9

TECHNICAL FOUNDATIONS IN PROGRAMMING LANGUAGES

Reference Materials

Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation.* 4th Ed. Upper Saddle River, NJ: Prentice Hall. 1995. Chapter 4.
C. Hankin. *Lambda Calculi: A Guide for Computer Scientists.* London: King's College Publications. 2004.
Kenneth C. Louden. *Programming Languages: Principles and Practice.* 2d Ed. Belmont, CA: Thompson, Brooks/Cole. 2003. Chapters 9 and 13.

Web Sites

Wikipedia: search "lambda calculus"

9.1 INTRODUCTION

Early models of programming languages were very ad hoc simply because there was so little accumulated experience and research. In the early 1960s, the predominant theoretical model of computation was the Turing machine, the model received from logic and mathematics. The Turing machine made sense as a model because computers of the day "looked" similar. Interestingly enough, there were many other programming models available.

CASE 49. EARLY HISTORY OF PROGRAMMING LANGUAGES

There are many interesting Web sites on the history of programming languages, but the more interesting study is that of computing from the time of Charles Babbage's original analytical engine (1819) through 1970 when there was an explosion of programming paradigms.

Write a paper that includes a timeline that marks events in the development of programming languages/machines/systems. For each event:

1. What was the major innovation? What was the problem that demanded this innovation?
2. Where did the innovation come from and from whom?
3. How does the innovation enhance computing?

Merge your timeline with others in the class to form the class's perception of the history of innovations.

9.2 BINDING TIMES: WHEN DO WE KNOW?

We probably don't think about it, but history plays an overarching role in the evolution of language. We pick up the current Java release manual and don't think much about how we got it. That's unfortunate, because we lose perception of the workings.

The term *binding time* is used to denote when, along the development timeline, a word in the lexicon is bound to (associated with) its definition. As an example, consider the program snippet `sin(x)` and ask "When do `sin` and `x` receive their values?" The use of the abbreviation `sin` for the sine of an angle is inherited from mathematical practice many centuries ago. The actual code for computing the sine is relatively recent but is based on the Taylor series, named for Brook Taylor (1685–1731). The code itself (which is a definition) was most likely written in the language itself, but it is a system routine and the regular programmers treat it as available. What about `x`? `x` is invented by the programmer during program design and long after the compiler is written. Assuming it is not a constant, it received its definition (value) during the running of the program. Therefore, stretched out on a timeline, the binding times for such a simple snippet stretch to antiquity through the program's running.

The reason this discussion takes places is that it is the prelude to the discussion of *scope*.

9.3 SCOPE

Scope is the term that denotes the rules under which a symbol can be replaced by a definition. For students with only C-based language experience (C, C++, Java), scope rules are relatively straightforward. This is because of the relative inability to introduce new procedures. The functional languages tend to have much richer definitional capabilities.

The basic issue in scope is the implementation of a *visibility function*. An example of how this issue affects us is the following simple C program:

```
int mystery(float x){
double xp = x;
  {
  int xp = 17;
  return xp;
  }
}
```

What is returned? Why?

The metaphor of *seeing* is very real here: the `return` statement sees the *nearest* definition of `xp` and hence returns the 17. How can the compiler deal with this?

The issue is *environment-producing statements*, such as the braces in C. An innovation in the 1970s with structured programming was the realization that all programs can be written with properly nested environments. An environment is a set of all the definitions that have occurred in the scope of the environment producing statements.

In `mystery`, there are *three* environments. The program header/name is in the file environment. The `x` and the first definition of `xp` are in the second and the last definition of `xp` is in the third environment. The `return` statement is in the third environment, so it uses the definition of `xp` in that environment. What about this problem?

```
int mystery(float x){
double xp = x;
  {
      statements using and changing xp;
      return xp;
  }
}
```

In this case, there are still three environments, but no definition of `xp` occurs in the third. Therefore, the `return` statement uses the definition of `xp` from the second environment. This is why the proper nesting of definitional environment is so crucial.

CASE 50. SOL DEFINITIONAL ENVIRONMENTS

Using the SOL specification, identify all the environment-producing statements of the language. Develop several examples of how the environments can be nested; these examples should serve as test cases in implementation.

9.4 THE λ-CALCULUS MODEL

All programming languages have the ability to define independent procedures. The complication comes when a procedure wants to use another procedure—or even itself—to compute values. As straightforward as it seems today, the concept of how to think about variables, arguments, and invocation was not always spelled out. The first and most influential was the idea of the λ-calculus, first developed by Alonzo Church in the 1920s, and later refined in *The Calculi of Lambda-Conversion* in 1941. Because of Church's interests, the calculus was developed strictly over the integers; naïve extensions to other types is straightforward, although a complete theoretical explanation had to wait until Barendregt in 1984.

More importantly for our purposes, the λ-calculus was the basis of the programming system Lisp, primarily motivated by John McCarthy, which gained prominence in the 1960s. McCarthy wrote "A Basis for a Mathematical Theory of Computation" in 1963, which had the major impact of moving the basis of programming from Turing machines to recursive functions as founded on λ-calculus. The Lisp 1.5 manual in 1965 presented a programming system we would recognize: Lisp is coded in Cambridge Polish. What follows is a simple introduction to the λ-calculus suitable for our purposes.

9.4.1 Syntax

A suitable syntax for λ-expressions would be Old Faithful with just one or two additions. For reference,

$$
\begin{aligned}
E &\rightarrow T + E\\
E &\rightarrow T - E\\
E &\rightarrow T\\
T &\rightarrow F * T\\
T &\rightarrow F/T\\
T &\rightarrow F\%T\\
T &\rightarrow F\\
F &\rightarrow \text{any integer}\\
F &\rightarrow (E)
\end{aligned}
$$

9.4.1.1 Variables

The first task question is to define the lexical rule for variable. The obvious answer is that we use elements composed of letters. In the original formulation, variables were single, lower-case Latin letters.

Where should variables be recognized? A moment's thought indicates that variables are stand-ins for integers. Therefore, we need a new rule for F:

$$F \rightarrow \text{V : any lower-case Latin letter}$$

9.4.1.2 Introducing Variable Names

The numeral 17 is converted to the same number everywhere in a program, but, as we discussed above while dealing with scope, a single variable has meaning only in scope. How do we indicate scope? Old Faithful has no means; in fact, neither did arithmetic in general until Church came along.

Church introduced an *improper function* λ to introduce a new variable. For example, $\lambda\ x\ .\ x + 2$ introduces the variable x. Notice the period: the expression on the right of the period is the expression "guarded" by the λ.

There is no obvious place in the existing grammar to place the λ, but it is clear that the right-hand side could be any complex expression. Therefore, we add a whole new nonterminal and we'll call it L:

$$L \rightarrow \lambda\text{V : any variable.}E$$

An obvious problem is that F has a rule for (E); this should be changed to (L). In order to have parenthesized expressions on the right-hand side of the period, we need a renaming production for L:

$$L \rightarrow E$$

Finally, there is no way to define a function. Adopting a common format we use the `let` keyword:

$$\begin{aligned} letprod &\rightarrow \texttt{let}\text{V : any variable} = L \\ &\rightarrow E \end{aligned}$$

9.4.2 Semantics

Consider the following two statements in the extended calculus:

$$let f = \lambda\ x.x+2 f(3)$$

What is the value of $f(3)$? To answer this we must define the semantics of the λ-calculus. Once the λ is taken care of, the semantics are those of ordinary arithmetic.

The semantics we chose to demonstrate are those of ***substitution***, ***rewriting***, and ***unfolding***.

9.4.3 Substitution and Rewriting

The concept of substitution should be clear, at least in the naïve sense. Restrict the substitution operation to one variable only at a time. Substitution would be a three-place function that defines what it means for a value v to be substituted for a variable x in expression e. As sample semantics, look at $f(3)$: it should be clear that we want to take the 3 and substitute that value everywhere x occurs on the right-hand side of the λ-expression. In order to keep track of this substitution, we need to ***rewrite*** the expression. We propose that the semantics of the λ is this substitution/rewrite action. We can display that in a tabular form:

Old Expression	*New Expression*	*Reason*
$f(3)$	$(\lambda x.x+2)(3)$	Substitute definition of f
$(\lambda x.x+2)(3)$	$3+2$	Definition of λ
$3+2$	5	Definition of +

9.4.4 Recursion and the Unfolding Model

Consider a slightly different definition: factorials.

$$let\ ! = \lambda x.x * f(x-1)$$

Actually, this would be a nonterminating computation unless we have some method of checking for a zero argument. It should be obvious by now how to add an "if" statement to Old Faithful so that

$$let\ g = \lambda x.\ \ if\ \ (x = 0)1\ \ else\ \ x * g(x-1)\, f(3)$$

This is clearly much different from the first function we defined. It also adds a complication that has plagued theoreticians for 100 years: the g that is defined is not necessarily the g on the right side of the expression. Programmers recognize this same problem when they are required to use definitions in C to cause the loader to correctly match the definition. There have been several different solutions to this problem: a special form of the `let`, usually termed a `letrec`, is common. Another solution is to require that every `let` is actually `letrec`. Let's use this latter idea.

The semantics for recursion are effectively the same, substitution and rewriting, but with some wrinkles. The tabular presentation is wasteful of space, so we will resort to a simplification.

$$
\begin{aligned}
g(3) &\Rightarrow (\lambda x.\ \mathit{if}\ \ (x = 0)\ \ \mathit{then}\ 1\ \ \mathit{else}\ \ x * g(x - 2))(3) \\
&\Rightarrow \mathit{if}\ \ (3 = 0)\ \ \mathit{then}\ 1\ \ \mathit{else}\ \ 3 * g(3 - 1) \\
&= 3 * g(2) \\
&\Rightarrow 3 * (\lambda x.\ \mathit{if}\ \ (x = 0)\ \ \mathit{then}\ 1\ \ \mathit{else}\ \ x * g(x - 1))(2) \\
&\Rightarrow 3 * (\mathit{if}\ \ (2 = 0)\ \ \mathit{then}\ 1\ \ \mathit{else}\ \ 2 * g(2 - 1)) \\
&= 3 * (2 * g(1)) \\
&\Rightarrow 3 * (2 * (\lambda x.\ \mathit{if}\ \ (x = 0)\ \ \mathit{then}\ 1\ \ \mathit{else}\ \ x * g(x - 1))) \\
&\Rightarrow 3 * (2 * (\mathit{if}\ \ (1 = 0)\ \ \mathit{then}\ 1\ \ \mathit{else}\ \ 1 * g(1 - 1))) \\
&= 3 * (2 * (1 * g(0))) \\
&\Rightarrow 3 * (2 * (1 * (\lambda x.\ \mathit{if}\ \ (x = 0)\ \ \mathit{then}\ 1\ \ \mathit{else}\ \ x * g(x - 1)))) \\
&\Rightarrow 3 * (2 * (1 * (\mathit{if}\ \ (0 = 0)\ \ \mathit{then}\ 1\ \ \mathit{else}\ \ g(0 - 1)))) \\
&= 3 * (2 * (1 * 1)) \\
&= 3 * (2 * 1) \\
&= 3 * 2 \\
&= 6
\end{aligned}
$$

where $\Rightarrow$ indicates a substitution move and $=$ indicates a rewrite move.

We can now state what compilers actually do: they generate code that, when executed, carries out the subsitution, rewrite, and unfolding operations. The question before the compiler designer is how to do this efficiently.

9.4.5 Arguments, Prologues, and Epilogues

All this looks very easy when you can work out a small problem on paper, but in large systems, the bookkeeping is very difficult. Actually, the code for computing values is reasonably straightforward compared to all the bookkeeping: getting values to the called functions, remembering which function was the caller and where to return, how to store values so they can be retrieved easily, and how to reclaim space when it is no longer needed.

9.4.5.1 Argument Passing

Consider the following snippet:

```
void caller( ...){

int x;

...

a = called(3 + 3.7 - x);

...

}


float called ( double x ){


int q,z;

...

return (q + 3.0)/z;

}
```

This problem shows many of the issues in actually implementing a programming language with regard to functions, control, and argument passing.

1. The argument $3 + 3.7 - x$ must first be evaluated.
2. The result must be stored appropriately.
3. Control must be transferred to `called` with enough information so that `called` can return the value to the `caller`.
4. `called` must save information regarding the state of `caller`.
5. `called` then computes the requested information.

6. `called` then must restore `caller`'s state and put the return value where `caller` can find it.
7. `called` must return control to `caller`.
8. `caller` must store the return value into `a`.

There are many possible protocols that can be used for these steps. Generally, the tasks cover the two steps starting at 3, called the ***prologue***, and the two steps starting with 6, called the ***epilogue***.

CASE 51. ARGUMENT PASSING

The actual protocol for how arguments are passed is many and varied. Write a short paper on how these protocols have been handled in the past. Choose from the list the protocol you want for SOL.

9.5 IMPLEMENTATION ISSUES

This chapter has presented a number of concepts that must be defined in order to produce a compiler. Here is a partial list of tasks that must be finished before a design is completed.

1. Give an operational definition of the visibility function.
2. How are you going to allocate and free storage for arguments?
3. How are you going to allocate and free storage for local variables?
4. What are the scope-producing statements in SOL? When does the scope start and when does it end?
5. How will values be returned?
6. How will control be transferred from the calling procedure to the called procedure?