# Programming Language Notes

April 9, 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 well-written program is a poem. Both are powerful because their content is condensed without being inscruitable and because the form is careful chosen to give insight into the topic. For a program, the topic is an algorithm and the implementation should emphasize the key steps while minimizing the details. The most elegant implementation is not always the most efficient, although it often is within a constant factor of optimal. The choice of programming language most closely corresponds to the choice of poem structure, e.g., sonnet or villanelle, not the choice of natural language, e.g., English or French. The structure of each enforces certain patterns and ways of thinking on the author and reader.

To author elegant programs, one must master a set of languages and language features. Then, one must subjectively but precisely choose among them to express specific algorithms. Languages are themselves designed. A well-designed *language* is a meta-poem. A language designer crafts a set of expressive tools suited to the safety, performance, and expressive demands of a problem domain. As with literature, the difficult creative choice is often not what to include, but what to omit.

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 (and Kleene) showed that the minimal semantics of the $\lambda$ 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 $\lambda$ 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. In these notes, our aperature on programming languages is features, which can increase and decrease the expressiveness of the language for certain domains.

### 1.1 Types

Every language makes some programs easy to express and others difficult. When a language is well-suited to a problem domain, the programs it makes easy to express are correct solutions to problems in that domain. A well-suited language furthermore makes it hard to express programs 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 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 at **run-time**–while the program is executing.

We say that a language exhibits **type soundness** if well-typed programs in that language cannot "go wrong" [Mil78]. That is, if well-typed programs cannot reach 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]

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

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 (even assembly 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 $\lambda$ **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 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, increasingly, 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,

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

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.

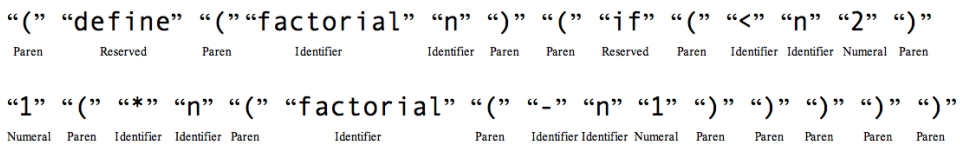


Figure 2: Token stream for factorial.

## 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 sub-expressions. 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.

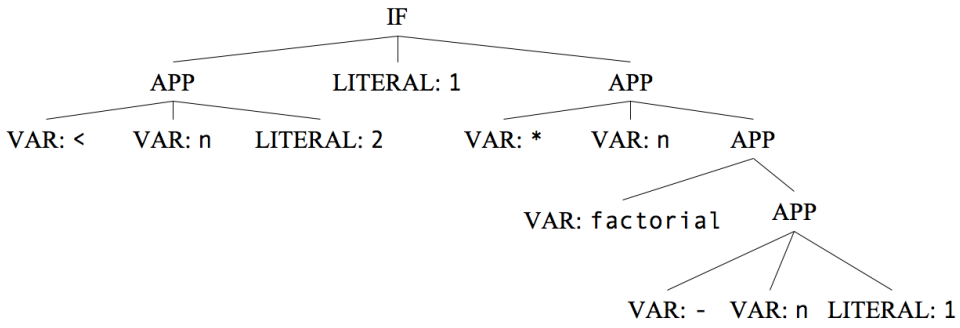


Figure 3: Parse tree for factorial.

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 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,

[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.

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 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,

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

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

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

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):

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

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:

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

### 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. Such expressions are also referred to as being **macro-expressive**. They can naturally be implemented entirely within the parser or as **macros** in languages with reasonable macro systems.

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.

We can express the Java `FOR` loop's **reduction** in more formal notation as:

$$
\begin{array}{c}
\texttt{for}\ \texttt{(}\ \textit{exp}_{\textit{init}}\ \texttt{;}\ \textit{exp}_{\textit{test}}\ \texttt{;}\ \textit{exp}_{\textit{incr}}\ \texttt{)}\ \textit{exp}_{\textit{body}} \\
\Rightarrow \\
\textit{exp}_{\textit{init}}\ \texttt{;}\ \texttt{while}\ \texttt{(}\ \textit{exp}_{\textit{test}}\ \texttt{)}\ \texttt{\{}\ \textit{exp}_{\textit{body}}\ \ \textit{exp}_{\textit{incr}}\ \texttt{;}\ \texttt{\}}
\end{array}
\tag{2}
$$

Here the pattern on the left may be reduced to the simpler form on the left during evaluation. This is an example of a general mechanism for ascribing formal semantics to syntax that is described further in the following section.

## 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. Equivalently, it gives a set of **progress rules** for progressing from complex expressions to simpler ones, and eventually to values. Each rule has preconditions: to be applied, the rule must match the pattern of the expression.

Any rule whose preconditions are met can be applied. When no rule applies the program halts. If it halts with a value, that is the result of the computation. If the semantics are self-consistent and rule expansion halts with an expression that is not a value, that indicates an error in the target program. The nature and location of the error are determined by the remaining expression and the last rule applied.

Note that the semantics influence but ultimately 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 interchangeable at the semantic level. We say that two implementations are **semantically equivalent** if they always reduce identical expressions to identical vales, even if they use different underlying algorithms to implement the semantics.

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

$$x \Rightarrow y \tag{3}$$

"Expressions matching $x$ reduce to $y$"

where $x$ is a placeholder in this notation, not a variable in the programming language. These placeholders will be filled by terms, or by term variables such as $exp_{subscript}$ and $val_{subscript}$. Here, the name of the variable indicates its domain (often, expression or value), and the subscript is a tag to distinguish multiple variables in the same statement[3] Sometimes rules are written more casually using the subscripts as the variables and ignoring the domains, when the domain is irrelevant.

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

$$exp_x \texttt{ + } 0 \Rightarrow exp_x \tag{4}$$

We can specify general addition by:

$$num_x \texttt{ + } num_y \Rightarrow \widehat{num_x + num_y} \tag{5}$$

The addition sign on the right of the reduction indicates actual addition of value; the one on the left denotes syntax for an expression. Variables $num_x$ and $num_y$ are expressions; the hat on the right side indicates that we mean the value corresponding to that operation.

To make this more concrete, we could give a very specific reduction:

$$\texttt{1 + 2} \Rightarrow \hat{3} \tag{6}$$

Here, 1 and 2 on the left of the arrow are syntax for literals, i.e., they are **numerals**. The hat on the 3 to the right of the arrow indicates that it is a value, i.e., it represents an actual **number** and not just the syntax for a number. See [Kri07, 231] for further discussion of this hat notation. We will see some cases in which the line between the expression domain and the value domain is blurry. In those, the hat notation is unnecessary.

Sometimes we have complicated conditions on a rule that constrain the pattern on the left side of $\Rightarrow$. These are notated with a conditional statement the form:

$$\frac{a}{b} \tag{7}$$

"If mathematical statement $a$ is true, then statement $b$ is true (applicable)."

In general, both $a$ and $b$ are reductions. Furthermore, there may be multiple conditions in $a$, notationally separated by spaces, that must all be true for the reduction(s) in $b$ to be applied. These rules are useful for making progress when no other rule directly applies. For example, to evaluate the mathematical expression $1+(7+2)$, we must first resolve $(7+2)$. The rule for making progress on addition with nested subexpression on the right of the plus sign is:

$$\frac{exp_2 \Rightarrow num_2}{exp_1 \texttt{ + } exp_2 \Rightarrow exp_1 \texttt{ + } num_2} \tag{8}$$

which reads, "if expression #2 can be reduced to some number (by some other rules), then the entire sum can be reduced to the sum of expression #1 and that number." We of course need some way of reducing expressions on the *left* of the plus sign as well:

$$\frac{exp_1 \Rightarrow num_1}{exp_1 \texttt{ + } exp_2 \Rightarrow num_1 \texttt{ + } exp_2} \tag{9}$$

[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.

Applying combinations of these rules allows us to simplify arbitrarily nested additions to simple number additions.

There are two major interpreter designs: substitution and evaluation interpreters. Substitution interpreters transform expressions to expressions and terminate when the final expression is a literal. Their value domain is their literal domain. Evaluation/environment interpreters are implemented with an `EVAL` procedure that reduces expressions directly to values. Such a procedure recursively invokes itself, but program execution involves a single top-level `EVAL` call. These two models of implementation correspond to two styles of semantics:

1. **Small-step** operational semantics rules reduce expressions to **expressions** (like a substitution interpreter)
2. **Big-step** operational semantics rules reduce expressions to **values** (like an `EVAL` interpreter)

The following two subsections demonstrate how semantics are assigned in each. We use the context of a simple language that contains only single-argument procedure definition, variable, conditional, application, and booleans. Let these have the semantics of the equivalent forms in the Scheme language:

$$\begin{aligned} exp \quad ::= \quad & \texttt{(}\ \lambda\ \texttt{(}\ id\ \texttt{)}\ exp\ \texttt{)} \mid \\ & id \mid \\ & \texttt{(}\ if\ exp\ exp\ exp\ \texttt{)} \mid \\ & \texttt{(}\ exp\ exp\ \texttt{)} \mid \\ & \texttt{true} \mid \texttt{false} \end{aligned} \tag{10}$$

## 5.2 Small-step Example

### 5.2.1 Rules

Small-step operational semantics rules reduce expressions to expressions. Although the choice of implementation is not constrained by the style of semantics (much), small step maps most directly to a substitution-based interpreter (e.g., [Kri07, 15]). Under small-step semantics the value domain is merely the terminal subset of the expression domain, plus procedure values. That is, literals are values. It is useful to us later to define a subdomains in this definition:

$$\begin{aligned} ok \quad ::= \quad & \texttt{true} \mid \texttt{(}\ \lambda\ \texttt{(}\ id\ \texttt{)}\ exp\ \texttt{)} \\ val \quad ::= \quad & \texttt{false} \mid ok \end{aligned} \tag{11}$$

Those expressions require no progress rules because they are values. Variable expressions require no progress rules because variables are always substituted away by applications. Only conditional and application need be defined. Conditionals naturally have two obvious rules, that I name E-IfOk and E-IfFalse (the "E" stands for "Evaluate"):

$$\text{E-IfOk:} \quad \texttt{(}\ \texttt{if}\ ok\ exp_{then}\ exp_{else}\ \texttt{)} \Rightarrow exp_{then} \tag{12}$$

$$\text{E-IfFalse:} \quad \texttt{(}\ \texttt{if}\ \texttt{false}\ exp_{then}\ exp_{else}\ \texttt{)} \Rightarrow exp_{else} \tag{13}$$

Had Scheme's semantics dictated that the test expression must be a boolean, the first rule would have replaced ok with `true`, which is likely what you first expected to see there. However, recall that Scheme treats any non-false value as true for the purpose of an `IF` expression.

A perhaps less obvious rule, E-IfProgress, is required as well to complete the semantics of `IF`. When the test expression for the conditional is not in the value domain, we need some way of making progress on reducing the expression.

$$\text{E-IfProgress:} \frac{exp_{test} \Rightarrow val_{test}}{\texttt{(}\ \texttt{if}\ exp_{test}\ exp_{then}\ exp_{else}\ \texttt{)} \Rightarrow \texttt{(}\ \texttt{if}\ val_{test}\ exp_{then}\ exp_{else}\ \texttt{)}} \tag{14}$$

Application requires a notation for expressing variable substitution. This is:

$$[id \rightarrowtail v]\, body \tag{15}$$

"Substitute $v$ for $id$ in $body$".

The *body* is the expression in which all instances of variable named *id* are to be replaced with the expression *v*. In an eager language, the *v* expression must be in the value domain. In a lazy language, *v* can be any expression. For semantic purposes the distinction between eager and lazy is irrelevant in a language without mutation and small step semantics are almost always restricted to languages without mutation.

Using this notation we can express application as a reduction. If we choose lazy evaluation, it is:

$$\text{E-App}_{\text{lazy}}: \quad \texttt{(}\ \texttt{(}\ \lambda\ \texttt{(}\ id\ \texttt{)}\ exp_{body}\ \texttt{)}\ \ exp_{arg}\ \texttt{)} \ \Rightarrow\ \left[id \rightarrowtail exp_{arg}\right] exp_{body} \tag{16}$$

plus a progress rule:

$$\text{E-AppProgress1}: \ \frac{exp_1 \ \Rightarrow\ val_1}{\texttt{(}\ exp_1\ exp_2\ \texttt{)} \ \Rightarrow\ \texttt{(}\ val_1\ exp_2\ \texttt{)}} \tag{17}$$

To specify eager evaluation, we simply require the argument to be a value:

$$\text{E-App}_{\text{eager}}: \quad \texttt{(}\ \texttt{(}\ \lambda\ \texttt{(}\ id\ \texttt{)}\ exp_{body}\ \texttt{)}\ \ val_{arg}\ \texttt{)} \ \Rightarrow\ \left[id \rightarrowtail val_{arg}\right] exp_{body} \tag{18}$$

and introduce another progress rule (we still require rule E-AppProgress1):

$$\text{E-AppProgress2}_{\text{eager}}: \ \frac{exp_2 \ \Rightarrow\ val_2}{\texttt{(}\ exp_1\ exp_2\ \texttt{)} \ \Rightarrow\ \texttt{(}\ exp_1\ val_2\ \texttt{)}} \tag{19}$$

#### 5.2.2 A Proof

Because each rule is a mathematical statement, we can prove that a complex expression reduces to a simple value by listing the rules that apply. That is, by giving a list of true statements that reduce the expression to a specific value. Operational semantics are used this way, but they are more often used as a rigorous way of specifying what an interpreter should do. The reason for exploring this proof structure is that we will later us the same structure on type judgements (another kind of rule set) to prove that an expression has some interesting property, rather than a specific value.

We list the statements in the order they are applied. When reaching a conditional we must prove its antecedants. To do this, we replace each antecedant with its own proof; that is, we start nesting the conditional statements until all are satisfied.

**Theorem 1.** `(if ((λ (x) x) true) false true)` *reduces to* `false`.

*Proof.*
$$\frac{\texttt{((}\lambda\ \texttt{(x) x) true)} \overset{\text{(E-App)}}{\Rightarrow} \left[\texttt{x} \rightarrowtail \texttt{true}\right] \texttt{x} \overset{\text{(subst.)}}{\Rightarrow} \texttt{true}}{\texttt{(if ((}\lambda\ \texttt{(x) x) true) false true)} \Rightarrow \texttt{false}} \ \text{(E-IfTrue)}$$

□

### 5.3 Big-step Example

#### 5.3.1 Rules

Under big-step operational semantics, the left side of the progress arrow is always an expression and the right side of the arrow is always a value. Thus, each rule takes a "big step" to the final value of an expression. The value and expression domains must be disjoint to make this distinction, so we define the value domain more carefully here. For our simple language from eq. 10, the big-step value domain is:

$$\begin{aligned} proc &::= \langle id, exp, \mathscr{E} \rangle \\ ok &::= proc \mid \hat{t} \\ val &::= ok \mid \hat{f} \end{aligned} \tag{20}$$

The angle brackets in the procedure notation $\langle id, exp, \mathscr{E} \rangle$ denote a **tuple**, in this case, a 3-tuple. That is simply a mathematical vector of values, or equivalently, a list in an interpreter implementation. The specific tuple we're defining here is a 1-argument closure value, which has the expected three parts: formal parameter identifier, body expression, and captured environment. Big step operational semantics use environments in the same way that interpreters do for representing deferred substitutions. The environment is placed on the left side of the progress arrow and separated by an expression by a comma.

Now that the value domain is disjoint from the expression domain, we need some trivial rules for reducing literal and `LAMBDA` expressions to their corresponding values.

Forget all of the previous small-step rules. We begin big step with:

$$\begin{aligned}
\text{E-True}: &\quad \texttt{true}, \mathscr{E} &&\Rightarrow \hat{t} \\
\text{E-False}: &\quad \texttt{false}, \mathscr{E} &&\Rightarrow \hat{f} \\
\text{E-Lambda}: &\quad \texttt{(}\ \lambda\ \texttt{(}\ id\ \texttt{)}\ exp\ \texttt{)}, \mathscr{E} &&\Rightarrow \langle id, exp, \mathscr{E} \rangle
\end{aligned} \tag{21}$$

Note that `LAMBDA` captures the identifier and environment, and saves and delays the expression, and that the literals ignore the environment in which they are reduced.

The rules for evaluating `IF` expressions are largely the same as for small step, but we can ignore the progress rules because they are implicit in the big steps:

$$\text{E-IfOk:} \quad \frac{\neg\left(exp_t, \mathscr{E} \Rightarrow \hat{f}\right) \qquad exp_c, \mathscr{E} \Rightarrow val_c}{\texttt{(}\ \texttt{if}\ exp_t\ exp_c\ exp_a\ \texttt{)}, \mathscr{E} \Rightarrow val_c}$$

$$\text{E-IfFalse:} \quad \frac{exp_t, \mathscr{E} \Rightarrow \hat{f} \qquad exp_a, \mathscr{E} \Rightarrow val_a}{\texttt{(}\ \texttt{if}\ exp_t\ exp_c\ exp_a\ \texttt{)}, \mathscr{E} \Rightarrow val_a} \tag{22}$$

Each reduction, whether a conditional or a simple $a \Rightarrow b$, is a mathematical **statement**. Statements are either true or false[4] in the mathematical sense. Each progress rule is really a step within a proof whose theorem is "this program reduces to (whatever the final value is)." The $\neg$ operator negates the truth value of a statement. Thus the E-IfOk rule has as an antecedent "the test expression does not reduce to $\hat{f}$". We need to express it this way because our desired semantics follow Scheme's, which allow any value other than $\hat{f}$ to act like "true" for the purpose of an `IF` expression.

To express the semantics of `APP` we need a notation for extending an environment with a new binding. Since environments and substitutions are not used simultaneously, the substitution notation is repurposed to denote extending an environment:

$$\mathscr{E}[id \leftarrowtail val] \tag{23}$$

"Environment $\mathscr{E}$ extended with *id* bound to *val*".

Note that the arrow inside the brackets points in the opposite direction as for substitution, following the convention of [Kri07, 223]. The application rule for our toy language under big step semantics is:

$$\text{E-App}: \quad \frac{exp_p, \mathscr{E}_1 \Rightarrow \langle id, exp_b, \mathscr{E}_b \rangle \qquad exp_a, \mathscr{E}_1 \Rightarrow val_a \qquad exp_b, \mathscr{E}_b[id \leftarrowtail val_a] \Rightarrow val_b}{\texttt{(}\ exp_p\ \ exp_a\ \texttt{)}, \mathscr{E}_1 \Rightarrow val_b} \tag{24}$$

This reads,

> "**if**(expression $exp_p$ reduces to a procedure in the current environment ($\mathscr{E}_1$),
>
> **and** argument expression $exp_a$ reduces to value in the current environment, and
>
> **and** the body of that procedure evaluated in the procedure's stored environment ($\mathscr{E}_b$) extended with *id* bound to the argument's value reduces to value $val_b$,)
>
> **then** the application of the procedure expression to the argument expression in the current environment is $val_b$."

Observe that the body expression is evaluated in the stored environment extended with the new binding, creating lexical scoping. Were we to accidentally use the current environment there we would have created dynamic scope. We conclude with the trivial variable rule:

$$\text{E-Var}: id, \mathscr{E}[id \leftarrowtail val] \Rightarrow val \tag{25}$$

[4] ...although the function to evaluate the truth value of a statement may be non-computable, as in the case of the Halting Problem.

Examining our big-step rules, we see that they map one-to-one to the implementation of an interpreter for the language. Each rule is one case inside the `EVAL` procedure, or inside the parser for ones that we choose to rewrite at parse time. Within a rule, each antecedant corresponds to one recursive call to `EVAL`. For example in the E-App rule, there are three antecedants. These correspond to the recursive calls to evaluate the first subexpression (which should evaluate to a procedure), the second (i.e., the argument), and then the body. That last call to evaluate the body is usually burried inside `APPLY`. The environments specified on the left sides of the antecedants tell us which environments to pass to `EVAL`.

See [Kri07] chapter 23 for an excellent set of examples of progress rules for big-step operational semantics.

### 5.3.2 A Proof

This is a proof of the same statement from the small-step example, now proven with the big-step semantics. Because the nesting gets too deep to fit the page width, I created a separate lemma for the conditional portion of the proof.

**Lemma 1.** `((λ (x) x) true)` , $\mathscr{E}$ *reduces to* $\hat{t}$

*Proof.* $$\frac{\texttt{(λ (x) x)},\mathscr{E} \overset{\text{(E-Lambda)}}{\Rightarrow} \langle \texttt{x}, \texttt{x}, \mathscr{E} \rangle \qquad \texttt{true},\mathscr{E} \overset{\text{(E-True)}}{\Rightarrow} \hat{t} \qquad \texttt{x},[\texttt{x} \longleftarrow \hat{t}]\mathscr{E} \overset{\text{(E-Var)}}{\Rightarrow} \hat{t}}{\texttt{((λ (x) x) true)},\mathscr{E} \Rightarrow \hat{t}}\text{(E-App)}$$ □

**Theorem 2.** `(if ((λ (x) x) true) false true)` , $\mathscr{E}$ *reduces to* $\hat{f}$ .

*Proof.*

1. Because `((λ (x) x) true)` $,\mathscr{E} \overset{\text{(Lemma 1)}}{\Rightarrow} \hat{t}$ , and $\hat{t} \neq \hat{f}$
   the negation of the contradiction of Lemma 1, $\neg\left(\texttt{((λ (x) x) true)},\mathscr{E} \Rightarrow \hat{f}\right)$, is also true.

2. $$\frac{\neg\left(\texttt{((λ (x) x) true)},\mathscr{E} \Rightarrow \hat{f}\right) \qquad \texttt{false},\mathscr{E} \Rightarrow \hat{f}}{\texttt{(if ((λ (x) x) true) false true)},\mathscr{E} \Rightarrow \hat{f}}\text{E-IfOk}$$

□

# 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[5] 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 [G$\ddot{3}$1][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[6] 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.

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

[6] ...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.

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 $g$:

$$S_g(x) = \text{“}x > 4\text{”} \tag{26}$$

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

$$S_i(n) = \text{“}S_n \text{ is not provable.”} \tag{27}$$

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 3.** *$H(P,x)$ 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[7] 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 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.[8]

## 7.1 Syntax

The λ calculus is a language with surprisingly few primitives in the expression domain[9]:

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

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:

$$val ::= \lambda \ id \ . \ exp$$

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)

$$\frac{exp_p \Rightarrow exp'_p}{exp_p exp_a \Rightarrow exp'_p exp_a} \tag{28}$$

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

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

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

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

$$\frac{exp_a \Rightarrow exp'_a}{exp_p exp_a \Rightarrow exp_p exp'_a} \tag{29}$$

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

$$\lambda \ id \ . \ exp_{body} \ \ val \ \Rightarrow \ [id \rightarrowtail val] exp_{body} \tag{30}$$

The App-Abs rule relies on the same syntax for the *val* value and *abs* expression, which is fine in $\lambda$ 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:

$$\frac{val_p = \lambda \ id \ . \ exp_{body}}{val_p \ val_a \Rightarrow \ [id \rightarrowtail val_a] exp_{body}} \tag{31}$$

## 7.3 Examples

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

$$\lambda \ x \ . \ (x+3) \ 7 \tag{32}$$

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:

$$\begin{aligned}
&\lambda \ x \ . \ (x+3) \ 7 && (33)\\
&\quad \Rightarrow \ [x \rightarrowtail 7](x+3) && (34)\\
&\quad = (7+3) && (35)\\
&\quad = 10 && (36)
\end{aligned}$$

Arithmetic then reduces this to 10.

Now consider using a procedure as an argument:

$$(\lambda f \ . \ (f \ 3)) \ (\lambda x.(1+x)) \tag{37}$$

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:

$$\begin{aligned}
&(\lambda f \ . \ (f \ 3)) \ (\lambda x.(1+x)) && (38)\\
&\quad \Rightarrow [ \ f \ \rightarrowtail \ (\lambda x.(1+x)) \ ](f \ 3)) && (39)\\
&\quad = \lambda x.(1+x) \ 3 && (40)\\
&\quad \Rightarrow [x \rightarrowtail 3](1+x) && (41)\\
&\quad = (1+3) && (42)\\
&\quad = 4 && (43)
\end{aligned}$$

## 7.4 Partial Function Evaluation

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

$$f \ x \ y = (f \ x) \ y \tag{44}$$

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

$$\lambda x.\lambda y.(x+y) \tag{45}$$

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:

$$\begin{aligned}
&(\lambda x.\lambda y.(x+y)) \ \ 1 \ \ 2 && (46)\\
&\quad \Rightarrow \lambda y.(1+y) \ \ 2 && (47)\\
&\quad \Rightarrow 1+2 && (48)\\
&\quad = 3 && (49)
\end{aligned}$$

The process of converting a 0th-order function of $n$ arguments into an $n$th-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[10].

When an $n$th-order function is applied to $k < n$ arguments, in $\lambda$ 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:

$$(\lambda x.\lambda y.(x+y))\ \ 5 \tag{50}$$

$$\Rightarrow \lambda y.(5+y) \tag{51}$$

## 7.5 Creating Recursion

A **fixed point** of a function $f$ is a value $v$ such that $f(v) = v$ (in mathematical notation; in $\lambda$ 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 $\lambda$ calculus. For example, define factorial (for convenience, we temporarily extend the language with conditionals and integers; those can be reduced as shown previously):

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

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 $f$:

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

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 $n-1$:

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

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:

$$(\lambda z.z\ z)\ g\ \Rightarrow\ g\ g \tag{52}$$

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

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

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

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:

$$\begin{aligned} Y \quad = \quad & \lambda f. \\ & ((\lambda z\,.\,z\,z) \\ & (\lambda x\,.\,f(\lambda y\,.\,x\,x\,y))) \end{aligned} \tag{53}$$

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)))))))

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

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

# 8 Macros

We've seen that a parser can reduce macro-expressive forms to other forms to minimize the number of cases that need to be handled in the compiler/interpreter. For example, a short-circuiting `OR` expression like the one in Scheme can be reduced within the parser by the small-step rule:

$$\texttt{(}\ \texttt{or}\ exp_1\ exp_2\ \texttt{)} \Rightarrow \texttt{(}\ \texttt{let}\ \texttt{(}\ \texttt{[}\ id\ exp_1\ \texttt{]}\ \texttt{)}\ \texttt{(}\ \texttt{if}\ id\ id\ exp_2\ \texttt{)}\ \texttt{)} \tag{54}$$

Languages with a **macro** system feature allow the programmer to add rules such as this to the parser. They are effectively plugin-modules for the parser. Macros are written in a separate language that is often similar to the base language, and they generally describe pattern matching behavior. They extend the syntax of the language in ways that cannot be achieved using procedures alone. For example, a short-circuiting `OR` cannot be written using only procedures, `IF`, and application in a language with eager evaluation of procedure arguments.

In the `OR` example, it is important that the identifier *id* does not appear as a free variable on the other expressions. If it did, the macro would accidentally capture that variable and change the meaning of $exp_1$ and $exp_2$. A **hygienic** macro system is one in which identifiers injected into code by the macro system cannot conflict with ones already present in expressions. One way to achieve this is to append the level of evaluation at which an identifier was inserted to the end of its name. Identifiers in the original source code are at level 0, those created by first-level macro expansion are at level 1, those created by macros emitted by the first-level macros are at level 2, and so on. Not all macro systems are hygienic. While Scheme's macro system is (since R5R6) and is generally considered both clean and powerful, the most frequently used macro system–that of C/C++ –is not. This does not mean that C macros are useless, just that extreme care must be taken when using them.

## 8.1 C Macros

The C macro system contains two kinds of statements: `#if` and `#define`. Without defining its semantics here, an example[11] illustrates how they are typically employed:

```
#include <stdio.h>

#if defined(_MSC_VER)
//      Windows
#       define BREAK   ::DebugBreak();
#elif defined(__i386__) && defined(__GNUC__)
//      gcc on some Intel processor
#       define BREAK __asm__ __volatile__ ("int $3");
#else
//      Hopefully, some other gcc
#       define BREAK ::abort()
#endif

#define ASSERT(test_expr, message) \
    if (! (test_expr)) {\
        printf("%s\nAssertion \"%s\" failed in %s at line %d.\n", \
        message, #test_expr, __FILE__, __LINE__);\
        BREAK;\
    }

int main(int argc, char** argv) {
    int x = 3;
    ASSERT(x > 4, "Something bad");
    return 0;
}
```

The `#if` statements are used to determine, based on expressions including variables such as `_MSC_VER` that are defined at compile time, what compiler and operating system the code is being compiled for. They allow the program to function differently on different machines without the expense of a run-time check. Furthermore, certain statements that are not legal on one compiler can be avoided entirely, such as the inline assembly syntax used for gcc. The `#define` statement creates a new macro. By convention, macros are given all uppercase names in C. Here, two macros are defined: `BREAK`, which halts execution of the program when the code it emits is invoked, and `ASSERT`, which conditionally halts execution if a test expression returns false. `ASSERT` cannot be a procedure for two reasons: first, it would be nice to define it so that in an optimized build the assertions are removed (not shown here), and second, because we do not want to evaluate the message expression if the test passes.

Within the body of the `ASSERT` definition we see several techniques that are typical of macro usage. The special variables `__FILE__` and `__LINE__` indicate the location at which the macro was invoked. Unlike procedures in most languages, macros have access to the source code context from which they were invoked. This allows them to customize error reporting behavior. The expression `#test_expr` is applying the `#` operator to the `test_expr` macro variable. This operator quotes the source code, converting it from code into a string that may then be printed. Procedures have no way of accessing the expressions that produced their arguments, let alone the source code for those expressions. Note that where it is used as the conditional for `if`, `test_expr` is wrapped in parentheses. This is necessary because C macros operate at a pre-parse (in fact, pre-tokenizer!) level, unlike Scheme macros. Without these extra parentheses, the application of the not operator (`!`) might be parsed differently depending on the operator precedence of other operators inside the expression. This is generally considered a poor design decision of the C language, not a feature, although it can be exploited in useful ways to create tokens at compile time.

[11] Adapted from the G3D source code, http://g3d-cpp.sf.net.

C's macro system is practical, though ugly. The C++ language addresses many of its shortcomings by introducing two other language features for creating new syntax: **templates** and **operator overloading**. Templates were originally introduced as a straightforward polymorphic type mechanism, but have since been exploited by programmers as a general metaprogramming mechanism that has Turing-equivalent computational power.

### 8.2 Scheme Macros

## 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.

Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. 04 2007.

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.