Programming Language Notes

March 4, 2009

Morgan McGuire[*]
Williams College

This is a series of lecture notes for CS334 addressing some of the theoretical topics from the course. I will extend them periodically throughout the semester. These supplement the lectures and required reading. Those contain other topics, especially implementation details, and expand on this condensed form.

1 Introduction

A programming language is a mathematical calculus, or **formal language**. Its goal is to express algorithms in a manner that is unambiguous to people and machines. Like any calculus, a language defines both syntax and semantics. Syntax is the grammar of the language; the notation. Semantics is the meaning of that notation. Since syntax can easily be translated, the semantics are more fundamental.

Church and Turing showed that the minimal semantics of the λ calculus and Turing machine are sufficient to emulate the semantics of any more complicated programming language or machine. However, reducing a particular language to the λ calculus may require holistic restructuring of programs in that language. We say that a particular language feature (e.g., continuations, macros, garbage collection) is **expressive** if it cannot be emulated without restructuring programs that use it.

1.1 Types and Features

Every language makes some programs easy to express and others difficult. When a language is well-suited to a problem domain, it the programs it makes easy to express are solutions to problems in that domain. The programs that the language makes hard to express are ones that are incorrect. This is desirable! One way to design a language is to selectively add restrictions until it is hard to express incorrect programs for the target domain. The cost of a language design is that some correct and potentially useful programs also become hard to express in the language.

The **type system** is one tool for restricting a language. A type system associates metadata with values and the variables that can be bound to them. A well-typed program is one where constraints on the metadata imposed by the language and program itself are satisfied. When these are violated, e.g., by assigning a "String" value to an "int" variable in Java, the program is likely incorrect. Some kinds of program errors can be detected by static analysis, which means examining the program without executing it. Some kinds of errors cannot be detected efficiently through static analysis, or are statically undecidable. Many of these can be detected by dynamic analysis, which means executing type checks while the program is executing.

We say that a language exhibits **type soundness** if well-typed programs cannot "go wrong" [Mil78], i.e., by reaching stuck states [WF94] from which further execution rules are undefined. Another view of this is that "A language is **type-safe** if the only operations that can be performed on data in the language are those sanctioned by the type of the data." [Sar97]

By declaring undesirable behaviors–such as dereferencing a null pointer, accessing a private member of another class, or reading from the filesystem–to be type errors and thus unsanctioned, the language designer can leverage type soundness to enforce safety and security.

All languages assign a type to a value at least before it is operated on, since operations are not well-defined without an interpretation of the data. Most languages also assign types to values that are simply stored in memory. One set of languages that does not is assembly languages: values in memory (including registers) are just bytes and the programmer must keep track of their interpretation implicitly. **Statically typed** languages contain explicit declarations that limit the types of values a to which a variable may be bound. C++ and Java are statically typed languages. **Dynamically typed** languages such as Scheme and Python allow a variable to be bound to any type of value. Some languages, like ML, are dynamically typed but the interpreter uses **type inference** to autonomously assign static types where possible.

1.2 Imperative and Functional

The discipline of computer science grew out of mathematics largely due to the work of Church and his students, particularly Turing. Church and Kleene created a mathematical system called the λ **calculus** (also written out as the lambda calculus) that treats mathematical functions as first-class values within mathematics. It is minimalist in the sense that it contains the fewest possible number of expressions, yet can encode any decidable function. Turing created the **Turing machine** abstraction of a minimal machine for performing computations. These were then shown to be equivalent an minimal models of computation, which is today called the **Church-Turing Thesis**.

These different models of computation are inherited by different styles of programming. Turing's machine model leads to **imperative programming**, which operates by mutating (changing) state and proceeds by iteration. Java and C++ are languages that encourage this style. Church's mathematical model leads to **functional programming**, which operates by invoking functions and proceeds by recursion. Scheme, ML, Unix shell commands, and Haskell are languages that encourage this style. So-called scripting languages like Python and Perl encourage blending of the two styles, since they favor terseness in all expressions.

2 Life of a Program

A program goes through three major stages: Source, Expressions, and Values. Formal specifications describe the syntax of the source and the set of expressions using an **grammar**, typically in BNF. This is called the **expression domain** of the language. The **value domain** is described in set notation or as BNF grammars. Expressions are also called **terms**. Expressions that do not reduce to a value are sometimes called **statements**.

An analogy to a person reading a book helps to make clear the three stages. The physical ink on the printed page is source. The reader scans the page, distinguishing tokens of individual letters

[*]morgan@cs.williams.edu, http://graphics.cs.williams.edu

and symbols from clumps of ink. In their mind, these are assigned the semantics of words–i.e., expressions. When those expressions are evaluated, the value (meaning) of the words arises in the readers mind. This distinction is subtle in the case of literals. Consider a number written on the page, such as “32”. The curvy pattern of ink is the source. The set of two digits next to each other is the expression. The interpretation of those digits in the reader’s mind is the number value. The number value is not something that can be written, because the act of writing it down converts it back into an expression. Plato might say that the literal expression is a shadow on the cave wall of the true value, which we can understand but not directly observe. [1]

2.1 Source Code and Tokens

A program begins as **source code**. This is the ASCII (or unicode!) string describing the program, which is usually in a file stored on disk. A **tokenizer** converts the source to a stream of **tokens** in a manner that is specific to the language. For example, in Java the period character “.” becomes a separate token if it separates two identifiers (variables) but is part of a floating-point number if it appears in the middle of a sequence of digits, e.g., string.length() versus 3.1415. See java.StringTokenizer or G3D::TextInput for an example of an implementation.

Figures 2.1 and 2.1 show an example of the source code and resulting token stream for a simple factorial function implemented in the Scheme programming language. The tokenizer is often language-specific. For this example, the tokenizer tags each token as a parenthesis, reserved word, identifier, or numeral. Source code is usually stored in a string. A typical data structure for storing the token stream is an array of instances of a token class.

```
(define (factorial n)
     (if (< n 2)
          ; Base:
         1
          ; Recurse:
         (* n (factorial (- n 1)))))
```

Figure 1: Scheme source code for factorial.

“(”	“define”	“(”	“factorial”	“n”	“)”	“(”	“if”	“(”	“<”	“n”	“2”	“)”
Paren	Reserved	Paren	Identifier	Identifier	Paren	Paren	Reserved	Paren	Identifier	Identifier	Numeral	Paren

“1”	“(”	“*”	“n”	“(”	“factorial”	“(”	“-”	“n”	“1”	“)”	“)”	“)”	“)”	“)”
Numeral	Paren	Identifier	Identifier	Paren	Identifier	Paren	Identifier	Identifier	Numeral	Paren	Paren	Paren	Paren	Paren

Figure 2: Token stream for factorial.

2.2 Expressions

A **parser** converts the token stream into a **parse tree** of **expressions**. The legal expressions are described by the **expression domain** of the language, which is often specified in **BNF**. The nodes of a parse tree are instances of expressions (e.g., a FOR node, a CLASS-DEFINITION node) and their children are the subexpressions. The structure of the parse tree visually resembles the indenting in the source code. Figure 2.2 shows a parse tree for the expressions found in the token stream from figure 2.1.

[1]For the truly philosophical, what is in the mind, or what is represented as a value by bits in a computer’s memory, is still only a *representation* of the value. The actual value 32 simply exists is unique. There can be only one 32, which means it can’t be in multiple places at once–the bits representing 32 in a computer’s memory therefore act a pointer to the ideal 32. AI, PL, and philosophy meet when we consider whether the human mind is different, or just shuffling around around representations like a computer.

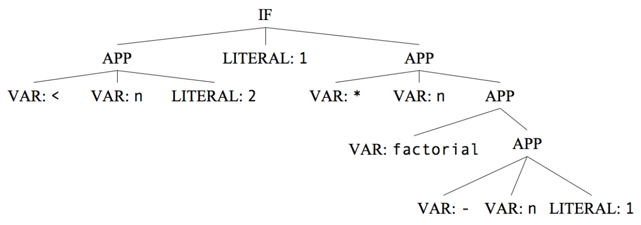

Figure 3: Parse tree for factorial.

The Scheme language contains the QUOTE special form for conveniently specifying parse trees directly as literal values, omitting the need for a tokenizer and parser when writing simple interpreters for languages that have an abstract syntax. The drawback of this approach is that simply quoting the factorial code in figure 2.1 would not produce the tree in figure 2.2. Instead, the result would be a tree of symbols and numbers without appropriate expression types labeling the nodes.

2.3 Values

When the program executes (if compiled, or when it is evaluated by an interpreter if not), expressions are reduced to values. The set of legal values that can exist during execution is called the **value domain**. The value domain typically contains all of the first-class values, although some languages have multiple value domains and restrict what can be done to them. In general, a value is **first-class** in a language if all of the following hold:

1. The value can be returned from a function
2. The value can be an argument to a function
3. A variable can be bound to the value
4. The value can be stored in a data structure

Java generics (a polymorphism mechanism) do not support primitive types like `int`, so in some sense those primitives are second-class in Java and should be specified in a separate domain from `Object` and its subclasses, which are first-class. In Scheme and C++, procedures (functions) and methods are first-class because all of the above properties hold. Java methods are not first-class, so that language contains a `Method` class that describes a method and acts as a proxy for it.

The value domain can be specified using set notation, e.g.,

$$\begin{aligned} real &= int \cup decimal \\ complex &= real \times real \\ number &= real \cup complex \end{aligned}$$

or using a BNF grammar (at least, for a substitution interpreter), which is described later.

2.4 Implementation Issues

There is a design tradeoff when implementing a language between compactness and abstraction. Using the same types in the implementation language for source, expressions, and values reduces the amount of packing and unpacking of values that is needed, and allows procedures in the implementation language to operate directly on the values in the target language. Furthermore, in Scheme, the READ procedure and QUOTE special form allow easy creation of tree values using literals that are syntactically identical to Scheme source code. This avoids the need for an explicit tokenizer and parser. Using the same types across domains violates the abstraction of those domains. This can make the implementation of the language harder to understand (when it grows large), and limits the ability of the type checker to detect errors in the implementation. For example, when implementing a Scheme interpreter in Java, one could choose to implement Scheme symbols, strings, identifiers, and source all as Java strings, without a wrapper class to distinguish them. It would be easy to accidentally pass a piece of source code to a method that expected an identifier, and the Java compiler could not detect that error at compile time because the method was only typed to expect a String, not a SchemeIdentifier.

3 Interpreters and Compilers

A compiler is a program that translates other programs in a high-level language to the machine language of a specific computer. The result is sometimes called a **native** binary because it is in the native language of the computer[2]. An interpreter is a program that executes other programs without compiling them to native code. There is a wide range of translation within the classification of interpreters. At one end of this range, some interpreters continuously re-parse and interpret code as they are moving through a program. At the other end, some interpreters essentially translate code down to native machine language at runtime so that the program executes very efficiently.

Although most languages can be either compiled or interpreted, they tend to favor only one execution strategy. C++, C, Pascal, Fortran, Algol, and Ada are typically compiled. Scheme, Python, Perl, ML, Matlab, JavaScript, HTML, and VisualBasic are usually interpreted. Java is an interesting case that compiles to machine language for a computer that does not exist. That language is then interpreted by a virtual machine (JVM).

Compilers tend to take advantage of the fact that they are run once for a specific instance of a program and perform much more static analysis. This allows them to produce code that executes efficiently and to detect many program errors at compile time. Detecting errors before a program actually runs is important because it reduces the space of possible runtime errors, which in turn increases reliability. Compiled languages often have features, such as static types, that have been added specifically to support this kind of compile-time analysis.

Interpreters tend to take advantage of the fact that code can be easily modified while it is executing to allow extensive interaction and debugging of the source program. This also makes it easier to patch a program without halting it, for example, when upgrading a web server. Many interpreted languages were designed with the knowledge that they would not have extensive static analysis and therefore omit the features that would support it. This can increase the likelihood of errors in the programs, but can also make the source code more readable and compact. Combined with the ease of debugging, this makes interpreted languages often feel “friendlier” to the programmer. This typically comes at the cost of decreased runtime performance cost increased runtime errors.

Compiled programs are favored for distributing proprietary algorithms because it is hard to reverse engineer a high-level algorithm from machine language. Interpreted programs by their nature require that the source be distributed, although it is possible to obfuscate or, in some languages, encrypt the source to discourage others from reading it.

[2] Although in practice, most modern processors actually emulate their published interface using a different set of operations and registers. This allows them include new architectural optimizations without changing the public interface, for compatibility.

4 Syntax

Although we largely focus on semantics, some notable points about syntax:

- A parser converts source code to expressions
- Backus-Naur Form (BNF) formal grammars are a way of describing syntax using recursive patterns
- **Infix** syntax places an operator between its arguments, e.g., “`1 + 2`”. Java uses infix syntax for arithmetic and member names, but prefix syntax for method application.
- Prefix syntax places the operator before the operands, e.g., “`add(1, 2)`”, which conveniently allows more than one or two operands and unifies operator and function syntax. Scheme uses prefix syntax for all expressions.
- Postfix places the operator after the operands, which allows nested expressions where the operators take a fixed number of arguments, without requiring parentheses. Postscript and some calculators use postfix.
- Scheme’s “abstract syntax” makes it easy to parse
- **Macros** allow a programmer to introduce new syntax into a language
- Python has an interesting syntax in which whitespace is significant. This reduces visual clutter but makes the language a little difficult to parse and to edit (in some cases)
- **Syntactic sugar** makes a language sweeter to use without increasing its expressive power

4.1 Backus-Naur Form

Backus-Naur Form (**BNF**) is a formal way of describing context-free grammars for formal languages. A grammar is **context-free** when the the grammar is consistent throughout the entire language (i.e., the rules don’t change based on context). BNF was first used to specify the ALGOL programming language.

Beyond its application to programming language syntax, BNF and related notations are useful for representing the grammars of any kind of structured data. Examples include file formats, types, database records, and string search patterns.

A BNF grammar contains a series of rules (also known as **productions**). These are patterns that legal programs in the specified language must follow. The patterns are typically recursive. In the BNF syntax, the **nonterminal** being defined is enclosed in angular brackets, followed by the “::=” operator, followed by an expression pattern. The expression pattern contains other nonterminals, terminals enclosed in quotation marks, and the vertical-bar operator “|” that indicates a choice between two patterns. For example,

$$\begin{array}{rcl} \langle \textit{digit} \rangle & ::= & \text{‘0’} \mid \text{‘1’} \mid \text{‘2’} \mid \text{‘3’} \mid \text{‘4’} \mid \text{‘5’} \mid \text{‘6’} \mid \text{‘7’} \mid \text{‘8’} \mid \text{‘9’} \\ \langle \textit{digits} \rangle & ::= & \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{digits} \rangle \end{array}$$

In this document, these are typeset using an unofficial (but common) variation, where terminals are typeset as $\boxed{\texttt{x}}$ and nonterminals as x. This improves readability for dense expressions. With this convention, digits are:

$$\begin{array}{rcl} \textit{digit} & ::= & \boxed{0} \mid \boxed{1} \mid \boxed{2} \mid \boxed{3} \mid \boxed{4} \mid \boxed{5} \mid \boxed{6} \mid \boxed{7} \mid \boxed{8} \mid \boxed{9} \\ \textit{digits} & ::= & \textit{digit} \mid \textit{digit}\ \textit{digits} \end{array}$$

It is common to extend BNF with regular expression patterns to avoid the need for helper productions. These include the following notation:

$(x) = x$; parentheses are for grouping only

$[x]$ = zero or one instances of x (i.e., x is optional)

x^* = zero or more instances of x

x^+ = one or more instances of x

An example of these patterns for expressing a simple programming language literal expression domain (e.g., a subset of Scheme's literals):

$$\begin{aligned}
\mathit{boolean} &::= \texttt{\#t} \mid \texttt{\#f} \\
\mathit{digit} &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\mathit{integer} &::= [\,\texttt{+} \mid \texttt{-}\,]\ \mathit{digit}^+ \\
\mathit{rational} &::= \mathit{integer}\ \texttt{/}\ \mathit{digit}^+ \\
\mathit{decimal} &::= [\,\texttt{+} \mid \texttt{-}\,]\ \mathit{digit}^*\ \texttt{.}\ \mathit{digit}^+ \\
\mathit{real} &::= \mathit{integer} \mid \mathit{rational} \mid \mathit{decimal}
\end{aligned}$$

BNF can be applied at both the character level (e.g., to describe a lexer/tokenizer) and the token level (e.g., to describe a parser). The preceding example operates on individual characters within a string and is useful to a tokenizer. An example of a subset of Scheme's expression domain represented in BNF at the token level is:

$$\begin{aligned}
\mathit{variable} &::= \mathit{id} \\
\mathit{let} &::= \texttt{(}\ \texttt{let}\ \texttt{(}\ (\ \texttt{[}\ \mathit{id}\ \mathit{exp}\ \texttt{]}\ )^*\ \texttt{)}\ \mathit{exp}\ \texttt{)} \\
\mathit{lambda} &::= \texttt{(}\ \texttt{lambda}\ \texttt{(}\ \mathit{id}^*\ \texttt{)}\ \mathit{exp}\ \texttt{)} \\
\mathit{exp} &::= \mathit{variable} \mid \mathit{let} \mid \mathit{lambda}
\end{aligned} \tag{1}$$

4.2 Syntactic Sugar

Some expressions make a language's syntax more convenient and compact without actually adding expressivity: they make the language sweeter to use. We say that an expression is **syntactic sugar** and adds no expressive power if it can be reduced to another expression with only local changes. That is, without rewriting the entire body of the expression or making changes to other parts of the program.

For example, in Java any FOR statement, which has the form:

$\texttt{for}\ \texttt{(}\ \mathit{init}\ \texttt{;}\ \mathit{test}\ \texttt{;}\ \mathit{incr}\ \texttt{)}\ \mathit{body}$

can be rewritten as a WHILE statement of the form:

$\mathit{init}\ \texttt{;}\ \texttt{while}\ \texttt{(}\ \mathit{test}\ \texttt{)}\ \{\ \mathit{body}\quad \mathit{incr}\ \}$

FOR therefore does not add expressivity to the language and is syntactic sugar. In Scheme, LET adds no power over LAMBDA, and LET* adds no power over LET. Java exceptions are an example of an expressive form that cannot be eliminated without completely rewriting programs in the language.

5 Semantics

5.1 Operational Semantics

An **operational semantics** is a mathematical representation of the semantics of a language. It defines how expressions are reduced to other expressions by applying a set of rules. Each rule has preconditions. Any rule whose preconditions are met can be applied. Note that the semantics do not imply the implementation. For example, an operation that requires $O(n^2)$ rule applications may require only $O(n)$ operations on an actual computer. Likewise, eager and lazy substitution are often ambiguous at the semantic level.

The rules are expressed using a notation for reductions, conditional reductions, and substitutions. The most general form of a rule is:

$$x \Rightarrow y \tag{2}$$

"Expressions matching x reduce to y"

where x is a variable of the notation itself, not necessarily a variable in the programming language. Variables in operational semantics notation correspond to expressions, so they follow the BNF notation. Recall that in this document, I'm typesetting BNF nonterminals as x and terminals as $\texttt{x}$. Where multiple expressions of the same kind appear in a rule, subscripts[3] distinguish them, e.g., exp_1, exp_2, $\mathit{exp}_{\mathit{body}}$.

A specific example of a reduction rule is the additive identity in arithmetic:

$$\texttt{0} + \mathit{exp} \Rightarrow \mathit{exp} \tag{3}$$

Raw reduction rules are rare. Often a reduction can only be applied when some precondition is met. These conditional rules are expressed as:

$$\frac{a}{b} \tag{4}$$

"If mathematical statement a is true, then reduction b may be applied." There may be multiple conditions in a, separated by space, that must all be true for b to be applied.

For example,

$$\frac{\mathit{exp}_1 \Rightarrow \texttt{0}}{\mathit{exp}_2\ \texttt{/}\ \mathit{exp}_2 \Rightarrow \text{divide by zero error}} \tag{5}$$

means, "if an expression can be reduced to zero (by some other rules), then any division with that expression in the denominator can be reduced to a divide-by-zero error."

Variable substitution is handled by the notation:

$$[\mathit{id} \rightarrowtail \mathit{val}]\,\mathit{body} \tag{6}$$

"Substitute val for id in body".

The *body* is the expression in which all instances of variable named *id* are to be replaced with the expression *val*. In an eager language, the *val* expression is only in the value domain. In a lazy language, *val* does not need to be reduced first.

6 Computability

6.1 The Incompleteness Theorem

At the beginning of the 20th century, mathematicians widely believed that all true theorems could be reduced to a small set of axioms. The assumption was that mathematics was sufficiently powerful to prove all true theorems. Hilbert's program[4] was to actually reduce the different fields of mathematics to a small and consistent

[3] This is admittedly a suboptimal notation, since the "name" that carries the meaning for the reader but is buried in the subscript, while the "type" dominates. However, it is standard in the field.

[4] "program" as in plan of action, not code

set of axioms, thus putting them all on a solid and universal foundation.

In 1931 Gödel [G3̈1][vH67, 595] proved that in any sufficiently complex system of mathematics (i.e., formal language capable of expressing at least arithmetic), there exist true statements that cannot be proven using that system, and that the system is therefore incomplete (unable to prove its own consistency). This **Incompleteness Theorem** was a surprising result and indicated that a consistent set of axioms could not exist. That result defeated Hilbert's program[5] and indicated for the first time the limitations of mathematics. This is also known as the First Incompleteness Theorem; there is a second theorem that addresses the inconsistency of languages that claim to prove their own consistency.

Here is a proof of the Incompleteness Theorem following Gödel's argument. Let every statement in the language be encoded by a natural number, which is the **Gödel Number** of that statement. This encoding can be satisfied by assigning every operator, variable, and constant to a number with a unique prefix and then letting each statement be the concatenation of the digits of the numbers in it. (This is roughly equivalent to treating the text of a program as a giant number containing the concatenation of all of its bits in an ASCII representation.) For example, the statement "$x > 4$" might be encoded by number g:

$$S_g(x) = \text{“}x > 4\text{”} \tag{7}$$

Now consider the self-referential ("recursive") statement,

$$S_i(n) = \text{“}S_n \text{ is not provable.”} \tag{8}$$

evaluated at $n = i$. This statement is a formal equivalent of the **Liar's Paradox**, which in natural language is the statement, "This sentence is not true." $S_n(n)$ creates an inconsistency. As a paradox, it can neither be proved (true), nor disproved (false).

As a result of the Incompleteness Theorem, we know that there exist functions whose results cannot be computed. These **non-computable functions** (also called **undecidable**) are interesting for computer science because they indicate that there are mathematical statements whose validity cannot be determined mechanically. For computer science, we define computability as:

A function f is **computable** if there exists a program P that computes f, i.e., for any input x, the computation $P(x)$ halts with output $f(x)$.

Unfortunately, many of undecidable statements are properties of programs that we would like a compiler to check. A constant challenge in programming language development is that it is mathematically impossible to prove certain properties about arbitrary programs, such as whether a program does not contain an infinite loop.

6.2 The Halting Problem

Let the **Halting Function** $H(P,x)$ be the function that, given a program P and an input x to P, has value "halts" if $P(x)$ would halt (terminate in finite time) were it to be run, and has value "does not halt" otherwise (i.e., if $P(x)$ would run infinitely, if run). The Halting Problem is that of solving H; Turing [Tur36] proved in 1936 that H is undecidable in general.

Theorem 1. *$H(P,x)$ is undecidable.*

Proof. Assume program $Q(P,x)$ computes H (somehow). Construct another program $D(P)$ such that

```
$D(P)$:
    if $Q(P,P)$ = "halts" then loop
    else halt
```

In other words, $D(P)$ exhibits the opposite halting behavior of $P(P)$.

Now, consider the effect of executing $D(D)$. According to the program definition, $D(D)$ must halt if $D(D)$ would run forever, *and* $D(D)$ must run forever if $D(D)$ would halt. Because $D(D)$ cannot both halt and run forever, this is a contradiction. Therefore the assumption that Q computes H is false. We made no further assumption beyond H being decidable, therefore H must be undecidable. □

The proof only holds when H must determine the status of every program and every input. It *is* possible to prove that a specific program with a specific input halts. For a sufficiently limited language, it is possible to solve the Halting Problem. For example, every finite program in a language without recursion or iteration must halt.

The theorem and proof can be extended to most observable properties of programs. For example, within the same structure one can prove that it is undecidable whether a program prints output or reaches a specific line in execution. Note that it is critical to the proof that $Q(P,x)$ does not actually run P; instead, it must decide what behavior P *would* exhibit, were it to be run, presumably by examining the source code of P. See http://www.cgl.uwaterloo.ca/ csk/halt/ for a nice explanation of the Halting Problem using the C programming language.

7 The λ Calculus

The λ calculus is Church's [Chu32] minimalist functional model of computation. Church showed that all other programming constructs can eliminated by reducing them to single-argument procedure definition (i.e., abstraction; lambda), variables, and procedure application. Variations of λ calculus are heavily used in programming language research as a vehicle for proofs. Outside research, there are several motivations for studying λ calculus and reductions to it from more complex languages.

Philosophically, λ calculus is the[6] foundation for our understanding of computation and highlights the power of abstraction. Practically, understanding the language and how to reduce others to it changes the way that one thinks about (and applies) constructs in other languages. This leads the way to emulating constructs that are missing in a language at hand, which makes for a better programmer. For example, Java lacks lambda. The Java API designers quickly learned to use anonymous classes to create anonymous closures, enabling the use of first-class function-like objects in a language that does not support functions. C++ programmers discovered a way to use the polymorphic mechanism of templates as a complete macro language.

On learning a new language, the sophisticated programmer does not learn the specific forms of that language blindly but instead asks, "which forms create closures, recursive bindings, iteration, etc. in this language?". If any of the desired features are missing, that programmer then emulates them, using techniques learned by emulating complex features in the minimalist λ calculus. So, although implementing Church Booleans is just an academic puzzle

[5]...and answered Hilbert's "second problem": prove that arithmetic is self-consistent. Whitehead and Russell's *Principia Mathematica* previously attempted to derive all mathematics from a set of axioms.

[6]or at least, one of the two...

for most programmers, that kind of thought process is valuable in implementing practical applications.

André van Meulebrouck describes an alternative motivation:

> Perhaps you might think of Alonzo Church's λ-calculus (and numerals) as impractical mental gymnastics, but consider: many times in the past, seemingly impractical theories became the underpinnings of future technologies (for instance: Boolean Algebra [*i.e., today's computers that operate in binary build massive abstractions using only Boole's theoretical logic!*]).
>
> Perhaps the reader can imagine a future much brighter and more enlightened than today. For instance, imagine computer architectures that run combinators or λ-calculus as their machine instruction sets.[7]

7.1 Syntax

The λ calculus is a language with surprisingly few primitives in the expression domain[8]:

$$\begin{aligned} var &::= id \\ abs &::= \boxed{\lambda}\ id\ \boxed{.}\ exp \\ app &::= exp\ \ exp \\ exp &::= var \mid abs \mid app \mid \boxed{(}\ exp\ \boxed{)} \end{aligned}$$

The last expression on the right simply states that parentheses may be used for grouping.

The language contains single value type, the single-argument procedure, in the value domain. In set notation this is:

$$val = proc = var \times exp$$

and in BNF:

$$val \quad ::= \quad \boxed{\lambda}\ id\ \boxed{.}\ exp$$

The abbreviated names used here and in the following discussions are mnemonics for: 'id' = 'identifier', 'abs' = 'abstraction' (since λ creates a procedure, which is an abstraction of computation), 'app' = 'procedure application', 'exp' = 'expression', 'proc' = 'procedure', and 'val' = 'value'.

7.2 Semantics

The formal semantics are simply those of substitution [Pie02, 72]:

App-Part 1: (reduce the procedure expression towards a value)

$$\frac{exp_p \Rightarrow exp'_p}{exp_p exp_a \Rightarrow exp'_p exp_a} \tag{9}$$

App-Part 2: (reduce the actual parameter towards a value)

$$\frac{exp_a \Rightarrow exp'_a}{exp_p exp_a \Rightarrow exp_p exp'_a} \tag{10}$$

App-Abs: (apply a procedure to a value)

$$\boxed{\lambda}\ id\ \boxed{.}\ exp_{body}\ \ val\ \Rightarrow\ [id \rightarrowtail val]\, exp_{body} \tag{11}$$

The App-Abs rule relies on the same syntax for the *val* value and *abs* expression, which is fine in λ calculus because we're using pure textural substitution. In the context of a true value domain that is distinct from the expression domain, we could express it as an *abs* evaluation rule for reducing a procedure expression to a procedure value and and an application rule written something like:

$$\frac{val_p = \boxed{\lambda}\ id\ \boxed{.}\ exp_{body}}{val_p\ val_a \Rightarrow\ [id \rightarrowtail val_a]\, exp_{body}} \tag{12}$$

7.3 Examples

For the sake of giving simple examples, momentarily expand λ calculus with the usual infix arithmetic operations and integers. Consider the evaluation of the following expression:

$$\lambda\ x\ .\ (x+3)\ 7 \tag{13}$$

This is an app expression. The left sub-expression is a abs, the right sub-expression is an integer (7). Rule App-Abs is the only rule that applies here, since both the procedure and the integer cannot be reduced further. App-Abs replaces all instances of x in the body expression $(x+3)$ with the right argument expression, 7:

$$\begin{aligned} &\lambda\ x\ .\ (x+3)\ 7 && (14)\\ &\Rightarrow\ [x \rightarrowtail 7](x+3) && (15)\\ &= (7+3) && (16)\\ &= 10 && (17) \end{aligned}$$

Arithmetic then reduces this to 10.

Now consider using a procedure as an argument:

$$(\lambda f\ .\ (f\ 3))\ (\lambda x.(1+x)) \tag{18}$$

This is again an app. In this case, both the left and right are abs expressions. Applying rule App-Abs substitutes the right expression for f in the body of the left procedure, and then we apply App-Abs again:

$$\begin{aligned} &(\lambda f\ .\ (f\ 3))\ (\lambda x.(1+x)) && (19)\\ &\Rightarrow [\ f\ \rightarrowtail\ (\lambda x.(1+x))\ ](f\ 3)) && (20)\\ &= \lambda x.(1+x)\ 3 && (21)\\ &\Rightarrow [x \rightarrowtail 3](1+x) && (22)\\ &= (1+3) && (23)\\ &= 4 && (24) \end{aligned}$$

7.4 Partial Function Evaluation

Because procedure application is left associative and requires no function application operator:

$$f\ x\ y = (f\ x)\ y \tag{25}$$

we can emulate the functionality of multiple-argument procedures using single argument procedures. For example, the "two-argument" procedure that sums its arguments is:

$$\lambda x.\lambda y.(x+y) \tag{26}$$

The outer expression is an abs (procedure definition), whose body is another abs. This is a first-order procedure. When placed into nested app expressions, the procedure returns another procedure, which then consumes the second app's argument:

$$\begin{aligned} &(\lambda x.\lambda y.(x+y))\ \ 1\ \ 2 && (27)\\ &\Rightarrow \lambda y.(1+y)\ \ 2 && (28)\\ &\Rightarrow 1+2 && (29)\\ &= 3 && (30) \end{aligned}$$

[7]http://www.mactech.com:16080/articles/mactech/Vol.07/07.06/ChurchNumerals/

[8]This is specifically a definition of the *untyped* λ-calculus.

The process of converting a 0th-order function of n arguments into an nth-order function of one argument, like the one above, is called **currying**. It is named for the logician Haskell Curry, who was not its inventor[9].

When an nth-order function is applied to $k < n$ arguments, in λ calculus, the result reduces to an $(n-k)$th order function. The resulting function "remembers" the arguments that have been provided because they have already been substituted, and it will complete the computation when later applied to the remaining arguments. This is called **partial function evaluation**, and is a feature of some languages including Haskell (which is *also* named for Haskell Curry, who did not invent it either.) For example, the addition function above can be partially evaluated to create an "add 5" function:

$$(\lambda x.\lambda y.(x+y))\ 5 \tag{31}$$
$$\Rightarrow \lambda y.(5+y) \tag{32}$$

7.5 Creating Recursion

A **fixed point** of a function f is a value v such that $f(v) = v$ (in mathematical notation; in λ calculus, we would say $f\ v = v$. A function may have zero or more fixed points. For example, the identity function $\lambda x.x$ has infinitely many fixed points. Let $s = \lambda x.x^2$ be the function that squares its argument; it has fixed points at 0 and 1.

A **fixed point combinator** is a function that computes a fixed point of another function. This is interesting because it is related to recursion. Consider the problem of defining a recursive function in λ calculus. For example, define factorial (for convenience, we temporarily extend the language with conditionals and integers; those can be reduced as shown previously):

$$\begin{aligned}&\lambda n.(\text{if } (\text{iszero } n)\\ &\qquad 1\\ &\qquad (n*(f\ (n-1))))\end{aligned}$$

The problem with this definition is that we need the f embedded inside the recursive case to be bound to the function itself, but that value does not exist at the time that the function is being defined. Alternatively, the problem is that f is a free variable. Adding another abs expression captures f:

$$\begin{aligned}&\boxed{\lambda f.}\ \lambda n.(\text{if } (\text{iszero } n)\\ &\qquad 1\\ &\qquad (n*(f\ (n-1))))\end{aligned}$$

This just says that if we already had the factorial function that operated on values less than n, we could implement the factorial function for n. That's close to the idea of recursion, but is not fully recursive because we've only implemented one inductive step. We need to handle all larger values. To let this inductive step run further, say that f is the function that we're defining, which means that the inner call requires two arguments: f and $n-1$:

$$\begin{aligned}&\lambda f.\lambda n.(\text{if } (\text{iszero } n)\\ &\qquad 1\\ &\qquad (n*(f\ \boxed{f}\ (n-1))))\end{aligned}$$

Call this entire function g. It is a function that, given a factorial function, creates the factorial function. At first this does not sound useful–if we had the factorial function, we wouldn't need to write it! However, consider that what we have defined g such that $(gf) = f$...in other words, the factorial function is the fixed point of g. For this particular function, we can find the fixed point by binding it and then calling it on itself. Binding and applying values are accomplished using abs and app expressions. An expression of the form:

$$(\lambda z.z\ z)\ g \Rightarrow g\ g \tag{33}$$

applies g to itself. Wrapping our entire definition with this:

$$\begin{aligned}&\boxed{(\lambda z.z\ z)}\\ &\quad (\lambda f.\lambda n.(\text{if } (\text{iszero } n)\\ &\qquad 1\\ &\qquad (n*(f\ f\ (n-1)))))\end{aligned}$$

produces a function that is indeed factorial, albeit written in a strange manner. Convince yourself of this by running it in Scheme, using the following translation and application to 4:

```
(
  ((lambda (z) (z z))
   (lambda (f)
     (lambda (n)
       (if (zero? n)
           1
          (* n ((f f) (- n 1)))))))

4)
```

When run, this correctly produces $4! = 4*3*2*1 = 24$.

This exercise demonstrates that it is possible to implement a recursive function without an explicit recursive binding construct like LETREC. For the factorial case, we manually constructed a generator function g and its fixed point. Using a fixed point combinator we can automatically produce such fixed points, simplifying and generalizing the process.

Curry discovered the simplest known fixed point combinator for this application. It is known as the **Y combinator** (a.k.a. applicative-order Z combinator as expressed here), and is defined as:

$$\boxed{\begin{aligned}Y\ =\ &\lambda f.\\ &\quad ((\lambda z\ .\ z\ z)\\ &\qquad (\lambda x\ .\ f(\lambda y\ .\ x\ x\ y)))\end{aligned}} \tag{34}$$

When applied to a generator function, Y finds its fixed point and produces that recursive function. The Y combinator is formulated so that the generator need not apply its argument to itself. That is, the step where we rewrote $(f\ (n-1))$ as $(f\ f\ (n-1))$ in our derivation is no longer necessary.

A Scheme implementation of Y and its use to compute factorial are below. The use of the DEFINE statement is merely to make the implementation more readable. Those can be reduced to LAMBDA and application expressions.

```
; Creates the fixed point of its argument
(define Y
  (lambda (f)
    ((lambda (z) (z z))
     (lambda (x) (f (lambda (y) ((x x) y)))))))

; Given the factorial function, returns the factorial function
(define generator
  (lambda (fact)
     (lambda (n)
       (if (zero? n)
           1
          (* n (fact (sub1 n)))))))
```

[9] The idea is commonly credited to Schönfinkel in the 20th century, and was familiar to Frege and Cantor in the 19th [Pie02, 73]

```
; The factorial function: prints (lambda...)
(Y generator)

; Example: prints 24
((Y generator) 4)
```

References

Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33:346–366, 1932.

Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme. *I. Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

Vijay Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. http://www.research.att.com/ vj/bug.html.

Alan Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. http://www.turingarchive.org/browse.php/B/12.

Jean van Heijenoort, editor. *From Frege to Gdel: A Source Book in Mathematical Logic, 1979-1931*. Harvard University Press, 1967.

Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.