Programming Language Foundations

and Models of Computation

Kent D. Lee

December 21, 2006

Preface

Computer Science has matured into a very large discipline. It is universally accepted that you will not be taught everything you will need in your career. The goal of a Computer Science education is to prepare you for a life of learning. The creativity encouraged by a lifetime of learning makes Computer Science one of the most exciting fields today according to Money Magazine's Best Jobs in America in 2006.

The words *Computer Science* don't really reflect what most computer programmers do. The field really should be named something like Computer Program Engineering. It is more of an engineering field than a science. That's why programmers are often called Software Engineers. There *is* science in Computer Science generally relating to the gathering of empirical evidence pertaining to performance of algorithms and hardware. There are also theoretical aspects of the discipline, some of which this text will explore. However, the bulk of what computer scientists do is related to the creative process of building programs. As computer scientists we know some things about how to write good programs, but it is still a very creative process.

Because computer programming is such a creative process there are currently no formal ways of automatically generating programs, nor will there be for the foreseeable future. It is imperative that we be able to *predict* what our programs will do. To predict what a program will do, you must understand how the program works. The programs of a language execute according to a model of computation. The model may be implemented in many different ways depending on the targeted hardware architecture. However, it is not necessary to understand all the different architectures out there to understand the model of computation used by a language.

For several years in the late 1980's and 1990's I worked at IBM on the operating system for the AS/400 (what is now called the iSeries). My understanding of compilers and language implementation helped me make better decisions about how to write code and several times helped me find problems in code I was testing. An understanding of the models of computation used by the languages I programmed in aided me in predicting what my programs would do. After completing a course in Programming Language Foundations, you should understand some of the basics of language implementation. This book is not intended to be a complete text on compiler or interpreter implementation, but there are aspects of language implementation that are included in it. My belief is that we are better users of tools when we understand how the tools we use work.

I hope you enjoy learning from this text and the course you are about to take. The text is meant to be used interactively. You should read a section and as you read it do the practice exercises listed in the gray boxes. Each of the exercises are meant to give you a goal in reading a section of the text. If you can do the practice exercise you understand the material you just read.

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 like Python or Java. 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 like polymorphism, recursion, and logical implication are relatively new to the student reading this book. The functional and logic paradigms, while not the mainstream, have their place and have been successfully used in interesting applications.

The Object-Oriented language presented in this book is C++ or Ruby. Teachers may choose between the chapter on Ruby or the chapter on C++, but probably won't assign both unless you wish to compare and contrast a compiled language to an interpreted language. The same project is presented in both chapters with the C++ chapter requiring a little more explanation in terms of the compiler and organization of the code. Either language is interesting to choose from and the chapters do cross-reference each other to compare and contrast the two styles of programming.

C++ has many nuances that are worthy of several chapters in a programming languages book. Notably the pass by value and pass by reference mechanisms in C++ create considerable complexity in the language. Polymorphism is another interesting aspect of Object-Oriented languages that is studied in this text.

Ruby is relatively new to the programming language arena, but is widely accepted and is a large language when compared to Python and Java. In addition, its object-centered approach is very similar to Smalltalk. Ruby is also interesting due to the recent development of "Ruby on Rails" as a code generation tool.

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 correct. This has some implications in writing programs. While ML's 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 like Lisp, Scheme, Ruby, or Python.

The logic language is Prolog. While Prolog is an Artificial Intelligence language, it originated as a meta-language for expressing other languages. The text concentrates on using Prolog to implement other languages. Students learn about logical implication and how a problem they are familiar with can be re-expressed in a different paradigm.

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 teach interactively. 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 practice with 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 if 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. Look for the light gray *Practice* boxes.

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

Finally, it is expected that while teaching a class using this text, lab time will be liberally sprinkled throughout the course as the instructor sees fit. Reinforcing lectures with experience makes students appreciate the difficulty of learning new paradigms while making them stronger programmers, too.

Supplementary materials including answers to exercises, review questions, and programming assignments are available to instructors upon request.

Contents

Small Action Semantic Description

Sorts

(1) function = an abstraction of an action [producing the empty-map]

(2) value = integer | truth-value

(3) bindable = value | [value]cell | function

Semantics

- run _ :: Prog → action

(1) run ⟦ E:Expr ⟧ =
    | | bind "output" to the native abstraction of an action
    | | [using a given value] [giving ()]
    | before
    | | bind "input" to the native abstraction of an action
    | | [giving an integer] [using the given ()]
    hence
    | evaluate E .

- evaluateSeq _ :: ExprSeq → action

(2) evaluateSeq ⟦ E:Expr ";" E_s:ExprSeq ⟧ =
    | evaluate E
    then
    | evaluateSeq E_s

(3) evaluateSeq ⟦ E:Expr ⟧ =
    evaluate E

- evaluate _ :: Expr → action

(4) evaluate ⟦ L_v:Expr ":=" E:Expr ⟧ =
    | | evaluate L_v
    | and then
    | | evaluate E
    then
    | store the given value#2 in the given cell#1

(5) evaluate ⟦ "while" B:Expr "do" E:Expr ⟧ =
    unfolding
    | | evaluate B
    | then
    | | | evaluate E
    | | | and then
    | | | | unfold
    | | else
    | | | complete

(6) evaluate ⟦ "let" D_s:DecSeq "in" E_s:ExprSeq "end" ⟧ =
    | furthermore elaborateDecSeq D_s
    hence
    | evaluateSeq E_s

(7) evaluate ⟦ "if" S:SExpr "then" E_1:Expr "else" E_2:Expr ⟧ =
    | evaluateSExpr S
    then
    | | evaluate E_1
    | else
    | | evaluate E_2

(8) evaluate ⟦ S:SExpr ⟧ =
    evaluateSExpr S

- elaborateDecSeq _ :: DecSeq → action

(9) elaborateDecSeq ⟦ D:Decl D_s:DecSeq ⟧ =
    elaborate D before elaborateDecSeq D_s

(10) elaborateDecSeq ⟦ D:Decl ⟧ =
    elaborate D

- elaborate _ :: Decl → action

(11) elaborate ⟦ "fun" F_s:FunSeq ⟧ =
    recursively elaborateFunSeq F_s

(12) elaborate ⟦ "val" I:Identifier "=" E:Expr ⟧ =
    | evaluate E
    then
    | bind I to the given (value | [value]cell)

- elaborateFunSeq _ :: FunSeq → action

(13) elaborateFunSeq ⟦ F:FunDecl "and" F_s:FunSeq ⟧ =
    elaborateFun F and then elaborateFunSeq F_s

(14) elaborateFunSeq ⟦ F:FunDecl ⟧ =
    elaborateFun F

- elaborateFun _ :: FunDecl → action

(15) elaborateFun ⟦ Id:Identifier "(" ")" "=" E:Expr ⟧ =
    bind I to the closure of the abstraction of
    | evaluate E

(16) elaborateFun ⟦ I:Identifier "(" P_s:FormalParmSeq ")" "=" E:Expr ⟧ =
    bind I to the closure of the abstraction of
    | | furthermore
    | | | bindParameters P_s
    | thence
    | | evaluate E

- bindParameters _ :: FormalParmSeq → action

(17) bindParameters ⟦ I:Identifier ⟧ =
    bind I to the given value

(18) bindParameters ⟦ I:Identifier ":" "int" ⟧ =
    bind I to the given integer

(19) bindParameters ⟦ I:Identifier ":" "bool" ⟧ =
    bind I to the given truth-value

(20) bindParameters ⟦ P:FormalParm "," P_s:FormalParmSeq ⟧ =
    | | bindParameter P
    | and then
    | | give the rest of the given data
    then
    | bindParameters P_s

- bindParameter _ :: FormalParm → action

(21) bindParameter ⟦ I:Identifier ⟧ =
    bind I to the given value#1

(22) bindParameter ⟦ I:Identifier ":" "int" ⟧ =
    bind I to the given integer#1

(23) bindParameter ⟦ I:Identifier ":" "bool" ⟧ =
    bind I to the given truth-value#1

- evaluateSExpr _ :: SExpr → action

(24) evaluateSExpr ⟦ "ref" E:SExpr ⟧ =
    | | allocate a cell
    | and then
    | | evaluateSExpr E
    then
    | | store the given value#2 in the given cell#1
    | and then
    | | give the given cell#1

(25) evaluateSExpr ⟦ E:SExpr "orelse" T:Term ⟧ =
    | evaluateSExpr E
    then
    | | give true
    | else
    | | evaluateTerm T

(26) evaluateSExpr ⟦ E:SExpr "+" T:Term ⟧ =
    | | evaluateSExpr E
    | and then
    | | evaluateTerm T
    then
    | give the sum of the given data

(27) evaluateSExpr ⟦ E:SExpr "-" T:Term ⟧ =
    | | evaluateSExpr E
    | and then
    | | evaluateTerm T
    then
    | give the difference of (the given integer#1, the given integer#2)

(28) evaluateSExpr ⟦ T:Term ⟧ =
    evaluateTerm T

- evaluateTerm _ :: Term → action

(29) evaluateTerm ⟦ T:Term "andalso" N:Neg ⟧ =
    | evaluateTerm T
    then
    | | evaluateNeg N
    | else
    | | give false

(30) evaluateTerm ⟦ T:Term "*" N:Neg ⟧ =
    | | evaluateTerm T
    | and then
    | | evaluateNeg N
    then
    | give the product of the given data

(31) evaluateTerm ⟦ T:Term "/" N:Neg ⟧ =
    | | evaluateTerm T
    | and then
    | | evaluateNeg N
    then
    | give the (quotient of (the given integer#1, the given integer#2))
    | [yielding an integer]

(32) evaluateTerm ⟦ T:Term "mod" N:Neg ⟧ =
| | evaluateTerm T
| and then
| | evaluateNeg N
then
| give remainder of (the given integer#1, the given integer#2)

(33) evaluateTerm ⟦ N:Neg ⟧ =
evaluateNeg N

- evaluateNeg _ :: Neg → action

(34) evaluateNeg ⟦ "not" N:Neg ⟧ =
| evaluateNeg N
then
| give not of the given truth-value

(35) evaluateNeg ⟦ "-" N:Neg ⟧ =
| evaluateNeg N
then
| give (difference of (0, the given integer)) [yielding an integer]

(36) evaluateNeg ⟦ C:Comparison ⟧ =
evaluateComparison C

- evaluateComparison _ :: Comparison → action

(37) evaluateComparison ⟦ F_1:Factor "<" F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give the given integer#1 is less than the given integer#2

(38) evaluateComparison ⟦ F_1:Factor ">" F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give the given integer#1 is greater than the given integer#2

(39) evaluateComparison ⟦ F_1:Factor "=" F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give the given (value | [value]cell) #1 is the given (value | [value]cell)#2

(40) evaluateComparison ⟦ F_1:Factor "<>" F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give not (the given (value | [value]cell)#1 is
| the given (value | [value]cell)#2)

(41) evaluateComparison ⟦ F_1:Factor ">=" F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give not (the given integer#1 is less than the given integer#2)

(42) evaluateComparison ⟦ F_1:Factor "<=" F_2:Factor ⟧ =
| | evaluateFactor F_1
| and then
| | evaluateFactor F_2
then
| give not (the given integer#1 is greater than the given integer#2)

(43) evaluateComparison ⟦ F:Factor ⟧ =
evaluateFactor F

- evaluateFactor _ :: Factor → action

(44) evaluateFactor ⟦ I:Integer ⟧ =
give I

(45) evaluateFactor ⟦ "true" ⟧ =
give true

(46) evaluateFactor ⟦ "false" ⟧ =
give false

(47) evaluateFactor ⟦ "!" F:Factor ⟧ =
| evaluateFactor F
then
| give the value stored in the given cell

(48) evaluateFactor ⟦ I:Identifier ⟧ =
| give the value bound to I
or
| give the [value]cell bound to I

(49) evaluateFactor ⟦ I:Identifier "(" A:ArgSeq ")" ⟧ =
| evaluateArgs A
then
| enact the application of the function bound to I to the given data

(50) evaluateFactor ⟦ I:Identifier "(" ")" ⟧ =
    | complete
    then
    | enact the application of the function bound to I to the given data

(51) evaluateFactor ⟦ "(" S:ExprSeq ")" ⟧ =
    evaluateSeq S

- evaluateArgs _ :: ArgSeq → action

(52) evaluateArgs ⟦ E:Expr ⟧ =
    evaluate E

(53) evaluateArgs ⟦ E:Expr "," A:ArgSeq ⟧ =
    | evaluate E
    and then
    | evaluateArgs A