Programming Language Notes

February 24, 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, which 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's notation contains both syntax and semantics. Since syntax can be easily translated, the semantics are more fundamental.

Church and Turing showed that the minimal semantics of the Lambda Calculus and Universal Turing Machine are sufficient to emulate the semantics of any more complicated programming language or machine. Reducing a particular language to the Lambda 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 are solutions to problems in that domain. The programs that the language makes hard to express are ones that are incorrect. This is a feature! 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 create type errors. By declaring undesirable behaviors within the problem domain (such as dereferencing a null pointer, accessing a private member of another class, or reading from the filesystem) to be type errors, the language designer can leverage type soundness to enforce safety and security.

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

1.2 Imperative and Functional

The discipline of computer science grew out of mathematics largely due to the work of Turing and his advisor, Church. Church 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 **Universal 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 support this style. Church's mathematical model leads to **functional programming**, which operates by invoking functions and proceeds by recursion. Scheme, ML, and Haskell are languages that support this style.

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.

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 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

[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.

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.

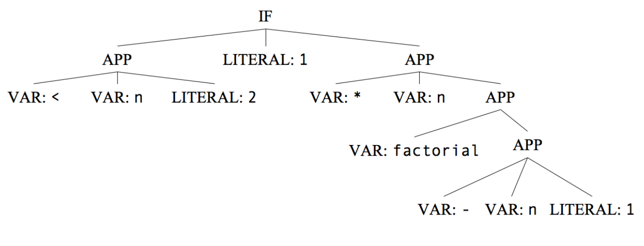

Figure 3: Parse tree for factorial.

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

[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.

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.

4 Syntax

Although we largely focus on semantics, some points about syntax covered in the course:

- 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.

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,

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

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

$$\begin{array}{rcl} \langle \textit{boolean} \rangle & ::= & \text{`\#t'} \mid \text{`\#f'} \\ \langle \textit{digit} \rangle & ::= & \text{`0'} \mid \text{`1'} \mid \text{`2'} \mid \text{`3'} \mid \text{`4'} \mid \text{`5'} \mid \text{`6'} \mid \text{`7'} \mid \text{`8'} \mid \text{`9'} \\ \langle \textit{integer} \rangle & ::= & [\text{`+'} \mid \text{`-'}] \langle \textit{digit} \rangle^+ \\ \langle \textit{rational} \rangle & ::= & \langle \textit{integer} \rangle \text{`/'} \langle \textit{digit} \rangle^+ \\ \langle \textit{decimal} \rangle & ::= & [\text{`+'} \mid \text{`-'}] \langle \textit{digit} \rangle^* \ \text{`.'} \ \langle \textit{digit} \rangle^+ \\ \langle \textit{real} \rangle & ::= & \langle \textit{integer} \rangle \mid \langle \textit{rational} \rangle \mid \langle \textit{decimal} \rangle \end{array}$$

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 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:

`for (` *init* `;` *test* `;` *incr* `)` *body*

can be rewritten as a WHILE statement of the form:

init `; while (` *test* `)` { *body*   *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 Computability

5.1 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[3] was to actually reduce the different fields of mathematics to a small and consistent 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), there exist true statements that cannot be proven using that system. This **Incompleteness Theorem** was a surprising result and indicated that a consistent set of axioms could not exist. That result defeated Hilbert's program 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:

$$S_g(x) = \text{“}x > 4\text{”} \tag{1}$$

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

$$S_i(n) = \text{“}S_n \text{ is not provable.”} \tag{2}$$

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.

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

5.2 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.

References

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

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.