
   val member     : ''a -> ''a set -> bool
   val union      : ''a set -> ''a set -> ''a set
   val intersect  : ''a set -> ''a set -> ''a set
   val setdif     : ''a set -> ''a set -> ''a set
   val card       : 'a set -> int
   val subset     : ''a set -> ''a set -> bool
   val simetdif   : ''a set -> ''a set -> ''a set
   val forall     : ''a set -> (''a -> bool) -> bool
   val forsome    : ''a set -> (''a -> bool) -> bool
   val forsomeone : 'a set -> ('a -> bool) -> bool
end
```

Fig. 5.22 The Set Signature

An implementation must be provided before the program can be run. However, if a better implementation comes along later it can be substituted without changing any of the code that uses the set signature.

5.21.2 Implementing a Signature

To implement a signature we can use the struct construct that we've seen before. In this case it is done as follows. A partial implementation of the *SetSig* signature is provided in Fig. 5.23.

Of course, the entire implementation of all the set functions in the signature is required. Some of these functions are left as an exercise.

Practice 5.24 1. Write the card function. Cardinality of a set is the size of the set.
2. Write the intersect function. Intersection of two sets are just those elements that the two sets have in common. Sets do not contain duplicate elements.

You can check your answer(s) in Section 5.26.24.

```
(***** An Implementation of Sets as a SML datatype *****)

structure Set : SetSig =
struct

exception Choiceset
exception Restset

datatype 'a set = Set of 'a list

val emptyset = Set []

fun singleton e = Set [e]

fun member e (Set [])     = false
  | member e (Set (h::t)) = (e = h) orelse member e (Set t)

fun notmember element st = not (member element st)

fun union (s1 as Set L1) (s2 as Set L2) =
    let fun noDup e = notmember e s2
     in
       Set ((List.filter noDup L1)@(L2))
    end

...
end
```

Fig. 5.23 A Set Structure

5.22 Type Inference

Perhaps Standard ML's strongest point is the formally proven soundness of its type inference system. ML's type inference system is guaranteed to prevent any run-time type errors from occurring in a program. This turns out to prevent many run-time errors from occurring in your programs. Projects like the Fox Project have shown that ML can be used to produce highly reliable large software systems.

The origins of type inference include Haskell Curry and Robert Feys who in 1958 devised a type inference algorithm for the simply typed lambda calculus. In 1969 Roger Hindley worked on extending this type inference algorithm. In 1978 Robin Milner independently from Hindley devised a similar type inference system proving its soundness. In 1985 Luis Damas proved Milner's algorithm was complete and extended it to support polymorphic references. This algorithm is called the Hindley-Milner type inference algorithm or the Milner-Damas algorithm. The type inference system is based on a very powerful concept called unification.

Unification is the process of using type inference rules to bind type variables to values. The type inference rules look like this.

IfThen

$$\frac{\varepsilon \vdash e_1 : bool \quad \varepsilon \vdash e_2 : \alpha \quad \varepsilon \vdash e_3 : \alpha}{\varepsilon \vdash if\ e_1\ then\ e_2\ else\ e_3 : \alpha}$$

This rule says that for an if-then expression to be correctly typed, the type of the first expression must be a *bool* and the types of the second and third expression must be unifiable. If those preconditions hold, then the type of the if-then expression is given by the type of either of the second two expressions (since they are the same). Unification happens when α is written twice in the rule above. The ε is the presence of type information that is used when determining the types of the three expressions and is called the type environment.

Here are two examples that suggest how the type inference mechanism works. In this example we determine the type of the following function.

```
fun f(nil,nil) = nil
  | f(x::xs,y::ys) = (x,y)::f(xs,ys);
```

The function f takes one parameter, a pair.

```
f: 'a * 'b -> 'c
```

From the nature of the argument patterns, we conclude that the three unknown types must be lists.

```
f: ('p list) * ('s list) -> 't list
```

The function imposes no constraints on the domain lists, but the codomain list must be a list of pairs because of the cons operation *(x,y) ::*. We know *x:'p* and *y:'s*. Therefore *'t='p *'s*.

```
f: 'p list * 's list -> ('p * 's) list
```

where *‘p* and *‘s* are any ML types. In this example the type of the function g is inferred.

```
fun g h x = if null x then nil
            else
              if h (hd x) then g h (tl x)
              else (hd x)::g h (tl x);
```

The function g takes two parameters, one at a time.

```
g: 'a -> 'b -> 'c
```

The second parameter, x, must serve as an argument to *null*, *hd*, and *tl*; it must be a list.

```
g: 'a -> ('s list) -> 'c
```

The first parameter, *h*, must be a function since it is applied to *hd x*, and its domain type must agree with the type of elements in the list. In addition, *h* must produce a boolean result because of its use in the conditional expression.

```
g: ('s -> bool) -> ('s list) -> 'c
```

The result of the function must be a list since the base case returns *nil*. The result list is constructed by the code *(hd x) ::g h (tl x)*, which adds items of type *‘s* to the resulting list.

Therefore, the type of g must be:

```
g: ('s -> bool) -> 's list -> s list
```

Chapter 8 explores type inference in much more detail. A type checker for Standard ML is developed using Prolog, a programming language ideally suited to problems involving unification.

5.23 Building a Prefix Calculator Interpreter

The datatype definition in Fig. 5.12 provided an abstract syntax tree definition for a calculator language with one memory location. A related prefix calculator expression language is relatively easy to define and from that we can build an interpreter of prefix calculator expressions. Prefix expressions are comprised of an operator first followed by an expression or expressions. The prefix calculator expression language is defined by this LL(1) grammar.

$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = \{E\}$

$\mathcal{T} = \{S,\ R, number,\ \ , +, -, *, /\}$

$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow +\ E\ E \mid -\ E\ E \mid *\ E\ E \mid /\ E\ E \mid\sim E \mid S\ E \mid R \mid number$$

```
1  fun delimiter #" " = true
2    | delimiter #"\t" = true
3    | delimiter #"\n" = true
4    | delimiter _ = false
5
6  fun run() =
7    (TextIO.output(TextIO.stdOut,"Please enter a prefix calculator expression: ");
8     TextIO.flushOut(TextIO.stdOut);
9     let val line = TextIO.inputLine(TextIO.stdIn)
10        val tokens = String.tokens delimiter (valOf line)
11        val (ast,remainingTokens) = E(tokens)
12        val result = eval(ast)
13    in
14      if length(remainingTokens) <> 0 then
15        raise(eofException)
16      else ();
17      TextIO.output(TextIO.stdOut,"The answer is: "^Int.toString(result)^"\n")
18    end
19    handle eofException =>
20               TextIO.output(TextIO.stdOut,
21                 "You entered an invalid prefix expression.\n")
22           | Option =>
23               TextIO.output(TextIO.stdOut,
24                 "You entered invalid characters in the prefix expression.\n"))
```

Fig. 5.24 The Prefix Calc Interpreter Run Function

The only non-terminal in this grammar is *E*. The *S* is the store operator which stores the expression that follows it in the memory location. The *R* is the recall operator. The tilde (i.e. ~) is the unary negation operator. To implement an interpreter for this language we must first parse the expression and build an abstract syntax tree. Then the abstract syntax tree can be evaluated. The entire process can be encapsulated in a *run* function.

The *run* function that provides the overall design of the prefix calculator interpreter is provided in Fig. 5.24.

A number of things should be explained about this code. Line 8 flushes standard output. Without it the prompt does not print before the program starts waiting for input. Line 9 gets a line of input from the user. It is returned as a string option so on line 10 the *valOf* function is applied to get the string or raise an Option exception if *NONE* had been returned.

Line 10 calls the *tokens* function. All the tokens must be separated by spaces or tabs for the program to read the tokens correct. Here is an example of running this code.

```
- run();
Please enter a prefix calculator expression: + * S ~ 6 R 5
The answer is: 41
val it = () : unit
```

Line 11 calls the parser to parse the list of tokens. In this case, the list of tokens is passed to the parsing function. The parser returns a tuple with the AST as the first item of the tuple and the rest of the tokens as the second result. After parsing, line

14 checks to see that there are no more tokens left after parsing. If there are, then the *eofExceptioni* is raised.

Line 12 calls the evaluator function *eval* to interpret the AST. The *eval* function returns the result of evaluating the tree. Line 17 prints the result to the screen.

There are two handled exceptions. If the *eofException* is thrown then the expression did not parse correctly. If the Option exception is thrown there was a bad token in the input. Note that only integers are allowed for numbers in this implementation. This was decided by the AST definition in Fig. 5.12.

5.23.1 The Prefix Calc Parser

Parsing the expression is easy thanks to the LL(1) grammar for prefix calculator expressions. The *E* function is defined using pattern-matching in Fig. 5.25. Each time a token is consumed it is simply omitted from the remaining list of tokens. The tokens are single-threaded through the function. This just means the left over tokens are always passed on to the next piece to be parsed and the remaining tokens are always returned along with the AST when the *E* function returns.

The parser doesn't do any evaluation of the data. It simply works on building an AST for the expression. The evaluation of the AST comes later, by the evaluator.

Notice in line 39 that the *valOf* function is used on the result of the *Int.fromString* function. If the string being converted is not a valid value, the *valOf* will raise the Option exception terminating the run function with an appropriate error message.

Line 43 of Fig. 5.25 handles getting to the end of the input (i.e. the list of tokens) earlier than is expected. If the parser reaches this case the original expression was mal-formed and throwing the *eofException* is the appropriate response.

5.23.2 The AST Evaluator

To complete implementation of the prefix calculator the AST needs to be evaluated. The *eval* function presented in Fig. 5.26 provides this evaluation function. Line 1 declares a memory reference that is imperatively updated with the value stored in the calculator's memory.

Lines 2–9 provide the traditional binary operations of addition, subtraction, multiplication, and division. Because this calculator is only an integer calculator, the integer division *div* is used. Unary negation occurs on lines 10 and 11.

Line 12 stores a value in the memory of the calculator by first evaluating the subtree and then storing the value before returning it. Line 18 is responsible for recalling the value by returning the dereferenced memory location.

```
exception eofException;

fun E ("+"::rest) =
  let val (ast1,rest1) = E(rest)
    val (ast2,rest2) = E(rest1)
  in
    (add'(ast1,ast2),rest2)
  end
  | E ("-"::rest) =
  let val (ast1,rest1) = E(rest)
    val (ast2,rest2) = E(rest1)
  in
    (sub'(ast1,ast2),rest2)
  end
  | E ("*"::rest) =
  let val (ast1,rest1) = E(rest)
    val (ast2,rest2) = E(rest1)
  in
    (prod'(ast1,ast2),rest2)
  end
  | E ("/"::rest) =
  let val (ast1,rest1) = E(rest)
    val (ast2,rest2) = E(rest1)
  in
    (div'(ast1,ast2),rest2)
  end
  | E ("~"::rest) =
  let val (ast,rest1) = E(rest)
  in
    (negate'(ast),rest1)
  end
   | E ("S"::rest) =
  let val (ast,rest1) = E(rest)
  in
    (store'(ast),rest1)
  end
  | E ("R"::rest) = (recall',rest)
  | E (x::rest) =
    let val i = valOf(Int.fromString(x))
    in
      (integer'(i),rest)
    end
  | E nil = raise eofException;
```

Fig. 5.25 The Parser

```
val memory = ref 0;
fun eval(add'(t1,t2)) =
      eval(t1) + eval(t2)
  | eval(sub'(t1,t2)) =
      eval(t1) - eval(t2)
  | eval(prod'(t1,t2)) =
      eval(t1) * eval(t2)
  | eval(div'(t1,t2)) =
      eval(t1) div eval(t2)
  | eval(negate'(t)) =
       ~1 * eval(t)
  | eval(store'(t)) =
    let val x = eval(t)
    in
      memory := x;
      x
    end
  | eval(recall') = !memory
  | eval(integer'(x)) = x
```

Fig. 5.26 The Evaluator

5.23.3 Imperative Programming Observations

There are a couple of Standard ML syntax issues that are good to recognize at this point. In Fig. 5.24, line 7 begins with a left paren. The left paren can be used to construct a tuple in Standard ML, but it is also used to *begin* a sequence of expressions. The last right paren on line 24 *ends* the sequence. Expressions are separated by semicolons in a sequence of expressions. This occurs on line 16 of Fig. 5.24. No semicolon appears after the expression on line 17 because semicolons only separate expressions, they do not terminate them. On line 16 the else clause has a *unit* (i.e. *()*) as its result. This is because the type generated by raising an exception is a *unit*, and the *then* and *else* clause return types must match.

5.24 Chapter Summary

This chapter introduced functional programming. For many this is a new way of thinking about programming. Recursion is the main pattern used in computing when writing in a functional programming style. Higher-order functions are an important part of functional programming. Certain patterns appear often in functional programs and these patterns have been implemented as some common higher-order functions like *map*, *filter*, *foldr*, and others.

An important thing to learn from this chapter is that functional programming is more declarative and less prescriptive than programming in an imperative language like C++ or Java. Standard ML is a good functional programming language but other languages like C++, Java, and Python support functional programming as well.

Standard ML has a strong type checker that has been proven sound and complete. That means that while more time is spent removing type errors from programs, much less time is spent debugging Standard ML programs. Experiments like the Fox Project at Carnegie Mellon have shown this is true for large software systems written in Standard ML as well.

Much more can be learned about Standard ML and the next chapter not only looks at some Standard ML tools for language implementation, but it also describes the implementation of a compiler that translates Standard ML to JCoCo assembly language.

Jeffrey Ullman's book on functional programming in Standard ML is a very good introduction and reference for Standard ML. It is more thorough than the topics provided in this text and contains many topics not covered here including discussion of arrays, functors, and sharings along with a few of the Basis structures. The topics presented here and in the next chapter give you a good introduction to the ideas and concepts associated with functional programming. Ullman's book and on-line tutorials and manual pages are another great resource for learning functional programming.

5.25 Exercises

In the exercises below you are encouraged to write other functions that may help you in your solutions. You might have better luck with some of the harder ones if you solve a simpler problem first that can be used in the solution to the harder problem.

You may wish to put your solutions to these problems in a file and then

```
- use "thefile";
```

in SML. This will make writing the solutions easier. You can try the solutions out by placing tests right within the same file. You should always comment any code you write. Comments in SML are preceded with a *(** and terminated with a **)*.

1. Reduce $(\lambda z.z + z)((\lambda x.\lambda y.x + y)\ 4\ 3)$ by normal order and applicative order reduction strategies. Show the steps.
2. How does the SML interpreter respond to evaluating each of the following expressions? Evaluate each of these expression in ML and record what the response of the ML interpreter is.

   (a) *8 div 3;*
   (b) *8 mod 3;*
   (c) *"hi"^"there";*

(d) *8 mod 3 = 8 div 3 orelse 4 div 0 = 4;*
(e) *8 mod 3 = 8 div 3 andalso 4 div 0 = 4;*

3. Describe the behavior of the *orelse* operator in exercise 2 by writing an equivalent *if then* expression. You may use nested if expressions. Be sure to try your solution to see you get the same result.
4. Describe the behavior of the *andalso* operator in exercise 2 by writing an equivalent *if then* expression. Again you can use nested if expressions.
5. Write an expression that converts a character to a string.
6. Write an expression that converts a real number to the next lower integer.
7. Write an expression that converts a character to an integer.
8. Write an expression that converts an integer to a character.
9. What is the signature of the following functions? Give the signature and an example of using each function.

   (a) hd
   (b) tl
   (c) explode
   (d) concat
   (e) :: - This is an infix operator. Use the prefix form of *op ::* to get the signature.

10. The greatest common divisor of two integers, x and y, can be defined recursively. If y is zero then x is the greatest common divisor. Otherwise, the greatest common divisor of x and y is equal to the greatest common divisor of y and the remainder x divided by y. Write a recursive function called *gcd* to determine the greatest common divisor of x and y.
11. Write a recursive function called *allCaps* that given a string returns a capitalized version of the string.
12. Write a recursive function called *firstCaps* that given a list of strings, returns a list where the first letter of each of the original strings is capitalized.
13. Using pattern matching, write a recursive function called *swap* that swaps every pair of elements in a list. So, if *[1,2,3,4,5]* is given to the function it returns *[2,1,4,3,5]*.
14. Using pattern matching, write a function called *rotate* that rotates a list by n elements. So, *rotate(3,[1,2,3,4,5])* would return *[4,5,1,2,3]*.
15. Use pattern matching to write a recursive function called *delete* that deletes the nth letter from a string. So, *delete(3,"Hi there")* returns *"Hi here"*. HINT: This might be easier to do if it were a list.
16. Again, using pattern matching write a recursive function called *intpow* that computes x^n. It should do so with $O(log\ n)$ complexity.
17. Rewrite the *rotate* function of exercise 14 calling it *rotate2* to use a helper function so as to guarantee $O(n)$ complexity where n is the number of positions to rotate.
18. Rewrite exercise 14's *rotate(n,lst)* function calling it *rotate3* to guarantee that less than l rotations are done where l is the length of the list. However, the

outcome of rotate should be the same as if you rotated *n* times. For instance, calling the function as *rotate3(6,[1,2,3,4,5])* should return *[2,3,4,5,1]* with less than 5 recursive calls to *rotate3*.

19. Rewrite the *delete* function from exercise 15 calling it *delete2* so that it is curried.
20. Write a function called *delete5* that always deletes the fifth character of a string.
21. Use a higher-order function to find all those elements of a list of integers that are even.
22. Use a higher-order function to find all those strings that begin with a lower case letter.
23. Use a higher-order function to write the function *allCaps* from exercise 11.
24. Write a function called *find(s,file)* that prints the lines from the file named *file* that contain the string *s*. You can print the lines to *TextIO.stdOut*. The *file* should exist and should be in the current directory.
25. Write a higher-order function called *transform* that applies the same function to all elements of a list transforming it to the new values. However, if an exception occurs when transforming an element of the list, the original value in the given list should be used. For instance,

```
- transform (fn x => 15 div x) [1,3,0,5]
val it = [15,5,0,3] : int list
```

26. The natural numbers can be defined as the set of terms constructed from 0 and the $succ(n)$ where n is a natural number. Write a datatype called *Natural* that can be used to construct natural numbers like this. Use the capital letter O for your zero value so as not to be confused with the integer 0 in SML.
27. Write a *convert(x)* function that given a natural number like that defined in exercise 26 returns the integer equivalent of that value.
28. Define a function called *add(x,y)* that given *x* and *y*, two natural numbers as described in exercise 26, returns a natural number that represents the sum of *x* and *y*. For example,

```
- add(succ(succ(O)),succ(O))
val it = succ(succ(succ(O))) : Natural
```

You may NOT use *convert* or any form of it in your solution.

29. Define a function called *mul(x,y)* that given *x* and *y*, two natural numbers as described in exercise 26, returns a natural that represents the product of *x* and *y*. You may NOT use *convert* or any form of it in your solution.
30. Using the *add* function in exercise 28, write a new function *hadd* that uses the higher order function called *foldr* to add together a list of natural numbers.
31. The prefix calculator intpreter presented at the end of this chapter can be implemented a little more concisely by having the parser not only parse the prefix expression, but also evaluate the expression at the same time. If this is to be done, the parser ends up returning a *unit* because the parser does not need to return an AST since the expression has already been evaluated. This means the definition of the *AST* is no longer needed. Rewrite the prefix calculator code presented at the end of this chapter to combine the *parse* and *eval* functions.

Remove any unneeded code from your implementation but be sure to cover all the error conditions as the version presented in this chapter.

32. Alter the prefix expression calculator to accept either integers or floating point numbers as input. The result should always be a float in this implementation.
33. Add an input operator to the prefix calculator. In this version, expressions like *+ S I 5* when evaluated would prompt the user to enter a value when the *I* was encountered. This expression, when evaluated, would cause the program to respond as follows.

```
Please enter a prefix calculator expression: + S I 5
? 4
The answer is: 9
```

34. The prefix calculator intrepeter presented in this chapter can be transformed into a prefix calculator compiler by having the program write a file called *a.casm* with a JCoCo program that when run evaluates the compiled prefix calculator expression. Alter the code at the end of this chapter to create a prefix caclulator compiler. Running the compiler should work like this.

```
% sml
- use "prefixcalc.sml";
- run();
Please enter a prefix calculator expression: + S 6 5
- <ctrl-d>
% coco a.casm
The answer is: 11
```

35. For an extra hard project, combine the previous two exercises into one prefix calc compiler whose programs when run can gather input from the user to be used in the calculation.
36. Rewrite the prefix calculator project to single thread the memory location through the *eval* function as shown in pattern Completing this project removes the imperatively updated memory location from the code and replaces it with a single-threaded argument to the *eval* function.

5.26 Solutions to Practice Problems

These are solutions to the practice problems. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

5.26.1 Solution to Practice Problem 5.1

Addition is not commutative in Pascal or Java. The problem is that a function call, which may be one or both of the operands to the addition operator, could have a side-effect. In that case, the functions must be called in order. If no order is specified within expression evaluation then you can't even reliably write code with side-effects within an expression.

Here's another example of the problem with side-effects within code. In the code below, it was observed that when the code was compiled with one C++ compiler it printed 1,2 while with another compiler it printed 1,1. In this case, the language definition is the problem. The C++ language definition doesn't say what should happen in this case. The decision is left to the compiler writer.

```
int x = 1;
cout << x++ << x << endl;
```

The practice problem writes 17 as written. If the expression were b+a() then 15 would be written.

5.26.2 Solution to Practice Problem 5.2

With either normal order or applicative order function application is still left-associative. There is no choice for the initial redex.

$\underline{(\lambda xyz.xz(yz))(\lambda x.x)}(\lambda xy.x)$
$\Rightarrow (\lambda yz.\underline{(\lambda x.x)z}(yz))(\lambda xy.x)$
$\Rightarrow \underline{(\lambda yz.z(yz))(\lambda xy.x)}$
$\Rightarrow \lambda z.z(\underline{(\lambda xy.x)z})$
$\Rightarrow \lambda z.z(\lambda y.z)\Box$

5.26.3 Solution to Practice Problem 5.3

Normal Order Reduction

$\underline{(\lambda x.y)((\lambda x.xx)(\lambda x.xx))}$
$\Rightarrow y$

Applicative Order Reduction

$(\lambda x.y)(\underline{(\lambda x.xx)(\lambda x.xx)})$
$\Rightarrow (\lambda x.y)(\underline{(\lambda x.xx)(\lambda x.xx)})$
$\Rightarrow (\lambda x.y)(\underline{(\lambda x.xx)(\lambda x.xx)})$
$\Rightarrow (\lambda x.y)(\underline{(\lambda x.xx)(\lambda x.xx)})$ ...
You get the idea.

5.26.4 Solution to Practice Problem 5.4

```
x div 6;
Real.round(Real.fromInt(x) * y);
x / 6.3;
x mod y
```

5.26.5 Solution to Practice Problem 5.5

```
fun factorial(n) = if n=0 then 1 else n*factorial(n-1)
```

5.26.6 Solution to Practice Problem 5.6

The recursive definition is fib(0) = 0, fib(1) = 1, fib(n)=fib(n-1)+fib(n-2). The recursive function is:

```
fun fib(n) = if n = 0 then 1 else
             if n = 1 then 1 else
             fib(n-1) + fib(n-2)
```

5.26.7 Solution to Practice Problem 5.7

The solutions below are example solutions only. Others exist. However, the problem with each invalid list is not debatable.

1. You cannot cons a character onto a string list. “a” ::[”beautiful day”]
2. You cannot cons two strings. The second operand must be a list. “hi” :: [”there”]
3. The element comes first in a cons operation and the list second. “you” ::[”how”,”are”]
4. Lists are homogeneous. Reals and integers can’t be in a list together. [1.0,2.0,3.5,4.2]
5. Append is between two lists.2 ::[3,4] or *[2]@[3,4]*
6. Cons works with an element and a list, not a list and an element. 3 ::[]

5.26.8 Solution to Practice Problem 5.8

```
fun explode(s) =
  if s ="" then []
  else String.sub(s,0){~:}{:}
       (explode(String.substring(s,1,String.size(s)-1)))
```

5.26.9 Solution to Practice Problem 5.9

```
fun reverse(L) =
  if null L then []
  else append(reverse(tl(L)),[hd(L)])
```

5.26.10 Solution to Practice Problem 5.10

```
fun reverse([]) = []
  | reverse(h{~:}{:}t) = reverse(t)@[h]
```

5.26.11 Solution to Practice Problem 5.11

```
let val x = 10
in
    (* 1. Value of x = 10 *)
    let val x = x+1
    in
      (* 2. Value of x = 11 (hidden x still is 10) *)
      x
    end;
    (* 3. Value of x = 10 (hidden x is visible again) *)
    x
end
```

5.26.12 Solution to Practice Problem 5.12

```
datatype intlist = nil' | cons of int * intlist;
```

5.26.13 Solution to Practice Problem 5.13

```
fun maxIntList nil' = valOf(Int.minInt)
  | maxIntList (cons(x,xs)) = Int.max(x,maxIntList xs)
```

or

```
fun maxIntList (cons(x,nil')) = x
| maxIntList (cons(x,xs)) = Int.max(x,maxIntList xs)
```

The second solution will cause a pattern match nonexhaustive warning. That should be avoided, but is OK in this case. The second solution will raise a pattern match exception if an empty list is given to the function. See the section on exception handling for a better solution to this problem.

5.26.14 Solution to Practice Problem 5.14

The first step in the solution is to determine the number of calls required for values of n. Consulting Fig. 5.16 shows us that the number of calls are 1, 1, 3, 5, 9, 15, 25, etc. The next number in the sequence can be found by adding together two previous plus one more for the initial call.

The solution is that for $n \geq 3$ the function 1.5^n bounds the number of calls on the lower side while 2^n bounds it on the upper side. Therefore, the number of calls increases exponentially.

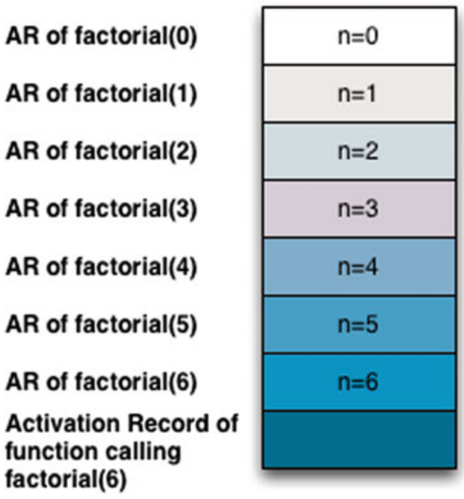


Fig. 5.27 The run-time stack when factorial(6) is called at its deepest point

5.26.15 Solution to Practice Problem 5.15

The cons operation is called n times where n is the length of the first list when append is called. When reverse is called it calls append with $n-1$ elements in the first list the first time. The first recursive call to reverse calls append with $n-2$ elements in the first list. The second recursive call to reverse calls append with $n-3$ elements in the first list. If we add up $n-1+n-2+n-3+...$ we end up with $\sum_{i=1}^{n-1} i = ((n-1)n)/2$. Multiplying this out leads to an n^2 term and the overall complexity of reverse is $O(n^2)$.

5.26.16 Solution to Practice Problem 5.16, see Fig. 5.27

5.26.17 Solution to Practice Problem 5.17

This solution uses the accumulator pattern and a helper function to implement a linear time reverse.

```
fun reverse(L) =
    let fun helprev (nil, acc) = acc
          | helprev (h::t, acc) = helprev(t,h::acc)
    in
      helprev(L,[])
    end
```

5.26.18 Solution to Practice Problem 5.18

This solution is surprisingly hard to figure out. In the first, f is certainly an uncurried function (look at how it is applied). The second requires f to be curried.

```
- fun curry f x y = f(x,y)
val curry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

- fun uncurry f (x,y) = f x y
val uncurry = fn : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

5.26.19 Solution to Practice Problem 5.19

The first takes a list of lists of integers and adds one to each integer of each list in the list of lists.

The second function takes a list of functions that all take the same type argument, say *a'*. The function returns a list of functions that all take an *a' list* argument. The example below might help. The list of functions that is returned by *(map map)* is suitable to be used as an argument to the *construction* function discussed earlier in the chapter.

```
- map (map add1);
val it = fn : int list list -> int list list

(map map);
stdIn:63.16-64.10 Warning: type vars not generalized because
of value restriction are instantiated to dummy types
(X1,X2,...)
val it = fn : (?.X1 -> ?.X2) list ->
                      (?.X1 list -> ?.X2 list) list
- fun double x = 2 * x;
val double = fn : int -> int
- val flist = (map map) [add1,double];
val flist = [fn,fn] : (int list -> int list) list
- construction flist [1,2,3];
val it = [[2,3,4],[2,4,6]] : int list list
```

5.26.20 Solution to Practice Problem 5.20

foldl is left-associative and *foldr* is right-associative.

```
- foldr op :: nil [1,2,3];
val it = [1,2,3] : int list
- foldr op @ nil [[1],[2,3],[4,5]];
val it = [1,2,3,4,5] : int list
```

5.26.21 Solution to Practice Problem 5.21

```
- List.filter (fn x => x mod 7 = 0) [2,3,7,14,21,25,28];
val it = [7,14,21,28] : int list
- List.filter (fn x => x > 10 orelse x = 0)
          [10, 11, 0, 5, 16, 8];
val it = [11,0,16] : int list
```

5.26.22 Solution to Practice Problem 5.22

```
cpslen [1,2,3] (fn v => v)
= cpslen [2,3] (fn w => ((fn v => v) (1 + w)))
= cpslen [3]
       (fn x => ((fn w => ((fn v => v) (1 + w)))(1 + x)))
= cpslen []
       (fn y => ((fn x => ((fn w => ((fn v => v)
       (1 + w)))(1 + x)))(1 + y)))
= (fn y => ((fn x => ((fn w => ((fn v => v)
       (1 + w)))(1 + x)))(1 + y))) 0
= (fn x => ((fn w => ((fn v => v) (1 + w)))(1 + x))) 1
= (fn w => ((fn v => v) (1 + w))) 2
= (fn v => v) 3
= 3
```

5.26.23 Solution to Practice Problem 5.23

```
datatype bintree = termnode of int
       | binnode of int * bintree * bintree;

val tree = (binnode(5,binnode(3,termnode(4),binnode(8,
             termnode(5),termnode(4))), termnode(4)));

fun depth (termnode _) = 0
  | depth (binnode(_,t1,t2)) = Int.max(depth(t1),depth(t2))+1

fun cpsdepth (termnode _) k = k 0
  | cpsdepth (binnode(_,t1,t2)) k =
      Int.max(cpsdepth t1 (fn v => (k (1 + v))),
              cpsdepth t2 (fn v => (k (1 + v))))
```

5.26.24 Solution to Practice Problem 5.24

```
fun card (Set L) = List.length L;

fun intersect (Set L1) S2 =
      Set ((List.filter (fn x => member x S2) L1))
```

Compiling Standard ML

6

The ML in the name *Standard ML* stands for meta-language. SML was designed as a language for describing languages when it was used as part of the Logic for Computable Functions (LCF) system [9]. Two tools were designed to work with Standard ML for language implementation, *ML-lex* and *ML-yacc*. The pattern matching, ease of defining recursive datatypes, the functional nature of the language along with these two tools make Standard ML an excellent language choice for implementing interpreters and compilers. This chapter introduces these two tools through a case study involving the development of a compiler for a subset of the Standard ML language called the *Small* language. Over the years the Small language has grown into a pretty robust subset of Standard ML.

Depicted in Fig. 6.1 are all the relevant pieces in constructing and using the *mlcomp* compiler which can be downloaded from http://github.com/kentdlee/mlcomp. Compiling an SML program begins by scanning the source file for tokens. The scanner is called by the parser to get each of the tokens from the SML source file. The parser, a bottom-up parser, performs a reverse right-most derivation of the source program forming an abstract syntax tree along the way. When the AST is returned by the parser, the compiler calls the code generator to evaluate the tree and produce the target code, in this case JCoCo assembly language. In this chapter the scanner and the parser won't have to be written by hand. *ML-lex* and *ML-yacc* are used to generate these parts of the compiler from specifications that are provided to these tools.

Two commonly used terms in compiler construction are the *front end* and the *back end*. The *front end*, referring to the scanner and the parser, reads the tokens and builds an AST of a program. The *back end* generates the code given the AST representation of the program. ML-lex and ML-yacc are used to generate the front end from two specifications, provided by the compiler writer. The back end is written by the compiler writer to generate the code given an AST of the program. In Fig. 6.1 the light green objects are the parts of the compiler provided by the compiler writer. The dark green box represents the SML program provided by the user of the compiler, the

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_6

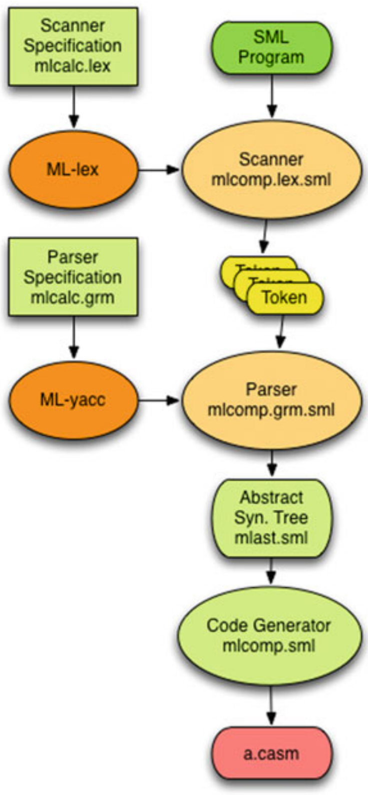


Fig. 6.1 Structure of MLComp

SML programmer who is compiling his or her code. Summarizing, the files written by the compiler writer include the following.

- The tokens of the language are defined in a file called *mlcomp.lex*.
- The datatype for the AST is defined in a file called *mlast.sml*.
- The grammar of the language is defined in a file called *mlcomp.grm*. This file also contains a mapping from productions in the grammar to nodes in an AST. The parser reads tokens and builds an AST of the expression being compiled.
- The code generator is defined in a file called *mlcomp.sml*.

The next sections introduce ML-lex, ML-yacc, and code generation. The rest of this chapter explores parts of the compiler that are already completed and other possible enhancements to the language. Building and using this compiler requires installation of Standard ML and the ML-yacc and ML-lex tools.

Don't be intimidated! The suggested enhancements to the language are accompanied by test programs that use these enhancements. By attempting to compile one of these tests you will be pointed at the location in the compiler where new code is required. Adding that code will lead you to another location within the compiler, and so on. The compiler is designed so that it will tell you where enhancements are needed when you attempt to compile a test that is not currently supported. By repeatedly attempting to build the compiler and compile a new test, you will be given a hands-on tour of the compiler. That, along with the descriptions in this chapter of how the compiler currently works will teach you about compiler construction for a non-trivial language! Good luck. With a little work you will learn a lot about compiler construction and implementing a functional programming language!

6.1 ML-lex

ML-lex is a scanner generator. ML-lex generates a function that can be used to get tokens from the input. It is based on a similar tool called *lex* that generates scanners for C programs. The input to the two tools is similar but not exactly the same. The input to ML-lex is a file consisting of three sections, where each section is separated by *%%*. The format of an ML-lex input file is:

```
User declarations
%%
ML-lex definitions
%%
Token Rules
```

The user declarations include any ML code that will assist you in defining the tokens. Typically, a variable is used to keep track of the line of input being read. There might also be some functions for converting strings to other values like integers. An error function that handles bad tokens is a common function for this section to get the scanner and the parser to work together.

The ML-lex definitions follow the user declarations. Sets of characters are declared in this section. In addition a functor must be declared. A functor is a module that takes a structure as a parameter and returns a new structure as a result. A functor is used by ML-lex and ML-yacc to create the scanner.

The last section of an ML-lex definition is composed of a set of rules that define the tokens of the language. Each rule has the form:

```
reg_exp => (return_value);
```

The *reg_exp* is a regular expression. The language of regular expressions can be used to define tokens. Every regular expression can be expressed as a finite state machine. Finite state machines can be used to recognize tokens. The set of *reg_exp* is eventually translated into a finite state machine that can be used to recognize tokens in the language. When a string of characters is recognized as a token, its matching return value is constructed from the rules and that value is returned by the scanner to the parser. Figures 6.2, 6.3, and 6.4 contain the three parts of the lexer specification given to ML-lex for the mlcomp compiler. The file is called mlcomp.lex.

```
 1  (* mlcomp.lex -- lexer spec *)
 2  type pos = int
 3  type svalue = Tokens.svalue
 4  type ('a, 'b) token = ('a, 'b) Tokens.token
 5  type lexresult = (svalue, pos) token
 6  val pos = ref 1
 7  val error = fn x => TextIO.output(TextIO.stdErr, x ^ "\n")
 8  val eof = fn () => Tokens.EOF(!pos, !pos)
 9  fun countnewlines s =
10      let val lst = explode s
11          fun count (c:char) nil = 0
12            | count c (h::t) =
13              let val tcount = count c t
14              in
15                if c = h then 1+tcount else tcount
16              end
17      in
18        pos:= (!pos) + (count #"\n" lst)
19      end
```

Fig. 6.2 mlcomp.lex part one

```
20  %%
21  %header (functor mlcompLexFun(structure Tokens : mlcomp_TOKENS));
22  alpha=[A-Za-z];
23  alphanumeric=[A-Za-z0-9_\.];
24  digit=[0-9];
25  ws=[\ \t];
26  dquote=[\"];
27  squote=[\'];
28  anycharbutquote=[^"];
29  anychar=[.];
30  pound=[\#];
31  tilde=[\~];
32  period=[\.];
```

Fig. 6.3 mlcomp.lex part two

In Fig. 6.2 lines 1–19 make up the first part of the ML-lex specification, the user declarations. The *pos* type must be defined and is used to define the position within the source program where a token is found. In this case, the position is the line on which it is found. Later a variable called *pos* is also initialized to 1 for the first line of the source program.

A structure called Tokens is used to contain information about the tokens returned by ML-lex. The *Tokens.svalue* is the actual string representing the characters of each token. Line 3 just equates a type called *svalue* to the *Tokens.svalue*. Line 4 does the same for the type *token*. Those two type names are used in line 6 where *lexresult* is declared. This *lexresult* is required to be defined in the user declarations section of *ML-lex*.

The *error* function is used later in the lexer specification. The *eof* function is used to return the *EOF* token and is called automatically by the lexer when it reaches the

```
%%
\(\*([^*]|[\r\n]|(\*+([^*\)]|[\r\n])))*\*+\) => (countnewlines yytext; lex());
\n  => (pos := (!pos) + 1; lex());
{ws}+  => (lex());
"+"  => (Tokens.Plus(!pos,!pos));
"*"  => (Tokens.Times(!pos,!pos));
"-"  => (Tokens.Minus(!pos,!pos));
"@"  => (Tokens.Append(!pos,!pos));
"=" => (Tokens.Equals(!pos,!pos));
"("  => (Tokens.LParen(!pos,!pos));
")"  => (Tokens.RParen(!pos,!pos));
"[" => (Tokens.LBracket(!pos,!pos));
"]" => (Tokens.RBracket(!pos,!pos));
"::" => (Tokens.ListCons(!pos,!pos));
"," => (Tokens.Comma(!pos,!pos));
";" => (Tokens.Semicolon(!pos,!pos));
"_" => (Tokens.Underscore(!pos,!pos));
"=>" => (Tokens.Arrow(!pos,!pos));
"|" => (Tokens.VerticalBar(!pos,!pos));
">" => (Tokens.Greater(!pos,!pos));
(* a few token are omitted here *)
{tilde}?{digit}+  => (Tokens.Int(yytext,!pos,!pos));
{pound}{dquote}{anychar}{dquote} => (Tokens.Char(yytext,!pos,!pos));
{dquote}{anycharbutquote}*{dquote} => (Tokens.String(yytext,!pos,!pos));
{alpha}{alphanumeric}*=>
   (let val tok = String.implode (List.map (Char.toLower)
              (String.explode yytext))
    in
      if      tok="let" then Tokens.Let(!pos,!pos)
      else if tok="val" then Tokens.Val(!pos,!pos)
      else if tok="in" then Tokens.In(!pos,!pos)
      else if tok="end" then Tokens.End(!pos,!pos)
      else if tok="if" then Tokens.If(!pos,!pos)
      else if tok="then" then Tokens.Then(!pos,!pos)
      else if tok="else" then Tokens.Else(!pos,!pos)
      else if tok="div" then Tokens.Div(!pos,!pos)
      else if tok="mod" then Tokens.Mod(!pos,!pos)
      else if tok="fn" then Tokens.Fn(!pos,!pos)
      else if tok="while" then Tokens.While(!pos,!pos)
      else if tok="do" then Tokens.Do(!pos,!pos)
      else if tok="and" then Tokens.And(!pos,!pos)
      else if tok="rec" then Tokens.Rec(!pos,!pos)
      else if tok="fun" then Tokens.Fun(!pos,!pos)
      else if tok="as" then Tokens.As(!pos,!pos)
      else if tok="handle" then Tokens.Handle(!pos,!pos)
      else if tok="raise" then Tokens.Raise(!pos,!pos)
      else if tok="true" then Tokens.True(!pos,!pos)
      else if tok="false" then Tokens.False(!pos,!pos)
      else Tokens.Id(yytext,!pos,!pos)
    end);
.  => (error ("error: bad token "^yytext); lex())
```

Fig. 6.4 mlcomp.lex part three

end of file. The *countnewlines* function is also used later in the lexer specification when skipping over whitespace likes spaces, tabs, and newline characters.

The ML-lex declarations begins with declaring a functor in Fig. 6.3. The functor is required by the parser. A functor is a parameterized type in Standard ML and this functor is expected by the parser and is parameterized by the Tokens structure. Declaring the functor in this way is required to get the parser generated by ML-yacc to talk to the scanner generated by ML-lex.

The alpha declaration declares a class of characters called *alpha* that consists of letters *a* to *z* in lower and upper case. The *alphanumeric* characters include letters, digits, underscores, and the period character. The *digit* declaration defines the class of digits as being *0* to *9*. The *ws* stands for whitespace. It defines blanks and tabs as whitespace. The *dquote* class is for double quote and *squote* for single quote. The ^ means *not* so *anycharbutquote* is exactly as it reads. The period represents any character whatsoever. The actual period character must be escaped by preceding it with a backslash.

Finally, the rules define all the tokens in the third part of the lexer definition in Fig. 6.4. The first rule discards comments in the source file. It says that comments look like *(* any text *)*. Unfortunately, this is a complex regular to start the rule definitions with. It is first because the rules will be matched in order of their definition. It begins by saying look for a left paren followed by an asterisk. Then the next part of the regular expression is one of two possibilities.

- A character which is not an asterisk or it is a carriage return (i.e. the \r) or a newline (i.e. the \n).
- A string of characters which is some number of asterisks followed by a either not an asterisk or a right paren or a carriage return or a newline.

Those two preceeding bullets may repeat zero or more times (the Kleene star that appears near the end of the regular expression says this). Finally, the whole regular expression ends by saying that the comment ends with one or more asterisks followed by a right paren. The action to take in this case is to call the *countnewlines* function to count any newline characters in the comment to update the *pos* variable accordingly. Finally, calling *lex* at the end of the action causes the lexer to get the next token, effectively ignoring the comment so the parser never sees it. The next regular expression skips newlines that are not in a comment. The third regular expression skips blanks and tabs that might appear in the program.

The next several rules define short simple tokens like infix operators. The token is defined within the *Tokens* structure and each rule returns a particular token value, defined in the parser. Every token value carries with it two integers. In this case, the line number is provided for both values. Tokens that consist of more than one or two characters should not be defined in this way since the length of each token string makes the number of states grow exponentially. The remaining rules define tokens that can't be explicitly given along with the keywords that are defined like identifiers in the language.

Line 54 defines positive and negative numbers. The *?* indicates 0 or 1 occurrence of the *negation* symbol. This is followed by 1 or more digits, followed by a possible period and other digits. Line 55 defines character constants in Standard ML like #"a" for instance. Escape characters like the newline character, "n", are not currently supported by could be. Line 56 defines string tokens which start with a double quote followed by zero or more of any character but double quote followed by a double quote. Lines 57–82 recognize identifiers and keywords. Defining one rule to handle all these different tokens with an *if-else-if* expression reduces the final number of states in the scanner. If each keyword were handled by a separate rule the number of states in the scanner would explode. Finally, line 83 handles any other character that might be found in the source file by writing an error message to the screen and skipping over it.

The scanner generated by ML-lex returns each token described in Fig. 6.4 with the line number in the source program where it was found. In some cases, the *lexeme*, the actual string of characters making up the token, is also returned. The *lexeme* is returned for tokens where the token type is not enough information. For instance, *Int*, *String*, *Char*, and *Id* tokens all need to carry along the lexeme, the *yytext*, because that information is needed by the parser. From a definition like the *mlcomp.lex* file shown in Figs. 6.2, 6.3, and 6.4 the ML-lex tool has enough information to generate a scanner for the tokens of the language.

Practice 6.1 Given the ML-lex specification in Figs. 6.2, 6.3, and 6.4, what more would have to be added to allow expressions like this to be correctly tokenized by the scanner? What new tokens would have to be recognized? How would you modify the specification to accept these tokens?

```
case x of
    1 => "hello"
  | 2 => "how"
  | 3 => "are"
  | 4 => "you"
```

You can check your answer(s) in Section 6.15.1.

6.2 The Small AST Definition

The parser reads tokens and builds an abstract syntax tree of a source program. Figure 6.5 contains the abstract syntax definition for the Small language. In SML, the abstract syntax definition is given by an SML datatype. Each type of node in the tree is tagged with its type. Some nodes in the tree include the subtrees such as the infixexp node. The datatype can consist of multiple types which may all be mutually recursive. For the multiple types to be mutually recursive, the keyword *and* is used to separate the datatype definitions.

The Small subset of Standard ML is primarily composed of expressions. The *exp* datatype describes trees representing expressions in the language. An expression is either an integer, character, boolean value, identifier, list constant, tuple constant, function application, infix expression, a sequence of expressions, a let declaration, a raised exception, an exception handler, an *if then* expression, a *while do* expression, or a function definition.

A function definition and an exception handler contain a list of matches. A match is composed of a pattern and an expression as in *4 =>"you"* for instance. The allowed

```
structure MLAS =
struct
datatype
  exp = int of string
      | ch of string
      | str of string
      | boolval of string
      | id of string
      | listcon of exp list
      | tuplecon of exp list
      | apply of exp * exp
      | infixexp of string * exp * exp
      | expsequence of exp list
      | letdec of dec * (exp list)
      | raisexp of exp
      | handlexp of exp * match list
      | ifthen of exp * exp * exp
      | whiledo of exp * exp
      | func of int * match list
and
  match = match of pat * exp
and
  pat = intpat of string
      | chpat of string
      | strpat of string
      | boolpat of string
      | idpat of string
      | wildcardpat
      | infixpat of string * pat * pat
      | tuplepat of pat list
      | listpat of pat list
      | aspat of string * pat
and
  dec = bindval of pat * exp
      | bindvalrec of pat * exp
      | funmatch of string * match list
      | funmatches of
end
```

Fig. 6.5 mlast.sml

patterns are described by the *pat* datatype and include integers, characters, strings, boolean values, identifiers, the underscore pattern (called *wildcardpat* in the AST definition), tuples, lists, and a special *as* pattern which lets the programmer specify an identifer as a pattern as in *z as (x,y)*. This would match a pattern where *x* and *y* match the elements of a tuple and *z* matches the entire tuple.

A *let* expression binds identifiers to values and the *dec* datatype defines binding declarations. In Standard ML it is possible to bind the identifiers in a pattern to an expression. The *bindvalrec* represents a recursive binding which is needed in the case of recursive function definitions. The *funmatch* is used in a function which is defined with a series of pattern matching cases. The *funmatches* comes into play when a series of mutually recursive function definitions are being defined, somewhat like the mutually recursive AST datatype definition given in Fig. 6.5.

Practice 6.2 How would you modify the abstract syntax so expressions like this could be represented?

```
case x of
    1 => "hello"
  | 2 => "how"
  | 3 => "are"
  | 4 => "you"
```

You can check your answer(s) in Section 6.15.2.

6.3 Using ML-yacc

ML-yacc is a parser generator. The name stands for *Yet Another Compiler Compiler* (i.e. yacc). *Yacc* is a tool that generates parsers for compilers written in C or C++. ML-yacc is the SML version of this tool. ML-yacc is a little different than yacc but provides mostly the same functionality. ML-yacc's input format is similar to ML-lex's input format. An ML-yacc specification consists of three parts.

```
User declarations
%%
ML-yacc definitions
%%
Rules
```

The user declarations include providing the AST definition and any functions, variables, or exceptions that might be useful while parsing the input. Figures 6.6, 6.7, 6.8, and 6.9 contains the parser specification for the Small language.

The user declarations of the parser are on lines 1–42 of Fig. 6.6. This part of the parser contains useful utility functions much like the user declaration section of ML-lex. The abstract syntax definition is opened in the parser. This is similar to the *using*

```
1  open MLAS;
2  val idnum = ref 0
3  fun nextIdNum() =
4    let val x = !idnum
5    in
6      idnum := !idnum + 1;
7      x
8    end
9  exception emptyDecList;
10 exception argumentMismatch;
11 fun uncurryIt nil = raise emptyDecList
12   | uncurryIt (L as ((name,patList,exp)::t)) =
13     let fun len nil = raise argumentMismatch
14           | len [(n,p,e)] = length(p)
15           | len ((n,p,e)::t) =
16             let val size = length(p)
17             in
18               if size = len t then size else
19                 (TextIO.output(TextIO.stdOut,
20                 "Syntax Error: Number of arguments does not match in function "
21                 ^name^"\n");
22                  raise argumentMismatch)
23             end
24         val tupleList = List.map (fn x => "v"^Int.toString(nextIdNum())) patList
25      in
26        len(L); (* just check the parameter list sizes of all patterns *)
27        (name,[match(idpat(hd(tupleList)),
28                  List.foldr (fn (x,y) => func(nextIdNum(),[match(idpat(x), y)]))
29                     (apply (func(nextIdNum(),
30                             List.map (fn (n,p,e) => match(tuplepat(p),e)) L),
31                             tuplecon(List.map (fn x => id(x)) tupleList)))
32                         (tl tupleList))])
33      end
34 fun makeMatchList (nil) = raise emptyDecList
35   | makeMatchList (L as (name,pat,exp)::t) =
36     (name, List.map (fn (n,p,e) =>
37                (if name <> n then (
38                    TextIO.output(TextIO.stdOut,
39                    "Syntax Error: Function definition with different names "
40                    ^name^" and "^n^" not allowed.\n");
41                    raise argumentMismatch)
42                 else match(p,e))) L)
```

Fig. 6.6 mlcomp.grm part one

namespace std in C++. Lines 2–8 define a function that can return a unique integer which is needed in some code in the parser. Line 10–33 define a function and two exceptions that are used in defining curried functions. This is covered in detail later in the chapter. Lines 34–42 convert a list of (name, pattern, expression) tuples to a tuple of (name,list) where the list is a list of (pattern,expression) pairs. It also checks that all names in the original tuples were for the same function.

The ML-yacc definitions start on line 43 if Fig. 6.7. They include a name to prefix functions in the scanner with, in this case *mlcomp*. The *verbose* helps in debugging. The *eop*, or *end of parse*, says that *EOF* is the last token returned. This helps in terminating the parser. The *pos* type is redeclared in Fig. 6.7 for use with the scanner.

The *nodefault* tells the parser not to insert tokens it thinks might have been left out. This helps in finding syntax errors earlier than they would be otherwise. If this were omitted the parser would insert a token when it is reasonably sure the program

```
%%
%name mlcomp (* mlcomp becomes a prefix in functions *)
%verbose
%eop EOF
%pos int
%nodefault
%pure (* no side-effects in actions *)
%term EOF | LParen | RParen | Plus | Minus | Times | Div | Mod | Greater | Less
    | GreaterEqual | LessEqual | NotEqual | Append | ListCons | Negate | Comma
    | Semicolon | Underscore | Arrow | Equals | VerticalBar | LBracket
    | RBracket | Fun | As | Let | Val | In | End | If | Then | Else | Fn
    | While | Do | Handle | Raise | And | Rec | String of string
    | Char of string | Int of string | True | False | Id of string
    | SetEqual | Exclaim
%nonterm Prog of exp | Exp of exp | Expressions of exp list
       | ExpSequence of exp list | MatchExp of match list
       | Pat of pat | Patterns of pat list
       | PatternSeq of pat list | Dec of dec | ValBind of dec
       | FunBind of (string * match list) list
       | FunMatch of (string * pat * exp) list
       | Con of exp | FuncExp of exp | DecSeq of dec list
       | CurriedFun of (string * pat list * exp) list
%right SetEqual
%left Plus Minus Append Equals NotEqual
%left Times Div Mod Greater Less GreaterEqual LessEqual
%right ListCons
%right Exclaim
```

Fig. 6.7 mlcomp.grm part two

```
%%
Prog : Exp EOF                          (Exp)
Exp : Con                               (Con)
    | Id                                (id(Id))
    | FuncExp Exp                       (apply(FuncExp,Exp))
    | Exclaim Exp                       (apply(id("!"),Exp))
    | Id SetEqual FuncExp               (infixexp(":=",id(Id),FuncExp))
    | Exp Plus Exp                      (infixexp("+",Exp1,Exp2))
    | Exp Minus Exp                     (infixexp("-",Exp1,Exp2))
    | Exp Times Exp                     (infixexp("*",Exp1,Exp2))
    | Exp Div Exp                       (infixexp("div",Exp1,Exp2))
    | Exp Mod Exp                       (infixexp("mod",Exp1,Exp2))
    | Exp Greater Exp                   (infixexp(">",Exp1,Exp2))
    | Exp GreaterEqual Exp              (infixexp(">=",Exp1,Exp2))
    | Exp Less Exp                      (infixexp("<",Exp1,Exp2))
    | Exp LessEqual Exp                 (infixexp("<=",Exp1,Exp2))
    | Exp Equals Exp                    (infixexp("=",Exp1,Exp2))
    | Exp NotEqual Exp                  (infixexp("<>",Exp1,Exp2))
    | Exp Append Exp                    (infixexp("@",Exp1,Exp2))
    | Exp ListCons Exp                  (infixexp("::",Exp1,Exp2))
    | LParen Exp RParen                 (Exp)
    | LParen Expressions RParen         (tuplecon(Expressions))
    | LParen ExpSequence RParen         (expsequence(ExpSequence))
    | LBracket Expressions RBracket     (listcon(Expressions))
    | LBracket RBracket                 (id("nil"))
    | Let DecSeq In ExpSequence End
                (List.hd (List.foldr (fn (x,y) =>
                    [letdec(x,y)]) ExpSequence DecSeq))
    | Raise Exp                         (raisexp(Exp))
    | Exp Handle MatchExp               (handlexp(Exp,MatchExp))
    | If Exp Then Exp Else Exp          (ifthen(Exp1,Exp2,Exp3))
    | While Exp Do Exp                  (whiledo(Exp1,Exp2))
    | Fn MatchExp                       (func(nextIdNum(),MatchExp))
```

Fig. 6.8 mlcomp.grm part three

```
FuncExp : Exp                          (Exp)
Expressions : Exp                      ([Exp])
            | Exp Comma Expressions    (Exp::Expressions)
ExpSequence : Exp                      ([Exp])
            | Exp Semicolon ExpSequence
                                       (Exp::ExpSequence)
MatchExp : Pat Arrow Exp               ([match(Pat,Exp)])
         | Pat Arrow Exp VerticalBar MatchExp
                                       (match(Pat,Exp)::MatchExp)
Pat : Int                              (intpat(Int))
    | Char                             (chpat(Char))
    | String                           (strpat(String))
    | True                             (boolpat("true"))
    | False                            (boolpat("false"))
    | Underscore                       (wildcardpat)
    | Id                               (idpat(Id))
    | Pat ListCons Pat                 (infixpat("::",Pat1,Pat2))
    | LParen Pat RParen                (Pat)
    | LParen Patterns RParen           (tuplepat(Patterns))
    | LBracket Patterns RBracket       (listpat(Patterns))
    | LBracket RBracket                (idpat("nil"))
    | Id As Pat                        (aspat(Id,Pat))
Patterns : Pat                         ([Pat])
         | Pat Comma Patterns          (Pat::Patterns)
PatternSeq : Pat                       ([Pat])
           | Pat PatternSeq            (Pat::PatternSeq)
Dec : Val ValBind                      (ValBind)
    | Fun FunBind                      (funmatches(FunBind))
DecSeq : Dec                           ([Dec])
       | Dec DecSeq                    (Dec::DecSeq)
ValBind : Pat Equals Exp               (bindval(Pat,Exp))
        | Rec Id Equals Exp            (bindvalrec(idpat(Id),Exp))
FunBind : FunMatch                     ([makeMatchList FunMatch])
        | CurriedFun                   ([uncurryIt CurriedFun])
        | FunBind And FunBind          (FunBind1@FunBind2)
FunMatch : Id Pat Equals Exp           ([(Id,Pat,Exp)])
         | Id Pat Equals Exp VerticalBar FunMatch
                                       ((Id,Pat,Exp)::FunMatch)
CurriedFun :
           Id PatternSeq Equals Exp    ([(Id,PatternSeq,Exp)])
         | Id PatternSeq Equals Exp VerticalBar CurriedFun
                                       ((Id,PatternSeq,Exp)::CurriedFun)
Con : Int                              (int(Int))
    | Char                             (ch(Char))
    | String                           (str(String))
    | True                             (boolval("true"))
    | False                            (boolval("false"))
    | LParen RParen                    (tuplecon([]))
```

Fig. 6.9 mlcomp.grm part four

being parsed is missing a token. The *pure* declarations says that the parser has no side-effects. It simply builds a tree and returns it. This means that ML-yacc can undo certain parsing operations if it needs to without fear of a side-effect not being undone.

Most importantly the terminals and nonterminals of the language are declared in the ML-yacc declarations. Those tokens that carry along their lexeme are declared as a token *of* something. For instance, *Int of string* where the string is a string containing the token's number. The nonterminals include all the *nonterm* defined identifiers and represent the syntactic categories of the grammar.

There are just a few more declarations in the ML-yacc definitions section. The grammar rules, given in the next section, have some ambiguity in them. Specifically, some of the operators have ambiguous precedence. The associativity and precedence rules are defined on lines 65–69 with those operators with lowest precedence coming first and higher precedence operators later. So *SetEqual* has the lowest precedence and is right associative. The *Plus*, *Minus*, *Append*, *Equals*, and *NotEqual* operator tokens have the next lowest precedence and are all left associative. These precedence rules simplify the writing of the grammar while disambiguating it.

Lines 70–150 of Figs. 6.8 and 6.9 make up the *Rules* section and define the grammar for Small. Each production of the grammar is given on the left of the AST it returns when matched. Consider a Small program like this:

```
4 * x + 5
```

When matching the rule on line 77 of the grammar specification the *4*x* will match the expression on the left side of the *Plus* token. The AST that results from parsing *4*x* is named *Exp1* by ML-yacc. Remember, the parser is a bottom-up parser so the *4*x* has already been parsed. The *5* on the right side of the *Plus* has also been parsed when this rule is matched. The *5* is referred to as *Exp2* by ML-yacc. The rule on line 74 says when this rule is matched to return an AST of *infixexp("+",Exp1,Exp2)*. The full AST for this expression, and the value returned for this example, would be

infixexp("+", infixexp("*", int("4"), id("x")), int("5"))

To the right of each production is a value that is returned when that production is matched during parsing. In most cases, this is a straight-forward construction of an AST. In a few cases a list is returned instead as in the *MatchExp* nonterminal or the *PatternSeq* nonterminal. In a couple of cases, the *uncurryIt* or *makeMatchList* functions are called which in turn generate an AST node to be returned. In the end, the parser returns a description of the source program as an abstract syntax tree.

Practice 6.3 What modifications would be required in the *mlcomp.grm* specification to parse expressions like this?

```
case x of
    1 => "hello"
  | 2 => "how"
  | 3 => "are"
  | 4 => "you"
```

You can check your answer(s) in Section 6.15.3.

```
5 + 4
```

Fig. 6.10 SML addition

```
Function: main/0
Constants: None, 5, 4
BEGIN
    LOAD_CONST 1
    LOAD_CONST 2
    BINARY_ADD
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Fig. 6.11 JCoCo addition

```
infixexp("+", int("5"),
              int("4"))
```

Fig. 6.12 Addition AST

6.4 Compiling and Running the Compiler

Code generation is essential to any compiler. The code generator translates the abstract syntax tree into a language that may either be executed directly or interpreted by some low-level interpreter like the JCoCo Virtual Machine. For this text, the *mlcomp* compiler generates JCoCo assembly language. The code generator for *mlcomp* is in the file named *mlcomp.sml*. The entire file is too big to include here. The remainder of this chapter will examine code generation in parts. First, consider code generation for the addition of two integers.

Adding 5 and 4 in the Small language is written as shown in Fig. 6.10. Adding 5 and 4 in JCoCo can be written as shown in the code of Fig. 6.11. The compiler for the Small language is given a source file as shown in Fig. 6.10 and parses it to produce an abstract syntax tree as shown in Fig. 6.12. The abstract syntax tree is passed to the code generator. It is the job of the code generator, given the abstract sytnax tree shown in Fig. 6.12, to generate code similar to that of Fig. 6.11.

```
1  fun codegen(int(i),outFile,indent,consts,...) =
2      let val index = lookupIndex(i,consts)
3      in
4        TextIO.output(outFile,indent^"LOAD_CONST "^index^"\n")
5      end
6    | codegen(infixexp("+",t1,t2),outFile,indent,consts,...) =
7      let val _ = codegen(t1,outFile,indent,consts,...)
8          val _ = codegen(t2,outFile,indent,consts,...)
9      in
10       TextIO.output(outFile,indent^"BINARY_ADD\n")
11 end
```

Fig. 6.13 Addition code generation

The *codegen* function of *mlcomp.sml* is responsible for generating code. To generate code for the AST shown in Fig. 6.12 the two patterns shown in Fig. 6.13 are needed. When the *infixexp* code generation is called, it recursively calls code generation on the two subtrees. The subtrees in this example are the two *int* nodes in the AST, resulting in calling the code generator on *int(i)*. When code is generated for *int("5")*, line 2 looks up the index of the "5" in the constants which is a list of the function's constants much like it appears on line 2 of Fig. 6.11. Line 4 of the code generator then writes the *LOAD_CONST* instruction to the file. The recursive call of codegen on line 8 of Fig. 6.13 similarly calls the *int* codegen to generate the other *LOAD_CONST* instruction. Finally, line 10 of the code in Fig. 6.13 generates the *BINARY_ADD* instruction.

Every JCoCo program must contain code like lines 1–3 of Fig. 6.11. Likewise, lines 7–10 are needed to finish up the *main* function of every JCoCo program. Lines 1–3 are often referred to as the *prolog* of a compiled program and lines 7–10 are commonly referred to as the *epilog* of the program. The *prolog* and *epilog* code is generated by the code that calls the code generator.

The compiler starts when the *run* function is called in the code of Fig. 6.15. The run function is written with two arguments so it can be exported. Exporting a function in Standard ML causes the SML interpreter to export it and all dependent functions into an executable program that can be started from the command-line. The *run* function is like the *main* function in C or C++ program. The arguments to run include the list of command-line arguments to the program. The first item in that list is the first command-line argument. In this case that is the *filename* of the source program. The argument *a* to the *run* function is the name of the SML interpreter used to run the program.

The *run* function then calls the *compile* function passing it the *filename*. Line 2 calls the parser to parse it which returns the AST. Two output files are opened and the *termFile* is written by the *writeTerm* function. This is covered in more detail in Chap. 8.

Lines 7–16 create various bindings of identifiers to locations or functions within the JCoCo virtual machine. Lines 17–23 check for any unbound identifiers in the

```
 1 #!/bin/bash
 2 set -f
 3 export file="$1"
 4 if [ -z $file ]; then
 5   echo -n "Enter a file name: "
 6   read file
 7 fi
 8 if [ -e $file ]; then
 9   rm a.casm >& /dev/null
10   rm a.term >& /dev/null
11   echo ******* Source File ********
12   cat $file
13   sml @SMLload=mlcompimage $file
14   echo * Target Program Execution *
15   coco a.casm
16 else
17   echo FILE DOES NOT EXIST
18 fi
```

Fig. 6.14 The mlcomp script

program which are not allowed. Lines 24–35 are responsible for writing the *prolog* and generating any code for functions that are defined within the program. The *codegen* function is called on line 36. The *epilog* is written by lines 37–41.

The *run* function is invoked via the bash script *mlcomp* in Fig. 6.14. The script is invoked as *mlcomp test0.sml* for instance. The *test0.sml* command-line argument is *$1* in the code. Line 4 checkes to see if a non-empty filename was provided. If not, then lines 5–6 prompt for and get a filename from the user. Line 13 invokes the exported run function by loading the compiled image file *mlcompimage*. The compiler writes a file called *a.casm* which was opened for output on line 3 of Fig. 6.15 on page 201. Then the JCoCo virtual machine is invoked on the target program on line 15 of the script in Fig. 6.14. The *mlcomp* script both compiles and runs the intended SML program.

The Bash script of Fig. 6.16, found in *Makefile.gen*, runs Standard ML's compiler manager. It does this by starting *sml* and then executing the function *CM.make* on the file *sources.cm*. The *exportFn* function creates the binary image executable that is started on line 13 of Fig. 6.14. The target image is created by Standard ML's compiler manager. This is a tool provided with Standard ML much like the *make* utility for Unix except better because Standard ML's compiler manager figures out all dependencies by itself, without the need for a *Makefile*.

The *sources.cm* file is needed to indicate which files to include in the project. Lines 2–3 include the ML-yacc tool (which in turn include ML-lex), the basis library, and some utility code to help with debugging. The last four lines are the compiler source

```
fun compile filename  =
    let val (ast, _) = parse filename
        val outFile = TextIO.openOut("a.casm")
        val termFile = TextIO.openOut("a.term")
        val _ = writeTerm(termFile,ast)
        val _ = TextIO.closeOut(termFile)
        val consts =
            removeDups ("None"::"'Match Not Found'"::"0"::(constants ast))
        val globalBindings = [("println","print"),...]
        val (newbindings,freeVars,cells) =
            localBindings(ast,[],globalBindings,0)
        val bindingVars = removeDups (List.map (fn x => #2(x)) newbindings)
        val cellVars =
            List.map (fn x => boundTo(x,newbindings@globalBindings)) cells
        val locals = listdiff bindingVars cellVars
        val globals = removeDups (List.map (fn (x,y) => y) globalBindings)
    in
      if length(freeVars) <> 0 then
         (TextIO.output(TextIO.stdOut,
            "Error: Unbound variable(s) found in main expression => " ^
            (commaSepList freeVars) ^ ".\n");
          raise notFound)
      else ();
      TextIO.output(outFile,"Function: main/0\n");
      nestedfuns(ast,outFile,"    ",globals,[],globalBindings,0);
      TextIO.output(outFile,"Constants: "^(commaSepList consts) ^ "\n");
      if not (List.null(locals)) then
        TextIO.output(outFile,"Locals: "^(commaSepList locals) ^ "\n")
      else ();
      if not (List.null(cellVars)) then
        TextIO.output(outFile,"CellVars: "^(commaSepList cellVars) ^ "\n")
      else ();
      TextIO.output(outFile,"Globals: "^(commaSepList globals) ^ "\n");
      TextIO.output(outFile,"BEGIN\n");
      makeFunctions(ast,outFile,"    ",consts,...);
      codegen(ast,outFile,"    ",consts,...);
      TextIO.output(outFile,"    POP_TOP\n");
      TextIO.output(outFile,"    LOAD_CONST 0\n");
      TextIO.output(outFile,"    RETURN_VALUE\n");
      TextIO.output(outFile,"END\n");
      TextIO.closeOut(outFile)
    end
    handle _ => (TextIO.output(TextIO.stdOut,
                    "An error occurred while compiling!\n\n"));
 fun run(a,b::c) = (compile b; OS.Process.success)
   | run(a,b) = (TextIO.print("usage: sml @SMLload=mlcomp\n");
                  OS.Process.success)
```

Fig. 6.15 MLComp run function

```
#!/bin/bash
sml << EOF
CM.make "sources.cm";
SMLofNJ.exportFn("mlcompimage",
    mlcomp.run);
EOF
```

Fig. 6.16 Makefile.gen

```
Group is
  $/ml-yacc-lib.cm
  $/basis.cm
  $smlnj-tdp/back-trace.cm
  mlcomp.lex
  mlcomp.grm
  mlcomp.sml
  mlast.sml
```

Fig. 6.17 sources.cm

code for the *mlcomp* compiler. From this simple specification, the Standard ML Compiler Manager will run ML-lex and ML-yacc if needed and recompile only the parts of the project that have changed, just as *make* does for Unix. To make compiling the compiler even easier a *Makefile* is part of the project which simply invokes the *Makefile.gen* script.

```
make
mlcomp test0.sml
```

To compile and run the *mlcomp* compiler simply type in the mlcomp directory to compile the compiler and run the first test, test0.sml. If all succeeds there will be no errors printed and the program will print nothing to the screen, although other output will be printed like the AST and the compiled and assembled source program.

The remainder of this chapter will cover parts of the code generator that are already implemented and worth taking a look at. It will also cover parts of the compiler that are not yet implemented and suggest how they can be implemented. After working through this chapter you will have a working compiler for the Small language.

6.5 Function Calls

Running *test0.sml* is not very satisfying because no output is printed. Calling a function like *println* will print the output to the screen. The Small language includes a number of functions that can be called for input and output operations. Small differs some from Standard ML in this regard and *println* is one of those differences. Adding a *println* to *test0.sml* results in *println* $5 + 4$, the contents of a file called *test1.sml* in the *mlcomp* distribution file found on Github. Parsing the program results in the AST

```
apply(id("println"),infixexp("+",int("5"),int("4")))
```

Code generation for this program yields the program in Fig. 6.18. The code contains two additional instructions, the *LOAD_GLOBAL* and the *CALL_FUNCTION*

```
1  Function: main/0
2  Constants: None, 'Match Not Found', 0, 5, 4
3  Globals: print, fprint, input, int, len,
4           type, Exception, funlist, concat
5  BEGIN
6      LOAD_GLOBAL 0
7      LOAD_CONST 3
8      LOAD_CONST 4
9      BINARY_ADD
10     CALL_FUNCTION 1
11     POP_TOP
12     LOAD_CONST 0
13     RETURN_VALUE
14 END
```

Fig. 6.18 test1.sml JCoCo code

instructions. The code generator is called for two additional AST nodes as well, the *id* node and the *apply* node. Each of these two calls to *codegen* are provided in Fig. 6.19. The elipses (i.e. …) indicate abbreviated code. The *mlcomp.sml* file can be consulted for the full details.

When *codegen* is called a list of *globals* and *globalBindings* are provided to each call to *codegen*. The *globals* list can be seen in Fig. 6.18 on lines 3 and 4. The *env* in Fig. 6.19 is a list of bindings including a binding of the Small *println* function to a built-in function in JCoCo called *print* which does the same thing in the target language. The *env* list contains the tuple *("println", "print")*. Initially *env* and *globalBindings* are the same list (see line initial call to *codegen* in the *mlcomp.sml* file).

When *codegen* is called for the *apply* in the AST, it immediately calls *codegen* on the *id("println")*.. This results in calling the *load* function which searches all the different bindings to find the binding for *println*. It finds this in the *env* list, finds the corresponding *print* JCoCo function, and looks up *print* in the list of *globals* generating the *LOAD_GLOBAL 0* since it finds *print* in the first position in the *globals* list. The *load* function was written because the type of load necessary depends on where the identifier is found.

The call to *codegen* for *apply* first calls *codegen* to load the print function onto the stack. Then addition code is generated by the call to *codegen* on line 5 of Fig. 6.19. Finally, the *apply* codegen call generates the *CALL_FUNCTION* instruction. There is only one argument passed to any function in the Small language so the *1* is hard-coded.

```
| codegen(id(name),outFile,indent,...,globals,env,globalBindings,...) =
    load(name,outFile,indent,...,freeVars,cellVars,globals,globalBindings,env)
| codegen(apply(t1,t2),outFile,indent,...,globals,env,globalBindings,...) =
    let val _ = codegen(t1,outFile,indent,...,globals,env,globalBindings,...)
        val _ = codegen(t2,outFile,indent,...,globals,env,globalBindings,...)
    in
       TextIO.output(outFile,indent^"CALL_FUNCTION 1\n")
    end
```

Fig. 6.19 Code generation for function calls

```
let val x = 5
in
    println x
end
```

Fig. 6.20 test2.sml

Calling a function is relatively easy as shown in this example. Maintaining and understanding all the bindings is the trickier part, but further examples will serve to make this clearer as well. The next example takes a look at user-defined bindings.

6.6 Let Expressions

Let expressions provide a means for a value or function to be bound to a value. Consider the code in Fig. 6.20 that binds *x* to *5*. This SML program is compiled into the JCoCo program of Fig. 6.21. From the source program, this AST is built.

```
letdec(bindval(idpat("x"),int("5")),
       [apply(id("println"),id("x"))])
```

The AST has the new binding first, followed by the sequence of expressions between the *in* and *end* keywords. In this case there is one expression in the body of the *let* expression. Examining the code in Fig. 6.21 there are two new instructions on lines 7 and 8. These two lines take care of storing the 5 in a local variable called *x@0*. The 0 refers to the scope level of the variable is added to the variable name to be sure that variable names in JCoCo are unique. Line 10 has the *LOAD_FAST* instruction, another new instruction in the program. The *LOAD_FAST* loads from the list of *locals*. We have seen the call to the *load* function in Fig. 6.19 that loads this value from the *locals*.

The Small program contains the binding of *x* to *5*. When compiling this code, the *x* is bound to a location in the locals called *x@0* which contains the *5*. The *let expression* must create this binding to make the *x* visible in the body of the *let*

```
Function: main/0
Constants: None, 'Match Not Found',
           0, 5
Locals: x@0
Globals: print, ...
BEGIN
    LOAD_CONST 3
    STORE_FAST 0
    LOAD_GLOBAL 0
    LOAD_FAST 0
    CALL_FUNCTION 1
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Fig. 6.21 test2.sml JCoCo code

```
| codegen(letdec(d,L2),...,globals,env,globalBindings,scope) =
  let val newbindings = decgen(d,...,globals,env,globalBindings,scope)
  in
    codegenseq(L2,...,globals,newbindings@env,globalBindings,scope+1)
  end
```

Fig. 6.22 Let expression code generation

expression. It does this in the code in Fig. 6.22 by calling the function *decgen* which generates the code for storing the value in the local location and also creates a new binding *("x","x@0")*. This new binding is added to the *env* environment bindings. When the *load* function is called in the body of the *let* expression, the *x@0* will be found in the list of locals.

Building the list of *locals* for the *main* function is handled by lines 12 in Fig. 6.15. The *locals* is computed in part from the bindings computed by the *localBindings* function. The *localBindings* function traverses the body of a function looking at all identifiers found in the code. If an identifier is free in the body of a function, it is added to the *freeVars* returned by the *localBindings* function. If the identifier is bound to a value or inner function, the bindings is returned in the *newbindings*. If the code passed to the *localBindings* function contains nested functions, then the *freeVars* of those nested functions must be *cellVars* in the current function because a closure will be necessary when the inner function is called. The *localBindings* function finds those identifiers that must be bound to *cellVars* and returns them as well.

```
let val x = 5
    val y = 6
in
  println (x + y)
end
```

Fig. 6.23 test10.sml

```
| Let DecSeq In ExpSequence End
    (List.hd (List.foldr (fn (x,y) => [letdec(x,y)]) ExpSequence DecSeq))
```

Fig. 6.24 The folded let

```
let val x = 5
in
    let val y = 6
    in
        println (x+y)
    end

```

Fig. 6.25 Unsweetened

For the code in Fig. 6.20, the *newbindings* of Fig. 6.22 consist of *[("x","x@0")]* and these bindings are added to the environment *env* when the code for the body of the *let* declaration is generated. The list of the *locals* already is set to *["x@0"]* so when the *load* function is called to load the value of *x*, the combination of the environment *env* and the *locals* results in the correct index being found to generate the *STORE_FAST* and *LOAD_FAST* instructions.

With Standard ML it is possible to define more than one value in a *let* expression. Consider the program in Fig. 6.23. This program has two bindings created in one *let* expression. However, the program is not the program compiled by the *mlcomp* compiler. The parser transforms this program into a program like the one given in Fig. 6.25. The ability to write a program like Fig. 6.23 is called *syntactic sugar*. It is certainly nicer to write programs like that in Fig. 6.23 rather than being limited to one binding per let expression all the time. However, the Small abstract syntax does not include support for multiple bindings. That's what is meant by *syntactic sugar*. When a programming language feature like *let* expressions of multiple bindings is implemented in terms of some other simpler but less desirable form it is called *syntactic sugar*. The *mlcomp* compiler handles multiple bindings by using a *foldr* call to fold those multiple bindings into multiple nested *let* expressions. Figure 6.24 contains the code in the parser that forms this folded *let*.

6.7 Unary Negation

It turns out that unary negation is not implemented correctly in the *mlcomp* compiler. Presently, it is possible to print a negative 5. However, the program in Fig. 6.26 should compile and run, but instead the scanner deletes the ~ as a bad token and a 5 is printed to the screen instead. This is not the behavior of Standard ML. The tilde serves as a unary negation operator in Standard ML. To fix this, several changes are necessary. Starting with the scanner, the tilde must be recognized as its own token. To do this, the tilde is removed from the *Int* token and added as its own token in the *mlcomp.lex* file.

```
{tilde} => (Tokens.Negate(!pos,!pos));
{digit}+  => (Tokens.Int(yytext,!pos,!pos));
```

Adding the token in the parser specification is next. So the tokens are now defined as follows in *mlcomp.grm*

```
%term EOF
    | Negate
    | ...
```

Then we define the precedence of the *Negate* token in *mlcomp.grm*. Unary negation has very high precedence and is right-associative.

```
%right ListCons Negate
```

The last bit in the *mlcomp.grm* is to write a production that uses the *Negate* token. To negate an expression we just write an expression as possibly being negated as in this bit of code.

```
| Negate Exp          (negate(Exp))
```

Writing this production requires a new node definition for the AST in *mlast.sml*. A *negate* node in an AST is another kind of expression. Unary negation can be represented by defining another expression for *negate* as follows.

```
let val x = 5
in
    println ~x
end
```

Fig. 6.26 test3.sml

```
| negate of exp
```

Finally, to finish the correct implementation of unary negation, the code generator module must be modified. The *mlcomp.sml* file must be edited in a few spots to add support for unary negation. The *infixexp* expression is an AST node like the *negate* node. Searching for *infixexp* in the *mlcomp.sml* file helps determine where the changes must be made in *mlcomp.sml*. The first change is in the *nameOf* function.

```
| nameOf(infixexp(operator,e1,e2)) = operator
| nameOf(negate(e)) = "~"
```

The next match is found inside the *constants* function where this code must be added.

```
| con(infixexp(operator,t1,t2)) = (con t1) @ (con t2)
| con(negate(e)) = "0" :: (con e)
```

This code adds a zero to the list of constants. This is because to implement unary negation the generated code will subtract the value from zero. The *bindingsOf* function is the next location where *infixexp* appears in the *mlcomp.sml* file. The code to write here looks like this.

```
| bindingsOf(infixexp(operator,exp1,exp2),bindings,scope) =
      (bindingsOf(exp1,bindings,scope); bindingsOf(exp2,bindings,scope))
| bindingsOf(negate(exp),bindings,scope) = bindingsOf(exp,bindings,scope)
```

The *bindingsOf* function is looking for any new bindings introduced by the new unary negation expression. There are no new bindings created by Unary negation so it just calls the *bindingsOf* function on its sub-expression. The *codegen* function is the next place where *infixexp* is found and the following code is added to generate code for unary negation.

```
| codegen(negate(t),outFile,indent,consts,...) =
  let val _ = codegen(int("0"),outFile,indent,consts,...)
      val _ = codegen(t,outFile,indent,consts,...)
  in
    TextIO.output(outFile,indent^"BINARY_SUBTRACT\n")
  end
```

In the *codegen* function a “fake” *int(“0”)* node is created to get a zero loaded onto the stack. Then the value for the sub-expression is loaded onto the stack and the *BINARY_SUBTRACT* instruction causes the unary negation to be computed. Both the *nestedfuns* and the *makeFunctions* function need a line for unary negation added as well. In both cases the code is identical and looks like this:

```
| functions(infixexp(operator,exp1,exp2)) = (functions exp1;functions exp2)
| functions(negate(exp)) = functions exp
```

The *nestedfuns* code is looking for any nested functions within the expression. Unary negation is not a nested function so the code just calls the check by calling the *functions* function on the sub-expression. The *makeFunctions* function generates some code for any nested functions to have JCoCo create the closure or function objects for any nested functions. Finally, the *writeTerm* function must be modified. While not needed by the compiler, the *writeTerm* function is useful when reading Chap. 8. Here is the code for writing a unary negation term.

```
1  Function: main/0
2  Constants: None, 'Match Not Found', 5, 0
3  Locals: x@0
4  Globals: print, ...
5      LOAD_CONST 2
6      STORE_FAST 0
7      LOAD_GLOBAL 0
8      LOAD_CONST 3
9      LOAD_FAST 0
10     BINARY_SUBTRACT
11     CALL_FUNCTION 1
12     POP_TOP
13     LOAD_CONST 0
14     RETURN_VALUE
15 END
```

Fig. 6.27 test3.sml JCoCo code

```
| writeExp(indent,negate(exp)) =
         (print("negate(");
          writeExp(indent,exp);
          print(")"))
```

The final result of these changes is code as it appears in Fig. 6.27. The value of ~*x* is computed by subtracting from 0. The new code consists of lines 8 and 10 in the JCoCo code in Fig. 6.27.

6.8 If-Then-Else Expressions

Comparing two values in SML is as simple as writing $x < y$. In JCoCo it involves pushing two values on the operand stack and calling the *COMPARE_OP* instruction. When comparing values in an *if-then-else* expression the result of the comparison will be used to jump to one label or another. Consider the Small program in Fig. 6.28. Again, this code differs a bit from Standard ML. The *input* function is unique to Small as are the *print* and *println* functions. The *input* function returns a string of input from the user. The *print* function prints without a newline character. The *println* prints with a newline at the end of the line.

Compiling the code in Fig. 6.28 should result in the JCoCo code in Fig. 6.29. However, code generation for *if-then-else* expressions is not currently implemented. The abstract syntax tree for the program in Fig. 6.28 includes a node for the *if-then-else* expression like this.

```
ifthen(infixexp(">",id("x"),id("y")),id("x"),id("y"))
```

The AST definition for this program is already in the *mlast.sml* file and the scanner and parser are already able to parse *if-then-else* expressions. Generating code for this AST involves some of the same changes that were needed to add unary negation to the code generator. Those steps can be followed to add all the necessary code to handle *if-then-else* expressions in the code generator. By attempting to compile the code in Fig. 6.28 you will discover places in the compiler where code is missing. The compiler is written to report where code is missing. Attempt to compile *test4.sml*, see where the problem is, fix it, and repeat as many times as is necessary.

Implementing the *codegen* code is the hardest part of adding support for *if-then-else* expressions, but it's not too hard. The AST expression above has three sub-expressions: the greater than comparison, the *id("x")*, and the *id("y")*. Code generation is already done for identifiers so the *id* nodes for *x* and *y* are already handled. Generating code for the *if-then-else* expression involves generating the code for the comparison and then jumping to one place or another depending on the result of the comparison.

The *if-then-else* generated code begins on line 26 of Fig. 6.29 with the comparison code. Calling *codegen* on the infix expression generates the code on lines 26–28. Line 29 begins some of the code for the *if-then-else* expression. Line 29 begins by jumping to *L0* if the condition is false. The label *L0* labels the *else* clause of the expression. Line 30 is the code generated for the *id("x")* which is the *then* expression. Line 31 is generated by the *if-then-else* again to jump past the code in the *else* expression. Line 34 is the last bit of code generated by the *if-then-else* expression.

There are two labels needed by the code generator. The *nextLabel* function in *mlcomp.sml* is designed just for that purpose. Calling it will return a unique label that can be used in the code. Code generation for *if-then-else* expressions calls this function twice. In summary, there are several actions that must occur to generate code for *if-then-else* expressions.

```
let val x = Int.fromString(
              input("Please enter an integer: "))
    val y = Int.fromString(
              input("Please enter an integer: "))
in
  print "The maximum is ";
  println (if x > y then x else y)
end
```

Fig. 6.28 test4.sml

```
Function: main/0
Constants: None, 'Match Not Found',
  0, "Please enter an integer: ",
  "The maximum is "
Locals: y@1, x@0
Globals: print, fprint, input, int, len,
  type, Exception, funlist, concat
BEGIN
    LOAD_GLOBAL 3
    LOAD_GLOBAL 2
    LOAD_CONST 3
    CALL_FUNCTION 1
    CALL_FUNCTION 1
    STORE_FAST 1
    LOAD_GLOBAL 3
    LOAD_GLOBAL 2
    LOAD_CONST 3
    CALL_FUNCTION 1
    CALL_FUNCTION 1
    STORE_FAST 0
    LOAD_GLOBAL 1
    LOAD_CONST 4
    CALL_FUNCTION 1
    POP_TOP
    LOAD_GLOBAL 0
    LOAD_FAST 1
    LOAD_FAST 0
    COMPARE_OP 4
    POP_JUMP_IF_FALSE L0
    LOAD_FAST 1
    JUMP_FORWARD L1
L0:
    LOAD_FAST 0
L1:
    CALL_FUNCTION 1
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Fig. 6.29 test4.sml JCoCo code

- Two labels need to be created.
- The comparison code is generated.
- The *POP_JUMP_IF_FALSE* instruction is written along with the *else* clause label.
- The *then* clause code is generated.
- A jump to jump past the *else* clause code is written.
- The *else* clause label is written.
- The *else* clause code is generated.
- The final label is written to the file.

Successfully completing this code will get *if-then-else* expressions compiling correctly and *test4.sml* will run printing the maximum of two numbers entered at the keyboard.

6.9 Short-Circuit Logic

Short-circuit logic is a common feature of programming languages. If you have two boolean expressions, *E1* and *E2*, and you want to know if both are true or false there are situations where it is not necessary to test both the conditions. For instance, when testing *E1 and E2* if *E1* is false, there is no reason to evaluate *E2*. Likewise, if evaluating *E1 or E2* if *E1* is true there is no reason to evaluate *E2*. This logic is called *short-circuit logic* and is commonly used by *and* and *or* operators in programming languages. C++ and Java use this logic in their && and || operators. In Standard ML the operators are called *andalso* and *orelse* to indicate their short-circuit nature.

Neither the *andalso* or *orelse* operators are implemented in the *mlcomp* compiler. Support can be added pretty easily by following many of the steps in adding unary negation to the language. These steps include:

- Add two tokens for *andalso* and *orelse* to the scanner. Both are keywords and should be added to the keywords section of the scanner specification in *mlcomp.lex*.
- Add the tokens to the grammar specification in *mlcomp.grm* and define their precedence. Both operators have the same precedence which is at the same level as addition. They are also both left-associative.
- Add two productions to the grammar so the expressions can be parsed. The productions should return AST nodes as described next.
- Implement the code generation for these operators.

A correctly generated AST for this code will both include *infixexp* nodes like this.

```
infixexp("orelse",id("x"),
    infixexp("div",id("y"),int("0")))
infixexp("andalso",id("y"),
    infixexp("*",id("x"),int("5")))
```

The code for line 4 of Fig. 6.30 starts on line 14 of Fig. 6.31. The *print* function is loaded first. This is already implemented of course. Line 15 begins the code

```
let val x = true
    val y = false
in
  println (x orelse y div 0);
  println (y andalso x * 5)
end
```

Fig. 6.30 test5.sml

generation for the *orelse* operator. For the expression *E1 orelse E2* the code for *E1* is generated first, followed by *DUP_TOP*, *POP_JUMP_IF_TRUE*, and the *POP_TOP* instructions. The idea is if the first value is true, leave it on the stack and skip evaluating *E2*. However, if the value of *E1* is false, pop its value, and leave the value of *E2* on the stack after executing the code for *E1 orelse E2*.

A label is needed as the target for the jump instruction. The *nextLabel* function returns a unique label as was described in Sect. 6.8 on compiling *if-then-else* expressions.

The code for *andalso* appearing on lines 26–33 of Fig. 6.31 is analogous to the *orelse* code jumping if the first value is false and evaluating *E2* if *E1* is true.

The program in Fig. 6.30 is of some interest because it is not a valid Small program, yet the *mlcomp* compiler will generate code and it is possible to run the program on the JCoCo virtual machine. Since the short-circuit logic prevents the badly typed expressions from being evaluated, the error is never encountered. Chapter 8 will explore how the program in Fig. 6.30 fails to pass typechecking by looking at how the Standard ML type inference algorithm is implemented.

The difference between Python and Standard ML is that Python will allow a program like this to run as long as no run-time error occurs and Standard ML will complain that it doesn't pass type checking and will abort. Is the type inference of Standard ML better than the dynamic type checking of Python? Type inference catches many errors in logic. Debugging most Standard ML programs is trivial compared to debugging Python programs. However, passing the type checker is often more difficult and often requires tedious type conversion code. Standard ML is a bit better in that regard given its polymorphic type inference algorithm. In general, research like the Fox project at Carnegie Mellon has shown that large software systems benefit enormously from strong type checking by reducing the time it takes to test code.

The tradeoff is in convenience vs safety while writing code and the amount of time spent testing and debugging after the code is written. Standard ML is somewhat less convenient for writing, but debugging costs are negligible. Python is more convenient to write but in a large software system you might pay for it later. Other factors in language selection include appropriateness for the task at hand, whether similar code has already been written in a particular language, the existence of libraries

```
Function: main/0
Constants: None,
   'Match Not Found',
   True, False, 0, 5
Locals: y@1, x@0
Globals: print, fprint, input,
   int, len, type, Exception,
   funlist, concat
BEGIN
    LOAD_CONST 2
    STORE_FAST 1
    LOAD_CONST 3
    STORE_FAST 0
    LOAD_GLOBAL 0
    LOAD_FAST 1
    DUP_TOP
    POP_JUMP_IF_TRUE L0
    POP_TOP
    LOAD_FAST 0
    LOAD_CONST 4
    BINARY_FLOOR_DIVIDE
L0:
    CALL_FUNCTION 1
    POP_TOP
    LOAD_GLOBAL 0
    LOAD_FAST 0
    DUP_TOP
    POP_JUMP_IF_FALSE L1
    POP_TOP
    LOAD_FAST 1
    LOAD_CONST 5
    BINARY_MULTIPLY
L1:
    CALL_FUNCTION 1
    POP_TOP
    LOAD_CONST 0
    RETURN_VALUE
END
```

Fig. 6.31 test5.sml JCoCo code

providing APIs, and the availability of tools like compilers, interpreters, and IDEs (i.e. Integrated Development Environments). All these factors must be weighed to decide what language is most appropriate for a project.

6.10 Defining Functions

Function definitions in Standard ML may appear literally anywhere within the program. Functions are first class values and may appear anywhere a declaration may appear. In addition, anonymous functions may appear anywhere an expression may appear in an SML program. Not so in JCoCo. In the JCoCo virtual machine function definitions may be provided at the top level, outside any other functions, or may be nested inside another function but must be written immediately after the *Function* statement of their outer function. In addition, in JCoCo all functions must be named. There are no anonymous functions.

The *nestedfuns* function traverses an AST for an SML expression looking for any function definitions. If it finds one it generates the code for the nested function immediately. Consider the *compile* function of the *mlcomp.sml* module.

```
TextIO.output(outFile,"Function: main/0\n");
nestedfuns(ast,outFile,"   ",globals,[],globalBindings,0);
```

This code prints the *Function* statement for the *main* function. Then it immediately called the *nestedfuns* function to look for any nested functions and generate their code before continuing with the code generation for the *main* function. Again, this is the order required by the JCoCo virtual machine. When a nested function definition is found in the AST, the *nestedfun* function is called to generate the code for it. There is too much code to include here, but the *nestedfun* function gathers information about the constants, locals, cell variables, and bindings of the inner function before calling *codegen* to generate the body of it. Of course, it also looks for any nested functions inside it before continuing.

When an anonymous function is found it must be assigned a name since that is required by the JCoCo virtual machine. Naming anonymous functions occurs in the parser in the production for anonymous functions.

```
Fn MatchExp (func(nextIdNum(),MatchExp))
```

```
let fun factorial 0 = 1
      | factorial n = n * (factorial (n-1))
in
  println (factorial 5)
end
```

Fig. 6.32 test6.sml

```
Function: main/0
    Function: factorial/1
    Constants: None,
        'Match Not Found', 0, 1
    Locals: factorial@Param, n@1
    FreeVars: factorial
    Globals: print, fprint, input,
        int, len, type, Exception,
        funlist, concat
    BEGIN
        LOAD_FAST 0
        LOAD_CONST 2
        COMPARE_OP 2
        POP_JUMP_IF_FALSE L0
        LOAD_CONST 3
        RETURN_VALUE
L0:
        LOAD_FAST 0
        STORE_FAST 1
        LOAD_FAST 1
        LOAD_DEREF 0
        LOAD_FAST 1
        LOAD_CONST 3
        BINARY_SUBTRACT
        CALL_FUNCTION 1
        BINARY_MULTIPLY
        RETURN_VALUE
L1:
        LOAD_GLOBAL 6
        LOAD_CONST 1
        CALL_FUNCTION 1
        RAISE_VARARGS 1
    END
...
```

Fig. 6.33 test6.sml JCoCo code

In this code the *nextIdNum* function returns a unique integer. In the code generator this unique integer is used to form a name for the anonymous function of *anon@i* where *i* is the unique integer assigned by the parser.

Function definitions are always defined for functions of exactly one argument. Pattern matching may be used in matching the argument as it is in Standard ML. The parameter of the function is matched to each pattern in the function definition. Consider the code in Fig. 6.32. There are two patterns in the function definition, a number pattern, and an identifier pattern, which always matches. The *patMatch* function in *mlcomp.sml* takes care of generating code to match the argument to the pattern.

For the number pattern, the code on lines 12–14 of Fig. 6.33 checks to see if the number matches. If not, the code jumps to the end of its case. There is no code to check the identifier pattern matching because it always matches.

Take note of the code on lines 28–32 of Fig. 6.33. Each time the *patmatch* code is called it is passed the label of the next pattern to jump to if the current pattern does not match. In this case, the last pattern always matches, but if it hadn't the code might have jumped to *L1*. In that case, since all the patterns are exhausted at that point, an exception would be raised by the code. In this particular function, lines 28–32 are an example of *dead code*. The code will never be reached and could be removed.

The *patmatch* function matches patterns for *nil*, numbers, true or false, strings, identifiers, the :: cons operator (i.e. a non-empty list pattern), and tuples. The tuple pattern in turn matches each element of the tuple pattern to the elements of the tuple argument by calling *patmatch*.

6.10.1 Curried Functions

It was said earlier that all functions are functions of one argument in Small (and in Standard ML as well) and it's true. Curried functions are another example of *syntactic sugar*. A curried function appears to be a function of more than one argument where the arguments can be provided one at a time. The truth is that a curried function is transformed into a series of anonymous functions, each of one argument. Consider the program in Fig. 6.34. The *append* function is written in curried form. *appendOne* is a function of one argument. When the program is run they both do exactly the same thing appending two lists together. Calling *append* and *appendOne* look identical. That's because the two functions are identical. Function application is left associative so each function is applied to its first, and only, argument which returns a function that is applied to its second argument.

The *mlcomp parser* reduces curried functions like *append* to a function of one argument with one anonymous function for each of the curried arguments. This is done via a rather complex function that gathers each of the different pattern matches of a curried function and rewrites the code so that each pattern match is a pattern match of exactly one argument returning a function that takes the next argument. This function is called *uncurryIt* and is given in Fig. 6.35 on page 213.

```
let
  fun append nil L = L
    | append (h::t) L =
        h :: (append t L)

  fun appendOne x =
    (fn nil => (fn L => L)
     | h::t => (fn L =>
         h :: (appendOne t L))) x
in
  println(append [1,2,3] [4]);
  println(appendOne [1,2,3] [4])
end
```

Fig. 6.34 test7sml

```
fun uncurryIt nil = raise emptyDecList
  | uncurryIt (L as ((name,patList,exp)::t)) =
    let fun len nil = raise argumentMismatch
          | len [(n,p,e)] = length(p)
          | len ((n,p,e)::t) =
            let val size = length(p)
            in
              if size = len t then size else
                (TextIO.output(TextIO.stdOut,
                     "Syntax Error: Number of arguments does not match ...."
                 raise argumentMismatch)
            end

        val tupleList = List.map (fn x => "v"^Int.toString(nextIdNum())) patList
    in
      len(L); (* check that all patterns have same length *)
      (name,[match(idpat(hd(tupleList)),
                 List.foldr (fn (x,y) => func(nextIdNum(),[match(idpat(x), y)]))
                   (apply (func(nextIdNum(),List.map (fn (n,p,e) =>
                       match(tuplepat(p),e)) L),
                          tuplecon(List.map
                          (fn x => id(x)) tupleList))) (tl tupleList))])
    end
```

Fig. 6.35 The uncurryIt function

6.10.2 Mutually Recursive Functions

Functions in Small and SML are often recursive. Sometimes, functions may be mutually recursive as is the case in Fig. 6.36. The function *f* calls *g* and vice versa. In C++, to write two functions like this, a forward declaration is required using the function prototype for at least *g*. In Standard ML, the use of the *and* keyword between the two function definitions indicates that they are mutually recursive functions. The AST for this program is specified like this:

```
let fun f(0,y) = y
      | f(x,y) = g(x,x*y)
    and g(x,y) = f(x-1,y)
in
  println (f(10,5))
end
```

Fig. 6.36 test11.sml

```
| dec(funmatches(L)) =
  let val nameList = List.map (fn (name,matchlist) => name) L
  in
    List.map (fn (name,matchList) =>
    let val adjustedBindings =
        List.map (fn x => (x,x)) (listdiff nameList [name])
    in
      nestedfun(name,matchList,outFile,indent,globals,
          adjustedBindings@env,globalBindings,scope)
    end) L;
    ()
  end
```

Fig. 6.37 Mutually recursive function declarations

```
letdec(funmatches([funmatch("f",f's body),funmatch("g",g's body)]))
```

When a *funmatches* AST node is encountered, the bindings of all the functions in the *funmatches* list are passed to the code generation of each function. This is seen in the *nestedfuns* function when matching a declaration for a *funmatch* as shown in Fig. 6.37.

In this code the list of all function names is gathered in *nameList* and then passed to each recursive call of *nestedfun* after taking out the name of the function on which *nestedfun* is being called. Mutually recursive functions are more common than you might think. Look for uses of *and* in the *mlcomp.sml* file to see when it is needed in the implementation of the compiler.

6.11 Reference Variables

Adding variables to the Small language turns out to be almost trivial. Examining Fig. 6.38 the new code involves the *ref* keyword, the exclamation point used as the dereference operator, and the := operator (pronounced *set equal*). The scanner includes support for the dereference and the set equal operators. The *ref* will be recognized as an identifier, which turns out to be just fine.

The grammar specification in *mlcomp.grm* already has support for both the dereference and set equal operators. The productions for the two are of some interest. In Fig. 6.39 the set equal production demands that an identifier be on the left hand side. A variable cannot be an expression. If the reference variable is to point to a new value, the left hand side must name the reference variable. Yet, the AST is an *infixexp* by creating an expression node from the identifier using *id(Id)*. The dereference production is even more interesting creating a fake function application node with a *!* identifier. No production is needed for the *ref* keyword addition because the grammar already parses this as function application of the *ref* function to the value *0*.

Code generation for variables is handled by a series of special cases. The *decBindingsOf* function must be modified because binding for a variable is different than the binding for a regular identifier. The code in Fig. 6.40 must be placed before the pattern for regular identifiers.

The code in Fig. 6.40 binds the variable name to a unique identifier in the JCoCo program and it adds the variable name to the list of identifiers that will be associated with cell variables. A cell variable is a reference and variables are references in Standard ML.

The dereference operator must be handled as a special case in the *bindingsOf* function. Normally an identifier is looked up to see if it is bound or free in a function. The parser generated AST for the dereference operator makes it look like an identifier in Fig. 6.39. To handle this, the following code is a special case and must appear before the normal look up of identifiers in the *bindingsOf* function.

```
let val x = ref 0
in
  x := !x + 1;
  println (!x)
end
```

Fig. 6.38 test8.sml

```
| Exclaim Exp (apply(id("!"),Exp))
| Id SetEqual FuncExp
             (infixexp(":=",id(Id),FuncExp))
```

Fig. 6.39 Set equal and deref operators

```
and decbindingsOf(bindval(idpat(name),apply(id(l+s+s2{r}ef"),exp)),bindings,scope) =
    let val newbindings = patBindings(idpat(name),scope)
    in
      bindingsOf(exp,newbindings@bindings,scope+1);
      addIt(name,cellVars);
      [addIt((name,name^l+s+s2{@}n{^}Int.toString(scope)),theBindings)]
    end
```

Fig. 6.40 Reference variable bindings

```
1   | codegen(id(name),outFile,...) =
2     load(name,outFile,...)
3   | codegen(apply(id("ref"),t2),outFile,...) =
4     codegen(t2,outFile,...)
5   | codegen(infixexp(":=",id(name),t2),...) =
6   let val _ = codegen(t2,outFile,...)
7         val noneIndex = lookupIndex("None",consts)
8   in
9      store(name,outFile,...);
10     TextIO.output(outFile,
11         indent^"LOAD_CONST "^noneIndex^"\n")
12  end
```

Fig. 6.41 Variable code generation

```
| bindingsOf(id("!"),bindings,scope) = ()
```

Finally, code generation must be done for the *ref* declaration, the dereference operator, and the set equal operator. The *ref* code generation is another special case and must be done before normal function application. What is interesting is that all the work of code generation was actually done by the *decBindingsOf* function when the variable was added to the cell variables list. In lines 1–2 of Fig. 6.41, the code for a *ref* expression is identical to the code for a *non-reference* expression because the *store* function will find the variable in the cell variables and then generate the appropriate store instruction.

Lines 3–4 generate the code for dereferencing a variable. Indirectly, this calls *load* which will automatically generate the appropriate load instruction because the *decBindingsOf* function placed the variable in the list of cell variables. Finally, the code for the set equal operator is pretty straightforward. The *LOAD_CONST* instruction is needed because every expression in Standard ML has a result and at the end of the assignment statement the result is popped from the stack. The result of assignment is *unit* which translates to the *None* value in the JCoCo virtual machine.

When a binding to an identifier is used in an inner function, the identifier must be bound to a cell variable so a closure can be constructed when the inner function is

```
let val x = 0
    fun f y = (x:=!x+1)
in
  f 0;
  println x
end
```

Fig. 6.42 test9.sml

called. Reference variables are also bound to cell variables so they can be updated. Having two different sorts of bindings both map to the same implementation leads to some interesting possibilities in the code. Consider the program in Fig. 6.42. This program is not a legal Small program. The binding of *x* to 0 is a constant binding. It should not be possible to update the contents of the variable. However, the assignment statement on line 2 works because *x* is used in the inner function *f* and therefore is assigned to a cell variable.

The code in Fig. 6.42 is an example of when type checking is needed to prevent an illegal program from executing. The program is incorrect. The programmer made a mistake and would like to know about this mistake. Yet JCoCo doesn't care and neither does the *mlcomp* compiler. A typechecker should flag this as an error and terminate the code generator before any program is generated. This example, and the need for type checking, will be studied in more detail in Chap. 8.

6.12 Chapter Summary

The goal of the chapter was to provide an introduction to language features by studying the implementation of the Small language. Those wishing to learn more about compiler construction may want to consult a full text on the subject. For instance Aho, Sethi, and Ullman's dragon book [2]. There are many other good texts on compiler writing as well.

The case study in this chapter illustrated several features of programming languages. The implementation of functions in block structured languages is perhaps the most difficult of the concepts presented. Important concepts and skills presented in this chapter include the scope of bindings and how bindings are created, mutually recursive functions, reference variables, code generation for several language features, how to extend a language, how to use ML-lex and ML-yacc, syntactic sugar and its uses in the Small language, and short-circuit logic. Exception handling was not covered in this chapter and is a part of the mlcomp compiler.

As the need for embedded systems grows so will the demand for new programming languages targeting those platforms. The demands of a fast-paced work environment have also spurred interest in programming language design and development. This is an exciting time for experts in programming languages and this text only scratches the surface of a vast and exciting area of study.

6.13 Review Questions

1. The language of regular expressions can be used to define the tokens of a language. Give an example for a regular expression from the chapter and indicate what kind of tokens it represents.
2. What does ML-lex do? What input does it require? What does it produce?
3. Why do keywords have to be recognized by an if-else-if statement in the ML-lex definition? Why couldn't each keyword just be recognized like other fixed tokens in a language?
4. How is an abstract syntax tree declared in ML?
5. Using the grammar specification for Small, what is the AST of the following expression?

```
fun abs(x) = if x > 0 then x else ~1*x
```

6. How does the load function of the code generator decide which load instruction to generate?
7. In the code generation for function calls in Fig. 6.19, what is the purpose of the two recursive calls to *codegen*?
8. Which function in the code generator is responsible for returning the new bindings created by a *let* expression?
9. What does it mean for the *Small* language to support short-circuit logic? What happens in the code generation?
10. In Fig. 6.37 what do *nameList* and *adjustedBindings* refer to for the program given in Fig. 6.36? Give the actual contents of the three lists? Why three lists?

6.14 Exercises

1. Modify the compiler to support unary negation as described in this chapter. Upon completion *test3.sml* should compile and run correctly.
2. Add >=, <=, and <> (not equal) operators to the Small language. Provide all the pieces in all the files so programs using these operators can be compiled. Write a Small program that demonstrates that this functionality works.

3. Add support for *if-then-else* expressions to the Small compiler as described in this chapter. Follow the instructions of the chapter and be sure to test your implementation using *test4.sml*.
4. Implement short-circuit logic as described in this chapter for the *andalso* and the *orelse* operators.
5. Follow the step in this chapter to add support for compiling expressions with variables. Then, implement a *while do* loop for the *mlcomp* compiler. A while loop is written *while Exp1 do Exp2*. The *Exp1* expression is evaluated first to see if it yields true. If it does, then *Exp2* is evaluated. This repeats until *Exp2* returns false. Remember your job is to generate code for a while loop, not execute it. Use examples like adding *if-then-else* to help you determine where the changes need to be made to add support for *while do* loops. Successfully writing this code will result in successfully compiling and running test12.sml.
6. Add support for *case* expressions in the *mlcomp* Small compiler. The concrete syntax of a case statement is

```
Expression : ...
  | Case Exp Of MatchExp   (caseof(Exp,MatchExp))
```

while the abstract syntax of a case expression is given here.

```
caseof of exp * match list
```

Follow an example like adding support for unary negation to see what all is required to support the *case* expression in JCoCo. Write a program to test the use of the *case* expression in your code. There is currently no support for case expressions in the mlcomp compiler. This project will require you to add support to all facets of the compiler including the scanner, parser, and code generator. When you have successfully implemented the code to parse and compile case expressions, you will be able to compile this program which is test15.sml in the mlcomp distribution.

```
let val x = 4
in
  println
    (case x of
       1 => "hello"
     | 2 => "how"
     | 3 => "are"
     | 4 => "you")
end
```

The generated code for this program is given below. The program, when run, will print *you* to the screen.

```
Function: main/0
Constants: None, 'Match Not Found', 0, 1, "hello", 2, "how", 3, "are", 4, "you"
Locals: x@0
Globals: print, fprint, input, int, len, type, Exception, funlist, concat
BEGIN
    LOAD_CONST 9      # Here the 6 is stored in x.
    STORE_FAST 0
    LOAD_GLOBAL 0     # This is the println pushed onto stack.
    LOAD_FAST 0       # x is loaded onto stack.
    DUP_TOP           # Case expression code where x's value is duplicated.
    LOAD_CONST 3      # This is a pattern match for the first pattern.
    COMPARE_OP 2
    POP_JUMP_IF_FALSE L1
    POP_TOP           # Case expression code to pop x from stack
    LOAD_CONST 4      # This is the expression for the first match.
    JUMP_FORWARD L0   # Case expression code to jump to end of case.
L1:                   # Case expression code for label for end of first pattern.
    DUP_TOP           # Case expression code where x's value is duplicated.
    LOAD_CONST 5      # This is a pattern match for the second pattern.
    COMPARE_OP 2
    POP_JUMP_IF_FALSE L2
    POP_TOP           # Case expression code to pop x from stack
    LOAD_CONST 6      # This is the expression for the second match.
    JUMP_FORWARD L0   # Case expression code to jump to end of case.
L2:                   # Case expression code for label for end of second pattern.
    DUP_TOP           # Case expression code where x's value is duplicated.
    LOAD_CONST 7      # This is a pattern match for the third pattern.
    COMPARE_OP 2
    POP_JUMP_IF_FALSE L3
    POP_TOP           # Case expression code to pop x from stack
    LOAD_CONST 8      # This is the expression for the third match.
    JUMP_FORWARD L0   # Case expression code to jump to end of case.
L3:                   # Case expression code for label for end of third pattern.
    DUP_TOP           # Case expression code where x's value is duplicated.
    LOAD_CONST 9      # This is a pattern match for the fourth pattern.
    COMPARE_OP 2
    POP_JUMP_IF_FALSE L4
    POP_TOP           # Case expression code to pop x from stack
    LOAD_CONST 10     # This is the expression for the fourth match.
    JUMP_FORWARD L0   # Case expression code to jump to end of case.
L4:                   # Case expression code for label for end of fourth pattern.
L0:                   # This is the end of case expression label.
    CALL_FUNCTION 1   # print the result which was left on the stack
    POP_TOP           # Pop the None left by println
    LOAD_CONST 0      # Push a None to return
    RETURN_VALUE      # Return the None
END
```

7. The following program does not compile correctly using the mlcomp compiler and type inference system. However, it is a valid Standard ML program. Modify the mlcomp compiler to correctly compile this program.

```
let val [(x,y,z)] = [(1+s+s2{h}ellop{,}1,true)] in println x end
```

8. Currently, the abstract syntax and parser of *Small* includes support for the wildcard pattern in pattern matching, but the code generator does not support it. Add support for wildcard patterns, write a test program, and test the compiler and code generation.
9. Currently, the abstract syntax and parser of *Small* includes support for the *as* pattern in pattern matching, but the code generator does not support it. Add support for *as* patterns, write a test program, and test the compiler and code generation. The *as* pattern comes up when you write a pattern like *L as h::t* which assigns *L* as a pattern that represents the same value as the compound pattern of *h::t*.

6.15 Solutions to Practice Problems

These are solutions to the practice problem s. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

6.15.1 Solution to Practice Problem 6.1

The keywords *case* and *of* must be added to the scanner specification in *mlcomp.lex*. All the other tokens are already available in the scanner.

6.15.2 Solution to Practice Problem 6.2

You need to add a new AST node type.

```
| caseof of exp * match list
```

6.15.3 Solution to Practice Problem 6.3

The grammar changes required for case expressions are as follows.

```
Expression : ...
  | Case Exp Of MatchExp  (caseof(Exp,MatchExp))
```

Logic Programming 7

Imperative programming languages reflect the architecture of the underlying von Neumann stored program computer: Programs update memory locations under the control of instructions. Execution is (for the most part) sequential. Sequential execution is governed by a program counter. Imperative programs are prescriptive. They dictate precisely how a result is to be computed by means of a sequence of statements to be performed by the computer. Consider this program using the Small language developed in Chap. 6.

What do we want to know about the program in Fig. 7.1? Are we concerned with a detailed description of what happens when the computer runs this? Do we want to know what the PC is set to when the program finishes? Are we interested in what is in memory location 13 after the second iteration of the loop? These questions are not ones that need to be answered. They don't tell us anything about what the program does.

Instead, if we want to understand the program we want to be able to describe the relationship between the input and the output. The output is the remainder after dividing the first input value by the second input. If this is what we are really concerned about then why not program by describing relationships rather than prescribing a set of steps. In Logic Programming the programmer describes the logical structure of a problem rather than prescribing how a computer is to go about solving it. Languages for Logic Programming are called:

- **Descriptive languages**: Programs are expressed as known facts and logical relationships about a problem. Programmers assert the existence of the des ired result and a logic interpreter then uses the computer to find the desired result by making inferences to prove its existence.
- **Nonprocedural languages**: The programmer states only what is to be accomplished and leaves it to the interpreter to determine how it is to be accomplished.

```
let val m = ref 0
    val n = ref 0
in
  m:=Int.fromString(
      input("Please enter an integer: "));
  n:=Int.fromString(
      input("lease enter another: "));
  while !m >= !n do m:=!m-!n;
  println(!m)
end
```

Fig. 7.1 A small sample

- **Relational languages**: Desired results are expressed as relations or predicates instead of as functions. Rather than define a function for calculating a square root, the programmer defines a relation, say $sqrt(x, y)$, that is true exactly when $y^2 = x$.

While there are many application specific logic programming languages, there is one language that stands out as a general purpose logic programming language. Prolog is the language that is most commonly associated with logic programming. The model of computation for Prolog is not based on the Von Neumann architecture. It's based on the mechanism in logic called unification. Unification is the process where variables are unified to terms.

This text has explored a variety of languages from the JCoCo assembly language, to Java and C++, to Standard ML, and now Prolog. These languages reflect a continuum from prescriptive languages to descriptive languages.

- Assembly language is a very prescriptive language, meaning that you must think in terms of the particular machine and solve problems accordingly. Programmers must think in terms of the von Neumann machine stored program computer model.
- C++ is a high-level language and hence allows you to think in a more descriptive way about a problem. However, the underlying computational model is still the von Neumann machine.
- Standard ML is a high-level language too, but allows the programmer to think in a mathematical way about a problem. This language gets away from the traditional von Neumann model in some ways.
- Prolog takes the descriptive component of languages further and lets programmers write programs based solely on describing relationships.

Prolog was developed in 1972. Alain Colmerauer, Phillipe Roussel, and Robert Kowalski were key players in the development of the Prolog language. It is a surprisingly small language with a lot of power. The Prolog interpreter operates by doing a depth first search of the search space while unifying terms to try to come to a conclusion about a question that the programmer poses to the interpreter. The programmer describes facts and relationships and then asks questions.

This simple model of programming has been used in a wide variety of applications including automated writing of real estate advertisements, an application that writes legal documents in multiple languages, another that analyzes social networks, and a landfill management expert system. This is only a sampling of the many, many applications that have been written using this simple but powerful programming model.

7.1 Getting Started with Prolog

If you don't already have a Prolog interpreter, you will want to download one and install it. There are many versions of Prolog available. Some are free and some are not. The standard free implementation is available at http://www.swi-prolog.org. There are binary distributions available for Microsoft Windows, Mac OS X, and Linux, so there should be something to suit your needs.

Unlike SML, there is no way to write a program interactively with Prolog. Instead, you write a text file, sometimes called a database, containing a list of facts and predicates. Then you start the Prolog interpreter, consult the file, and ask yes or no questions that the Prolog interpreter tries to prove are true.

To start the Prolog interpreter you type either *pl* or *swipl* depending on your installation of SWI Prolog. To exit the interpreter type a *ctl-d*. A Prolog program is a database of facts and predicates that can be used to establish further relationships among those facts. A predicate is a function that returns true or false. Prolog programs describe relationships. A simple example is a database of facts about several people in an extended family and the relationships between them as shown in Fig. 7.2. Questions we might ask:

1. Is Gary's father Sophus?
2. Who are Kent's fathers?
3. For who is Lars a father?

These questions can all be answered by Prolog given the database in Fig. 7.2.

```
parent(fred, sophusw). parent(fred, lawrence).
parent(fred, kenny). parent(fred, esther).
parent(inger,sophusw). parent(johnhs, fred).
parent(mads,johnhs). parent(lars, johan).
parent(johan,sophus). parent(lars,mads).
parent(sophusw,gary). parent(sophusw,john).
parent(sophusw,bruce). parent(gary, kent).
parent(gary, stephen). parent(gary,anne).
parent(john,michael). parent(john,michelle).
parent(addie,gary). parent(gerry, kent).
male(gary). male(fred).
male(sophus). male(lawrence).
male(kenny). male(esther).
male(johnhs). male(mads).
male(lars). male(john).
male(bruce). male(johan).
male(sophusw). male(kent).
male(stephen). female(inger).
female(anne). female(michelle).
female(gerry). female(addie).
father(X,Y):-parent(X,Y),male(X).
mother(X,Y):-parent(X,Y), female(X).
```

Fig. 7.2 The family tree

7.2 Fundamentals

Prolog programs (databases) are composed of facts. Facts describe relationships between terms. Simple terms include numbers and atoms. Atoms are symbols like *sophus* that represent an object in our universe of discourse. Atoms MUST start with a small letter. Numbers start with a digit and include both integers and real numbers. Real numbers are written in scientific notation. For instance, 3.14159e0 or just 3.14159 when the exponent is zero.

A predicate is a function that returns true or false. Predicates are defined in Prolog by recording a fact or facts about them. For instance, Fig. 7.2 establishes the fact that Johan was the parent of Sophus. *parent* is a predicate representing a true fact about the relationship of *johan* and *sophus*.

Frequently terms include variables in predicate definitions to establish relationships between groups of objects. A variable starts with a capital letter. Variables are used to establish relationships between classes of objects. For instance, to be a father means that you must be a parent of someone and be male. In Fig. 7.2 the *father* predicate is defined by writing

```
father(X,Y):-parent(X,Y), male(X).
```

which means *X* is the *father* of *Y* if *X* is the *parent* of *Y* and *X* is *male*. The symbol :− is read as *if* and the comma in the predicate definition is read as *and*. So X is a father of Y *if* X is a parent of Y *and* X is male.

Practice 7.1 What are the terms in Fig. 7.2? What is the difference between an atom and a variable? Give examples of terms, atoms, and variables from Fig. 7.2.

You can check your answer(s) in Section 7.17.1.

To program in Prolog the programmer first writes a database like the one in Fig. 7.2. Then the programmer consults the database so the Prolog interpreter can internally record the facts that are written there. Once the database has been consulted, questions can be asked about the database. Questions asked of Prolog are limited to yes or no questions that are posed in terms of the predicates in the database. A question posed to Prolog is sometimes called a query. To discover if Johan is the father of Sophus you start Prolog using *pl* or *swipl*, then consult the database, and pose the query.

```
% swipl
?- consult('family.prolog').
?- father(johan,sophus).
Yes
?-
```

Queries may also contain variables. If we want to find out who the father of sophus is we can ask that of Prolog by replacing the father position in the predicate with a variable. When using a variable in a query Prolog will answer yes or no. If the answer is yes, Prolog will tell us what the value of the variable was when the answer was yes. If there is more than one way for the answer to be yes then typing a semicolon will tell Prolog to look for other values where the query is true.

```
?- father(X, sophus).
X = johan
Yes
?- parent(X,kent).
X = gary ;
X = gerry ;
No
?-
```

The final *No* is Prolog telling us there are no other ways for *parent(X,kent)* to be true.

7.3 The Prolog Program

Prolog performs *unification* to search for a solution. Unification is simply a list of substitutions of terms for variables. A query of the database is matched with its predicate definition in the database. Terms in the query are matched when a suitable pattern is found among the parameters of a predicate in the database. If the matched predicate is dependent on other predicates being true, then those queries are posed to the Prolog interpreter. This process continues until either Prolog finds that no substitution will satisfy the query or it finds a suitable substitution.

Prolog uses depth first search with backtracking to search for a valid substitution. In its search for truth it will unify variables to terms. Once a valid substitution is found it will report the substitution and wait for input. In Sect. 7.2 the interpreter reports that $X = gary$ is a substitution that makes *parent(X,kent)* true. Prolog waits until either *return* is pressed or a semicolon is entered. When the semicolon is entered, Prolog undoes the last successful substitution it made and continues searching for another substitution that will satisfy the query. In Sect. 7.2 Prolog reports that $X = gerry$ will satisfy the query as well. Pressing semicolon one more time undoes the $X = gerry$ substitution, Prolog continues its depth first search looking for another substitution, finds none, and reports *No* indicating that the search has exhausted all possible substitutions.

Unification finds a substitution of terms for variables or variables for terms. Unification is a symmetric operation. It doesn't work in only one direction. This means (among other things) that Prolog predicates can run backwards and forwards. For instance, if you want to know who Kent's dad is you can ask that as easily as who is Gary the father of. In the following example we find out that *gary* is the father of *kent*. We also find out who *gary* is the father of.

```
?- father(X,kent).
X = gary ;
No
?- father(gary,X).
X = kent ;
X = stephen ;
X = anne ;
No
```

Practice 7.2 Write predicates that define the following relationships.

1. brother
2. sister
3. grandparent
4. grandchild

Depending on how you wrote grandparent and grandchild there might be something to note about these two predicates. Do you see a pattern? Why?

You can check your answer(s) in Section 7.17.2.

7.4 Lists

Prolog supports lists as a data structure. A list is constructed the same as in ML. A list may be empty which is written as [] in Prolog. A non-empty list is constructed from an element and a list. The construction of a list with head, H, and tail, T, is written as[H | T]. So, [1,2,3] can also be written as [1 | [2 | [3 | []]]]. The list [a | []] is equivalent to writing [a]. Unlike ML, lists in Prolog do not need to be homogeneous. So [1, hi, 4.3] is a valid Prolog list.

By virtue of the fact that Prolog's algorithm is depth first search combined with unification, Prolog naturally does pattern matching. Not only does [H | T] work to construct a list, it also works to match a list with a variable. Append can be written as a relationship between three lists. The result of appending the first two lists is the third argument to the append predicate. The first fact below says appending the empty list to the front of *Y* is just *Y*. The second fact says that appending a list whose first element is *H* to the front of *L2* results in *[H|T3]* when appending *T1* and *L2* results in *T3*.

```
append([],Y,Y).
append([H|T1], L2, [H|T3]) :- append(T1,L2,T3).
```

Try out append both backwards and forwards! The definition of *append* can be used to define a predicate called *sublist* as follows:

```
sublist(X,Y) :- append(_,X,L), append(L,_,Y).
```

Stated in English this says that *X* is a sublist of *Y* if you can append something on the front of *X* to get *L* and something else on the end of *L* to get *Y*. The underscore is used in predicate definitions for values we don't care about.

To prove that *sublist([1],[1,2])* is true we can use the definition of *sublist* and *append* to find a substitution for which the predicate holds. Figure 7.3 provides a proof that *[1]* is a sublist of *[1,2]*.

Practice 7.3 What is the complexity of the append predicate? How many steps does it take to append two lists?

You can check your answer(s) in Section 7.17.3.

Practice 7.4 Write the reverse predicate for lists in Prolog using the append predicate. What is the complexity of this reverse predicate?

You can check your answer(s) in Section 7.17.4.

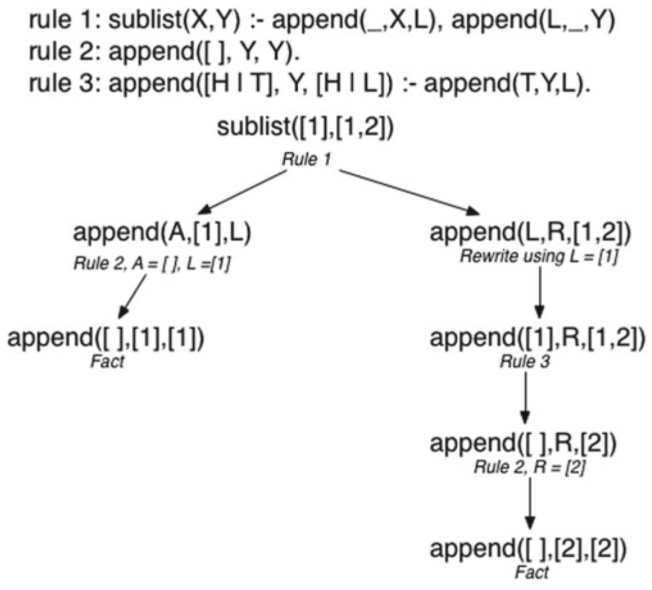


Fig. 7.3 A unification tree

7.5 The Accumulator Pattern

The slow version of reverse from practice problem 7.4 can be improved upon. The accumulator pattern can be applied to Prolog as it was in SML. Looking back at the solution to *practice problem* 5.17, the ML solution can be rewritten to apply to Prolog as well. In the ML version an accumulator argument was added to the function that allowed the *helprev* helper function to accumulate the reversed list without the use of append.

```
fun reverse(L) =
    let fun helprev (nil, acc) = acc
          | helprev (h::t, acc) = helprev(t,h::acc)
    in
      helprev(L,[])
    end
```

Unlike SML, Prolog does not have any facility for defining local functions with limited scope. If using helper predicates in a Prolog program the user and/or programmer must be trusted to invoke the correct predicates in the correct way.

Applying what we learned from the ML version of reverse to Prolog results in a helprev predicate with an extra argument as well. In many ways this is the same function rewritten in Prolog syntax. The only trick is to remember that you don't write functions in Prolog. Instead, you write predicates. Predicates are just like functions with an extra parameter. The extra parameter establishes the relationship between the input and the output.

Sometimes in Prolog it is useful to think of input and output parameters. For instance, with append defined as a predicate it might be useful to think of the first two parameters as input values and the third as the return value. While as a programmer it might sometimes be useful to think this way, this is not how Prolog works. As was shown in Sect. 7.4, append works both backwards and forwards. But, thinking about the problem in this way may help identifying a base case or cases. When the base cases are identified, the problem may be easier to solve.

Practice 7.5 Write the reverse predicate using a helper predicate to make a linear time reverse using the accumulator pattern.

You can check your answer(s) in Section 7.17.5.

7.6 Built-In Predicates

Prolog offers a few built in predicates. The relational operators ($<$, $>$, $<=$, $>=$, and $=$) all work on numbers and are written in infix form. Notice that not equals is written as \= in Prolog.

To check that a predicate doesn't hold, the *not* predicate is provided. Preceding any predicate with *not* insists the predicate returns false. For instance, *not(5 > 6)* returns true because *5 > 6* returns false.

The *atom* predicate returns true if the argument is an atom. So *atom(sophus)* is true but *atom(5)* is not. The *number* predicate returns true if the argument is a number. So *number(5)* is true but *number(sophus)* is not.

7.7 Unification and Arithmetic

The Prolog interpreter does a depth first search of the search space while unifying variables to terms. The primary operation that Prolog carries out is unification. Unification can be represented explicitly in a Prolog program by using the equals (i.e. =) operator. When equals is used, Prolog attempts to unify the terms that appear on each side of the operator. If they can be unified, Prolog reports yes and continues unifying other terms to try to find a substitution that satisfies the query. If no substitution is possible, Prolog will report no.

You might have caught yourself wanting to write something like $X = Y$ in some of the practice problem s. This is normal, but is the sign of a novice Prolog programmer. Writing $X = Y$ in a predicate definition is never necessary. Instead, everywhere Y appears in the predicate, write X instead.

Unification has one other little nuance that most new Prolog programmers miss. There is no point in unifying a variable to a term if that variable is used only once in a predicate definition. Unification is all about describing relationships. Unification doesn't mean much when a variable is not used in more than one place in a definition. In terms of imperative programming it's kind of like storing a value in a variable and then never using the variable. What's the point? Prolog warns us when we do this by saying

```
Singleton variables: [X]
```

If this happens, look for a variable called *X* (or whatever the variable name is) that is used only once in a predicate definition and replace it with an underscore (i.e. _). An underscore indicates the result of unification in that position of a predicate isn't needed by the current computation. Prolog warns you of singleton variables because they are a sign that there may be an error in a predicate definition. If an extra variable exists in a predicate definition it may never be instantiated. If that is the case, the predicate will always fail to find a valid substitution. While singleton variables should be removed from predicate definitions, the message is only a warning and does not mean that the predicate is wrong.

The use of equality for unification and not for assignment statements probably seems a little odd to most imperative programmers. The equals operator is not the assignment operator in Prolog. It is unification. Assignment and unification are different concepts. Writing $X = 6 * 5$ in Prolog means that the variable X must be equal to the term $6 * 5$, not 30. The equals operator doesn't do arithmetic in Prolog. Instead, a special Prolog operator called *is* is used. To compute $6 * 5$ and assign the result to the variable X the Prolog programmer writes *X* is $6 * 5$ as part of a predicate. Using the *is* operator succeeds when the variable on the left is unbound and the expression on the right doesn't cause an exception when computed. All values on the right side of the *is* predicate must be known for the operation to complete successfully. Arithmetic can only be satisfied in one direction, from left to right. This means that predicates involving arithmetic can only be used in one direction, unlike the append predicate and other predicates that don't involve arithmetic.

Practice 7.6 Write a length predicate that computes the length of a list.
You can check your answer(s) in Section 7.17.6.

7.8 Input and Output

Prolog programs can read from standard input and write to standard output. Reading input is a side-effect so it can only be satisfied once. Once read, it is impossible to unread something. The most basic predicates for getting input are *get_char(X)* which instantiates *X* to the next character in the input (whatever it is) and *get(X)* which instantiates *X* to the next non-whitespace character. The *get_char* predicate instantiates *X* to the character that was read. The *get* predicate instantiates *X* to the ASCII code of the next character.

There is also a predicate called *read(X)* which reads the next term from the input. When *X* is uninstantiated, the next term is read from the input and *X* is instantiated with its value. If *X* is already instantiated, the next term is read from the input and Prolog attempts to unify the two terms.

As a convenience, there are certain libraries that also may be provided with Prolog. The *readln* predicate may be used to read an entire line of terms from the keyboard, instantiating a variable to the list that was read. The *readln* predicate has several arguments to control how the terms are read, but typically it can be used by writing *readln(L, _, _, _, lowercase).*

```
? - readln(L,_,_,_,lowercase).
```

Reading input from the keyboard, no matter which predicate is used, causes Prolog to prompt for the input by printing a **|:** to the screen. If the *readln* predicate is invoked as shown above, entering the text below will instantiate *L* to the list as shown.

```
|: + 5 S R
L = [+, 5, s, r] ;
No
?-
```

The *print(X)* predicate will print a term to the screen in Prolog. The value of its argument must be instantiated to print it. Print always succeeds even if the argument is an uninstantiated variable. However, printing an uninstantiated variable results in the name of the variable being printed which is probably not what the programmer wants. When a query is made in Prolog, each variable is given a unique name to avoid name collisions with other predicates the query may be dependent on. Prolog assigns these unique names and they start with an underscore character. If an uninstantiated variable is printed, you will see it's Prolog assigned unique name.

```
?- print(X).
_G180
X = _G180 ;
No
```

The *print* predicate is satisfied by unifying the variable with the name of Prolog's internal unique variable name which is almost certainly not what was intended. The *print* predicate should never be invoked with an uninstantiated variable.

7.9 Structures

Prolog terms include numbers, atoms, variables and one other important type of term called a structure. A structure in Prolog is like a datatype in SML. Structures are recursive data structures that are used to model structured data. Computer scientists typically call this kind of structured data a tree because they model recursive,

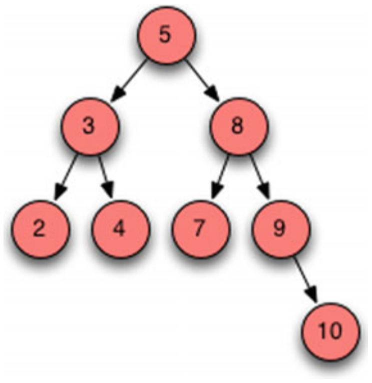


Fig. 7.4 Search tree

hierarchical data. A structure is written by writing a string of characters preceding a tuple of some number of elements. Consider implementing a lookup predicate for a binary search tree in Prolog. A tree may be defined recursively as either *nil* or a *btnode(Val, Left, Right)* where *Val* is the value stored at the node and *Left* and *Right* represent the left and right binary search trees. The recursive definition of a binary search tree says that all values in the left subtree must be less than *Val* and all values in the right subtree must be greater than *Val*. For this example, let's assume that binary search trees don't have duplicate values stored in them.

A typical binary search tree structure might look something like the term below and corresponds to the tree shown graphically in Fig. 7.4.

```
btnode(5,
  btnode(3,
    btnode(2, nil, nil),
    btnode(4, nil, nil)),
  btnode(8,
    btnode(7, nil, nil),
    btnode(9, nil,
      btnode(10, nil, nil))))
```

Items may be inserted into and deleted from a binary search tree. Since Prolog programmers write predicates, the code to insert into and delete from a binary search tree must reflect the before and after picture. Because a binary search tree is recursively defined, each part of the definition will be part of a corresponding case for the insert and delete predicates. So, inserting into a search tree involves the value to insert, the tree before it was inserted, and the tree after it was inserted. Similarly, a delete predicate involves the same three arguments.

Looking up a value in a binary search tree results in a true or false response, which is the definition of a predicate. Writing a lookup predicate requires the value and the search tree in which to look for the value.

Practice 7.7 Write a lookup predicate that looks up a value in a binary search tree like the kind defined in this section.

You can check your answer(s) in Section 7.17.7.

7.10 Parsing in Prolog

As mentioned earlier in the text, Prolog originated out of Colmerauer's interest in using logic to express grammar rules and to formalize the parsing of natural language sentences. Kowalski and Comerauer solved this problem together and Colmerauer figured out how to encode the grammar as predicates so sentences could be parsed efficiently. The next sections describe the implementation of parsing Colmerauer devised in 1972. Consider the following context-free grammar for English sentences.

Sentence ::= Subject Predicate .
Subject ::= Determiner Noun
Predicate ::= Verb | Verb Subject
Determiner ::= a | the
Noun ::= professor | home | group
Verb ::= walked | discovered | jailed

Given a sequence of tokens like "the professor discovered a group.", Chap. 2 showed that a parse tree can be used to demonstrate that a string is a sentence in the language and at the same time displays its syntactic structure.

Practice 7.8 Construct the parse tree for "the professor discovered a group." using the grammar in this section.

You can check your answer(s) in Section 7.17.8.

Prolog is especially well suited to parse sentences like the one in practice problem 6.8. The language has built in support for writing grammars and will automatically generate a parser given the grammar of a language. How Prolog does this is not intuitively obvious. The grammar is taken through a series of transformations that produce the parser. The next few pages present these transformations to provide insight into how Prolog generates parsers.

Parsing in Prolog requires the source program, or sentence, be scanned as in the parser implementations presented in Chaps. 2 and 3. The *readln* predicate will suffice to read a sentence from the keyboard and scan the tokens in it. Using the *readln* predicate to read the sentence, "the professor discovered a group.", produces the list [the, professor, discovered, a, group,'.'].

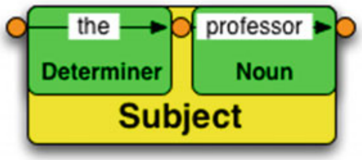


Fig. 7.5 Sentence structure

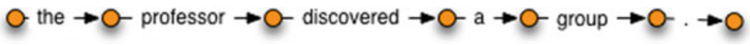


Fig. 7.6 A sentence graph

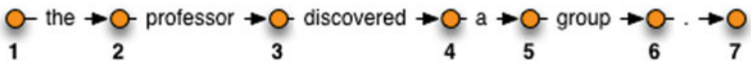


Fig. 7.7 A labeled sentence graph

A Prolog parser is a top-down or recursive-descent parser. Because the constructed parser is top-down, the grammar must be LL(1). There cannot be any left-recursive productions in the grammar. Also, because Prolog uses backtracking, there cannot be any productions in the grammar with common prefixes. If there are any common prefixes, left factorization must be performed. Fortunately, the grammar presented in this section is already LL(1).

The Prolog parser will take the list of tokens and produce a Prolog structure. The structure is the Prolog representation of the abstract syntax tree of the sentence. For instance, the sentence, "the professor discovered a group.", when parsed by Prolog, yields the term sen(sub(det(the), noun(professor)), pred(verb(discovered), sub(det(a), noun(group)))).

The logic programming approach to analyzing a sentence in a grammar can be viewed in terms of a graph whose edges are labeled by the tokens or terminals in the language. Figure 7.6 contains a graph representation of a sentence. Two terminals are contiguous in the original string if they share a common node in the graph.

A sequence of contiguous labels constitutes a nonterminal if the sequence corresponds to the right-hand side of a production rule in the grammar. The contiguous sequence may then be labeled with the nonterminal. In Fig. 7.5 three nonterminals are identified. To facilitate the representation of graphs like Fig. 7.6 in Prolog the nodes of the graph are given labels. Positive integers are convenient labels to use as shown in Fig. 7.7.

The graph for the sentence can be represented in Prolog by entering the following facts. These predicates reflect the end points of their corresponding labeled edge in the graph.

```
the(1,2).
professor(2,3).
discovered(3,4).
a(4,5).
```

```
group(5,6).
period(6,7).
```

Using the labeled graph in Fig. 7.7, nonterminals in the grammar can be represented by predicates. For instance, the subject of a sentence can be represented by a *subject* predicate. The *subject(K,L)* predicate means that the path from node K to node L can be interpreted as an instance of the subject nonterminal.

For example, subject(4, 6) should return true because edge (4, 5) is labeled by a determiner "a" and edge (5, 6) is labeled by the noun "group'‘. To define a sentence predicate there must exist a determiner and a noun. The rule for the sentence predicate is

```
subject(K,L) :- determiner(K,M), noun(M,L).
```

The common variable *M* insure the determiner immediately precedes the noun.

Practice 7.9 Construct the predicates for the rest of the grammar.
You can check your answer(s) in Section 7.17.9.

The syntactic correctness of the sentence, "the professor discovered a group." can be determined by either of the following queries

```
?- sentence(1,7).
yes
? - sentence(X,Y).
X = 1
Y = 7
```

The sentence is recognized by the parser when the paths in the graph corresponding to the nonterminals in the grammar are verified. If eventually a path for the sentence nonterminal is found then the sentence is valid. The paths in the graph of the sentence are shown in Fig. 7.8. Note the similarity of the structure exhibited by the paths in the graph with the tree of the sentence. If you use your imagination a bit you can see the parse tree upside down (or right-side up for your non-programming friends).

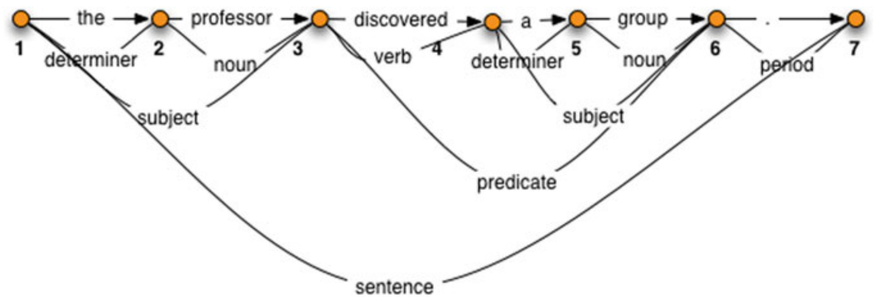


Fig. 7.8 An upside down parse tree

7.10.1 Difference Lists

There are a couple of problems with the development of the parser above. First, entering the sentence as facts like *the(1,2)* and *professor(2,3)* is impractical and awkward. There would have to be some preprocessing on the list to get it in the correct format to be parsed. While this could be done, a better solution exists. The other problem concerns what the parser does. So far the parser only recognizes a syntactically valid sentence and does not produce a representation of the abstract syntax tree for the sentence.

Labeling the nodes of the graph above with integers was an arbitrary decision. The only requirement of labeling nodes in the graph requires that it be obvious when two nodes in the graph are connected. Both problems above can be solved by letting sublists of the sentence label the graph instead of labeling the nodes with integers. These sublists are called difference lists. A difference list represents the part of the sentence that is left to be parsed. The difference between two adjacent nodes is the term which labels the intervening edge. The difference list representation of the graph is shown in Fig. 7.9. Using difference lists, two nodes are connected if their difference lists differ by only one element. This connection relationship can be expressed as a Prolog predicate.

This is the connect predicate and the grammar rewritten to use the connect predicate.

```
c([H|T],H,T).
```

The *c* (i.e. connect) predicate says that the node labeled *[H|T]* is connected to the node labeled *T* and the edge connecting the two nodes is labeled *H*. This predicate can be used for the terminals in the grammar in place of the facts given above.

```
determiner(K,L) :- c(K,a,L).
determiner(K,L):- c(K,the,L).

noun(K,L) :- c(K,professor,L).
noun(K,L) :- c(K,home,L).
noun(K,L) :- c(K,group,L).

verb(K,L) :- c(K,walked,L).
verb(K,L) :- c(K,discovered,L).
verb(K,L) :- c(K,jailed,L).
```

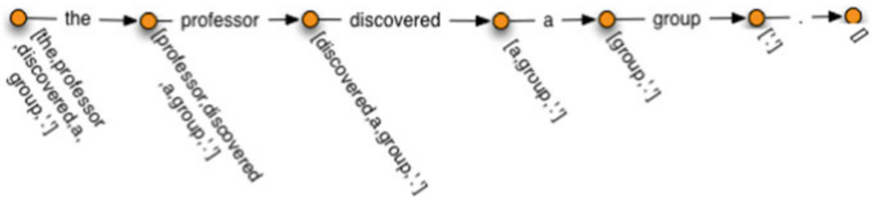


Fig. 7.9 Difference lists

The graph need not be explicitly created when this representation is employed. The syntactic correctness of the sentence, “the professor discovered a group.” can be recognized by the following query.

```
?- sentence([the,professor,discovered,a,group,'.'], [ ]).
yes
```

The parsing succeeds because the node labeled with [the, professor, discovered, a, group, ‘.’] can be joined to the node labeled with [] via the intermediate nodes involved in the recursive descent parse of the sentence. Because Prolog predicates work backwards as well as forward, it is just as easy to explore all the sentences of this grammar by posing this query to the Prolog interpreter.

```
?- sentence(S,[ ]).
```

This reveals that there are 126 different sentences defined by the grammar. Some of the sentences are pretty non-sensical like “the group discovered a group.”. Some of the sentences like “the group jailed the professor.” have some truth to them. Sophus Lie used to walk to many of the places he visited partly because he liked to walk and partly because he had little money at the time. He also liked to draw sketches of the countryside when hiking. He was jailed in France when France and Germany were at war because the French thought he was a German spy. It was understandable since he was walking through the countryside talking to himself in Norwegian (which the French thought might be German). When they stopped to question him, they found his notebook full of Mathematical formulas and sketchings of the French countryside. He spent a month in prison until they let him go. While in prison he read and worked on his research in Geometry. Of his prison stay he later commented, “I think that a Mathematician is comparatively well suited to be in Prison.”[20]. Other mathematicians may not agree with his assessment of the mathematical personality.

Some care must be taken when asking for all sentences of a grammar. If the grammar contained a recursive rule, say

```
Subject ::= Determiner Noun | Determiner Noun Subject
```

then the language would allow infinitely many sentences, and the sentence generator will get stuck with ever lengthening subject phrases.

7.11 Prolog Grammar Rules

Most implementations of Prolog have a preprocessor which translates grammar rules into Prolog predicates that implement a parser of the language defined by the grammar. The grammar of the English language example takes the following form as a logic grammar in Prolog.

```
sentence --> subject, predicate,['.'].
subject --> determiner, noun.
predicate --> verb, subject.
determiner --> [a].
determiner --> [the].
noun --> [professor]; [home]; [group].
verb --> [walked]; [discovered]; [jailed].
```

Note that terminal symbols appear inside brackets exactly as they look in the source text. Since they are Prolog atoms, tokens starting with characters other than lower case letters must be placed within apostrophes. The Prolog interpreter automatically translates these grammar rules into normal Prolog predicates identical to those defining the grammar presented in the previous section.

7.12 Building an AST

The grammar given above is transformed by a preprocessor to generate a Prolog parser. However, in its given form the parser will only answer yes or no, indicating the sentence is valid or invalid. Programmers also want an abstract syntax tree if the sentence is valid. The problem of producing an abstract syntax tree as a sentence is parsed can be handled by using parameters in the logic grammar rules.

Predicates defined using Prolog grammar rules may have arguments in addition to the implicit ones created by the preprocessor. These additional arguments are inserted by the translator to precede the implicit arguments. The grammar rule

```
sentence(sen(N,P)) --> subject(N), predicate(P), ['.'].
```

will be translated into the Prolog rule

```
sentence(sen(N,P),K,L) :- subject(N,K,M),
                              predicate(P,M,R),c(R,'.',L).
```

A query with a variable representing a tree produces that tree as its answer.

```
?- sentence(Tree, [the,professor,discovered,a,group,'.'],[]).
Tree = sen(sub(det(the),noun(professor)),
                pred(verb(discovered),sub(det(a),noun(group))))
```

Practice 7.10 Write a grammar for the subset of English sentences presented in this text to parse sentences like the one above. Include parameters to build abstract syntax trees like the one above.

You can check your answer(s) in Section 7.17.10.

Writing an interpreter or compiler in Prolog is relatively simple given the grammar for the language. Once the AST has been generated for an expression in the language the back end of the interpreter or compiler proceeds much like it does in other languages.

7.13 Attribute Grammars

Programming language syntax is specified by formal methods like grammars. Semantics, or the meaning of a computer program, are much harder to define. The study of formal methods of specifying the meaning, or *semantics*, of a program is a difficult but rewarding area of Computer Science. In Chap. 6 a compiler for the Small language was developed. Mapping the Small language into the language of JCoCo is a way of defining the semantics of Small. Mappings like this are sometimes called *Small Step Operational Semantics* meaning that the Small language was defined in terms of the smaller steps in the JCoCo language. Of course, the JCoCo language's semantics should also be formally defined in that case.

Another form of semantic definition is an *Attribute Grammar*. Attribute grammars are not ideal for larger languages, even languages as big as the Small language would be difficult and tedious to describe with an attribute grammar. But, a language like the prefix calculator language is perfect for an attribute grammar definition.

The prefix calculator expression language was first presented in Chap. 5. The contents of the memory location after evaluating an expression is not specified by the grammar of the language. In fact, the purpose of any of the operators is not made explicit in the grammar. Even though we know that $*$ stands for multiplication, there is nothing in the grammar itself that insists this be the case. Other means are necessary to convey that meaning. One such method of conveying the semantics of a language is called an *attribute grammar*. An attribute grammar adds attributes to each node of an abstract syntax tree for sentences in the language.

The attributes tell us how a program would be evaluated in terms of its abstract syntax tree. In other words, an attribute grammar provides a mapping of the syntax of a program into a set of attributes that describe the semantics of the program. Consider the prefix calculator grammar

$G = (\mathcal{N}, \mathcal{T}, \mathcal{P}, E)$ where

$\mathcal{N} = E$

$\mathcal{T} = S, R, number, \ , +, -, *, /$

$\mathcal{P}$ is defined by the set of productions

$$E \rightarrow +\ E\ E \mid -\ E\ E \mid *\ E\ E \mid /\ E\ E \mid \sim E \mid S\ E \mid R \mid number$$

Recall the grammar represents prefix expressions because the operation is written before its arguments. So, $+5 * 64$ results in 29 when evaluated. Notice that when written in prefix notation, the expression $S\ 5$ stores 5 in the memory location. S is a prefix operator.

The *prog* node in the abstract syntax definition in Fig. 7.10 was added to assist in the definition of the attribute grammar. This abstract syntax can be used in Prolog but does not need to be defined as a datatype as it would in Standard ML.

An attribute grammar attaches assignment statements for the attributes to each node in the abstract syntax tree. To distinguish between parts of the abstract syntax

```
AST = prog of AST
    | add of AST * AST
    | sub of AST * AST
    | prod of AST * AST
    | div of AST * AST
    | negate of AST
    | num of number
    | store of AST
    | recall
```

Fig. 7.10 AST definition

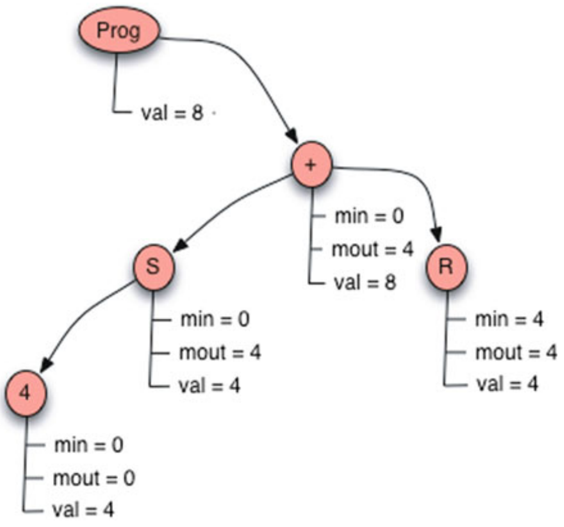


Fig. 7.11 Annotated AST for + S 4 R

tree, let *AST0* denote the AST on the left hand side of a production and *ASTi* where *i* > *0* represent an AST on the right hand side of the production. The attribute grammar for the calculator language is given in Fig. 7.12. Semantics rules are attached to each of the nodes in the AST definition. These rules govern the assignment of the attributes in the AST. The numbers to the left of each rule are there simply to number the rules and are not part of the attribute grammar. By deriving an AST for a sentence and then applying the semantic rules the tree is decorated with attributes that describe the meaning of the sentence, or program, in the language.

The attribute grammar given in Fig. 7.12 can be used to convey the meaning of evaluating an expression like + S 4 R. Figure 7.11 depicts the annotated AST according to the attribute grammar given in Fig. 7.12.

```
AST -> Prog AST
(1) AST1.min = 0
(2) AST0.val = AST1.val

AST -> op AST AST
(3) AST1.min = AST0.min
(4) AST2.min = AST1.mout
(5) AST0.mout = AST2.mout
(6) AST0.val = AST1.val op AST2.val
       where op is one of +,-,*,/

AST -> Store AST
(7) AST1.min = AST0.min
(8) AST0.mout = AST1.val
(9) AST0.val = AST1.val

AST -> Negate AST
(10) AST1.min = AST0.min
(11) AST0.mout = AST1.mout
(12) AST0.val = -1 * AST1.val

AST -> Recall
(13) AST0.val = AST0.min
(14) AST0.mout = AST0.min

AST -> number
(15) AST0.mout = AST0.min
(16) AST0.val = number
```

Fig. 7.12 Attribute grammar

Practice 7.11 Justify the annotation of the tree given in Fig. 7.11 by stating which rule was used in assigning each of the attributes annotating the tree.

You can check your answer(s) in Section 7.17.11.

7.13.1 Synthesized Versus Inherited

Attributes in an attribute grammar come in two flavors. Some attributes are *inherited* which means they are derived from values that are above or to the left in the AST. Some attributes are *synthesized* meaning they are derived from values that are below or to the right in the tree. The *val* attribute is a synthesized attribute in the attribute grammar presented in Fig. 7.12.

Practice 7.12 Is the *min* attribute synthesized or inherited? Is the *mout* attribute synthesized or inherited?

You can check your answer(s) in Section 7.17.12.

Attribute grammars work great for small languages. When a language is larger, the number of attributes can grow exponentially, resulting in a very large annotated tree. In addition, attribute grammars don't deal well with things like control flow and values that aren't determined until run-time. There are many aspects of programming languages that are difficult to assign as attributes in an AST. Typically, attribute grammars work well for small interpreted languages with little or no unknown information.

7.14 Chapter Summary

This chapter provided an introduction to programming in Prolog. List manipulation and building and traversing complex recursive terms are important skills in becoming an experienced Prolog programmer. Grammars and recursive-descent parsing are natural topics relating to Prolog. Building top-down parsers in Prolog is easy with the grammar extension provided in the Prolog language.

In addition, the chapter introduced a couple of formal semantic methods for describing programming languages. Small step operational semantics is one method where a language is defined in terms of smaller steps in a simpler well-defined language. Attribute grammars is another method of assigning meaning to programs.

There are several good books on Prolog programming. The Prolog presented in this chapter is enough to get a flavor of the language and a good start programming in the language. Things left out of the discussion include the *cut* operator and some nuances of how unification is done (i.e. the difference between = and ==). Reading from and writing to files was also left out. The definitive book for more information is Clocksin and Mellish [5]. This book lacks exercises but contains many examples and is a good reference once you understand something about how to program in Prolog (which I hope you do once you've read the chapter and worked through the problems).

7.15 Review Questions

1. What is a term made up of in Prolog? Give examples of both simple and complex terms.
2. What is a predicate in Prolog?
3. In Standard ML you can pattern match a list using (h::t). How do you pattern match a list in Prolog?
4. According to the definition of append, which are the input and the output parameters to the predicate?
5. How do you get more possible answers for a question posed to Prolog?
6. In the expression $X = 6 * 5 + 4$ why doesn't X equal 34 when evaluated in Prolog? What does X equal? What would you write to get X equal to 34?
7. Provide the calls to lookup to look up 7 in the binary tree in Fig. 7.4. Be sure to write down the whole term that is passed to lookup each time. You can consult the answer to practice problem 7.5 to see the definition of the lookup predicate.
8. What symbol is used in place of the:- when writing a grammar in Prolog?
9. What is a synthesized atrribute?
10. What is an inherited attribute?

7.16 Exercises

In these early exercises you should work with the relative database presented at the beginning of this chapter.

1. Write a rule (i.e. predicate) that describes the relationship of a sibling. Then write a query to find out if Anne and Stephen are siblings. Then ask if Stephen and Michael are siblings. What is Prolog's response?
2. Write a rule that describes the relationship of a brother. Then write a query to find the brothers of sophusw. What is Prolog's response?
3. Write a rule that describes the relationship of a niece. Then write a query to find all nieces in the database. What is Prolog's response?
4. Write a predicate that describes the relationship of cousins.
5. Write a predicate that describes the ancestor relationship.
6. Write a predicate called odd that returns true if a list has an odd number of elements.
7. Write a predicate that checks to see if a list is a palindrome.
8. Show the substitution required to prove that sublist([a, b], [c, a, b]) is true. Use the definition in Fig. 7.3 and use the same method of proving it's true.
9. Write a predicate that computes the factorial of a number.
10. Write a predicate that computes the nth fibonacci number in exponential time complexity.
11. Write a predicate that computes the nth fibonacci number in linear time complexity.

12. Write a predicate that returns true if a third list is the result of zipping two others together. For instance,

    ```
    zipped([1,2,3],[a,b,c],[pair(1,a),pair(2,b),pair(3,c)])
    ```

    should return true since zipping [1, 2, 3] and [a, b, c] would yield the list of pairs given above.
13. Write a predicate that counts the number of times a specific atom appears in a list.
14. Write a predicate that returns true if a list is three copies of the same sublist. For instance, the predicate should return true if called as

    ```
    threecopies([a, b, c, a, b, c, a, b, c]).
    ```

    It should also return true if it were called like

    ```
    threecopies([a,b,c,d,a,b,c,d,a,b,c,d]).
    ```

15. Implement insert, lookup, and delete on a binary search tree. The structure of a binary search tree was discussed in this chapter. Your main *run* predicate should be this:

    ```
    buildtree(T) :- readln(L,_,_,_,lowercase), processlist(L,nil,T).

    run :- print('Please enter integers to build a tree: '), buildtree(T),
           print('Here is the tree:'), print(T), print('\PYGZbs{n'}),
           print('Now enter integers to delete: '), readln(L,_,_,_,lowercase),
           delListFromTree(L,T,DT), print(DT).
    ```

    The *run* predicate calls the *buildTree* predicate to build the binary search tree from the list read by the readline. If *5 8 2 10* is entered at the keyboard, *L* would be the list containing those numbers. To complete this project there should be at least three predicates: *insert*, *lookup*, and *delFromTree*.
    The *lookup* predicate was a practice problem and the solution is provided if you need it. The *insert* predicate is somewhat like the *lookup* predicate except that a new node is constructed when you reach a leaf. Deleting a node is similiar to looking it up except that if it is found, the tree is altered to delete the node. Deleting a node from a binary search tree has three cases.

    (a) The node to delete is a leaf node. If this is the case, then deleting it is simple because you just return an empty tree. In Fig. 7.4 this occurs when 2, 4, 7, or 10 is deleted.
    (b) The node to delete has one child. If this is the case, then the result of deleting the node is the subtree under the deleted node. In Fig. 7.4, if the 9 is deleted, then the 10 is just moved up to replace the 9 in the tree.
    (c) The node to delete has two children. If this is the case, then you have to do two things. First, find the left-most value from the right subtree. Then, delete the left-most value from the right subtree and return a new tree with the left-most value of the right subtree at its root. Consider deleting 5 from the tree in Fig. 7.4. The left-most value of the right subtree is 7. To delete 5 we put the 7 at the root of the tree and then delete 7 from the right subtree.

To make this project easy, write it incrementally. Print the results as you go so you can see what works and what doesn't. The print predicate will print its argument while the nl predicate will print a newline. Don't start by writing the entire *run* predicate right away. Write one piece at a time, test it, and then move on to the next piece.

16. Implement a calculator prefix expression interpreter in Prolog as described in the section on attribute grammars in this chapter. The interpreter will read an expression from the keyboard and print its result. The interpreter should start with a *calc* predicate. Here is the *calc* predicate to get you started.

```
calc :- readln(L,_,_,_,lowercase), preprocess(L,PreL), print(PreL), nl,
        expr(Tree,PreL,[]), print(Tree), nl, interpret(Tree,0,_,Val),
        print(Val), nl.
```

The program reads a list of tokens from the keyboard. The *preprocess* predicate should take the list of values and add *num* tags to any number it finds in the list. This makes writing the grammar a lot easier. Any number like *6* in *L* should be replaced by *num((6)* in the list *PreL*. The *expr* predicate represents the start symbol of your grammar. Finally, the *interpret* predicate is the *attribute grammar* evaluation of the AST represented by *Tree*.
To make this project easy, write it incrementally. Print the results as you go so you can see what works and what doesn't. The print predicate will print its argument while the nl predicate will print a newline. Don't write the entire *calc* predicate right away. Write one piece, test it, and then move on to the next piece.

7.17 Solutions to Practice Problems

These are solutions to the practice problem s. You should only consult these answers after you have tried each of them for yourself first. Practice problems are meant to help reinforce the material you have just read so make use of them.

7.17.1 Solution to Practice Problem 7.1

Terms include atoms and variables. Atoms include sophus, fred, sophusw, kent, johan, mads, etc. Atoms start with a lowercase letter. Variables start with a capital letter and include X and Y from the example.

7.17.2 Solution to Practice Problem 7.2

1. *brother(X,Y):- father(Z,X), father(Z,Y), male(X).*
2. *sister(X,Y):- father(Z,X), father(Z,Y), female(X).*
3. *grandparent(X,Y):- parent(X,Z), parent(Z,Y).*
4. *grandchild(X,Y):- grandparent(Y,X).*

Grandparent and grandchild relationships are just the inverse of each other.

7.17.3 Solution to Practice Problem 7.3

The complexity of append is O(n) in the length of the first list.

7.17.4 Solution to Practice Problem 7.4

```
reverse([],[]).
reverse([H|T],L) :- reverse(T,RT), append(RT,[H],L).
```

This predicate has O(n^2) complexity since append is called n times and append is O(n) complexity.

7.17.5 Solution to Practice Problem 7.5

```
reverseHelp([],Acc,Acc).
reverseHelp([H|T], Acc, L) :- reverseHelp(T,[H|Acc],L).
reverse(L,R):-reverseHelp(L,[],R).
```

7.17.6 Solution to Practice Problem 7.6

```
len([],0).
len([_|T],N) :- len(T,M), N is M + 1.
```

7.17.7 Solution to Practice Problem 7.7

```
lookup(X,btnode(X,_,_)).
lookup(X,btnode(Val,Left,_)) :- X < Val, lookup(X,Left).
lookup(X,btnode(Val,_,Right)) :- X > Val, lookup(X,Right).
```

7.17.8 Solution to Practice Problem 7.8

7.17.9 Solution to Practice Problem 7.9

```
sentence(K,L) :- subject(K,M), predicate(M,N), period(N,L).
subject(K,L) :- determiner(K,M), noun(M,L).
predicate(K,L) :- verb(K,M), subject(M,L).
determiner(K,L) :- a(K,L); the(K,L).
verb(K,L) :- discovered(K,L); jailed(K,L); walked(K,L).
noun(K,L) :- professor(K,L); group(K,L); home(K,L).
```

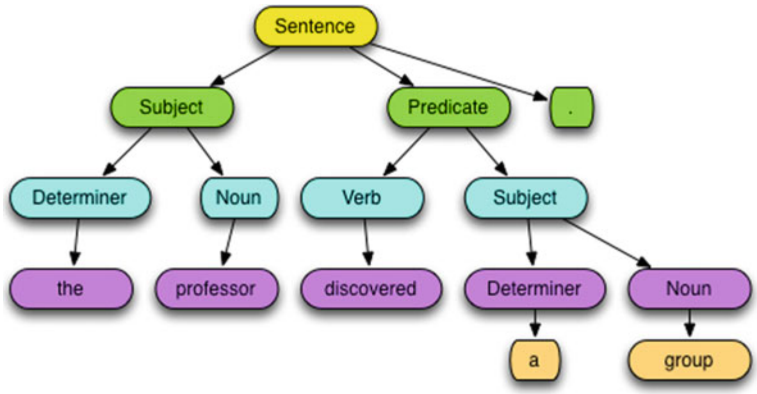


Fig. 7.13 The sentence structure for "The professor discovered a group"

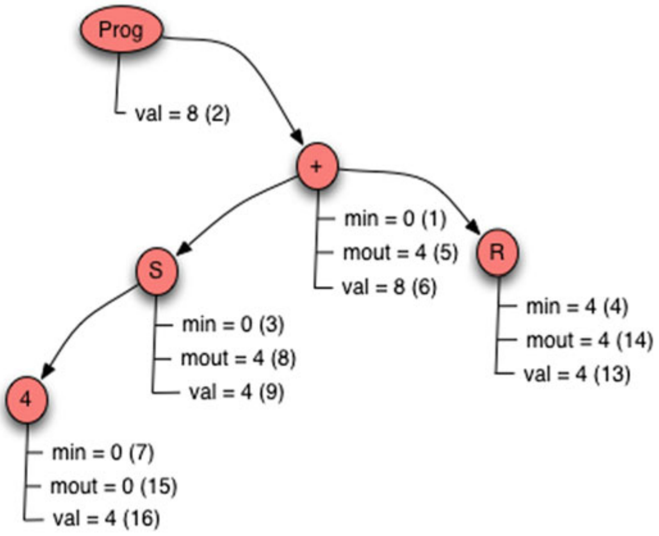


Fig. 7.14 Decorated tree for the prefix expression + S 4 R

7.17.10 Solution to Practice Problem 7.10

```
sentence(sen(N,P)) --> subject(N), predicate(P), ['.'].
subject(sub(D,N)) --> determiner(D), noun(N).
predicate(pred(V,S)) --> verb(V), subject(S).
determiner(det(the)) --> [the].
determiner(det(a)) --> [a].
noun(noun(professor)) --> [professor].
noun(noun(home)) --> [home].
noun(noun(group)) --> [group].
verb(verb(walked)) --> [walked].
verb(verb(discovered)) --> [discovered].
verb(verb(jailed)) --> [jailed].
```

7.17.11 Solution to Practice Problem 7.11 (See Fig. 7.13)

7.17.12 Solution to Practice Problem 7.12

The *val* attribute is synthesized. The *min* value is inherited. The *mout* value is synthesized (See Fig. 7.14).

8 Standard ML Type Inference

Many language implementations, like C++ and Java, check the types of values and operations to be sure each operation is supported for the types of its operands. An important feature of Standard ML is the type inference system which is somewhat like the type checkers of C++ and Java, but a bit more powerful. A type checker checks the types written by the programmer to be sure each type declaration is consistent with the operations being performed, values being passed to functions, and the values being returned. Compilers for Java and C++ even infer the types of some expressions when polymorphic operators are used. For instance, the addition operator has multiple result types depending on the types of its operands.

The type inference system of Standard ML distinguishes itself from other type inference systems by *inferring* almost all the types of an SML program, rather than requiring the programmer to declare the types of variables. The SML type inference system *infers* the types of values in its programs by using type information about constant values and the types supported by its built-in operators or functions. Many of the functions in Standard ML are polymorphic allowing more than one type of argument to be passed to them. The type inference system of Standard ML is able to handle this polymorphism. Robin Milner, Roger Hindley, and Luis Damas all contributed to this powerful polymorphic type inference system.

This chapter develops a polymorphic type inference system for the Small language using Prolog as the implementation language. A typical way to describe type inference is with type inference rules. Each of the type inference rules associated with the Small language is presented along with some of the type inference rule implementations. Not all code is provided since some problems are left as exercises for the reader, but the Prolog examples in this chapter come from a working type inference system for the Small subset of Standard ML.

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_8

8.1 Why Static Type Inference?

To motivate our discussion, consider the program found in Fig. 8.1. This is a valid Small program and when compiled to JCoCo and run it prints 1 to the screen. Contrast that to the program in Fig. 8.2, which is not a valid Small program or Standard ML program. It is missing the dereference operator in the expressions referring to *x*. This program should not execute. Executing such a program would, at best, have unpredictable results. With the target language as the JCoCo virtual machine the program actually does run and produces 1 as its output, which is even worse than it not running at all. Any change in the compiler could end up breaking this program when at one time it seemed to compile and run successfully.

8.1.1 Exception Program

Here is another example. This question was posted on stackoverflow.com. The question posed was,

When executing the code [from Fig. 8.3] in SML I get:

```
stdIn:216.8-216.12 Error: operator and operand don't agree [literal]
  operator domain: real
  operand:         int
  in expression:
    z 3
```

That's fine - I understand that the line z(3); causes an error, since z throws int instead of real. But my problem is with the line x(3.0); why doesn't it cause an error?

```
let val x = ref 0
in
  x := !x + 1;
  println (!x)
end
```

Fig. 8.1 test8.sml

```
let val x = ref 0
in
  x := x + 1;
  println (x)
end
```

Fig. 8.2 test13.sml

```
exception E of int;
fun g(y) = raise E(y);
fun f(x) =
    let
      exception E of real;
      fun z(y)= raise E(y);
    in
      x(3.0);
      z(3)
    end;
f(g);
```

Fig. 8.3 Exception program

The answer is that the program in Fig. 8.3 never executes in Standard ML because it is not correctly typed. Since it is not correctly typed, the type inference system finds the type error, not with the first sequentially evaluated expression, but with the function application of *z* to 3. Without static type checking before the program runs, the Small language that we developed in Chap. 6 would try to execute this program and would encounter an error when evaluating *x(3.0)*. We need type inference to prevent this from happening. Preventing an incorrectly typed program from running catches many unintended errors that might only be caught at run-time otherwise. The type checker helps us find errors that might otherwise go undetected until the code path gets executed.

8.1.2 A Bad Function Call

One more example helps to illustrate the need for type inference. Consider the program in Fig. 8.4. This program is incorrect because it is missing a semicolon between the two println expressions. However, in the absence of type inference it starts run-

```
let val x = 6
in
  println x
  println "Done"
end
```

Fig. 8.4 A bad function call

ning and produces a run-time error stating that *None* is not a callable object. The first call to *println* looks like a curried function call of *println x println "Done"*. The result of *println x* is *None*. That appears to Small to be a function that should be passed the next argument, *println*. Hence we get the *"None is not callable"* run-time error message from the JCoCo virtual machine when the correct error message should come from type inference on this program to say that the println function application does not match its signature.

It would be much better to report to the programmer that the programs in Sect. 8.1 are invalid and do not pass the type inference system. It is dangerous for a program to execute that has undefined results because while an implementation detail like the JCoCo Virtual Machine's use of cell variables may allow a program to execute with the correct output, the implementation of the virtual machine or even a completely different target architecture could then cause a once working program to suddenly stop working. As programmers we rely on the tools we use to produce correct code and to guarantee that once debugged the behavior of a program won't suddenly change due to external factors like a compiler change.

8.2 Type Inference Rules

A type inference system is defined in terms of *type inference rules*. The collection of these rules define a type inference system. Each type inference system defines its own set of rules. Type inference rules follow a pattern of necessary conditions, or premises, and a logical conclusion. The rules are written in this form.

RuleName

$$\frac{Premise_1, \quad Premise_2, \quad ..., \quad Premise_n}{Conclusion}$$

The way to read this is to say that if each of the premises hold in some model, then the conclusion holds as well in that model. An inference system contains a collection of inference rules. Normally each rule in an inference system is given a name so it can be referred to in proofs. The collection of inference rules can be used in constructing a proof. In this case a proof of an expression's type.

All the type inference rules for the Small language are provided in the sections in this chapter. Some of the type inference rules will contain braces surrounding sytantic elements of the language (i.e. { and }). These braces are used to indicate zero or more occurrences of syntactic elements.

Much of the Prolog implementation of this type inference system is provided as well, although some pieces of it are left as exercises for the reader.

8.3 Using Prolog

The Small language and grammar is sufficiently complex that writing a top-down parser for it would be difficult. Since Prolog's grammar support creates a top-down parser from a grammar, it is not powerful enough to parse programs in the Small language. So, the program is not parsed by Prolog. Instead, the *mlcomp* compiler writes a file called *a.term* which is a Prolog term representing the abstract syntax of the source program. This AST is read by the Prolog type inference system. Consider the program in Fig. 8.1. The AST Prolog term for this program is shown in Fig. 8.5. In most cases, even if the compiler has not been extended to generate the correct code for a program, the compiler will still write a correct Prolog term. If compiling a new extension to the language the *writeTerm* function in *mlcomp.sml* may have to be extended to support the new extension.

The code in Fig. 8.6 starts the type checker. The *run* predicate reads the abstract syntax tree for the program from the file called *a.term*. The *print* prints it back to

```
letdec(
  bindval(idpat('x'),apply(id('ref'),int('0'))),
  [apply(id(':='),tuple([id('x'),
   apply(id('+'),tuple([apply(id('!'),id('x')),
         int('1')]))])),
   apply(id('println'),apply(id('!'),id('x')))
  ]).
```

Fig. 8.5 test8.sml AST

```
finalStatus(typeerror) :- print('The program failed to pass the typechecker.'), nl, !.
finalStatus(_) :- print('The program passed the typechecker.'), nl, !.

warning([],_) :- !.
warning(_,fn(_,_)) :- !.
warning([_|_],_) :-
  print('Warning: type vars not instantiated in result type initialized to dummy types!'),
  nl, nl, !.

errorOut(error(E)) :-
     nl, nl, print('Error: Typechecking failed. Message was : '), nl,
     print(E), nl, nl, halt(0).
errorOut(typeerror(E)) :-
     nl, nl, print('Error: Typechecking failed due to type error. Message was : '), nl,
     print(E), nl, nl, halt(0).
errorOut(E) :-
     nl, nl, print('Error: Typechecking failed for unknown reason : '), nl,
     print(E), nl, nl, halt(0).
run :- print('Typechecking is commencing...'), nl,
       readAST(AST), print('Here is the AST'), nl, print(AST), nl, nl, nl,
       catch(typecheckProgram(AST,Type),E,errorOut(E)),
       nl, nl, print('val it : '), printType(Type,TypeVars), nl, nl,
       warning(TypeVars,Type), finalStatus(Type).
runNonInteractive :- run, halt(0).
```

Fig. 8.6 The type checker run predicate

the screen just for visual confirmation. The *catch* is a Prolog predicate that provides exception handling. The first argument to *catch* is a predicate to satisfy. If an exception occurs while attempting to satisfy the predicate the error is unified with *E* and the *errorOut* predicate is called which prints one of three messages depending on the error.

If no error occurs, the variable *Type* will hold the type returned by the Small program. The *printType* predicate prints the type in Standard ML format and returns a list of any type variables it finds. The *warning* predicate warns of any uninstantiated type variables found in the type.

The *cut operator* (i.e. !) stops Prolog from backtracking. Normally, if a point is reached where Prolog cannot satisfy a predicate, it will undo the last unification and look for another way to satisfy the original query. The type inference system has side-effects, like printing error messages, and the type inference is deterministic in its choices. There is only one way to satisfy predicates in the type inference system: by finding the type of the program. To prevent backtracking the cut operator can be used. Technically, the cut operator is not needed because different cases of a predicate should all be logically mutually exclusive. However, it is sometimes more convenient to use the cut. When Prolog comes across a cut operator, the search space is pruned. The predicate in which the cut is found may not be satisfied by any other choices in that predicate. In the *warning* predicate, once one of the patterns matches (from the top down), the warning predicate cannot be satisfied by any other *warning* definition. As a Standard ML programmer this is appealing because it leads to the same kind of pattern matching used in Standard ML programs.

So, a term like the one in Fig. 8.5 is read as the *AST* by the type checker and passed to the predicate called *typeCheckProgram* that does the type inference of the Small program. The AST description is given in Standard ML form in Fig. 8.7. Prolog does not require datatypes be declared so there is no explicit declaration of the AST datatype in the typechecker. Nevertheless, the datatype is coded into the expected values of AST nodes in the type checker predicates. The Prolog AST format is nearly an exact copy from the Standard ML AST definition except that *boolval* in the Standard ML implementation is called *bool* in the Prolog version, the *infixexp* in the Standard ML AST is replaced with an *apply* in the type checker, and the *raise* AST node is replaced with an apply. See the *writeExp* function for infix expressions in *mlcomp.sml* for the details of the conversion from *infixexp* to *apply* and *raise* to *apply*. The implementation of the type checker follows from the definition of the abstract syntax.

The Standard ML types used in the Small language include the types in Fig. 8.8. These types include the usual boolean, integer, and string types. The *exn* type is the type of exceptions. The *tuple* type is a tuple of some aggregation of other types. Lists must be homogeneous meaning they are a list of some one type of value. The type *fn(A,B)* is the type of all functions. Every function takes one argument, which may be a tuple, and returns one value. The *ref* types are the reference types and are defined by the type of value to which they point. Type variables are denoted by the *typevar* type. The string in a typevar is the name of the type variable. The type checker assigns variable names as a, b, c, d, etc. The type checker is strict in *typeerror*, meaning once

```
datatype
  exp = int of string
      | ch of string
      | str of string
      | bool of string
      | id of string
      | listcon of exp list
      | tuplecon of exp list
      | apply of exp * exp
      | expsequence of exp list
      | letdec of dec * (exp list)
      | handlexp of exp * match list
      | ifthen of exp * exp * exp
      | whiledo of exp * exp
      | func of string * match list
and
  match = match of pat * exp
and
  pat = intpat of string
      | chpat of string
      | strpat of string
      | boolpat of string
      | idpat of string
      | wildcardpat
      | infixpat of string * pat * pat
      | tuplepat of pat list
      | listpat of pat list
      | aspat of string * pat
and
  dec = bindval of pat * exp
      | bindvalrec of pat * exp
      | funmatch of string * match list
      | funmatches of (string * match list) list
```

Fig. 8.7 AST description

an expression results in a type error all other expressions that interact with it also result in *typeerror*.

The job of the type checker is to map a program in the syntax of Fig. 8.7 into its type as defined in Fig. 8.8. Type inference rules provide the mapping instructions. The rest of this chapter explores type inference for the simplest nodes first, working up to more complex language constructs.

```
type = bool
     | int
     | str
     | exn
     | tuple of type list
     | listOf of type
     | fn of type * type
     | ref of type
     | typevar of string
     | typeerror
```

Fig. 8.8 Small types

8.4 The Type Environment

Functions in Standard ML are typed by their signature as seen in Chap. 5. For instance, the *Int.fromString* function has a signature of

$$fn : str \rightarrow int$$

The *environment* of the type checker provides information about the signature of built-in functions and operators in the language. The environment is referred to as epsilon (i.e. ε), the *type environment*, or just the *environment*. More generally, the *environment* provides a mapping of identifiers to types which can be consulted during type checking as needed.

Some functions are polymorphic and therefore type variables are necessary to describe their type. For instance, the *print* function has a type of

$$fn : \alpha \rightarrow ()$$

The α represents a type variable in the signature of the *print* function. The existence of type variables makes it possible for functions in the Standard ML type inference system to be polymorphic.

In Prolog the environment is created by the *typecheckProgram* predicate which passes it to the *typecheckExp* predicate. Figure 8.9 provides the type environment given to the *typecheckExp* predicate.

There are a number of functions and operators provided in the environment. The function type begins with *fn*. All type variables are named *typevar* and the *unit* type is denoted as *tuple([ ])* in the type checker. The environment is represented as follows in the type inference rules.

$$\varepsilon = [Exception \mapsto \alpha \rightarrow exn,\ raise \mapsto exn \rightarrow \alpha,\ andalso \mapsto bool \times bool \rightarrow bool, ...]$$

```
typecheckProgram(Expression,Type) :-
  typecheckExp([('Exception',fn(typevar(a),exn)),
                ('raise',fn(exn,typevar(a))),
                ('andalso',fn(tuple([bool,bool]),bool)),
                ('orelse',fn(tuple([bool,bool]),bool)),
                (':=',fn(tuple([ref(typevar(a)),typevar(a)]),tuple([]))),
                ('!',fn(ref(typevar(a)),typevar(a))),
                ('ref',fn(typevar(a),ref(typevar(a)))),
                ('::',fn(tuple([typevar(a),listOf(typevar(a))]),listOf(typevar(a)))),
                ('>', fn(tuple([typevar(a),typevar(a)]),bool)),
                ('<', fn(tuple([typevar(a),typevar(a)]),bool)),
                (@,fn(tuple([listOf(typevar(a)),listOf(typevar(a))]),listOf(typevar(a)))),
                ('Int.fromString',fn(str,int)),
                ('input',fn(str,str)),
                ('explode',fn(str,listOf(str))),
                ('implode',fn(listOf(str),str)),
                ('println',fn(typevar(a),tuple([]))),
                ('print',fn(typevar(a),tuple([]))),
                ('cprint',fn(typevar(a),cprint)),
                ('type',fn(typevar(a), str)),
                (+,fn(tuple([int,int]),int)),
                (-,fn(tuple([int,int]),int)),
                (*,fn(tuple([int,int]),int)),
                ('div',fn(tuple([int,int]),int))],
         Expression,Type).
```

Fig. 8.9 The type environment

The environment is a list of bindings of identifiers to types. The environment is always searched from left to right to find a binding as needed by type inference rules. The symbol $\mapsto$ is pronounced *maps to*. For instance, Exception maps to a polymorphic type from *alpha* to *exn*.

8.5 Integers, Strings, and Boolean Constants

The types of integer, string, and boolean constant values are determined by the scanner when read in *mlcomp*. Determining their types then is just a matter of matching their scanned type to a type in the type checker. So we write the following statements about the types of simple constant values. In each case, there are no premises that must be satisfied. When we see a boolean constant we can immediately determine its type.

BoolCon

$$\frac{}{\varepsilon \vdash bool(v) : bool}$$

IntCon

$$\frac{}{\varepsilon \vdash int(v) : int}$$

StringCon

$$\frac{}{\varepsilon \vdash str(v) : str}$$

```
typecheckExp(_,int(_),int).
typecheckExp(_,bool(_),bool).
typecheckExp(_,str(_),str).
```

Fig. 8.10 Constant type inference

To keep things simpler in the type inference algorithm we'll limit our discussion to integers for all numbers. Each type inference rule will be named in bold and its definition will be indented underneath it as seen here. In Prolog constant types are given a type by the typecheckExp predicate as shown in Fig. 8.10. The environment is the first argument to the typecheckExp predicate and is a don't care value in this case since the environment is not needed to determine the type of a constant. The AST argument is the second argument to the predicate. The third argument is the type of the expression.

Consider the expression *5*. This is mapped into the term *int(5)* by the mlcomp compiler. Passing *int(5)* to the type checker matches the predicate in Fig. 8.10 and returns *int* for its type. The type is printed by the type checker. Output from the type checker looks like this.

```
Typechecking is commencing...
Here is the AST
int(5)
val it : int
The program passed the typechecker.
```

8.6 List and Tuple Constants

The type of a list is derived from its constituent type. Lists are homogeneous in Small as they are in Standard ML, meaning that all elements must have the same type. The type of a tuple is derived from its constituent types. For example consider this list and tuple.

$$[6, 5, 4] : int\ \ list$$
$$("hi", true, 6) : str * bool * int$$

In the abstract syntax, list and tuple constants are written as lists of values. For instance, written in Prolog syntax, to typecheck the two values above, *typecheckExp* is implemented as follows.

```
typecheckExp(Env,listcon(L),listOf(T)) :- typecheckList(Env,L,T).
typecheckExp(Env,tuple(L),tuple(T)) :- typecheckTuple(Env,L,T).
```

Typechecking the list and tuple constants above returns these type values.

```
listOf(int)
tuple([str,bool,int])
```

Note the type value of *listOf* here. *list* is a built-in predicate in Prolog and should not be used. Here is the type inference rule that describes the type of lists in Small.

ListCon

$$\frac{\forall i \;\; 1 \le i \le n, n \ge 0 \qquad \varepsilon \vdash e_i : \alpha}{\varepsilon \vdash [e_1, e_2, ..., e_n] : \alpha \;\; list}$$

The *List* type inference rule can be read as follows: If in the type environment the types of all elements of a list are found to be α, then the type of the list constant of these values is $\alpha \; list$ in the same type environment. In the vacuous condition, where $n = 0$, there are no premises with the type of the list being polymorphically $\alpha \; list$.

For tuples the type inference rule is somewhat similar. The $\times$ in the rule below is the cross product symbol and is the symbol that corresponds to $*$ printed by the Standard ML type checker. The writing of this cross product forms the type for tuples of n elements.

TupleCon

$$\frac{\forall \;\; 1 \le i \le n, n \ge 0 \qquad \varepsilon \vdash e_i : \alpha_i}{\varepsilon \vdash (e_1, e_2, ..., e_n) : \times_{i=1}^{n} \alpha_i}$$

In the vacuous condition of $n = 0$ in the *TupleCon* rule the type is the empty Cartesian product which is denoted as the *unit* type in Standard ML. In other words, the empty tuple has type *unit* in Standard ML.

Consider type checking the expression *[1,2,3,4]*. The type checker provides output as shown below. Typechecking the list constant calls *typecheckList* as shown earlier in this section. The *typecheckList* predicate proceeds through the list of elements making sure all the types match, resulting in the type you see below.

```
Typechecking is commencing...
Here is the AST
listcon([int(1),int(2),int(3),int(4)])
val it : int list
The program passed the typechecker.
```

8.7 Identifiers

When a program uses an identifier the type of the identifier must be looked up in the type environment. Lookup in the environment is denoted as $\varepsilon[id \mapsto \alpha]$ which says that in the type environment find *id* and its associated type *alpha*. The rule

```
exists(Env,Name) :-
    member((Name,_),Env), !.
find(Env,Name,Type) :-
    member((Name,Type),Env), !.
find(Env,Name,Type) :-
    writeMsg(['Failed to find ',
    Name,' with type ',Type,
    ' in environment : ']), print(Env), nl,
    throw(typeerror('unbound identifier')).
typecheckExp(Env,id(Name),Type) :-
    find(Env,Name,Type).
```

Fig. 8.11 Environment lookup predicates

below indicates the type of an identifier is its type in the environment. In the Prolog implementation a *find* predicate is written to look up an indentifier in an environment to find its type. Here is the identifier type inference rule.

Identifier

$$\frac{}{\varepsilon[id \mapsto \alpha] \vdash id : \alpha}$$

The code in Fig. 8.11 provides the details of the *find* predicate implementation in Prolog. There is also an *exists* predicate that is satisfied if an environment contains a binding. The *member* predicate is a built-in predicate in Prolog. Normally in a proof this lookup will be implied when an identifier is looked up in the bindings and this step will be omitted. Consider the expression containing just the name of a function, as in *println*. Type checking this expression will reveal the type of *println*, which is not a Standard ML function but is in the Small language.

```
Typechecking is commencing...
Here is the AST
id(println)
val it : 'a -> unit
The program passed the typechecker.
```

The type checker sees the identifier and looks in the environment, finding the *println* identifier and yielding its type.

8.8 Function Application

Function application in Small and Standard ML occurs when two expressions are written next to each other as in the expression

```
println 6
```

for instance. In the Prolog AST this appears as *apply(id('println'),int('6'))*. Function application is the act of calling a function. The type of *println* is $\alpha \rightarrow unit$. The println function is being applied to an integer. We need a type inference rule that formally defines a legal function application.

Before the function application type inference rule can be written one more operator is needed which may be a bit difficult to understand at first. Small and Standard ML support polymorphic type checking. When a type contains type variables the type variables place restrictions on the kinds of values to which the type may be instantiated. For instance, the *println* function has type $\alpha \rightarrow unit$ which says that the function *println* is polymorphic taking arguments of any type. The type is defined with the type variable α, but just when is *println* polymorphic? The answer is every time *println* is called. One application of *println* can be given an integer, while the next application could be given a tuple of an integer and a boolean value. In each case the α type variable is instantiated to a type, an integer in the first case and a tuple in the second. Type inference rules need a way of creating instances of polymorphic types. In this way, one instance of the polymorphic type $\alpha \rightarrow unit$ can be instantiated as $int \rightarrow unit$ while the next can be instantiated as $int \times bool \rightarrow unit$.

In type inference rules this instantiation operator is written as *inst*. It is given a type and returns an instance of that type where all type variables are replaced by fresh, unbound instances of variables. In the type inference rule below the result type of function application is the specialization of the instantiated result type given an instance of the type of the argument passed to the function.

FunApp

$$\frac{\varepsilon \vdash e_1 : \alpha \rightarrow \beta,\ \ \alpha' \rightarrow \beta' : inst(\alpha \rightarrow \beta),\ \ \varepsilon \vdash e_2 : \alpha_{e2}, \alpha' : inst(\alpha_{e2})}{\varepsilon \vdash e_1 e_2 : \beta'}$$

The Prolog implementation of instantiation will shed some light on instantiation. In Prolog, all type variables are written as *typevar(id)* where *id* is typically some letter from *a* to *z*, but could be any identifier. This corresponds to the way type variables appear in Standard ML's type inference system when types like

fn:'a->'a

are printed. In the Prolog implementation of the typechecker the function type *fn:'a ->'a* is represented as *fn(typevar(a),typevar(a))*. Making an instance of a type like this creates a type that can be unified with other types in Prolog. An instance of this type would be written as *fn(A,A)*. In this Prolog term the variable *A* is unbound since it is not unified with any other term. The Prolog term *fn(A,A)* is an instance of the type *fn(typevar(a),typevar(a))*. Instantiation is performed by the *inst* predicate shown in Fig. 8.12.

On line 17 of Fig. 8.12 the *inst* operator calls the *instanceOf* predicate with an empty environment. The *instanceOf* predicate recursively traverses the type, changing all occurrences of type variables to Prolog variables. The environment keeps track

```
 1 instanceOfList(Env,[],[],Env).
 2 instanceOfList(Env,[H|T],[G|S],NewEnv) :-
 3     instanceOf(Env,H,G,Env1), instanceOfList(Env1,T,S,NewEnv).
 4 instanceOf(Env,A,A,Env) :- var(A), !.
 5 instanceOf(Env,A,A,Env) :- simple(A), !.
 6 instanceOf(Env,fn(A,B), fn(AInst,BInst),Env2) :-
 7     instanceOf(Env,A,AInst,Env1), instanceOf(Env1,B,BInst,Env2), !.
 8 instanceOf(Env,listOf(A),listOf(B),NewEnv) :- instanceOf(Env,A,B,NewEnv), !.
 9 instanceOf(Env,ref(A),ref(B),NewEnv) :- instanceOf(Env,A,B,NewEnv), !.
10 instanceOf(Env,tuple(L),tuple(M),NewEnv) :- instanceOfList(Env,L,M,NewEnv), !.
11 instanceOf(Env,typevar(A),B,Env) :- exists(Env,A), find(Env,A,B), !.
12 instanceOf(Env,typevar(A),B,[(A,B)|Env]) :- !.
13 instanceOf(_,A,B,_) :-
14     print('Type Error: Type '), printType(B,_),
15     print(' is not an instance of '), printType(A,_), nl,
16     throw(typeerror('type mismatch')), !.
17 inst(X,Y) :- instanceOf([],X,Y,_).
```

Fig. 8.12 The instantiation operator

```
typecheckExp(Env,apply(Exp1,Exp2),ITT) :-
    typecheckExp(Env,Exp1,fn(FT,TT)),
    typecheckExp(Env,Exp2,Exp2Type),
    inst(Exp2Type,Exp2TypeInst),
    catch(inst(fn(FT,TT), fn(Exp2TypeInst,ITT)),_,
        printApplicationErrorMessage(Exp1,
            fn(FT,TT),Exp2,Exp2Type,ITT)), !.
```

Fig. 8.13 Function application type inference

of the mapping of type variables to Prolog variables so if a type variable appears more than once in a type it is replaced by the same Prolog variable as is evident with the example of the polymorphic type of function *f* in the preceeding paragraph. Line 11 insures the same Prolog variable is used when the type variable is found in the environment. Line 12 creates a new Prolog variable when the type variable is not found in the environment.

Line 4 of Fig. 8.12 uses the *var* predicate which returns true if *A* is an unbound Prolog variable. This clause is important because if *instanceOf* is called with an uninstantiated variable already, then it will unify with anything it is matched to, like the function type in line 6 for instance. Line 4 insures that an unbound variable stays unbound. Line 5 uses the *simple* predicate which just means that *A* is a simple term like *int*, or *bool*. It is not complex, meaning there are no subterms that are a part of this term. A complex term would be a type like *tuple([typevar(a),typevar(a)])*. Line 5 handles all the simple types by just returning them. Simple types are not polymorphic.

Type inference for function application in Prolog utilizes Prolog exception handling as shown in Fig. 8.13. If a function call is not correct due to a type error, the instantiation predicate in Fig. 8.12 will throw a type error exception. In that case it would be nice to know there was an error with a function call. The error is caught in this code and a message is printed.

8.8.1 Instantiation

When an instance of a type is created with free variables, the Prolog variables only stay free as long as the instantiated type is not unified with any other types. Once that instance of a type is unified some or all of the free variables will be bound. In this way, when an instance of a type is created, it moves towards being a type with no free variables as type inference proceeds. If unification is not possible due to a type error, then that condition is recognized and the resulting type is the special type *typeerror* which is handled in the Prolog implementation by throwing an exception.

Several of the rules in the next section use instantiation so that unification of types is possible. When an instance of a type is the result of a type inference rule, all free variables have been unified with bound values producing a valid type except in the cases of type errors in the original program. Consider the invocation of *println 6* and how we would arrive at a type. The following instance of the *FunApp* rules shows how it is proved to be a valid function application.

$$\frac{\varepsilon \vdash println : \alpha \rightarrow unit, \quad int \rightarrow unit : inst(\alpha \rightarrow unit), \quad \varepsilon \vdash 6 : int}{\varepsilon \vdash println\ 6 : unit}$$

8.9 Let Expressions

Binding identifiers to values is the job of *let* expressions in Standard ML and Small. Let expressions create bindings between identifiers and values through the use of patterns. Identifiers can be bound to one or more function definitions in a *let* expression because functions are values too in Small and Standard ML. A little new notation must be introduced to write type inference rules for *let* expressions.

Let expression build new environments. To properly define type inference for the newly created environment, environments must be considered values in the type checker. A declaration produces an environment mapping one or more identifiers to their types. To combine two environments a new overlay operator is defined. One environment can then be used to partially overlay another environment. Consider two environments ε_1 and ε_2. To combine the first with the second environment the overlay $\oplus$ operator is defined as demonstrated here.

$$\begin{aligned}
\varepsilon_1 &= [x \mapsto \alpha \rightarrow \beta,\ y \mapsto int,\ z \mapsto \alpha \times \beta] \\
\varepsilon_2 &= [u \mapsto \alpha \times \beta \rightarrow \beta,\ y \mapsto\ bool] \\
\varepsilon_2 \oplus \varepsilon_1 &= [u \mapsto \alpha \times \beta \rightarrow \beta,\ y \mapsto\ bool] \oplus [x \mapsto \alpha \rightarrow \beta,\ y \mapsto int,\ z \mapsto \alpha \times \beta] \\
&= [u \mapsto \alpha \times \beta \rightarrow \beta,\ y \mapsto\ bool, x \mapsto \alpha \rightarrow \beta,\ y \mapsto int,\ z \mapsto \alpha \times \beta]
\end{aligned}$$

Since environments are always searched from left to right, the result of the overlay operator is the concatenation of the two environments. In this example the result is that y is mapped to *bool* in the new environment $\varepsilon_2 \oplus \varepsilon_1$. In Prolog, environments are represented as lists of bindings just as described here. The overlay operator is simply the *append* predicate in Prolog. Recalling that the *find* predicate searches an environment from left to right the result of appending two lists is the overlay of the bindings in the second list. One more bit of notation is needed. When a declaration creates a new environment it will be written using a double right arrow as follows.

$$\varepsilon \vdash dec \Rightarrow \varepsilon_{dec}$$

This indicates that the declaration builds a new environment ε_{dec} that will be used later in the type inference rule. Now we are ready to define the *let* expression type inference rule.

Let

$$\frac{\varepsilon \vdash dec \Rightarrow \varepsilon_{dec},\quad \varepsilon_{dec} \oplus \varepsilon \vdash e_{sequence} : \beta}{\varepsilon \vdash let\ dec\ in\ e_{sequence}\ end : \beta}$$

The *dec* declaration in the rule above can be one of two types of declarations in Small: either a *val* declaration or a series of *fun* declarations. The type inference for these two types of declarations is provided in the rules below. The expression *e* in the rule above is a sequence of expressions. The type inference rule for sequential execution is provided in a later section of this chapter.

ValDec

$$\frac{pat : \alpha \Rightarrow \varepsilon_{pat},\quad \varepsilon \vdash e : close(\alpha)}{\varepsilon \vdash val\ pat = e \Rightarrow \varepsilon_{pat}}$$

In the *ValDec* rule there are pattern declarations. The type inference rules for pattern declarations are provided in the next section of the chapter. Each pattern declaration provides an environment mapping identifiers in the pattern to their associated types. The next section provides the type inference rules for pattern matching along with the environments yielded by each type of pattern.

ValRecDec

$$\frac{[id : \alpha] \oplus \varepsilon \vdash e : \alpha}{\varepsilon \vdash val\ rec\ id = e \Rightarrow [id : close(\alpha)]}$$

A *ValRecDec* is used when an identifier is bound to an anonymous function that calls itself recursively. Anonymous functions don't normally call themselves. In this one instance, the anonymous function can through the use of a recursive binding. The binding in this case binds the identifier to the type of the function in the body of the function.

FunDecs

$$\frac{\forall i\ 1 \le i \le n, \forall j\ 1 < j \le n,\ \ n \ge 1, \qquad [id_1 \mapsto \alpha_1 \to \beta_1\ \{,\ id_j \mapsto \alpha_j \to \beta_j\}] \oplus \varepsilon \vdash id_i\ matches_i : \alpha_i \to \beta_i}{\varepsilon \vdash fun\ id_1\ matches_1\ \{and\ id_j\ matches_j\} \Rightarrow [id_1 \mapsto close(\alpha_1 \to \beta_1)\ \{,\ id_j \mapsto close(\alpha_j \to \beta_j)\}]}$$

In the rule above the braces (i.e. { and }) are EBNF and represent zero or more occurrences as necessary. Since *j* must be greater than 1, if *n=1* then no occurrences of the parts written inside braces are necessary. This rule introduces *matches*. The type inference for matches appears right after the section on patterns.

A *FunDecs* is a series of mutually recursive function definitions. See *mlcomp.sml* for examples where the keyword *and* is used between function definitions. The rule above starts with the premise that each function in the *FunDecs* has a type $\alpha \to \beta$. The rule makes an instance of the function type and places it in the environment given the *matches*. The *matches* are the list of pattern matches for one function definition. This is done because all recursive function calls to functions in the *FunDecs* must have consistent types. As the type inference rules are satisfied the instance of the type is bound to type values. If these premises are met, the conclusion produces a new environment with each function bound to its type.

The newly built environment that results from the *FunDecs* rule contains a type function called *close*. This type function is important. Closing a type means that any free type variables (i.e. Prolog type variables) are instantiated to *typevar* type variables. This is needed because otherwise the first application of a function with free type variables would instantiate them to the types of that particular function application. This would not be a problem if functions were not polymorphic. However, functions in Standard ML often have polymorphic types. The *close* type function is needed to support polymorphic type inference. The *close* function is the inverse of the *inst* type function.

8.10 Patterns

Patterns are used in *ValDec* declarations and in *matches* which are discussed in the next section. When a pattern is used, it produces bindings of one or more identifiers to types. Constant values can be used as patterns as in the *IntPat*, *BoolPat*, *StrPat*, *NilPat*, and *UnitPat* rules. Patterns like this don't produce any bindings because identifiers are not part of these patterns.

IntPat

$$\frac{}{integer\_constant : int \Rightarrow [\,]}$$

BoolPat

$$\frac{}{true : bool \Rightarrow [\,]}$$

$$\frac{}{false : bool \Rightarrow [\,]}$$

StrPat

$$\frac{}{string\_constant : str \Rightarrow [\,]}$$

NilPat

$$\frac{}{nil : \alpha\ list \Rightarrow [\,]}$$

ConsPat

$$\frac{pat_1 : \alpha \Rightarrow \varepsilon_{pat_1},\quad pat_2 : \alpha\ list \Rightarrow \varepsilon_{pat_2}}{pat_1{::}pat_2 : \alpha\ list \Rightarrow \varepsilon_{pat_1} + \varepsilon_{pat_2}}$$

TuplePat

$$\frac{\begin{array}{l}\forall i\ 1 \le i \le n, n \ge 0\\ \qquad pat_i : \alpha_i \Rightarrow \varepsilon_{pat_i}\end{array}}{(pat_1, pat_2, ..., pat_n) : \times_{i=1}^{n}\alpha_i \Rightarrow \sum_{i=1}^{n} \varepsilon_{pat_i}}$$

ListPat

$$\frac{\begin{array}{l}\forall i\ 1 \le i \le n, n \ge 0\\ \qquad pat_i : \alpha \Rightarrow \varepsilon_{pat_i}\end{array}}{[pat_1, pat_2, ..., pat_n] : \alpha\ list \Rightarrow \sum_{i=1}^{n} \varepsilon_{pat_i}}$$

The *ConsPat*, *TuplePat*, and *ListPat* rules may contain other patterns. Each of them employ the disjoint union operator to build new environments from their sub-environments. Disjoint union is used because duplicate identifiers are not allowed in patterns. The $+$ and $\sum$ symbols are used to denote the disjoint union of sets of patterns.

The *TuplePat* rule forms the cross product type of all its constituent types and forms the environment that results from all the sub-pattern environments being overlayed

```
let
  val (x,y)::L = [(1,2),(3,4)]
in
  println x
end
```

Fig. 8.14 Pattern matching

on one another. In the vacuous case, when $n = 0$, the *TuplePat* rule derives the *unit* pattern (i.e. the empty tuple) and yields an empty environment.

The vacuous case of the *ListPat* rule, when $n = 0$, provides an alternative form of specifying the empty list. Both *nil* and *[]* represent the empty list in Standard ML with polymorphic type $\alpha\ list$.

IdPat

$$\frac{}{id : \alpha \Rightarrow [id \mapsto \alpha]}$$

Most patterns boil down to creating bindings of identifiers to values. The *IdPat* type inference rule yields a new binding environment, binding the identifier to its type. Consider the program in Fig. 8.14. Typechecking this program results in the following output.

```
letdec(
  bindval(infixpat({:}{:},tuplepat([idpat(x),idpat(y)]),idpat(L)),
    listcon([tuple([int(1),int(2)]),tuple([int(3),int(4)])])),
  [apply(id(println),id(x))])
val (x,y){:}{:}L : (int * int) list
val it : unit
The program passed the typechecker.
```

The type inference rules specify how the type checker works. To see this in action a proof is possible using the type inference rules. Each step in the proof is justified by a type inference rule written to the right side of the rule's use. To reach the conclusion (1) of the type checker, premises (2) and (3) must hold.

$$\frac{(2)\varepsilon \vdash val\ (x, y)::L = [(1, 2), (3, 4)] \Rightarrow \varepsilon_{dec} \quad (3)\varepsilon_{dec} \oplus \varepsilon \vdash println\ x : unit}{(1)\varepsilon \vdash let\ val\ (x, y)::L = [(1, 2), (3, 4)]\ in\ println\ x\ end : unit}(Let)$$
$$\varepsilon_{dec} = [x \mapsto int, y \mapsto int, L \mapsto int * int\ list]$$

To prove (2):

$$\frac{(4)(x, y)::L : int \times int\ list \Rightarrow \varepsilon_{dec} \quad (5)\varepsilon \vdash [(1, 2), (3, 4)] : int \times int\ list}{(2)\varepsilon \vdash val\ (x, y)::L = [(1, 2), (3, 4)] \Rightarrow \varepsilon_{dec}}(ValDec)$$

To prove (4):

$$\frac{(6)(x, y) : int \times int \Rightarrow [x \mapsto int, y \mapsto int] \quad (7)L : int \times int\ list \Rightarrow [L \mapsto int \times int\ list]}{(4)(x, y)::L : int \times int\ list \Rightarrow \varepsilon_{dec}}(ConsPat)$$

To prove (6):

$$\frac{(8)x : int \Rightarrow [x \mapsto int] \quad (9)y : int \Rightarrow [y \mapsto int]}{(6)(x, y) : int \times int \Rightarrow [x \mapsto int, y \mapsto int]}(TuplePat)$$

Premises (7), (8), and (9) are true by virtue of the *IdPat* inference rule. Considering (5):

$$\frac{(10)\varepsilon \vdash (1, 2) : int \times int \quad (11)\varepsilon \vdash (3, 4) : int \times int}{(5)\varepsilon \vdash [(1, 2), (3, 4)] : int \times int\ list}(ListCon)$$

Considering (10) and a similar argument for (11):

$$\frac{(12)\varepsilon \vdash 1 : int \quad (13)\varepsilon \vdash 2 : int}{(10)\varepsilon \vdash (1, 2) : int \times int}(TupleCon)$$

Both (12) and (13) are true by the *IntCon* rule. A similar argument holds for (11). The proof nears completion by proving (3):

$$\frac{(14)\varepsilon_{dec} \oplus \varepsilon \vdash println : \alpha \rightarrow unit \quad int \rightarrow unit : inst(\alpha \rightarrow unit) \quad (15)\varepsilon_{dec} \oplus \varepsilon \vdash x : inst(int)}{(3)\varepsilon_{dec} \oplus \varepsilon \vdash println\ x : unit}(FunApp)$$

Both (14) and (15) are true by the *Identifier* rule concluding the proof of the type correctness of this program. The sequence rule was glossed over in this proof. Sequence type checking appears later in the chapter.

Practice 8.1 Prove that the program in Fig. 8.15 is correctly typed. The abstract syntax for this program is provided here.

```
letdec(bindval(idpat('x'),int('5')),
  [letdec(bindval(idpat('y'),int('6')),
    [apply(id('println'),apply(id('+'),tuple([id('x'),id('y')])))
         ])
  ]).
```

You can check your answer(s) in Section 8.19.1.

Practice 8.2 Minimally, what must the type environment contain to correctly type check the program in Fig. 8.15.
You can check your answer(s) in Section 8.19.2.

```
let val x = 5
    val y = 6
in
  println (x + y)
end
```

Fig. 8.15 test10.sml

8.11 Matches

Matches

There are two alternatives to the *Matches* rule differing only in the syntax of the match.

$$\frac{\begin{array}{c}\forall i\ 1 \le i \le n, \forall j\ 1 < j \le n,\ n \ge 1\\ \varepsilon \vdash id : \alpha \rightarrow \beta,\quad pat_i : \alpha \Rightarrow \varepsilon_{pat_i},\quad \varepsilon_{pat_i} \oplus \varepsilon \vdash e_i : \beta\end{array}}{\varepsilon \vdash id\ pat_1 = e_1\{|\ id\ pat_j = e_j\} : \alpha \rightarrow \beta}$$

or

$$\frac{\begin{array}{c}\forall i\ 1 \le i \le n, \forall j\ 1 < j \le n,\ n \ge 1\\ \varepsilon \vdash id : \alpha \rightarrow \beta,\quad pat_i : \alpha \Rightarrow \varepsilon_{pat_i},\quad \varepsilon_{pat_i} \oplus \varepsilon \vdash e_i : \beta\end{array}}{\varepsilon \vdash id\ pat_1 => e_1\{|\ pat_j => e_j\} : \alpha \rightarrow \beta}$$

The *Matches* type inference rule handles one or more matches in a function definition or other matches occurrence. A match has an identifier (i.e. the name of the function), a pattern, and an expression. Each match takes an argument and returns a value. The argument and pattern must be of type α and the type of the expression must be of type β. In addition, the bindings created by the pattern are part of the environment when the type of the expression is inferred.

```
let fun f(0,y) = y
      | f(x,y) = g(x,x*y)
    and g(x,y) = f(x-1,y)
in
  println (f(10,5))
end
```

Fig. 8.16 test11.sml

Consider the program in Fig. 8.16. This is an example of a program with multiple function declarations separated by the keyword *and*, thus allowing them to be mutually exclusive, which they are. The first function, f has two matches, which the *Matches* rule handles. The abstract syntax for this program includes two *funmatches*, one for each function f and g.

```
letdec(
  funmatches(
    [funmatch(f,
       [match(tuplepat([intpat(0),idpat(y)]),id(y)),
        match(tuplepat([idpat(x),idpat(y)]),apply(id(g),
           tuple([id(x),apply(id(*),tuple([id(x),id(y)]))])))]),
     funmatch(g,
       [match(tuplepat([idpat(x),idpat(y)]),
          apply(id(f),tuple([apply(id(-),tuple([id(x),int(1)])),id(y)
              ])))])]),
  [apply(id(println),apply(id(f),tuple([int(10),int(5)])))])
```

Consulting the AST for the program the two matches for f each include a pattern and the expression after the equals sign. The first expression is the y that is returned for the first match of f. The second match of f returns $g(x, x * y)$.

8.12 Anonymous Functions

AnonFun

$$\frac{[id \mapsto \alpha \rightarrow \beta] \oplus \varepsilon \vdash id\ matches : \alpha \rightarrow \beta}{\varepsilon \vdash fn\ id\ matches : \alpha \rightarrow \beta}$$

An anonymous function is given a name by the parser before a Prolog term is created. Names are needed for code generation. The type checker uses the name only to provide consistency in the way the *Matches* type inference rule is satisfied. However, the identifier is not used by the type inference rule because an anonymous function never calls itself recursively except in the case of a *val rec* binding, where a different identifier is present to be bound to the function. Consider the anonymous function defined in Fig. 8.17. The abstract syntax for this program is as shown here.

```
func(anon@0,[match(idpat(x),apply(id(+),tuple([id(x),int(1)])))])
```

Notice that the compiler has assigned a name to this function. The name *anon@0* is needed by the code generator and also by the *Matches* rule above (only to syntactically match the rule though), but is not used during type inference. Applying this program to the *AnonFun* rule we get this instance.

```
(fn x => x+1)
```

Fig. 8.17 Anonymous function

$$\frac{[anon@0 \mapsto int \rightarrow int] \oplus \varepsilon \vdash anon@0\; x => x + 1 : int \rightarrow int}{\varepsilon \vdash fn\; anon@0\; x => x + 1 : int \rightarrow int}$$

In this instance it doesn't appear much has changed. The *fn* has dropped in the premise. The premise is now an instance of the *Matches* rule which can then be applied to further reduce the proof.

Practice 8.3 Provide a complete proof that the program in Fig. 8.17 is correctly typed.
You can check your answer(s) in Section 8.19.3.

8.13 Sequential Execution

Sequence

$$\frac{\begin{array}{c}\forall i\ 1 \le i \le n, \forall j\ 1 < j \le n,\ n \ge 1\\ \varepsilon \vdash e_i : \alpha_i\end{array}}{\varepsilon \vdash e_1\{; e_j\} : \alpha_n}$$

Sequential execution of expressions results in the last value of the sequence. All other values are discarded. So, the type of a sequence is the type of the last expression evaluated. In the degenerative case, where $n = 1$, the type of the sequence is the type of the only expression in the sequence.

8.14 If-Then and While-Do

If-Then expressions and While-Do expressions have type restrictions on the types of values they can process. The type inference rules provided here describe those restrictions. The IfThen type inference rule was first presented in Chap. 5.

IfThen

$$\frac{\varepsilon \vdash e_1 : bool, \quad \varepsilon \vdash e_2 : \alpha, \quad \varepsilon \vdash e_3 : \alpha}{\varepsilon \vdash if\ e_1\ then\ e_2\ else\ e_3 : \alpha}$$

WhileDo

$$\frac{\varepsilon \vdash e_1 : bool, \quad \varepsilon \vdash e_2 : \alpha}{\varepsilon \vdash while\ e_1\ do\ e_2 : \alpha}$$

```
typecheckExp(Env,ifthen(Exp1,Exp2,Exp3), RT) :-
  typecheckExp(Env,Exp1,bool), typecheckExp(Env,Exp2,RT),
  typecheckExp(Env,Exp3,RT), !.
typecheckExp(Env,ifthen(Exp1,Exp2,Exp3), _) :-
  typecheckExp(Env,Exp1,bool), typecheckExp(Env,Exp2,ThenType),
  typecheckExp(Env,Exp3,ElseType),
  print('Error: Result types of then and else expressions must match.'), nl,
  print('Then Expression type is: '), printType(ThenType,_), nl,
  print('Else Expression type is: '), printType(ElseType,_), nl,
  throw(typeerror('result type mismatch in if-then-else expression')).
typecheckExp(Env,ifthen(Exp1,_,_), _) :-
  typecheckExp(Env,Exp1,Exp1Type), Exp1Type \PYGZbs{=} bool,
  print('Error: Condition of if then expression must have bool type.'), nl,
  print('Condition Expression type was: '), printType(Exp1Type,_), nl,
  throw(typeerror('type not bool in if-then-else expression condition')).
```

Fig. 8.18 If-Then type inference

While reporting *yes* it type checked correctly and here is your type, or *no* it did not type check correctly is what Prolog would do by default, that isn't really enough information to determine where in a program the type checker failed. As the type checker proceeds, certain error messages can be printed. For instance, consider the code for type checking If-Then expressions in Prolog.

The first rule in Fig. 8.18 is the Prolog implementation of the If-Then type inference rule. If the first rule works the cut operator insures that no backtracking will occur to match it another way. If the first rule is not satisfied, then an error message is printed and an exception is thrown to terminate the type checker.

Strictly speaking, an exception does not need to be thrown in the code of Fig. 8.18. The result of the If-Then failure could be the special type *typeerror*. The type inference algorithm is said to by *strict* in *typeerror* which means that once a type results in *typeerror* all types in which it takes part must also result in *typeerror*. However, this still leads to the whole program failing type inference and throwing an exception is a quick and dirty way to terminate the type inference algorithm.

8.15 Exception Handling

Handler

$$\frac{\varepsilon \vdash e : \alpha, \quad [handle@ \mapsto \; exn \rightarrow \alpha] \oplus \varepsilon \vdash handle@\ matches : exn \rightarrow \alpha}{\varepsilon \vdash e\ handle\ matches : \alpha}$$

An exception handler is a polymorphic function as far as the type inference system is concerned, mapping from type *exn* to the type of the expression. Both the expression and its exception handler must have the same result type according to this definition. To implement the handler like a function the identifier *handle@* is bound to the type of the handler.

8.16 Chapter Summary

This is a shorter but denser chapter than some in the text. Type inference is difficult at best to demonstrate on paper. Section 8.10 carries out a complete proof of type correctness as one example from beginning to end of type inference. The type inference system implemented here relies heavily on the unification of Prolog variables to terms. Perhaps the best way to understand this code is to extend it. Implementing type inference rules demands an understanding of how Prolog works. Examining already written type inference rules can help as well.

In spite of it being a challenging topic, inference and unification are two very powerful techniques available to computer programmers through the use of Prolog. Unification provides the means to work both backwards and forwards or anywhere in between as was pointed out with the *append* predicate in the last chapter. In terms of type inference, one important aspect is being able to assign a type to an expression before you know what its type is. By assigning a Prolog variable that will be unified to an actual type later, the type inference can be written very declaratively, like the inference rules themselves, without regard to exactly the order that information is known. That's the power of Prolog. The unification algorithm makes declarative programming in Prolog possible.

Type checking, without type inference, is effective and simpler to implement but costs the programmer more in having to explicitly declare types of each variable. Being explicit about types is not always a bad thing. Even the SML compiler needs a little help sometimes by declaring the type of a function parameter. Regardless of the language, every type checker engages in some type inference. Standard ML's type inference system differs from other language implementations by the extent to which types are inferred.

8.17 Review Questions

1. What appears above and below the line in a type inference rule?
2. Why don't infix operators appear in the abstract syntax of programs handled by the type checker?
3. What does *typevar* represent in Fig. 8.8?
4. What does *typeerror* represent in Fig. 8.8?
5. What does the *type* of the list [("hello",1,true)] look like as a Prolog term?
6. What is the type environment?
7. Give an example of the use of the overlay operator.
8. What pattern(s) are used in this let expression?

```
let val (x,y,z) = (1+s+s2{h}ellop{,}1,true) in println x end
```

   What is the pattern as a Prolog term?
9. Give an example where the *Sequence* rule might be used to infer a type.
10. Give a short example of where the *Handler* rule might be used to infer a type.

8.18 Exercises

1. The following program does not compile correctly or typecheck correctly using the mlcomp compiler and type inference system. However, it is a valid Standard ML program. Modify both the mlcomp compiler and type checker to correctly compile and infer its type. This program is included in the compiler project as test20.sml.

```
let val [(x,y,z)] = [(1+s+s2{h}ellop{,}1,true)] in println x end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(bindval(listpat([tuplepat([idpat(x),idpat(y),idpat(z)])]),
       listcon([tuple([str("hello"),int(1),bool(true)])])),
       [apply(id(println),id(x))])
val [(x,y,z)] : (str * int * bool) list
val it : unit
The program passed the typechecker.
```

2. Implement the Prolog type predicates to get the following program to type check successfully. This program is test14.sml in the mlcomp compiler project. This will involve writing type checking predicates for matching, boolean patterns, integer patterns, and sequential execution.

```
let fun f(true,x) = (println(x); g(x-1))
      | f(false,x) = g(x-1)
    and g 0 = ()
      | g x = f(true,x)
in
       g(10)
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(funmatches([funmatch(f,[match(tuplepat([boolpat
    (true),idpat(x)]),
        expsequence([apply(id(println),id(x)),apply(id(
           g),apply(id(-),
        tuple([id(x),int(1)])))])),match(tuplepat([
           boolpat(false),idpat(x)]),
        apply(id(g),apply(id(-),tuple([id(x),int(1)])))
           )]),funmatch(g,[match(intpat(0),
        tuple([])),match(idpat(x),apply(id(f),tuple([
           bool(true),id(x)])))])]),
        [apply(id(g),int(10))])
val f = fn : bool * int -> unit
val g = fn : int -> unit
val it : unit
The program passed the typechecker.
```

3. Implement enough of the type checker to get test12.sml to type check correctly. This will mean writing the *WhileDo* inference rule as a Prolog predicate, implementing the *Match* rule's predicate called *typecheckMatch*, and the type inference predicate for sequential execution named *typecheckSequence* as defined in the *Sequence* rule. The code for test12.sml is given here for reference.

```
let val zero = 0
    fun fib n =
    let val i = ref zero
        val current = ref 0
        val next = ref 1
        val tmp = ref 0
    in
      while !i < n do (
        tmp := !next + !current;
        current := !next;
        next := !tmp;
        i := !i + 1
      );
      !current
    end
    val x = Int.fromString(input(l+s+s2{"lease enter an integer:"))
    val r = fib(x)
in
  print l+s+s2{F}ib(p{;}
  print x;
  print l+s+s2{)} is p{;}
  println r
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(bindval(idpat(zero),int(0)),[letdec(funmatches
    ([funmatch(fib,
   [match(idpat(n),letdec(bindval(idpat(i),apply(id(
       ref),id(zero))),
   [letdec(bindval(idpat(current),apply(id(ref),int(0)
       )),
   [letdec(bindval(idpat(next),apply(id(ref),int(1))),
   [letdec(bindval(idpat(tmp),apply(id(ref),int(0))),
   [whiledo(apply(id(<),tuple([apply(id(!),id(i)),id(n
       )])),
   expsequence([apply(id(:=),tuple([id(tmp),apply(id
       (+),tuple([apply(id(!),id(next)),
   apply(id(!),id(current))]))])),apply(id(:=),tuple([
       id(current),apply(id(!),
   id(next))])),apply(id(:=),tuple([id(next),apply(id
       (!),id(tmp))])),apply(id(:=),
   tuple([id(i),apply(id(+),tuple([apply(id(!),id(i)),
       int(1)]))]))])),apply(id(!),
   id(current))])])])])]))])]),[letdec(bindval(idpat(x),
       apply(id(Int.fromString),
   apply(id(input),str("Please enter an integer:")))),
  [letdec(bindval(idpat(r),apply(id(fib),id(x))),[
      apply(id(print),str("Fib(")),
  apply(id(print),id(x)),apply(id(print),str(") is"))
      ,apply(id(println),id(r))])])])])
val zero : int
val i : int ref
val current : int ref
val next : int ref
val tmp : int ref
val fib = fn : int -> int
val x : int
val r : int
val it : unit
The program passed the typechecker.
```

4. Add support to the type checker to correctly infer the types of *case* expressions in Small. The following program should type check correctly once this project is completed. This test is in test15.sml in the mlcomp compiler project. This will involve writing code to correctly type check matches according to the *Match* rule. If case statements are not yet implemented in the compiler, support must be added to the compiler to parse *case* expressions, build an AST for them, and write their AST to the *a.term* file.

```
let val x = 4
in
  println
    (case x of
       1 => "hello"
     | 2 => "how"
     | 3 => "are"
     | 4 => "you")
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(bindval(idpat(x),int(6)),[apply(id(println),caseof(id(x),
       [match(intpat(1),str("hello")),match(intpat(2),str("how")),
       match(intpat(3),str("are")),match(intpat(4),str("you"))]))])
val x : int
val it : unit
The program passed the typechecker.
```

5. Add support to the type checker to correctly infer the types for test7.sml. The code is provided below for reference. Support will need to be added to infer the types of anonymous functions defined in the rule *AnonFun*, matching defined in the rule *Matches*, and the *ConsPat* rule.

```
let fun append nil L = L
      | append (h{:}{:}t) L = h {:}{:} (append t L)
    fun appendOne x = (fn nil => (fn L => L)
                        | h{:}{:}t => (fn L => h {:}{:} (appendOne t L
                            ))) x
in
  println(append [1,2,3] [4]);
  println(appendOne [1,2,3] [4])
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
Here is the AST
letdec(funmatches([funmatch(append,[match(idpat(v0),
    func(anon@3,
[match(idpat(v1),apply(func(anon@2,[match(tuplepat([
    idpat(nil),idpat(L)]),id(L)),
match(tuplepat([infixpat({:}{:},idpat(h),idpat(t)),
    idpat(L)]),apply(id({:}{:}),
tuple([id(h),apply(apply(id(append),id(t)),id(L))])))
    ]),
tuple([id(v0),id(v1)]))))])))]),[letdec(funmatches([
    funmatch(appendOne,
[match(idpat(x),apply(func(anon@6,[match(idpat(nil),
    func(anon@4,
[match(idpat(L),id(L))])),match(infixpat({:}{:},idpat(
    h),idpat(t)),
func(anon@5,[match(idpat(L),apply(id({:}{:}),tuple([id
    (h),apply(apply(id(appendOne),id(t)),
id(L))])))]))]),id(x)))])]),[apply(id(println),apply(
    apply(id(append),
listcon([int(1),int(2),int(3)])),listcon([int(4)]))),
    apply(id(println),
apply(apply(id(appendOne),listcon([int(1),int(2),int
    (3)])),listcon([int(4)])))])])
val append = fn : 'a list -> 'a list -> 'a list
val appendOne = fn : 'a list -> 'a list -> 'a list
val it : unit
The program passed the typechecker.
```

6. Add support for type inference for recursive bindings. The following program, saved as test19.sml in the Small compiler project, is a valid program with a recursive binding. It will type check correctly if the *ValRecDec* type inference rule is implemented. Write the code to get this program to pass the type checker as a valid program.

```
let val rec f = (fn 0 => 1
                  | x => x * (f (x-1)))
in
   println(f 5)
end
```

Output from the type checker should appear as follows.

```
Typechecking is commencing...
    Here is the AST
    letdec(bindvalrec(idpat(f),func(anon@0,[match(
        intpat(0),int(1)),match(idpat(x),
       apply(id(*),tuple([id(x),apply(id(f),apply(id
          (-),tuple([id(x),int(1)]))))])))])),
       [apply(id(println),apply(id(f),int(5)))])
    val f = fn : int -> int
    val it : unit
    The program passed the typechecker.
```

7. Currently the type checker allows duplicate identifiers in compound patterns like listPat and tuplePat. Standard ML does not allow duplicate identifiers in patterns. The type checker uses the *append* predicate to combine pattern binding environments. This is not good enough. Find the locations in the type checker where pattern environments are incorrectly appended and rewrite this code to enforce that all identifiers within a pattern must be unique. If not, you should print an error message like *"Error: duplicate variable in pattern(s): x"* to indicate the problem and typechecking should end with an error.
8. Currently, the abstract syntax and parser of *Small* includes support for the wildcard pattern in pattern matching, but the type checker does not support it. Add support for wildcard patterns, write a test program, and test the compiler and type checker. Be sure to write a type inference rule for wildcard patterns first.
9. Currently, the abstract syntax and parser of *Small* includes support for the *as* pattern in pattern matching, but the type checker does not support it. Add support for *as* patterns, write a test program, and test the compiler and type checker. The *as* pattern comes up when you write a pattern like *L as h::t* which assigns *L* as a pattern that represents the same value as the compound pattern of *h::t*. Be sure to write a type inference rule for *as* patterns first.

8.19 Solutions to Practice Problems

8.19.1 Solution to Practice Problem 8.1

Proving this requires a proof like was done in the chapter. Rules involved include *Let*, *ValDec*, *IdPat*, *TupleCon*, and *FunApp*. Technically, the *Sequence* rule is also required, but only in the degenerative case (i.e. when $n = 1$).

8.19.2 Solution to Practice Problem 8.2

Minimally the environment must contain *println* bound to a function type of $\alpha \rightarrow unit$ and the $+$ function bound to a function type of $int \times int \rightarrow int$.

8.19.3 Solution to Practice Problem 8.3

The *AnonFun* rule is applied first which requires the *Matches* rule be applied. The *Matches* rule requires the use of the *IdPat* rule and the *FunApp* rule. Finally, the *IntCon* rule is needed to complete the proof.

Appendix A: The JCoCo Virtual Machine Specification

9

JCoCo is a virtual machine which includes a built-in assembler. JCoCo executes assembly language programs by first processing the assembly language program and then executing it. The processing of the assembly language program is called *assembling*. The assembly language supported by JCoCo is defined by a BNF grammar. The grammar specifies how JCoCo assembly language programs are constructed. The grammar for the JCoCo virtual machine assembly language is provided in Fig. 9.1.

According to the BNF in Fig. 9.1 a JCoCo program is a sequence of class or function definitions. Each class definition may consist of one or more function definitions. Each function definition has several parts including a sequence of JCoCo virtual machine instructions like *LOAD_CONST*, *STORE_FAST*, and many others. The complete specification of instructions supported by JCoCo is provided at http://cs.luther.edu/~leekent/JCoCo and in this appendix. The complete syntax of the language is given in Fig. 9.1. There are just a few things to note in the BNF.

- Instructions may have as many labels defined on them as necessary. The definition of labeled instruction is recursive.
- The use of <null> indicates an empty production. For instance, a FunctionList may be empty meaning that there might not be a function list in a function definition. In this case that simply means a function might or might not have some nested functions.
- `[` and `]` indicate an optional part of a JCoCo program.
- Of course, the ... indicates there are more Unary and Binary mnemonics that are not listed in the BNF. The complete list of instructions and descriptions of each of them are given in this appendix.
- The JCoCo language is not line oriented. This BNF completely describes the language which has no line requirements. However, formatting a program like the disassembler will help in the clarity of written programs.

```
CoCoAssemblyProg ::= ClassFunctionListPart EOF

ClassFunctionListPart ::= ClassFunDef ClassFunctionList

ClassFunctionList ::= ClassFunDef ClassFunctionList | <null>

ClassFunDef ::= ClassDef | FunDef

ClassDef ::= Class colon Identifier [ ( Identifier ) ] BEGIN ClassFunctionList END

FunDef ::= Function colon Identifier slash Integer ClassFunctionList ConstPart
           LocalsPart FreeVarsPart CellVarsPart GlobalsPart BodyPart

ConstPart ::= <null> | Constants colon ValueList

ValueList ::= Value ValueRest

ValueRest ::= comma ValueList | <null>

Value ::= None | True | False | Integer |
          Float | String | code(Identifier) | TupleVal

TupleVal ::= ( ValueList )
(* the Scanner sees None, True, False, as Identifiers. *)

LocalsPart ::= <null> | Locals colon IdList

FeeVarsPart ::= <null> | FreeVars colon IdList

CellVarsPart ::= <null> | CellVars colon IdList

IdList ::= Identifier IdRest

IdRest ::= comma IdList | <null>

GlobalsPart ::= <null> | Globals colon IdList

BodyPart ::= BEGIN InstructionList END

InstructionList ::= <null> | LabeledInstruction InstructionList

LabeledInstruction ::= Identifier colon LabeledInstruction |
                       Instruction | OpInstruction

Instruction ::= STOP_CODE | NOP | POP_TOP | ROT_TWO | ROT_THREE | ...

OpInstruction ::= OpMnemonic Integer | OpMnemonic Identifier

OpMnemonic ::= LOAD_CONST | STORE_FAST | SETUP_LOOP | COMPARE_OP |
               POP_JUMP_IF_FALSE | ...
```

Fig. 9.1 The BNF for the JCoCo assembly language

9.1 Types

JCoCo supports the types given in Fig. 9.2.

Type	Description
type	the type of all types, including itself
NoneType	the type of None; None is a special value in Python referring to nothing. In other words, it is the null pointer in the JCoCo machine.
bool	the type of boolean types; True and False are the two boolean values.
int	integer types implemented as a C++ int; Depending on the C++ compiler this could be a 32-bit or 64-bit integer.
float	the type for floating point numbers; has the same precision as a C++ double precision floating point number.
str	the type of all strings; implemented using the C++ string class.
str_iterator	the type for iterators over strings
function	the type of all user-defined functions
method	the type of all user-defined methods
built_in_func-tion_or_method	the type of all built-in functions or methods
range	the type of range objects; these are lazily generated sequences of integers
range_iterator	the type of range iterator objects; objects of this type yield consecutive integers in their associated range.
Exception	the type of all exceptions
list	the type of list objects like the original Python list objects
list_iterator	the type of iterators over lists
funlist	the type of functional list objects; This is a new type not supported in Python with the properties of lists from functional languages that are constructed from a head and a tail; funlist values are immutable as opposed to the list type.
funlist_iterator	the type of iterators over funlists.
tuple	Tuples are like lists, but are immutable.
tuple_iterator	the type of iterators over tuples
dict	the type of dictionaries, i.e. maps
dict_keyiterator	the type of dictionary key iterators
code	type of code objects (i.e. functions)
cell	the type of a reference objects
super	the type of super class objects used in the presence of inheritance
<class>	the type of any instance of the programmer-defined class.

Fig. 9.2 JCoCo supported types

9.2 JCoCo Magic and Attr Methods

One of the powerful features of the Python language comes from methods being looked up on objects at run-time. This means that new types of objects can easily be added to the language because the virtual machine instructions presented in this appendix will polymorphically call the proper methods since lookup happens at run-time. In support of this, JCoCo, like Python, has what have traditionally been called magic methods. These methods typically begin and end with two underscores. Magic methods are used by instructions as needed. For instance, the __add__ magic method is used by the BINARY_ADD instruction.

JCoCo includes support for all the magic methods that are defined by Python. While support is there for the whole list, not all magic methods are implemented on each type of object. The magic methods that are supported are controlled by the type of the object. When a magic method is called, the magic method is first looked up on the type and if it is supported, the call is made. Otherwise, an IllegalOperationException is raised. The use of magic methods is illustrated in the descriptions of the JCoCo instructions in this appendix.

The possible magic methods include the following: __cmp__, __eq__, __ne__, __lt__, __gt__, __le__, __ge__, __pos__, __neg__, __abs__, __invert__, __ round__, __floor__, __ceil__, __trunc__, __add__, __sub__, __mul__, __ floordiv__, __div__, __truediv__, __mod__, __divmod__, __pow__, __lshift__, __rshift__, __and__, __or__, __xor__, __radd__, __rsub__, __rmul__, __ rfloordiv__, __rdiv__, __rtruediv__, __rmod__, __rdivmod__, __rpow__, __ rlshift__, __rand__, __ror__, __rxor__, __iadd__, __isub__, __imul__, __ ifloordiv__, __idiv__, __itruediv__, __imod__, __ipow__, __ilshift__, __iand__, __ior__, __ixor__, __int__, __long__, __float__, __bool__, __cmplex__, __ oct__, __hex__, __index__, __coerce__, __str__, __list__, vfunlist__, __repr__, __unicode__, __formatv, __hash__, __nonzero__, __dir__, __sizeofv, __getattr __, __setattr__, __delattr__, __getattribute__, __len__, __getitem__, __setitem __, __delitem__, __reversed__, __contains__, __missing__, __instancecheck__, __subclasscheckv, __call__, __copy__, __deepcopyv, __iter__, __next__, __type __, __excmatchv.

The last two magic methods are specific to CoCo and JCoCo but not a part of Python. The __type__ magic method is called when the type function is called on an object. The __excmatch__ magic method is called when matching an exception in an exception handler.

In addition, some objects have additional methods defined on them that are accessed like traditional method calls on objects. For instance, str objects have a split method that can be called to split a string on separator characters. The list of attr methods defined in JCoCo are *split* on strings, *append* on lists, *head* on funlists, *tail* on funlists, *concat* on strings, *keys* on dictionaries, and *values* on dictionaries. The *head* and *tail* methods are not found in Python but are defined in CoCo and JCoCo to support *funlist* objects which are defined to have a head and a tail.

9.3 Global Built-In Functions

JCoCo supports the following globally available built-in functions. These functions are not associated with any one type. When they are called, they polymorphically handle the arguments passed to them in their own manner as described.

print is a built-in function that prints a variable number of arguments to standard output, followed by a newline character, and returns None, just as print does in Python. The objects passed to print are printed by calling the __str__ magic method on each of them and appending their strings with an extra space between each pair of objects.

fprint prints exactly one argument. This is a built-in function that is specific to JCoCo and is not part of the standard Python language. It prints its argument by calling the __str__ magic method on the object to convert it to a string. This function returns itself, which can be useful when chaining together fprint expressions.

tprint prints exactly one argument, which may be a tuple, and returns None. tprint can be thought of as tuple print, because if a tuple is provided, the contents of the tuple are printed, separated by spaces, just as print does. However, tprint takes only one argument which may be a tuple. print takes a variable number of arguments. tprint is specific to JCoCo and is not part of the standard Python language. The values of the tuple are converted to strings using the __str__ magic method on each object. None is returned by tprint.

input is a built-in function that prints its prompt to standard output and returns one line of input as a string, just as input does in Python.

iter is a built-in function that constructs and returns an iterator over the object that is passed to it, just as Python's iter function works. This is implemented by calling the viter__ magic method on the object.

len is a built-in function that returns the length of the sequence that is passed to it. It does this by calling the __len__ magic method on the object given to it.

concat is built-in function that returns a string representation of the elements of its sequence concatenated together. The concat function in turn calls the concat method on the object that is passed to it.

int, **float**, **str**, **funlist**, **type**, and **bool** are all calls to types. When the type is called, the corresponding magic method of __int__, __float__, __strv, __funlistv, __type__, or __bool__ is called on the object that is passed to it. In this way, the object itself is in charge of how it is converted to the specified type.

range is a call to the range type that constructs a range object over the specified range. As in Python, the range function has 1, 2, or 3 arguments passed to it, representing the start, stop, and increment of the range of integer values. The start and increment values are optional.

Exception is a call to the exception type that constructs and returns an exception object that may be raised or thrown and caught by an exception handler.

super may be called in an instance method to gain access to the base class of an object. Single inheritance is supported in JCoCo. Unlike Python, multiple inheritance is not supported.

9.4 Virtual Machine Instructions

This is a subset of the full Python 3.2 instruction set with the addition of a few extra instructions and a couple of minor differences.

In the instructions in this appendix, TOS refers to the top element on the operand stack. TOS1 refers to the element second from the top of the operand stack. TOS2, and so on are similarly defined.

JCoCo instructions each take up exactly one location of space. The Python Virtual Machine uses one or more bytes for each instruction and therefore some instructions are composed of multiple bytes. JCoCo does not store its instructions as bytes and therefore each instruction takes exactly one location within the JCoCo virtual machine interpreter.

The Python Virtual machine defines some branching instructions as absolute jumps and other as relative jumps, that being relative to the current PC. JCoCo differs from the Python Virtual Machine in this regard. In the instructions any jump or branch is to an absolute location. Generally, the target of a branch or jump will be specified using a label. If labels are used for all branch and jump targets then this difference will only be noticable when looking at the assembled program. When read by the JCoCo assembler, the labels are converted to target locations which are always absolute addresses.

9.5 Arithmetic Instructions

BINARY_ADD

Implements TOS = TOS1 + TOS by making the call TOS1.__add__(TOS).

BINARY_SUBTRACT

Implements TOS = TOS1 - TOS by making the call TOS1.__sub__(TOS).

BINARY_MULTIPLY

Implements TOS = TOS1 * TOS by making the call TOS1.__mul__(TOS).

BINARY_MODULO

Implements TOS = TOS1 % TOS by making the call TOS1.__mod__(TOS).

BINARY_FLOOR_DIVIDE

Implements TOS = TOS1 // TOS by making the call TOS1.__floordiv__(TOS).

BINARY_TRUE_DIVIDE

Implements TOS = TOS1 / TOS by making the call TOS1.__truediv__(TOS).

BINARY_POWER

Implements TOS = TOS1 ** TOS by making the call TOS1.__pow__(TOS).

INPLACE_ADD

Implements in-place TOS = TOS1 + TOS. Exactly the same as BINARY_ADD by making the call TOS1.__add__(TOS).

9.6 Load and Store Instructions

BINARY_SUBSCR

Implements TOS=TOS1[TOS]. This instruction provides indexing into a list, tuple, or other object that supports subscripting. This is implemented as TOS1. __getitem__(TOS).

DELETE_FAST(namei)

This instruction does nothing in JCoCo which varies from the Python implementation. The purpose of this instruction seems to be implementation dependent. In the Python Virtual Machine it performs cleanup after an exception has occurred. The handling of exceptions is different in JCoCo so this instruction exists to make it work with the disassembler, but it is ignored.

LOAD_ATTR(namei)

Replaces TOS with getattr(TOS,Globals[namei]). An attribute is usually a method associated with some object.

LOAD_CLOSURE(i)

Pushes a reference to the cell contained in slot i of the cell and free variable storage. The name of the variable is CellVars[i] if i is less than the length of CellVars. Otherwise it is FreeVars[i-len(CellVars)].

LOAD_CONST(consti)

Argument *consti* is a zero-based integer. Pushes Constants[consti] onto the stack.

LOAD_DEREF(i)

Loads the cell contained in slot i of the cell and free variable storage. Pushes a reference to the object the cell contains on the stack.

LOAD_FAST(namei)

Argument *namei* is a zero-based integer. Pushes a reference to Locals[namei] onto the stack.

LOAD_GLOBAL(namei)

Argument *namei* is a zero-based integer. Loads the Globals[namei] onto the stack.

LOAD_NAME(var_num)

Loads a value from the locals dictionary that is named in the globals at var_num. It pushes the loaded value onto the stack. Preference should be given to using LOAD_FAST if possible.

LOAD_BUILD_CLASS

Loads the built-in function for building classes onto the operand stack. This function, when called, takes two or three arguments. When there are two arguments passed to the build class function the name of the class must be at TOS. The function that will instantiate the class must be at TOS1. When called with CALL_FUNCTION, this built-in function for building classes will be at TOS2. This built-in function leaves the instantiated class on the top of the stack.

When called with three arguments the name of the base class, from which this class will inherit, is located at TOS and the other two arguments are in the same order under the name of the base class.

STORE_ATTR(var_num)

Stores the object found at TOS1 in the object found at TOS in an attribute name found in the globals at var_num.

STORE_DEREF(i)

Stores TOS into the cell contained in slot i of the cell and free variable storage.

STORE_FAST(namei)

Argument *namei* is a zero-based integer. Stores TOS into the Locals[namei].

STORE_LOCALS

Used during class instantiation. Pops the dictionary from TOS and uses it as the locals for the currently executing function, replacing any locals dictionary already in use. The popped dictionary is the attribute dictionary of the class which includes methods to be instantiated upon object instantiation for objects of the class.

STORE_NAME(var_num)

Uses the name found in the globals at var_num to store a named value in the locals dictionary. Preference should be given to STORE_FAST if possible.

STORE_SUBSCR

Implements TOS1[TOS]=TOS2. The instruction provides indexing into a mutable list or other object that supports subscripting. The instruction is implemented by calling TOS1.__setitem__(TOS,TOS2).

9.7 List, Tuple, and Dictionary Instructions

BUILD_MAP(initial_capacity)

Creates an empty dictionary object and pushes it onto the stack. The initial capacity is ignored by JCoCo.

STORE_MAP

Performs TOS2[TOS]=TOS1. TOS1 is the value to be stored at key TOS in dictionary TOS2.

BUILD_TUPLE(count)

Creates a tuple consuming count items from the stack, and pushes the resulting tuple onto the stack.

SELECT_TUPLE(count)

Pushes the contents of the tuple with count elements onto the operand stack. The count must match the tuple's size or an illegal operation exception will be thrown. The elements of the tuple are pushed so the left-most element is left on the top of the stack. This instruction is not part of the Python Virtual Machine. It is JCoCo specific.

BUILD_LIST(count)

Works as BUILD_TUPLE, but creates a list.

BUILD_FUNLIST

Works as BUILD_TUPLE, but creates a list.

SELECT_FUNLIST

This instruction pushes the head and the tail (which is a funlist) onto the operand stack. The head of the list is left on the top of the operand stack. The tail is below it on the stack. This instruction is JCoCo specific.

CONS_FUNLIST

Pops two elements from the operand stack. TOS should be a funlist and TOS-1 should be an element. The instruction create a new funlist from the two pieces with TOS-1 the head and TOS the tail of the new list. It pushes this new list onto the operand stack. This instruction is JCoCo specific.

9.8 Stack Manipulation Instructions

POP_TOP

Removes the top-of-stack (TOS) item.

ROT_TWO

Swaps the two top-most stack items.

DUP_TOP

Duplicates the reference on top of the stack.

9.9 Conditional and Iterative Execution Instructions

GET_ITER

Implements TOS=iter(TOS).

BREAK_LOOP

Terminates a loop due to a break statement.

POP_BLOCK

Removes one block from the block stack. Per frame, there is a stack of blocks, denoting nested loops, try statements, and such.

POP_EXCEPT

Removes one block from the block stack. The popped block must be an exception handler block, as implicitly created when entering an except handler. In addition to popping extraneous values from the frame stack, the last three popped values are used to restore the exception state.

END_FINALLY

Terminates a finally clause. The interpreter recalls whether the exception has to be re-raised, or whether the function returns, and continues with the outer-next block.

COMPARE_OP(opname)

Performs a Boolean operation. Both TOS1 and TOS are popped from the stack and the boolean result is left on the operand stack after the execution of this instruction. opname is an integer corresponding to the following comparisons. In each case the comparison corresponding to opname is shown along with the magic method call that implements the comparison.

opname	Comparison Operation
0	TOS1 < TOS as TOS1.__lt__(TOS)
1	TOS1 <= TOS as TOS1.__le__(TOS)
2	TOS1 = TOS as TOS1.__eq__(TOS)
3	TOS1 != TOS as TOS1.__ne__(TOS)
4	TOS1 > TOS as TOS1.__gt__(TOS)
5	TOS1 >= TOS as TOS1.vge__(TOS)
6	TOS1 contains TOS as TOS1.__contains__(TOS)
7	TOS1 not in TOS as TOS1.__notin__(TOS)
8	TOS1 is TOS as TOS1.is_(TOS)
9	TOS1 is not TOS as TOS1.is_not(TOS)
10	Exception TOS1 matches TOS as TOS1.__excmatch__(TOS)

JUMP_FORWARD(target)

Sets the Program Counter (PC) to target.

POP_JUMP_IF_TRUE(target)

If TOS is true, sets the bytecode counter to target. TOS is popped.

POP_JUMP_IF_FALSE(target)

If TOS is false, sets the bytecode counter to target. TOS is popped.

JUMP_ABSOLUTE(target)

Set bytecode counter to target.

FOR_ITER(target)

TOS is an iterator. Call its __next__() method. If this yields a new value, push it on the stack (leaving the iterator below it). If the iterator indicates it is exhausted TOS is popped, and the PC is set to target.

SETUP_LOOP(target)

Pushes a block for a loop onto the block stack. The block spans from the current instruction to target.

SETUP_EXCEPT(target)

Pushes a try block from a try-except clause onto the block stack. Target points to the first except block.

SETUP_FINALLY(target)

Pushes a try block from a try-except clause onto the block stack. Target points to the finally block.

RAISE_VARARGS(argc)

This instruction varies from the Python version slightly. In JCoCo the argc must be one. This is because exceptions in JCoCo automatically contain the traceback which is not necessarily the case in the Python Virtual Machine. The argument on the stack should be an exception. The exception is thrown by this instruction.

9.10 Function Execution Instructions

RETURN_VALUE

Returns with TOS to the caller of the function.

CALL_FUNCTION(argc)

Calls a function. The *argc* indicates the number of positional parameters, the high byte the number of keyword parameters. On the stack, the opcode finds the keyword parameters first. For each keyword argument, the value is on top of the key. Below the keyword parameters, the positional parameters are on the stack, with the right-most parameter on top. Below the parameters, the function object to call is on the stack. Pops all function arguments, and the function itself off the stack, and pushes the return value.

MAKE_FUNCTION(argc)

Pushes a new function object on the stack. TOS is the code associated with the function. The function object is defined to have argc default parameters, which are found below TOS.

MAKE_CLOSURE(argc)

Creates a new function object, sets its closure, and pushes it on the stack. TOS is the code associated with the function, TOS1 the tuple containing cells for the closure's free variables. The function also has argc default parameters, which are found below the cells.

9.11 Special Instructions

BREAK_POINT

Pauses execution of the program and drops the JCoCo virtual machine into the interactive debugger.

Appendix B: The Standard ML Basis Library

10

Following is a subset of the Standard ML Basis Library. The Basis Library is covered in more detail at http://www.standardml.org/Basis. Documentation for these structures is found in this appendix.

- Bool
- Int
- Real
- Char
- String
- List
- Array
- TextIO

Other structures exist on the Basis website. The descriptions provided here may be helpful as well. Each function, along with its signature, is provided for each of the structures listed in this appendix.

10.1 The Bool Structure

This is the signature of the functions Bool structure. In addition to the not operator, SML defines the andalso and orelse operators which implement shortcircuit logic. More information can be found at http://www.standardml.org/Basis/bool.html.

datatype bool = false | true

The bool datatype is either false or true.

K.D. Lee, *Foundations of Programming Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-319-70790-7_10

val not : bool -> bool

not true = false, not false = true.

val toString : bool -> string

Converts a true/false value to a string for printing or other purposes.

val fromString : string -> bool option

Converts from a string to a bool. An option is either *NONE* or *SOME val*. If the string cannot be converted to a bool (i.e. it does not contain true or false), then *NONE* is returned. Otherwise *SOME true* or *SOME false* is returned. Pattern-matching can be used to determine the return value.

val scan : (char,'a) StringCvt.reader -> (bool,'a) StringCvt.reader

This behaves like *fromString* except that the remaining character stream is returned along with the value if a bool is found in the stream.

10.2 The Int Structure

Implementing the INTEGER signature, the Int structure contains the *int* type. Integer precision is platform dependent. Normally 32-bit or 64-bit precision is available depending on the platform. More information can be found at http://www.standardml.org/Basis/integer.html.

type int

The type of integers.

val precision : Int31.int option

An option indicating the precision of integers. For instance, *SOME 31* indicating 32-bit integers from -2^{31} to $2^{31}-1$. If the value is *NONE* it indicates arbitrary precision.

val minInt : int option

val maxInt : int option

Minimum and maximum integer values given the precision available. *NONE* if integers have arbitrary precision.

val toLarge : int -> IntInf.int

val fromLarge : IntInf.int -> int

Conversion functions from and to large integers.

val toInt : int -> Int31.int

val fromInt : Int31.int -> int

Conversion functions from and to 32-bit integers. Depending on implementation these may be identity functions.

val ~ : int -> int

Unary negation. ~6 is a negative 6.

val + : int * int -> int

val - : int * int -> int

val * : int * int -> int

val div : int * int -> int

val mod : int * int -> int

Typical integer operations. Note that *div* and *mod* are infix operators returning the integer division and remainder respectively. For instance, *6 div 4 = 1* and *6 mod 4 = 2*. These operations are infix operators.

val quot : int * int -> int

val rem : int * int -> int

These two operations reflect that most hardware implementations of integer division behave differently than the mathematical definition used by *div* and *mod* for negative integers. Consider the following.

```
- val x = ~6;
val x = ~6 : int
- x mod 4;
val it = 2 : int
- x div 4;
val it = ~2 : int
- Int.quot(x,4);
val it = ~1 : int
- Int.rem(x,4);
val it = ~2 : int
```

This shows that *mod* and *div* factor *–6* as *–2 * 4 +2* while *quot* and *rem* factor *–6* as *–1 * 4 +–2*. The mathematical definition of *mod* always results in a positive remainder. However, computer hardware often calculates using *quot* and *rem* semantics possibly resulting in faster calculations.

val min : int * int -> int

val max : int * int -> int

Maximum and minimum functions of two integers. Returns the max or min value of the pair of integers.

val abs : int -> int

Returns the absolute value.

val sign : int -> Int31.int

Returns either 1 or -1 depending on the sign of the integer.

val sameSign : int * int -> bool

True or false depending on the two integers.

val > : int * int -> bool

val >= : int * int -> bool

val < : int * int -> bool

val <= : int * int -> bool

Relational operators for the ordering of integers. These operators are infix operators.

val compare : int * int -> order

Returns one of the *order* values of *GREATER*, *LESS*, or *EQUAL* depending on the integers.

val toString : int -> string

val fromString : string -> int option

val scan : StringCvt.radix-> (char,'a) StringCvt.reader -> (int,'a) StringCvt. reader

val fmt : StringCvt.radix -> int -> string

Conversion functions for integer to string and streams. See the Bool structure for descriptions. The *StringCvt.radix* may be one of StringCvt.BIN, StringCvt.OCT, StringCvt.DEC, or StringCvt.HEX for conversion to/from their respective bases.

10.3 The Real Structure

Real numbers in Standard ML, and any other programming language, are approximations for Real numbers in Mathematics. They are always precisely the same.

The Real numbers of Standard ML conform to the underlying architecture's implementation of double precision floating point numbers. Typically, this standard is attributed to the IEEE. More information on Standard ML Reals can be found at http://www.standardml.org/Basis/real.html.

type real

The type of Real numbers. Type *real* are approximations of Real numbers.

val pi : real

val e : real

Constant values for convenience for *pi* and *e*. *e* is the base of natural log values, *ln e = 1*.

val Math.sqrt : real -> real

The square root of a non-negative real yields a real. For negative numbers it yields *nan* which stands for *Not A Number*.

val Math.sin : real -> real

val Math.cos : real -> real

val Math.tan : real -> real

val Math.asin : real -> real

val Math.acos : real -> real

val Math.atan : real -> real

val Math.atan2 : real * real -> real

Various trigonometric functions.

val Math.exp : real -> real

This raises *e* to the specified power.

val Math.pow : real * real -> real

Raises the first argument to the power specified by the second argument.

val Math.ln : real -> real

val Math.log10 : real -> real

Natural and log base 10 functions.

val Math.sinh : real -> real

val Math.cosh : real -> real

val Math.tanh : real -> real

Hyperbolic functions.

val radix : int

The base used in the floating point representation, either 2 or 10.

val precision : int

The number of digits in the mantissa in the base specified by radix.

val maxFinite : real

val minPos : real

val minNormalPos : real

val posInf : real

val negInf : real

Various constant values.

val + : real * real -> real

val - : real * real -> real

val * : real * real -> real

val / : real * real -> real

Normal binary operations. These operators are infix operators.

val *+ : real * real * real -> real

val *- : real * real * real -> real

Multiply by a factor and add a term as in *+(6.0, 5.0, 3.0) which yields 33.0.

val ~ : real -> real

Unary negation.

val abs : real -> real

Absolute value.

val min : real * real -> real

val max : real * real -> real

Binary max and min.

val sign : real -> int

Returns -1 or 1 depending on the sign.

val signBit : real -> bool

True if negative and false otherwise.

val sameSign : real * real -> bool

True if both have same sign.

val copySign : real * real -> real

The result is the first argument with the sign of the second argument.

val compare : real * real -> order

val compareReal : real * real -> IEEEReal.real_order

Returns *GREATER*, *LESS*, or *EQUAL* depending on how the first argument compares to the second. The compareReal has slightly different semantics for unordered real numbers (i.e. *nan*) returning IEEEReal.UNORDERED in those cases.

val < : real * real -> bool

val <= : real * real -> bool

val > : real * real -> bool

val >= : real * real -> bool

val == : real * real -> bool

val != : real * real -> bool

val ?= : real * real -> bool

Binary relational operators. These are infix operators.

val unordered : real * real -> bool

Returns true if one is *nan*.

val isFinite : real -> bool

val isNan : real -> bool

val isNormal : real -> bool

Tests for real values.

val class : real -> IEEEReal.float_class

Returns the IEEE class to which the real belongs.

val fmt : StringCvt.realfmt -> real -> string

val toString : real -> string

val fromString : string -> real option

val scan : (char,'a) StringCvt.reader -> (real,'a) StringCvt.reader

Various real to string or stream conversion functions. See int or bool for details on these functions.

val toManExp : real -> {exp:int, man:real}

val fromManExp : {exp:int, man:real} -> real

val split : real -> {frac:real, whole:real}

val realMod : real -> real

val rem : real * real -> real

Mantissa, exponent and fractional part functions.

val checkFloat : real -> real

Determines if it is a proper real number (not *nan* or *inf*). If it is proper, it returns the argument, otherwise an exception is raised.

val floor : real -> int

val ceil : real -> int

val trunc : real -> int

val round : real -> int

val realFloor : real -> real

val realCeil : real -> real

val realTrunc : real -> real

val realRound : real -> real

Various truncation and rounding functions.

val toInt : IEEEReal.rounding_mode -> real -> int

val toLargeInt : IEEEReal.rounding_mode -> real -> IntInf.int

val fromInt : int -> real

val fromLargeInt : IntInf.int -> real

val toLarge : real -> Real64.real

val fromLarge : IEEEReal.rounding_mode -> Real64.real -> real

val toDecimal : real -> IEEEReal.decimal_approx

val fromDecimal : IEEEReal.decimal_approx -> real

Numeric conversion functions.

10.4 The Char Structure

The following functions are part of the Char structure for the *char* type. The *char* type is separate from the *string* type, covered in the next section. More information can be found at http://www.standardml.org/Basis/char.html.

type char

The character type.

val chr : int -> char

val ord : char -> int

Conversion from and to ASCII values.

val minChar : char

val maxChar : char

val maxOrd : int

Various constants.

val pred : char -> char

val succ : char -> char

Moves through ASCII values.

val < : char * char -> bool

val <= : char * char -> bool

val > : char * char -> bool

val >= : char * char -> bool

Infix relational operators.

val compare : char * char -> order

See other compare functions for a description of the order type.

val scan : (char,’a) StringCvt.reader -> (char,’a) StringCvt.reader

val fromString : String.string -> char option

val toString : char -> String.string

val fromCString : String.string -> char option

val toCString : char -> String.string

Various conversion functions to and from strings.

val contains : string -> char -> bool

val notContains : string -> char -> bool

String search functions.

val isLower : char -> bool

val isUpper : char -> bool

val isDigit : char -> bool

val isAlpha : char -> bool

val isHexDigit : char -> bool

val isAlphaNum : char -> bool

val isPrint : char -> bool

val isSpace : char -> bool

val isPunct : char -> bool

val isGraph : char -> bool

val isCntrl : char -> bool

val isAscii : char -> bool

Character test functions.

val toUpper : char -> char

val toLower : char -> char

Upper and lowercase conversion functions.

10.5 The String Structure

This is the String structure providing functions that operate on strings. Strings are not the same as characters. A string can be exploded into a list of characters, but strings are separate objects from character values. More information can be found at http://www.standardml.org/Basis/string.html.

type string

Character sequences fall under the *string* type in Standard ML. However, strings are NOT lists of characters. There are functions given here to explode and implode a string to and from a list of characters.

val maxSize : int

Maximum string size.

val size : string -> int

Current size of a string.

val sub : string * int -> char

String subscript operator.

val str : char -> string

Convert char to string.

val extract : string * int * int option -> string

val substring : string * int * int -> string

A couple of substring operations. Extract's third argument is either *SOME x* where *x* is the ending lcoation+1 for the substring, or *NONE* to have extract extend to the the end of the string.

val ^ : string * string -> string

Binary string concatenation.

val concat : string list -> string

N-ary string concatenation.

val concatWith : string -> string list -> string

A variation on the other two concatenation operations.

val implode : char list -> string

val explode : string -> char list

Conversion to/from a list of characters to a string. These are useful when writing recursive string functions.

val map : (char -> char) -> string -> string

This is a higher order function that applies a character to character function to each character of a string and returns the string of collected results.

val translate : (char -> string) -> string -> string

Same as map above, but applies a character to string function to each character returning the string of collected strings.

val tokens : (char -> bool) -> string -> string list

val fields : (char -> bool) -> string -> string list

These two functions return tokens from a string. The char to bool function defines the delimiters of tokens. In other words the first argument is a function that returns true when white space is encountered. The *tokens* function always returns a non-empty token, the *fields* function may return empty tokens.

val isPrefix : string -> string -> bool

val isSubstring : string -> string -> bool

val isSuffix : string -> string -> bool

These are substring dectecting functions.

val compare : string * string -> order

Returns one of *GREATER*, *LESS*, or *EQUAL* depending on the two values being compared.

val collate : (char * char -> order) -> string * string -> order

Compares two strings lexicographically according to the provided character ordering.

val < : string * string -> bool

val <= : string * string -> bool

val > : string * string -> bool

val >= : string * string -> bool

Four infix, normal lexicographical comparisons.

val toString : string -> String.string

Replaces non-printing characters with SML escape character sequences.

val scan : (char,'a) StringCvt.reader -> (string,'a) StringCvt.reader

val fromString : String.string -> string option

val toCString : string -> String.string

val fromCString : String.string -> string option

Various string conversion functions and stream reading functions.

10.6 The List Structure

This is the List structure for the *list* polymorphic datatype in SML. More information can be found at http://www.standardml.org/Basis/list.html.

datatype ‘a list = :: of ‘a * ‘a list | nil

A list is formed from an element and a list. It is a recursive data structure with O(n) access to any element of the list. This should not be confused with an array that provides O(1) element access. The :: is called *cons* and stands for list construction or constructor. It forms a list from an element, *e*, and a list, *lst* as in *e::lst*. The *nil* keyword is used to represent an empty list. Writing *[]* is equivalent to *nil* in Standard ML. Lists in Standard ML must be homogenous, containing all the same type of elements.

exception Empty

Raised as necessary by various functions should an empty list be used as an argument. Not raised unless necessary.

val null : ‘a list -> bool

Returns true if the given list is empty.

val hd : ‘a list -> ‘a

val tl : ‘a list -> ‘a list

hd e::lst returns *e* while *tl e::lst* returns lst. *hd* is short for head of the list and *tl* is short for tail of the list.

val last : ‘a list -> ‘a

Returns the last element of the given list. Raise *Empty* if given an empty list.

val getItem : ‘a list -> (‘a * ‘a list) option

Returns SOME of the head and tail of a list or *NONE* if the list is empty. Calling *getItem (e::lst)* returns *SOME (e,lst)*.

val nth : ‘a list * int -> ‘a

Returns the nth item of the list (zero based) and raise *Subscript* if the list is too short.

val take : ‘a list * int -> ‘a list

Returns the first *i* elements of a list given a list and *i*. Raises *Subscript* if the list is too short.

val drop : ‘a list * int -> ‘a list

Returns the rest of a list after the first *i* elements. Raises *Subscript* if the list is too short.

val length : ‘a list -> int

Returns the length of a list.

val rev : ‘a list -> ‘a list

Returns the reverse of a list.

val @ : ‘a list * ‘a list -> ‘a list

This is list concatenation, not to be confused with :: which is list construction. This is an infix operator. So *[1, 2, 3]@[4, 5, 6]* is legal and so is *1* ::*[2, 3, 4, 5, 6]* which both yield the same result.

val concat : ‘a list list -> ‘a list

This takes a list of lists of all the same element and concatenates each of the lists together returning one big list of all the elements.

val revAppend : ‘a list * ‘a list -> ‘a list

Reverses the first list and appends it to the second.

val app : (‘a -> unit) -> ‘a list -> unit

This function applies the first argument, a function with a side-effect, to each element of a list. The *unit* type is another name for the empty tuple (i.e. *()*) which is the return type of many functions that have side-effects.

val map : (‘a -> ‘b) -> ‘a list -> ‘b list

The map function applies a function to each element of a list, building a new list of all the results.

val mapPartial : (‘a -> ‘b option) -> ‘a list -> ‘b list

This is like *map* except that if *NONE* is returned by the function, it is omitted from the resulting list. Only values of *SOME val* are included in the final result.

val find : (‘a -> bool) -> ‘a list -> ‘a option

Given a predicate function and a list, the *find* function returns either *SOME val* for the found value or *NONE* indicating the predicate did not return true for any element of the list.

val filter : (‘a -> bool) -> ‘a list -> ‘a list

This function returns a new list of all elements of the list that satisfy the provided predicate function.

val partition : (‘a -> bool) -> ‘a list -> ‘a list * ‘a list

This function returns a tuple where the first list consists of all elements that satisfy the predicate function and the second is comprised of the elements that did not satisfy the predicate.

val foldr : (‘a * ‘b -> ‘b) -> ‘b -> ‘a list -> ‘b

This function applies a provided function to each element and an initial value, folding all the results into one finals result. This function is called *foldr* because it is right-associative. Here is an example of calling *foldr*.

```
- foldr (op -) 0 [1,2,3,4];
val it = ~2 : int
```

The use of *op* - in the example transforms the infix - operator to a prefix function. The example computed (1 – (2 – (3 – (4 – 0)))). If the list is empty then the initial value, the second argument, is returned.

val foldl : (‘a * ‘b -> ‘b) -> ‘b -> ‘a list -> ‘b

This function is the left-associative analog of *foldr* meaning that the initial value is applied along with the first element of the list and that result applied along with the second element of the list and so on. For example,

```
- foldl (op -) 0 [1,2,3,4];
val it = 2 : int
```

The example computed (4 – (3 – (2 – (1 – 0)))). If the list is empty, then the initial value, the second argument, is returned.

val exists : (‘a -> bool) -> ‘a list -> bool

Given a predicate function, *exists* returns true if the predicate function evaluates to true for at least one element of the list.

val all : (‘a -> bool) -> ‘a list -> bool

Given a predicate function, *all* returns true if the predicate function evaluates to true for all elements of the list.

val tabulate : int * (int -> ‘a) -> ‘a list

Builds a list of *n* elements. The *n* is the first argument to *tabulate*. The each element is generated by passing one of 0 to *n-1* to the second argument, a function. Raises *Size* if there are less than *n* elements in the list.

```
- List.tabulate(5,fn x => x + 1);
val it = [1,2,3,4,5] : int list
```

val collate : (‘a * ‘a -> order) -> ‘a list * ‘a list -> order

This performs a lexicographical comparison of two lists according to the provided ordering function for each element of the lists. Returns one of *LESS*, *GREATER*, or *EQUAL*.

10.7 The Array Structure

Arrays are mutable sequences that provide O(1) lookup and assignment complexities. Lists are immutable and provide O(n) lookup time. Lists are immutable so item assignment is not possible in a list. Since arrays are mutable, many of the functions on arrays return *unit* the type of *()* which is used as the return type of mutating functions in Standard ML. More information can be found at http://www.standardml.org/Basis/array.html.

type ‘a array

Arrays must be homogeneous in Standard ML, comprised of all the same type of elements.

val maxLen : int

Maximum size of an array.

val array : int * ‘a -> ‘a array

Build an array with size *n*, the first argument, and all elements initialized to the value of *a*, the second argument.

val fromList : ‘a list -> ‘a array

Build an array from a list.

val tabulate : int * (int -> ‘a) -> ‘a array

See List.tabulate.

val length : ‘a array -> int

The length of an array.

val sub : ‘a array * int -> ‘a

The O(1) element retrieval operation not provided by lists in Standard ML.

val update : ‘a array * int * ‘a -> unit

The array element assignment operation, a O(1) mutating operation.

val vector : ‘a array -> ‘a vector

Builds a vector from an array.

val copy : {di:int, dst:’a array, src:’a array} -> unit

val copyVec : {di:int, dst:’a array, src:’a vector} -> unit

Copy utility functions.

val appi : (int * ‘a -> unit) -> ‘a array -> unit

val app : (‘a -> unit) -> ‘a array -> unit

Applies a function to an array. The first supplies the function with *i* provided as the first argument where *i* is the index of the element. The second applies the function to each element of the vector without knowledge of its location. The function applied would have some side-effect.

val modifyi : (int * ‘a -> ‘a) -> ‘a array -> unit

val modify : (‘a -> ‘a) -> ‘a array -> unit

Applies a function to an array. The first supplies the function with *i* provided as the first argument where *i* is the index of the element. The second applies the function to each element of the vector without knowledge of its location. The function applied results in a value that replaces the value in the array at the same location.

val foldli : (int * ‘a * ‘b -> ‘b) -> ‘b -> ‘a array -> ‘b

val foldri : (int * ‘a * ‘b -> ‘b) -> ‘b -> ‘a array -> ‘b

val foldl : (‘a * ‘b -> ‘b) -> ‘b -> ‘a array -> ‘b

val foldr : (‘a * ‘b -> ‘b) -> ‘b -> ‘a array -> ‘b

The fold equivalents (see List.fold functions) for arrays. The foldli and foldri functions provide the index of the value in addition to the value at each element of the array.

val findi : (int * ‘a -> bool) -> ‘a array -> (int * ‘a) option

val find : (‘a -> bool) -> ‘a array -> ‘a option

val exists : (‘a -> bool) -> ‘a array -> bool

val all : (‘a -> bool) -> ‘a array -> bool

val collate : (‘a * ‘a -> order) -> ‘a array * ‘a array -> order

All similar to List functions. See the List equivalents for explanations.

10.8 The TextIO Structure

This is a subset of the entire TextIO structure. Detailed descriptions of all functions can be found on the Basis Library website at http://www.standardml.org/Basis/text-io.html.

type instream

type outstream

Standard ML supports stream operations for both input and output streams.

val input : instream -> vector

val input1 : instream -> elem option

val inputN : instream * int -> vector

val inputAll : instream -> vector

These are blocking input functions. The *input* returns an empty vector if the input stream is closed, otherwise returning one or more items in the stream. The *input1* reads just one element from the stream and returns *NONE* if the input stream is closed. The *inputN* returns at most *n* items. The *inputAll* returns everything up to the end of stream.

val canInput : instream * int -> int option

val lookahead : instream -> elem option

These two functions look at the state of the stream. They are useful in making input decisions.

val closeIn : instream -> unit

val endOfStream : instream -> bool

The *closeIn* function closes a stream and *endOfStream* closes the given stream.

val output : outstream * vector -> unit

val output1 : outstream * elem -> unit

Writes all elements of a vector and one element, respectively, to a stream.

val flushOut : outstream -> unit

val closeOut : outstream -> unit

Before input is read, it may be necessary to flush output if a prompt is printed for instance. Otherwise, the prompt may not appear on the screen. The *closeOut* function closes an output stream.

val inputLine : instream -> string option

Reads an input *line* and returns either *SOME line* or *NONE*.

val outputSubstr : outstream * substring -> unit

Writes a substring.

val openIn : string -> instream

Opens an input stream for reading. The argument is a filename.

val openString : string -> instream

Opens a string stream for reading.

val openOut : string -> outstream

Opens an output stream for writing. The argument is a filename.

val openAppend : string -> outstream

Opens an output stream for writing. The argument is a filename. If the file exists, the data written will be appended to the end of the file.

val stdIn : instream

val stdOut : outstream

val stdErr : outstream

These are the names of the default input, output, and error streams supplied with every program. They are precreated objects.

val print : string -> unit

Prints to standard output the given string.

val scanStream : ((elem,StreamIO.instream) StringCvt.reader -> instream -> (‘a,StreamIO.instream) StringCvt.reader) -> instream -> ‘a option

Uses a stream and converts it to an imperative stream where conversions can be done while reading input. See the Basis Library for a more complete description of how this works.

Bibliography

1. J. Adams, National science foundation press release 07-029. National Science Foundation Press Releases (2007)
2. A.V. Aho, M. Lam, R. Sethi, J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd edn. (Addison-Wesley Longman Publishing Co., Boston, 2006)
3. J. Backus, [Photograph]. Photograph provided courtesy of IBM and used with permission (2008)
4. E. Biagioni, A structured TCP in standard ML, in *SIGCOMM '94: Proceedings of the Conference on Communications Architectures, Protocols and Applications, New York, NY, USA* (ACM Press, 1994), pp. 36–45
5. W. Clocksin, C. Mellish, *Programming in Prolog* (Springer, Berlin, 2003)
6. A. Colmerauer, [Photograph]. Photograph provided courtesy of Alain Colmerauer and used with his permission (2008)
7. A. Colmerauer, P. Roussel, The birth of prolog, in *HOPL-II: The second ACM SIGPLAN Conference on History of Programming Languages, New York, NY, USA* (ACM, 1993), pp. 37–52
8. R. Girvan, Partial differential equations, *Scientific-Computing.com* (2006), http://www.scientific-computing.com/review4.html
9. M. Gordon, From LCF to HOL: a short history (2000), pp. 169–185
10. R. Harper, P. Lee, Advanced languages for systems software: the Fox project in 1994. Technical report CMU-CS-94-104, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1994. (Also published as Fox Memorandum CMU-CS-FOX-94-01)
11. R Kowalski, An interview with Robert Kowalski, 2008. Details of events provided by Robert Kowalski through an exchange of email from 12 Feb 2008 to 14 Feb 2008 (2008)
12. R. Kowalski, [Photograph]. Photograph provided courtesy of Robert Kowalski and used with his permission (2008)
13. P. Linz, *An Introduction to Formal Languages and Automata* (Jones and Bartlett, Sudbury, 2006)
14. J. McCarthy, [Photograph]. Photograph provided courtesy of John McCarthy and used with his permission (2008)
15. R. Milner, [Photograph]. Photograph provided courtesy of Robin Milner and used with his permission (2008)

16. R. Milner, M. Tofte, R. Harper, D. Macqueen, *The Definition of Standard ML - Revised* (The MIT Press, Cambridge, 1997)
17. B. Stroustrup, A history of C++: 1979–1991 (1996), pp. 699–769
18. B. Stroustrup, [Photograph]. Photograph provided courtesy of Bjarne Stroustrup and used with his permission (2006)
19. B. Stroustrup, *The C++ Programming Language*, 4th edn. (Addison-Wesley Professional, Boston, 2013)
20. A. Stubhaug, *The Mathematician Sophus Lie* (Springer, New York, 2002)
21. A. Stubhaug, *The Mathematician Sophus Lie [Photograph]* (Springer, Photograph reprinted with permission of Springer and Arild Stubhaug, New York, 2002)
22. The Web Education Community Group: A Member of the W3C, A short history of javascript (2012), https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript. Accessed 19 Mar 2017
23. A. Turing, On computable numbers, with an application to the entscheidungsproblem. Proc. Lond. Math. Soc. **42**, 230–265 (1936)
24. A.M. Turing, *A. M. Turing's ACE Report of 1946 and Other Papers* (MIT Press, Cambridge, 1986)
25. G. van Rossum, [Photograph]. Photograph provided courtesy of Guido van Rossum and used with permission (2013)
26. J. von Neumann, First draft of a report on the EDVAC, http://en.wikipedia.org/wiki/Von_Neumann_architecture (1945)
27. Wikipedia, Charles babbage (2006). Accessed 14 Jan 2006
28. Wikipedia, John vincent atanasoff (2006). Accessed 14 Jan 2006
29. Wikipedia, Prolog (2008). Accessed 13 Feb 2008