# CS 333 Programming Languages, Spring 2018

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

**Course Description**

This course is a survey of programming languages and paradigms. We will focus on the design of programming languages and compare and contrast different language families including imperative, object-oriented, functional, and logic paradigms. Topics include syntax, context-free grammars, parsing, semantics, abstract representations of programming processes and structures, memory management, and exceptions. Students will undertake small programming projects in various languages and more extensive projects in two languages of their choice. Students will present the characteristics of their chosen languages to their peers at the end of the term.

**Prerequisites:** CS 231 or permission of instructor.

**Desired Course Outcomes**

A. Students demonstrate an understanding of different language paradigms and implement algorithms in each paradigm.

B. Students demonstrate an ability to independently learn programming languages.

C. Students demonstrate an ability to describe the syntax, semantics and functionality of different languages in a common, rigorous manner.

D. Students demonstrate an understanding of the relationship between language and design.

E. Students work with partners to learn one or more languages and present them to the class.

F. Students present algorithms, languages, and their characteristics in an organized and competently written manner.

# 1 Programming Languages Overview

## 1.1 History of programming languages

The earliest computers did not have stored memory programs. The computer could be configured to run a single program by adjusting gears (Charles Babbage/Ada Lovelace) or connecting cables and flipping switches (ENIAC). Punch cards provided an input/output mechanism for later computers. A punch card is a card-stock piece of paper with holes punched in it. The card is placed into a card reader, and the location of the holes in a hard communicates a set of instructions or data to the computers. IBM began its involvement in computing by building punch-card reader devices.

In all three of these cases, the first programmers had to specify the actions of the computer using the actual binary instruction codes executed at the level of the hardware. A programmer had to know the binary sequence for each machine instruction or data value fed into the machine.

Assembly language was the first abstraction of machine code. It used text mnemonics to represent binary instructions and enable the first use of symbols to represent binary sequences (e.g. numbers). Punch-card readers became interpreters that converted assembly language instructions and data into the binary values required by the computers. Text files, editors, and an interactive command line had to wait until the development of the cathode-ray tube [CRT] monitor and keyboards.

The first programming languages were abstractions of assembly, but were still written on punch cards. Fortran, for example, was developed on punch cards. Each card represented a single line of Fortran code, and each position on the card represented one character in an 80 character line. The traditions of 80 character windows on a terminal date to the number of columns on a punch card. Because Fortran was originally written on punch cards, it incorporated language characteristics that were easy to encode using punch cards, such as special characters in the first few columns and rules like the actual code had to start in a particular column.

Programming languages were intended to bridge the gap between natural language and the strict requirements of machine instructions. Newer programming languages incorporate additional structures, organizational tools, and capabilities that enable different design methodologies and abstractions. Every programming language has a purpose that guides the design decisions behind it, but every language must, eventually, possibly through several layers of interpretation, translate to machine instructions that can run on physical hardware.

Early programming languages were developed by people looking to satisfy particular needs.

- Fortran: scientific computing
- Cobol: business computing
- Algol: general purpose programming
- Lisp: AI programming
- C: systems programming
- Prolog: natural language processing
- SEQUEL: database management

The early languages have influenced the development of later languages. Some languages have survived and continue to be heavily used. Others have evolved into or been replaced by other languages.

Developing programming languages also required the development of tools to enable and facilitate their use.

- A programming language must be describable in an unambiguous manner (grammar).
- Compilers and interpreters must enable the language to be converted into assembly/machine language.
- The underlying computer architecture must sufficiently support the features of the language.

In particular, compilers and interpreters cannot be separated from the languages themselves. Some languages have not survived because the compilers were too complex or too slow. Some computer architectures, such as the IA-64, have not survived in part because building an efficient compiler for them was too complex. Some computer architectures, such as Lisp machines, were designed to support a specific language (most of those were short-lived as well). The broad acceptance and commercial success of general purpose hardware and general purpose languages means that specialized languages or hardware will tend to exist only so long as their purpose is sufficiently necessary and it is sufficiently difficult to accomplish that purpose using a general purpose system.

## 1.2 Goals of programming languages

In addition to their intended applications, programming languages have a number of different design goals and constraints.

- The language should be easy to write
- The language should be efficient to write
- The language should be easy to read and understand
- The language should be portable across computers
- The language must be implementable on a computer (e.g. a Turing machine)
- The language should help minimize programming errors
- The language should fit its intended purpose and technical setting
- The language should be clear about the binding of values
- The language should produce a program that runs identically each time it is run, given the same inputs.
- The language should have good support
- The language should support abstraction of code and data
- The language should have orthogonal operations
- The language should permit efficient and fast execution
- The language should have a fast compile time/efficient interpreter

What other goals can you think of?

Clearly, not all of these goals are compatible. An efficient and fast language is more difficult to design if you allow run time type management and significant code and data abstraction. Likewise, a programming language that is efficient to write may not be as efficient to read and understand.

Discussion

- What programming languages have you used and on what tasks?
- For what purpose did you choose the language?
- What characteristics of the language were appropriate for the task?
- What characteristics of the language were inappropriate for the task?
- How would you describe the syntax of the language?
- Could you write a program in the language without a manual? Why/why not?

## 1.3 Examples of programming languages

There are many programming languages. In fact, most of the things we do on a computer can be defined as a programming language. Even the series of gestures we use to interact with a GUI follow a specific syntax, or ordering of events. Each of the events has a semantic meaning to it, and the end result is an unambiguous instruction, or sequence of instructions to the computer. Some operating systems even permit us to build macros out of gestural events, allowing us to build abstractions.

(An interesting project would be to develop a program that permits us to create variables in gestural interfaces.)

Production: used for building commercial software and large, enterprise scale systems.

- **C/C++**: efficient implementations, broad support, ADTs and classes, extensive libraries, fine control.
- **Java**: portable, broad support, extensive libraries, ADTs and classes
- **PHP**: broad support, extensive libraries
- **SQL**: natural declarative syntax
- COBOL: string and symbol manipulation, arbitrary precision mathematics
- C#: string and symbol manipulation, GUI development support
- **ADA**: strong typing, ADTs and classes, separation of interfaces and implementations
- Forth: designed for embedded processing

Science: used for programs that do lots of computations, often using parallel architectures.

- APL: vector processing language, complex syntax/symbology
- **Fortran** / High Performance Fortran: efficient implementations / explicitly parallel
- **C/C++**: efficient implementations, ADT support
- **Matlab**: efficient language, basic data type is a matrix, explicitly parallel
- Occam: explicitly parallel language
- **Python**: increasingly used in research and rapid prototyping situations

Artificial Intelligence [AI]: used for reasoning, search, or theorem proving.

- **Lisp**: functional programming language, basic data type is a list
- Scheme: functional programming language, basic data type is a list
- Haskell: functional programming language, basic data type is a list
- **Prolog**: Logic programming language, basic statement is an if-then rule

GUI Development

- **Tcl/Tk**: scripting language defining GUIs and input/output objects
- **Torquescript**: game design language that permits definition of GUI objects
- Visual Basic: often used for GUI development on Windows platforms

System: used for writing operating systems, compilers, device drivers, and other programs that work closely with hardware or file systems.

- **C/C++**: close to the machine, efficient, explicit control of memory
- **Perl**: powerful scripting language with built in support for regular expressions
- **Python**: scripting language used heavily in Mac OS X.
- lex: language for describing a lexical parser
- yacc: language for describing a compiler
- sh/bash/tcsh: shell scripting languages for automating tasks
- make: language for software project management
- cmake: abstraction of the make language

Provability: designed, in part, to enable programmers to prove the correctness of a program.

- **ADA**: strong typing
- ALGOL 68: first formally described programming language
- Java Modeling Language [JML]: behavioral specification language for Java
- Alloy: language designed to model program behavior and identify failure modes

Object-oriented: designed to make it easy to program in an OO paradigm

- Modula-2: one of the first OO languages
- Smalltalk: precursor for most OO language designs
- **Logo**: explicitly OO language (no assignment statement)
- **Java**: designed to be modular and OO
- **C++**: addition of OO design concepts to C
- **Python**: not a pure OO design, but incorporates OO structures
- OCaml: a general language focused on program safety and reliability

Teaching: designed to be easy to use and to read and understand.

- **Basic**: simple imperative programming language
- **Pascal**: imperative programming language, less constrained than Basic
- **Python**: imperative/OO programming language with flexible typing and classes
- **Logo**: simple OO programming language with built in support for turtle graphics
- Processing: language designed to manipulate digital media such as images, video, and music

Databases: designed to enable easy access and manipulation of databases

- **SQL**: declarative language for managing data
- **XML**: markup language that surrounds and describes data

Web: designed to build content for the web.

- **HTML**: markup language for documents
- **PHP**: C-like programming language for putting code in-line with HTML
- **Cascading Style Sheets [CSS]**: object-oriented style descriptions
- **Javascript**: compact scripting language for making pages active/reactive
- Ruby: scripting language that abstracts common web-design tasks
- **Perl**: extensive library support for building dynamic web pages
- **Python**: extensive library support for building dynamic web pages

Publication/Markup: designed to describe/create documents

- **Latex**: markup language
- **HTML**: markup language
- **Postscript**: generic language for describing documents
- **RTF**: markup language
- PDF: generic language for describing documents

Application-specific: designed to enable scripting within a specific program.

- **Torquescript**: flexible data typing, OO design support, event-based programming
- **Matlab**: matrix manipulation language
- IDL: matrix manipulation language
- Mathematica: symbolic manipulation
- **R**: declarative language for implementing statistical calculations
- **Excel**: support for data flow operations and basic calculations
- Visual Basic for Applications: enables fast development of extensions to existing applications
- **Python**: used to extend many programs, including Blender and Panda3D

Hardware description: designed for textual description of digital and analog circuits

- **VHDL**: strongly typed, Ada-like language for describing circuits and circuit functionality
- Verilog: C-like language for describing circuits and circuit functionality
- **SPICE**: language for designing and modeling analog circuits
- ALI: VLSI layout design language (silicon layout, 1983)
- Layla: a Pascal-based VLSI layout design language (silicon layout, 1985)
- Tangram: VLSI programming language designed by Philips Research Lab
- Many others...

## 1.4 Programming languages in context

Programming languages are used to describe the behavior of a computer at all levels of the hierarchy.

- Chip layout
- Analog circuit design
- Digital circuit design
- CPU modeling
- Hardware components
- Operating system
- User interface
- Compilers
- Applications
- Theory

Note that programming languages tend to get more abstract further from the hardware. At the level of the operating system, for example, it is important to know and keep track of all memory usage explicitly. The operating system must often make explicit decisions about memory usage, for example, and cannot abstract away the concept with a separate garbage collection system.

Applications, however, are not as interested in memory usage, per se, and simply need to have enough memory to do their task. While an application designer concerned about performance my want to explicitly manage memory, for small applications where minimizing development time is important a programming language that handles memory implicitly may be a better choice.

## 1.5 C / Project Overview

- C is a typed language: the type of a variable or expression determines how it is interpreted by the compiler. Casting gives the programmer control.
- Execution begins at the start of the main function, which takes two arguments: int argc, char *argv[].
- You manage your own memory: allocation, maintenance, deallocation.
- A pointer is an address, pointers and arrays are the same data type.
- C syntax was the basis for Java, so basic control flow and assignment are similar.
- The symbol & means “address of”.

## 1.6 PL Concepts

- Syntax - defines the structure of the language. Syntax is defined by a vocabulary that specifies the set of possible symbols and a grammar that defines the set of possible valid programs. Note that a valid program is not necessarily a correct or useful program.
- Names - programs require that we give names to entities in the program. Entities include constants, variables, expressions, functions, libraries, and programs. Names permit us to manipulate entities.
- Scope - given the importance of names, it is important to know when the name of an entity is available. Scope defines the part of the program in which a name refers to a specific entity.
- Visibility - sometimes we use the same name for different entities. The visibility rules define which entity a particular name usage accesses.
- Binding - the entity referred to by a name is not always defined when the programmer writes the code. Binding specifies when the connection between name and entity is made.
- Types - all data in a computer is a sequence of binary values. The data types of a programming language define the abstractions built on top of binary sequences to permit a programmer to generate and manipulate information. Data types can be simple, such as integers or characters, or more abstract such as lists, hash tables and functions.
- Semantics - the meaning of a program is defined by its semantics. Generally, semantics are defined in terms of the behavior of a particular architecture, or computing model that is independent of the actual underlying hardware (but not always).
- Organization - all programming languages contain constructs that permit us to build abstractions. In some languages this is easier than others. Macros, functions, classes, interfaces, and packages are all examples of organization constructs that can exist in a language.
- Memory Management - allocating, freeing, and making use of memory are central to writing programs. In some language memory management is hidden from the programmer, while in others the programmer is responsible for managing its use. The most important concepts in memory management are the system stack–which is generally used for handling local variables and function calls–and the heap–which is generally used for dynamically allocated objects. The system stack is rarely explicitly managed by the programmer, while the heap is often at least partly exposed.

# 2 Syntax

The *syntax* of a programming language is a precise description of all its grammatically correct programs.

Tucker and Noonan, *Programming Languages*, 2007

The syntax of a programming language is defined by a grammar. In general, programming languages are defined by *context-free grammars*. In a context-free grammar, the meaning of a symbol is independent of any surrounding symbols. Context-free grammars are one level of a hierarchy of grammars defined by Noam Chomsky. All grammars are defined by a set of symbols and rules. The symbols divide into two categories: terminal and non-terminal. The set of terminal symbols $T$ defines the alphabet of the language. The set of non-terminal symbols $N$ defines abstractions within the language such as *expression*, *name*, or *program*. The production rules define the relationships between terminal and non-terminal symbols, which defines the syntax of the language.

Chomsky defined four categories for grammars. In all of these grammars, the process of analyzing a string involves applying a single rule to a single location in serial order. There are other grammars that do not follow Chomsky's hierarchy, such as L-systems, because they apply one or more rules in parallel to all instances of a symbol in the string. However, L-systems are not suited for compilation of a program; they are intended as grammars to generate an output string. Chomsky's grammar categories are intended to support deconstruction of a string into non-terminal symbols to evaluate whether it is a correct program and enable deduction of its semantics (meaning).

- Regular - corresponds to a standard finite-state automata; its rules are all of the form $A \rightarrow \omega B$ or all of the form $A \rightarrow B\omega$, where $A$ and $B$ are single non-terminal symbols and $\omega$ is a valid string of terminal symbols. In both cases $B$ can be null if $\omega$ is not null. The ordering of $B$ and $\omega$ determine whether the language is a *right regular grammar* or a *left regular grammar*.
- Context-free - corresponds to a push-down automata; its rules are of the form $A \rightarrow \omega$, where $A$ is a single non-terminal symbol and $\omega$ is a valid string and can be mix of terminal and non-terminal symbols, including $A$. Context-free grammars allow recursive syntactic structuress.
- Context-sensitive - has rules of the form $\alpha \rightarrow \beta$, where both $\alpha$ and $\beta$ can be strings containing terminals and nonterminal symbols; the length of $\beta$ must be greater than or equal to $\alpha$ so that the program cannot shrink. A context-sensitive grammar is undecidable, in the sense that we cannot necessarily decide if a program is valid given the grammar.
- Unrestricted - identical to a context-sensitive grammar, but removes the length restriction on $\beta$.

Programming languages use context-free grammars because they are the most powerful grammars that are decidable. A regular grammar, for example, cannot represent the need for opening and closing brackets in an expression. An arbitrary context-sensitive grammar, on the other hand, is unlikely to be compilable in reasonable time and may not have an unambiguous interpretation.

Most programming languages are, in fact, part of a subset of context free grammars called LL(k) grammars, which means the grammar can be parsed in a single pass through the code by looking ahead $k$ symbols. Not all context free grammars are LL(k) grammars, but all LL(k) grammars can be converted into a parse tree in linear time in a single pass. An LL(k) grammar cannot contain left recursion (e.g. $A \rightarrow A + B$).

An LL(k) parser begins with the $start$, or $program$ symbol and builds the parse tree from the top down, ending with the leaves as rules that convert to the terminal symbols of the input string.

An alternative to LL(k) grammars are LR(k) grammars, a subset of context-free grammars that allow left recursion, but not right recursion. An LR(k) parser looks ahead $k$ symbols and starts with the leaves of the parse tree and builds up the tree from the bottom. Most production compilers, such as gcc, are recursive descent LL(1) compilers.

A context free grammar has the following properties.

- A set of terminal symbols $T$. These are the symbols in which a program is written, including keyword strings and the set of legal characters for creating symbols.
- A set of non-terminal symbols $N$. These represent abstract concepts in the programming language such as expressions or functions.
- A set of production rules $P$ that define the relationships between non-terminal and terminal symbols. As noted above, these all have the form $A \to \omega$, where $A$ is a single non-terminal symbol and $\omega$ is non-null and can be a mix of terminal and non-terminal symbols.
- A start symbol $S$.

As noted above, a production rule in a context free grammar has the form given in (1). $A$ is a single symbol from the set $N$. $\omega$ is a sequence of symbols from $(T \cup N)$, and $|\omega| \geq 1$.

$$A \to \omega \tag{1}$$

Grammars define the lexical processing step of compilation or interpretation and allow the computer (or programmer) to determine if the program has the potential to be converted into an executable program. A syntactically correct program cannot necessarily be converted into an executable program.

## 2.1 Backus-Naur Form

Most programming languages are formally described using variations of a notation called Backus-Naur Form [BNF]. A BNF grammar is a context-free grammar. The terminal and non-terminal symbols are disjoint. Terminal symbols are generally limited to those that can be entered on a keyboard. As noted above, non-terminal symbols represent organizational concepts in a programming language. The start symbol for a programming language is the highest level concept in the language, such as *Program*. Some languages permit libraries or packages to sit at the top level.

As an example, consider the definition of a floating point number (excluding scientific notation for now). We could try to define all the possible ways using different rules.

$$\begin{aligned} &Float \to 0 \\ &Float \to 1 \\ &\ldots \\ &Float \to 9 \\ &Float \to 0.0 \\ &Float \to 1.0 \\ &\ldots \\ &Float \to 9.0 \end{aligned} \tag{2}$$

As should be obvious, we very quickly become overwhelmed by the number of possible numbers. Instead, we can may want to define some simpler concepts and build the $Float$ concept out of those. We could start by defining the idea of a digit as the set of numbers from 0 to 9. BNF form allows us to use a vertical bar to separate different options. For compactness we can also use dots to represent a series of values instead of explicitly writing them.

$$\begin{array}{l} Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ Digit \rightarrow 0 \mid \cdots \mid 9 \end{array} \tag{3}$$

Now we can define a sequence of digits as an $Integer$, using a recursive expression to permit arbitrary length number sequences. A variation of BNF, called Extended BNF, uses curly brackets to stand for zero or more instances of the object inside the brackets. Therefore, the following two expressions are equivalent in EBNF.

$$\begin{array}{l} Integer \rightarrow Digit\ Integer \mid Digit \\ Integer \rightarrow Digit\ \boldsymbol{\{}Digit\boldsymbol{\}} \end{array} \tag{4}$$

The other pieces of EBNF are parentheses $\boldsymbol{(}A \mid B\boldsymbol{)}$, which indicate one of the options in parentheses must be chosen, and brackets $\boldsymbol{[}$ else $Statement$ $\boldsymbol{]}$, which indicate an optional sequence of symbols. Note that EBNF brackets are labeled in bold and are not literal symbols in the grammar.

Building on the $Integer$ type, we can now define a floating point value as two $Integer$ strings separated by a dot. Note that this definition requires the programmer to use a dot and at least one digit on either side of it when defining a floating point value. From a compiler's point of view, this is a good thing, as the number 6 could be either an $Integer$ or a $Float$ without that requirement.

$$Float \rightarrow Integer\ .\ Integer \tag{5}$$

We could relax the requirement somewhat by only requiring an $Integer$ on one side of the dot.

$$Float \rightarrow Integer\ .\ Integer \mid Integer\ . \mid\ .Integer \tag{6}$$

Without the dot, however, we would have an ambiguous grammar.

## 2.2 Derivation

How do we determine if a string is valid according to a grammar? We must be able to derive the string from the start symbol using a series of replacements defined by the production rules. The production rules iteratively replace non-terminal symbols with non-terminals and terminals until there are no more non-terminal symbols in the string and non-terminal symbols match the input string. The language consists of all the possible strings defined by the grammar.

As an example, consider the floating point value 42.24.

$$\begin{aligned} Float &\Rightarrow Integer\ .\ Integer \\ &\Rightarrow Digit\ Integer\ .\ Integer \\ &\Rightarrow Digit\ Digit\ .\ Integer \\ &\Rightarrow Digit\ Digit\ .\ Integer\ Digit \\ &\Rightarrow Digit\ Digit\ .\ Digit\ Digit \\ &\Rightarrow 4\ Digit\ .\ Digit\ Digit \\ &\Rightarrow 4\ 2\ .\ Digit\ Digit \\ &\Rightarrow 4\ 2\ .\ 2\ Digit \\ &\Rightarrow 4\ 2\ .\ 2\ 4 \end{aligned} \tag{7}$$

Note that there are many different orderings in production rules from the $Float$ non-terminal to the final result. All of them will give the same result, however. We can represent the same derivation as a parse tree, which shows the order of operations graphically.

The above derivation is representative of a recursive descent (LL(k)) parser that starts with the top of the parse tree. The goal is to replace all terminal symbols with non-terminal symbols and then continue to replace non-terminal symbols until the only remaining symbol is the start symbol. An LR(k) parser begins with the leftmost leaf of the parse tree and builds up the tree from there using left-recursion, as below.

$$\begin{aligned} 4\ 2\ .\ 2\ 4 &\Rightarrow Digit\ 2\ .\ 2\ 4 \\ &\Rightarrow Digit\ Digit\ .\ 2\ 4 \\ &\Rightarrow Integer\ Digit\ .\ 2\ 4 \\ &\Rightarrow Integer\ .\ 2\ 4 \\ &\Rightarrow Integer\ .\ Digit\ 4 \\ &\Rightarrow Integer\ .\ Digit\ Digit \\ &\Rightarrow Integer\ .\ Integer\ Digit \\ &\Rightarrow Integer\ .\ Integer \\ &\Rightarrow Float \end{aligned} \tag{8}$$

In both top-down and bottom-up derivations, the parser has to be smart about selecting the appropriate production rule. In this case, the lexical analyzer can pursue a greedy strategy whereby it searches for the longest string representable by a single non-terminal before attempting to satisfy the terminal's parent. In the case above, the lexical analyzer cannot convert the first Integer and dot into a Float before converting the decimal to an Integer because at that point the Float terminal, which is the start symbol, does not represent the entire string.

The order of non-terminals on the right side is important to generating parse trees. For example consider the simple grammar below for addition and subtraction on integers.

$$\begin{aligned} &Expr \rightarrow Expr + Term \mid Expr - Term \mid Term \\ &Term \rightarrow Integer\ Digit \mid Digit \\ &Integer \rightarrow Digit\ \{Digit\} \\ &Digit \rightarrow 0 \mid \cdots \mid 9 \end{aligned} \tag{9}$$

For the expression $A - B + C$, the $+$ is placed at the top of the parse tree because the rules are written as $Expr + Term$, and $Term$ must be a single number. This results in left to right application of the operators and an interpretation of the expressions as $(A - B) + C$. If we reverse the $Expr$ and $Term$ ordering in the first production rule, as in (10), we get a different interpretation: $A - (B + C)$.

$$Expr \rightarrow Term + Expr \mid Term - Expr \mid Term \tag{10}$$

Maintaining the order of operations is essential in programming languages. In order to specify hierarchies of operations, grammars can become extremely complex. For a right-recursive language to maintain order of operation, for example, the grammar must use a different style of rules and use a hierarchy of non-terminals to avoid left recursive rules.

### 2.3 Flexible/Ambiguous Grammars

Sometimes we might want to use an ambiguous grammar in order to simplify the number of rules required. Note that the following grammar makes it easy to add new operators.

$$\begin{aligned}
&Expr \rightarrow Expr\ Op\ Expr \mid (Expr) \mid Integer \\
&Op \rightarrow + \mid - \mid * \mid / \\
&Integer \rightarrow Digit\ \{Digit\} \\
&Digit \rightarrow 0 \mid \cdots \mid 9
\end{aligned} \tag{11}$$

Having an $Expr$ symbol in front of each operator, however, means we don't have to build a parse tree in a particular order. We can expand the left or right side first. For the expression $A - B + C$, this means we can build either of the possible parse trees using the same grammar. Ambiguities in grammars are generally resolved using additional rules. For example, if we have a table of precedence and a default left-to-right ordering of operators of equal precedence, then we can resolve any ambiguities that arise.

Another common ambiguity in language syntax is the dangling else. When an if statement is contained inside an if statement, which if statement does a subsequent else belong to?

$$\begin{aligned}
&ifStatement \rightarrow \text{if } (Expression)\ Statement \mid \text{if } (Expression)\ Statement \text{ else } Statement \\
&Statement \rightarrow Assignment \mid ifStatement \mid Block \\
&Block \rightarrow \{\ Statement\ \{Statement\}\ \}
\end{aligned} \tag{12}$$

Consider the following code snippet.

```
if( x < 0 )
    if( y < 0 )
        y = y + 1;
    else
        y = 0;
```

The second else could match with either if condition. Only by inserting brackets could the interpretation be unambiguous. In practice, such as in C, the ambiguity is resolved by the arbitrary rule, included as part of the description of the language, that an else clause is associated with the textually nearest if statement in any ambiguous case.

In Java, the ambiguity is resolved by not permitting an if statement without an else clause as the single statement after an if. The following code snippet, for example, will not do what the tabbing implies.

```
a = -1;
b = 1;

if( a < 0 )
    if( b < 0 )
        System.out.println( "here" );
else
    System.out.println("there");
```

If `a` is non-negative, the snippet will print nothing, because both the if and the else statement are inside the first if condition. If `a` is less than zero, the snippet will print out `there` because the else is forced to be part of the inner if statement.

### 2.4 Clite Grammar (from Tucker and Noonan, 2007)

$$
\begin{aligned}
Program &\rightarrow \text{int main ( ) } \{ \; Declarations \; Statement \; \} \\
Declarations &\rightarrow \boldsymbol{\{} Declaration \boldsymbol{\}} \\
Declaration &\rightarrow Type \; Identifier \; [\, [\; Integer \;]\,] \; \boldsymbol{\{} \; , \; Identifier \; [\, [\; Integer \;]\,] \; \boldsymbol{\}} \\
Type &\rightarrow \text{int} \mid \text{bool} \mid \text{float} \mid \text{char} \\
Statements &\rightarrow \boldsymbol{\{} \; Statement \; \boldsymbol{\}} \\
Statement &\rightarrow \; ; \mid Block \mid Assignment \mid IfStatement \mid WhileStatement \\
Block &\rightarrow \{ \; Statements \; \} \\
Assignment &\rightarrow Identifier \; [\; [\; Expression \;]\; ] \; = \; Expression; \\
IfStatement &\rightarrow \text{if} \; ( \; Expression \; ) \; Statement \; [\, \text{else} \; Statement \,] \\
WhileStatement &\rightarrow \text{while} \; ( \; Expression \; ) \; Statement \\
\\
Expression &\rightarrow Conjunction \; \boldsymbol{\{} \; || \; Conjunction \; \boldsymbol{\}} \\
Conjunction &\rightarrow Equality \; \boldsymbol{\{} \; \&\& \; Equality \; \boldsymbol{\}} \\
Equality &\rightarrow Relation \; [\; EquOp \; Relation \;] \\
EquOp &\rightarrow \; == \mid \; != \\
Relation &\rightarrow Addition \; [\; RelOp \; Addition \;] \\
RelOp &\rightarrow < \mid < = \mid > \mid > = \\
Addition &\rightarrow Term \; \boldsymbol{\{} \; AddOp \; Term \; \boldsymbol{\}} \\
AddOp &\rightarrow + \mid - \\
Term &\rightarrow Factor \; \boldsymbol{\{} \; MulOp \; Factor \; \boldsymbol{\}} \\
MulOp &\rightarrow * \mid / \mid \% \\
Factor &\rightarrow [\; UnaryOp \;] \; Primary \\
UnaryOp &\rightarrow - \mid \; ! \\
Primary &\rightarrow Identifier \; [\, [\; Expression \;]\,] \mid Literal \mid \\
&\quad ( \; Expression \; ) \mid Type \; ( \; Expression \; ) \\
\\
Identifier &\rightarrow Letter \; \boldsymbol{\{} \; Letter \mid Digit \; \boldsymbol{\}} \\
Letter &\rightarrow \text{a-zA-Z} \\
Digit &\rightarrow \text{0-9} \\
Literal &\rightarrow Integer \mid Boolean \mid Float \mid Char \\
Integer &\rightarrow Digit \; \boldsymbol{\{} \; Digit \; \boldsymbol{\}} \\
Boolean &\rightarrow \text{true} \mid \text{false} \\
Float &\rightarrow Integer \; . \; Integer \\
Char &\rightarrow \text{'} \; ASCIIChar \; \text{'}
\end{aligned}
\tag{13}
$$

There are two additional items required to completely specify the grammar of the language. First, the if/else ambiguity is resolved by attaching the else to the nearest prior if statement. Second, the set ASCIIChar is the set of printable ASCII characters less than 128.

## 2.5 Lexical Syntax

While the theoretical definitions are all nice and neat, we have to write programs to convert messy text files into some kind of abstract symbolic representation. For a non-optimizing compiler, lexical analysis–parsing the file–can take up to 75% of the time of compilation, so it is not a trivial part of the task.

Because it is such an important component, most compilers separate tokenization, or lexical analysis, from syntactic analysis and program generation. Tokenization is simply conversion from a string of characters–or whatever input format is being used–to a sequential string of symbols. Lexical analysis does not do syntax checking, but can identify improperly defined identifiers, for example.

Tokens include the following.

- Identifiers - variable names, function names, labels
- Literals - numbers (e.g. Integers and Floats), characters, true and false
- Keywords - `bool char else false float if int main true while`
- Operators - for example, `+ - / * && || = ==`
- Punctuation - for example, `; .` { } `( )`

In other words, lexical analysis handles at least part of all of the rules that have a terminal on the right side. In the case of something like an if statement, it converts the string `if` into a symbol that represents the keyword.

Because tokenization is such a common process, there are some nice tools for generating lexical analyzers automatically based on a description of the token grammar. Examples include lex and flex, both of which are freely available. These tools permit you to write the lexical syntax components of a language as a set of rules, generally based on regular expressions.

### 2.5.1 Regular Expressions

Regular expressions are a language on their own designed to compactly represent a set of strings as a single expression. Their syntax is similar to the EBNF conventions (not surprisingly, perhaps).

Special characters in regular expressions.

- [ ] - used to specify a set of alternatives
- \- used as an escape character to permit use of the other special characters
- ˆ - negates an expression when inside brackets, permits you to specify strings that don't include a certain expression, is the start of the string otherwise
- $ - the end of the string
- . - matches almost any character except line breaks
- — - specifies an or between expressions
- ? - makes an expression lazy (first possible completion) instead of greedy (largest possible completion)
- * - tells the engine to match the prior expression zero or more times

- + - concatenation (and)
- ( ) - groups tokens
- { } - can be used to specify ranges for repetition.

### 2.5.2 Flex

Flex makes use of regular expressions to define lexical tokens. A lexical parser is defined by a set of rules. Each rule is a regular expression followed by C code that expresses an action (including doing nothing) when flex finds a string matching the regular expression. The rules are tested in the order in which they appear in the flex file, which allows you to specify priority. Text that does not match any rule is passed along to the output.

A flex file has three parts: definitions, rules, and user code. They are separated by the expression %%.

- Definitions permit you do define macros and variables to be used in the rule section. You can also include C code to define additional variables used in the rule implementations. For example, the following two lines define a DIGIT as a character in the range [0-9], and an ID as a series of alphanumeric characters, starting with a letter.

  ```
  DIGIT    [0-9]
  ID    [a-z][a-z0-9]
  ```

- The rules are each defined by a regular expression and C code. The regular expression must not be indented. The code for a rule has to start on the same line as the regular expression. Multi-line code needs to be inside a block ( {...} ), with at least the opening curly-bracket on the same line as the rule. The following two lines look for digit sequences in a text file, discarding the rest.

  ```
  [0-9]+    printf("number: %10d\n", atoi(yytext) );

  .        /* skip all other input */
  ```

  If we defined `DIGIT` in the definition section, the rule `[0-9]+` could be replaced with `{DIGIT}+` . The variable yytext is a special variable available to the C code. It contains the text corresponding to the current token: the text matched by the regular expression.

- The user code is appended to the end of the C file generated by flex. If you put nothing there, then you need to write your own main function, link it with the flex output file (lex.yy.c, by default), and call the function `yylex()` inside your code. Alternatively, you can put the main function in the user code section of the flex file. For example, the following reads in from either stdin or a file specified on the command line. Note the special variable yyin, which is where yylex reads its input, and the call to yylex() to process the input.

  ```
  int main(int argc, char *argv[]) {
    if( argc > 1 )
      yyin = fopen( argv[1], "r" );
    else
      yyin = stdin;

    yylex();
  }
  ```

Taken together, the above three sections would read in from either stdin or a specified file and print out a list of all the integers in the file.

## 2.6 Stages of Compilation

Lexical analysis provides the compiler with abstract tokens rather than character strings. These are then passed on to the next stage in compilation. Overall, compilation has five stages.

- Lexical analysis - takes in a text file and produces a sequence of tokens. Lexical analysis does not generally catch errors beyond illegal characters. The output of the lexical analysis is a set of tokens that correspond to non-terminals in the BNF grammar. The lexical analysis will also associate actual values with the tokens, when appropriate.
- Syntactic analysis - parses the sequence of tokens and generates a parse tree or abstract syntax. The syntactic analyzer can catch syntax errors and ill-formed expressions. The output of the syntactic analysis is a parse tree that represents the abstract syntax.
- Semantic analysis - uses the parse tree or abstract syntax to generate intermediate code. The semantic analyzer can catch semantic errors like undefined variables, variable type conflicts, and implied conversions. Intermediate code is a more detailed, explicit parse tree where operators will generally be specific to the data they are processing.
- Code optimization - analyzes the intermediate code generated by semantic analysis to identify optimizations that speed up code execution without changing the program functionality.
- Code generation - converts the intermediate code into machine code. The code generation is tailored to a specific machine, while the intermediate code is general across platforms.

## 2.7 Abstract Syntax

The grammar of a language often contains many non-terminal symbols that are used to describe precedence of operators. Their use results in unique parse trees, which is essential for making a language unambiguous. However, it also results in large parse trees with long chains of non-terminals that lead to a single terminal. **Abstract syntax** is a simplified version of a parse tree that removes non-terminals that do not contribute to the semantic meaning of the program.

We can write the abstract syntax for a language slightly differently from the EBNF syntactic description because the abstract syntax does not care about how things are written, only about describing the essential relationships between components. We can think of it as a functional, or even data-flow representation of the program that can be represented independently from the text that generated it.

The concrete grammar of Clite, for example, uses many non-terminals to represent an $Expression.$ At its essence, an $Assignment$ makes use of an $Expression$ to represent moving data from one location to another. The abstract syntax for an $Assignment$ explicitly defines the type that needs to be on the left-hand side and the type that needs to be on the right-hand side of an assignment operator. Likewise, all of the operator terms are combined together into the $Binary$ and $Unary$ categories based on the form of their functionality rather than precedence. Precedence is handled by the concrete syntax, and once the unambiguous parse tree exists, we can reduce it to simpler form by replacing it with the abstract syntax.

Concrete grammar for $Assignment$ and $Expression$ (Tucker and Noonan, 2007).

$$Assignment \rightarrow Identifier\ [\ [\,Expression\,]\ ]\ =\ Expression;$$

$$\begin{aligned}
Expression &\rightarrow Conjunction \,\{\ ||\ Conjunction\,\} \\
Conjunction &\rightarrow Equality \,\{\ \&\&\ Equality\,\} \\
Equality &\rightarrow Relation\,[\,EquOp\ Relation\,] \\
EquOp &\rightarrow \text{==} \mid \text{!=} \\
Relation &\rightarrow Addition\,[\,RelOp\ Addition\,] \\
RelOp &\rightarrow < \mid < = \mid > \mid > = \\
Addition &\rightarrow Term\,\{\ AddOp\ Term\,\} \\
AddOp &\rightarrow + \mid - \\
Term &\rightarrow Factor\,\{\ MulOp\ Factor\,\} \\
MulOp &\rightarrow * \mid / \mid \% \\
Factor &\rightarrow [\,UnaryOp\,]\ Primary \\
UnaryOp &\rightarrow - \mid\ ! \\
Primary &\rightarrow Identifier\,[\,[\,Expression\,]\,] \mid Literal \mid \\
&\quad (\ Expression\ ) \mid Type\ (\ Expression\ )
\end{aligned} \tag{14}$$

Abstract grammar for $Conditional$, $Assignment$ and $Expression$ (Tucker and Noonan, 2007).

$$\begin{aligned}
&ConditionalExpression\ \text{test};\ Statement\ \text{thenbranch},\ \text{elsebranch} \\
Assignment &= Variable\ \text{target};\ Expression\ \text{source} \\
Expression &= Variable \mid Value \mid Binary \mid Unary \\
Binary &= Operator\ \text{op};\ Expression\ \text{term1},\ \text{term1} \\
Unary &= Operator\ \text{op};\ Expression\ \text{term} \\
Variable &= String\ \text{id} \\
Value &= IntValue \mid BoolValue \mid FloatValue \mid CharValue \\
IntValue &= Integer\ \text{value} \\
BoolValue &= Boolean\ \text{value} \\
FloatValue &= Float\ \text{value} \\
CharValue &= Char\ \text{value} \\
Operator &= + \mid - \mid * \mid / \mid ! \mid \text{==} \mid \text{!=} \mid < \mid > \mid < = \mid > =
\end{aligned} \tag{15}$$

The abstract grammar has two properties that make it a more useful representation for analysis than the concrete grammar.

1. The non-terminals in the grammar are more directly connected to their semantic meaning than in a concrete grammar, with differences based primarily on function rather than precedence or spatial relationship.
2. The organization of the grammar explicitly links inputs and outputs of functional operations, discarding syntactic formalities.

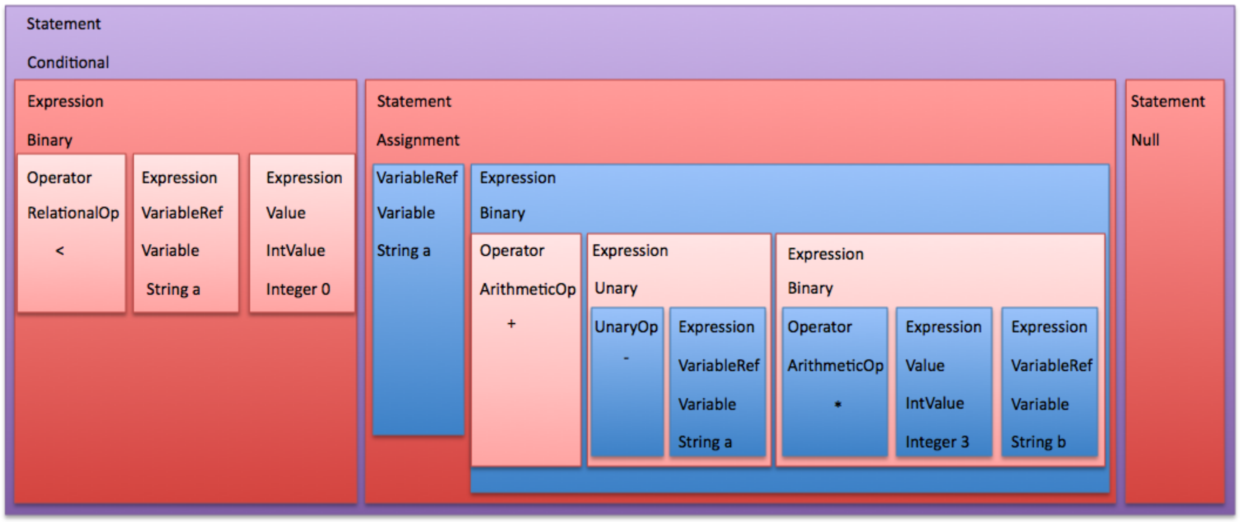
The first property makes it possible to start linking code to the non-terminals, as their definition is functional rather than syntactic. The second property makes it easy to internally represent the functionality, as we can represent each non-terminal by a structure that holds a reference to its inputs and output(s) and operator.

When drawing an abstract syntax tree, we can use boxes to represent each node, with a spot for each input, output, and operator.

**Example**

Consider the if statement below. The abstract syntax shows the fields of the various structures, such as the assignment, binary, and unary elements.

```
if( a < 0 ) {
  a = -a + 3*b;
}
```



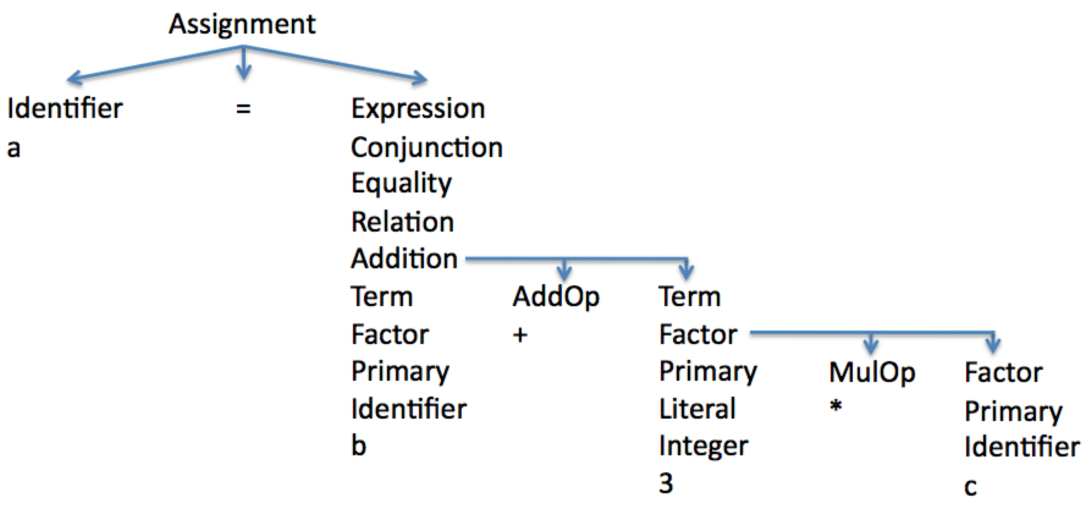


Figure 1: Concrete parse tree for the expression $a = b + 3 * c;$.

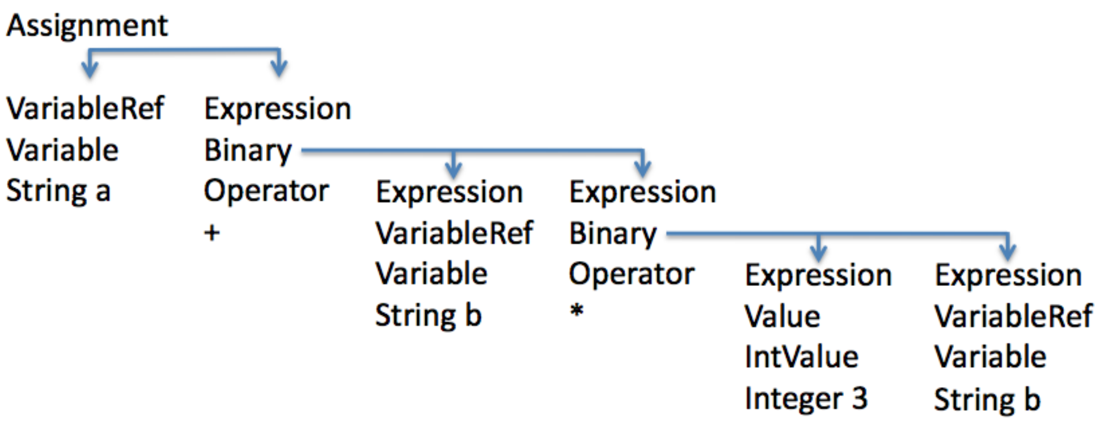


Figure 2: Abstract parse tree for the expression $a = b + 3 * c;$.

### 2.7.1 Converting a parse tree to abstract syntax

A three step process defines the transformation of a concrete parse tree to an abstract syntax tree.

1. Discard all separator or terminating symbols or tokens.
2. Discard all nonterminals that are trivial roots. A trivial root is a symbol with only one subtree.
3. Replace all remaining nonterminals with the operators which are a leaf or one of their immediate subtrees.

Consider the statement $a = b + 3 * c$. The parse tree for this expression is given in figure 1

This parse tree does not contain any separator symbols. There are many nonterminals, however, that have a trivial root. We can, for example, collapse All the terms from Expression down to Addition and replace it with a single Expression nonterminal. In the left subtree of the Addition term, we can replace from Term through Primary with Expression. In the right subtree we can replace Term and Factor with Expression and then replace the top two items of its subtrees with Expression. The result is given in figure 2. Replacing the term Assignment with = and the term Binary with the specific operator (+ or *) generates a slightly different version of the tree that is functionally identical.

The abstract parse tree has the following characteristics.

- There are no keywords or terminator symbols.
- There is no explicit indication of operator precedence. Precedence is completely defined by the shape of the tree.
- The tree explicitly represents the internal structure of each expression.

The lack of keywords and precedence means an abstract syntax representation of an expression should be common across any programming languages that has the ability to implement the operation. In some languages, the internal structure of the operators is potentially different. The meaning of the expression in figure 2 should be general across languages. The computer should multiply the contents of a variable with id $c$ by the integer 3, add the result to the contents of a variable with id $b$ and assign the result to a variable with id $a$. At its essence, the expression does three of the four things a computer can do: store data, move data, and manipulate data. (What's the fourth?)

The transformation between a concrete parse tree and an abstract parse tree is not trivial, but also not complex. It is straightforward to automate and represents the second stage in compilation. Once complete, the abstract parse tree becomes the basis for code generation. The abstract parse tree is not completely language independent, however, because different languages have different semantic rules. For example, in the expression above, the variables a, b and c could all be integers, which in most languages is the vanilla case. However, in Python, all three variables could be strings and it would be a perfectly valid expression. Furthermore, in Python, only b and c need to be defined prior to the expression, while in C or Java there must be some kind of prior declaration and typing of the three variables.

# 3 Naming

Naming is a critical ability of programming languages. If we can name an object, we can define rules for manipulating it. A name implies a binding, which is the connection between a definable entity and a symbol.

- A binding is *static* if it takes place before run time.
- A binding is *dynamic* if it takes place during run time.

Identifiers play a key role in language syntax. Identifiers can be case-sensitive, case-insensitive, bounded in length, or even predefined as keywords.

- Case-insensitive: Pascal, Fortran, Ada, Torquescript
- Case-sensitive: C, Python, Java,
- Mixed case sensitivity: PHP
- Bounded length: Fortran (6 characters through F77, 31 in F90)
- Special characters: C, Python, Java (underscore), Cobol (dash)

Most languages do not allow keywords to be used as identifiers. One reason for this rule is that it would lead to non-deterministic parsing. The string "if", for example, must convert only to a token with the semantic meaning "beginning of an if statement" in order to permit single-lookahead parsing.

A number of languages do have predefined identifiers, which may have different attributes than keywords. In some languages, predefined identifiers are reserved and cannot be redefined by the user (Cobol). In other languages (Pascal) they can be redefined by the user. In Pascal, for example, it is possible to redefine the reserved identifiers *true* and *false* as well as the base data types like *integer*. In Python, the statement `True = False` is perfectly legal (thought probably not recommended). In C/C++, it is not possible to redefine these boolean constants because they are defined as constants, which are immutable objects.

In some cases, an identifier may indicate more than a binding; it may actually specify other information about the entity to which the name is bound. For example, in Fortran, all variable names which begin with the letters A-H and O-Z represent, by default, single precision real numbers. Variable names that begin with I, J, K, L, M, and N represent, by default, integer data types. A programmer can override these default type bindings, but they are standard convention in Fortran code. The prevalence of the variable `i` as a loop variable is precisely because it is the first default integer variable in Fortran.

## 3.1 Variables

A variable represents the binding of an identifier to a memory address. Variables also have attributes such as their type, value, lifetime, and scope. At the very least, the binding of a symbol and an address requires four pieces of information.

- Identifier string
- Address (implementation specific)
- Type
- Value

Note that, even in a language without explicit variable types, the compiler or interpreter must, internally, handle typing. Otherwise, it does not know how to implement the basic operators on the data stored at the address. The data is just bits, and it is the type of manipulation applied to the bits that determines the semantic meaning (if any) of the bits.

Since variables provide both a memory address and a value, a language has to be clear about which to use in an expression. If a variable's identifier is placed on the left hand side of an expression, it must provide a memory reference. It doesn't make sense to assign a value to a value. If a variable's identifier is on the right hand side of an expression, it must provide a value. In some cases the value is, in fact, a memory address, depending upon the language.

Since Algol68, the convention has been to call the memory address meaning of a variable a *l-value*, and the value meaning of a variable as a *r-value*. Most languages do not require (or allow) explicit dereferencing of variables, although some languages (C) provide a suite of tools for doing so.

**Examples**

```
int x;
int y;
int *z;

// l-value and r-value meanings of x
x = x + 1;

// l-value meaning of y, r-value meaning of x
y = x + 1;

// l-value meaning of z, l-value meaning of x converted to an r-value
z = &x;

// l-value meaning of x, dereferenced r-value meaning of z
x = *z + 1;

// dereferenced l-value meaning of z, r-value meaning of x
*z = y;

// l-value meaning of z, l-value meaning of y converted to an r-value
z = &y + 4;

// l-value meaning of x, conversion of l-value meaning of z to an r-value
x = (int)z;
```

Some languages do not permit these kinds of explicit value manipulations of variables. However, all languages must be explicit in their internal semantic representation of the code's meaning to identify which meaning of an identifier should be used in a particular statement.

Python is an example of a language without explicit modifiers for a variable's interpretation. One reason is that it is possible to use a semantic model for Python that states that all variables represent a reference. In other words, all variables in Python hold memory addresses. Even a statement such as the assignment

```
a = 5
```

is possible to represent as `a` receiving a reference to a memory location holding the integer value 5. Other variables may also receive references to the same integer object. Because integer objects are immutable, it is not possible to change the value of the integer five, which eliminates the possibility of aliasing (changing the value of one variable by modifying a different variable). In truth, since integer objects are immutable, using a semantic model that says that the integer value 5 is copied into the memory location referenced by `a` produces the same results on a computer.

Java has an even more complex semantic model for variables, treating some types of variables to be memory locations while treating other types as references. Consider the example above, `a = 5;`. In Python, the meaning is always the same: the storage location referenced by `a` is assigned a reference to the value on the right. In Java, however, the semantic meaning of the statement is dependent upon the type of `a`.

If `a` is an `int`, for example, the meaning of `a` is that it is a memory location big enough to hold an integer, and the integer value 5 is written to that location. If `a` is an `Integer`, however, then the meaning of `a` is that it is a memory location big enough to hold an address, and it is assigned the address of an Integer object with the value 5. Note also, that Java is implicitly executing the statement `a = new Integer(5);` Java must implicitly handle the difference between primitive and Object types, because it does not contain operators to manipulate the interpretation of a variable, as provided in C.

## 3.2 Scope

The *scope* of an identifier is the collection of statements over which the identifier refers to the same entity, or has the same binding. Static scoping, or lexical scoping, defines scope in terms of the variable's declaration location in the program source code (C, Java, Python). Dynamic scoping defines the binding of a variable as the most recent binding in terms of program execution (Common Lisp, Perl). Dynamic scoping is generally not used because it makes type checking difficult and is prone to errors.

Scope can be *nested* or *disjoint*. Disjoint scope means that two variables with the same identifier cannot be confused because it is impossible to access both identifiers from from the same statement. An example is local variables in C functions. C functions cannot be nested, so variables local to one function are always inaccessible in another function. Java, on the other hand, permits nesting of both methods and classes.

Nesting can occur due to both syntactic nesting and semantic nesting. Syntactic nesting is generally obvious in the layout of the program. Semantic nesting is not generally obvious from the program layout. Inheritance in Java and C++, for example, permits nested name scope without any sort of syntactic linkage; only the semantic meaning of the class names links the name bindings.

Consider the following example of nesting in C.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  int i;
  int q = 5;

  printf("q = %d\n", q);

  for(i=0;i<1;i++) {
    int q = 4;

    printf("q = %d\n", q);

    {
      float t = 15.0;

      printf("t = %.2f  i = %d\n", t, i);
    }
  }

  printf("q = %d\n", q);
  return(0);
}
```

When executed, the function prints out:

```
q = 5
q = 4
t = 15.00  i = 0
q = 5
```

Scope applies to all forms of identifiers. The library function `printf`, for example, must be defined in `stdio.h` before you can use it. The scope of the variables `i` and `q` defined at the top of the main function is the entire main function. The scope of the variable `q` on the first line of the for loop is the for loop block.

As shown by the printout, the `q` variable inside the loop *shadows* the `q` variable declared outside the loop. In this case, the shadowing is complete; there is no way to access the outer `q` variable inside the for loop. Shadowing is not always complete, however. Classes in C++ or Java permit the programmer to access parent class entities with the same identifier as a child class entity by using the appropriate prefixes. Likewise, within a class method a programmer can usually use a predefined identifier such as `this` to differentiate a class field from a local variable or parameter. Shadowing is another way of describing the *visibility* of an entity. An entity is visible if the programmer can access it.

Note that in Ada, since all blocks, functions, and packages have names, all versions of an identifier are visible to the programmer using the appropriate qualifier.

Python has a unique method of enabling access to non-local variables. The interpreter creates a new scope for each file, class, and function. Blocks such as for loops and if-statements use their enclosing scope. If a variable is used only as an r-value, then Python will look in all of the enclosing scopes for the symbol. However, if a variable is used as an l-value anywhere inside a local scope then Python will look only in the local block for its definition, and create a new entry of the variable in the local block symbol table if it does not find a prior definition. The exception is that by using the `global` keyword, the programmer can tell Python to look in the global scope for the variable.

Nesting means we can differentiate variable accesses as local or non-local. A local access is an access within the level in which the variable is defined. A non-local access is anything else. In the example above, the print statement in the innermost block contains a local access of `t` and a non-local access of `i`.

Many languages implement a *no forward reference* rule for scoping, which means a name must be defined before it can be referenced. In languages such as PHP or Python, the rule is a bit more relaxed in that a name must be defined before it can be used as an r-value or function call, but the act of using a variable as an l-value defines it.

Java relaxes the *no forward reference* rule for methods and class field declarations. A programmer does not need to define methods and class fields before using them. This permits flexibility in writing code as well as heated arguments about proper style. The rules of Java compilation also means a program can reference classes and fields not defined in any scope within a code file, so long as those classes and fields are defined in other code files within the same directory.

Note that languages can change scoping rules over time. C used to require all variable declarations come before all statements. C++ relaxed the requirement and since relaxing the rule does not affect backward compatibility–and C compilers are strongly linked to C++ compilers–newer versions of C compilers permit inline declarations and for loop scoping. C still does not permit for loop scoping, while C++ allows variable declarations in the initialization step of a for loop.

An interesting contrast of scoping rules are loop variables in Python and C++. A Python for loop requires a loop variable name, and the loop variable continues to have a proper binding, and scope within the local code block after the for loop terminates. In C++, on the other hand, the compiler gives the error 'using obsolete binding'. The loop variable has scope only within the for loop.

## 3.3 Symbol Tables

A symbol table is the data type used by a compiler or interpreter to keep track of bindings. A symbol table is usually a dictionary. A dictionary is a data structure that links a symbol, or identifier, with other attributes, often using a hash table. In the case of a symbol table for a language, the dictionary contains at least the value and type of the variable. Scope is normally handled implicitly through structuring the order in which the compiler or interpreter creates, deletes, and accesses the dictionaries.

Let the dictionary for the current scope hold all variables declared in that scope and no others. Using the following algorithm the compiler can implement nested scoping rules.

1. When entering a new scope, push a new, empty dictionary on the symbol table stack
2. When exiting a scope, pop the top dictionary off the symbol table stack
3. When a name is bound to a value, push the entry onto the current dictionary
4. Given a name reference, look in the current dictionary
    - If the name reference has a binding, return the appropriate value
    - If the name reference is not found, repeat the process with the next dictionary in the stack

- If there are no more dictionaries, return failure and report a lookup error

In the most deeply nested block of the C example above, part of the symbol table for the C program above might look like the following.

```
<t, float, 15.0>
<q, int, 4>
<argc, int, #> <argv, **char, addr>  <i, int, 0> <q, int, 5>
<printf, func, addr> <main, func, addr>
```

Python makes heavy use of symbol tables. The flexibility of python identifiers to refer to data, functions, or classes means the symbol tables must handle many things. There are at least three symbol tables available to the program for l-values at any one time. In the case of r-values, there can be any number of symbol tables.

1. The local symbol table, searched first, which contains local names and a reference to the symbol table of the enclosing block.
2. The module symbol table, which contains the module's global names (e.g. the interpreted unit).
3. The global symbol table, which contains built-in names, variables, functions, and classes defined at the top level.

For an r-value, Python first searches the local symbol table, then its enclosing block, and continues the process until it reaches the global symbol table. The value bound to the symbol is the value in the first symbol table Python encounters. If the interpreter reaches the global symbol table without finding the symbol, it throws an error.

```
# Scoping example for Python
g = 10
f2 = 15
f1 = 12
print 'initial:', g, f2, f1

# a new symbol table gets made when f1 is executed
def f1():
    g = 5 # creates a new symbol g in the f1 symbol table

    print 'g in f1():',g

    global f2     # specifies that f2 should reference the global symbol table

    # modifies f2 in the global symbol table
    f2 = 25
    print 'f2 in f1():',f2

    # a new definition of f2, which modifies the global symbol table
    def f2():
        g = 6 # a new definition of g
        print 'g in f2:', g

    # a new definition of f3, which goes in the f1 symbol table
    def f3():
        f2 = 6     # a new definition of f2 in the f3 symbol table
        print 'f2 in f3:', f2

    # still refers to the function in the global symbol table
    print 'f2 in f1 post f2 def:', f2

    f2() # calls the function f2

    # refers to the g in the f1 symbol table
    print 'g in f1 post f2 call:', g

    f3() # calls the function f3 in f1 symbol table

    # still refers to the function f2 in the global symbol table
    print 'f2 in f1 post f3 call:', f2

# prints the global g
print 'g, f2, f1 post f1 definition', g, f2, f1

f1()  # calls f1 as defined in the global symbol table

# prints the global versions
print 'g, f2, f1 post f1 call', g, f2, f1

# f3 is not defined at the global symbol table level
try:
    f3()
except:
    print 'unable to call f3()'
```

For an l-value, Python's behavior is more complex. A reasonable model for how it works is that, while reading through a function definition, Python builds up a symbol table with all of the l-values in the local scope. Each l-value symbol gets an entry in the local symbol table initialized to *undefined*. While executing the function, Python will look first in the local symbol table for the l-value. If found, it uses the local binding. If not found, Python will look next only in the global symbol table, not in any intervening tables of enclosing functions or classes.

Python provides a method for specifying that the interpreter should look in the global symbol table for an l-value. The `global` keyword specifies that any subsequent l-value accesses to a variable should look in the global symbol table, skipping the local symbol table. It is not possible to use symbols from intervening symbol tables as l-values.

The above example shows several of the features provided by dynamic variable declaration and typing combined with scoping rules. Some of the interesting features include the following.

- Within the function f1, the assignment to variable g creates a local variable g and does not refer to the global variable g. The rule is that variables at a higher scope are read-only unless explicitly declared as global, as the code does for f2.
- Once declared global within a scope, it is not possible to make a local entry in the symbol table with the same name. Therefore, the function f2, even though it is defined within the function f1, is globally available. By contrast, the function f3, also defined within f2, is not globally available.
- If a variable is assigned within a scope, it cannot be used on the right side of an expression in the scope prior to its definition, even if the identifier is available at a higher scope. It seems that Python does not want to mix which scope is being accessed within a scope. The one exception to the rule is that you can create a variable in the local symbol table, then use the `global` keyword to specify that subsequent references go to the module symbol table.
- The keyword `global` sends the reference to the module symbol table only. If the identifier exists at an intermediate level but not the module level, Python still references the local version. If the symbol does not exist in the module symbol table, then the global statement throws an exception.

## 3.4 Overloading

Overloading is the use of the same symbol to have describe different functionality. Overloading applies to functions and operators.

Variables have scope rules, and the binding to which a variable identifier is linked is unique for statically scoped languages. Two variable bindings cannot have the same identifier within the local scope, and the scoping rules cause a local identifier to shadow a non-local binding. Languages that permit access to non-local variables with the same identifier require the use of a prefix to make its name unique.

Overloading of operators is also called making them orthogonal. The concept of orthogonality in languages is making the operators independent of the data type. For example, in Python the + operator functions on the base data types string, int, float, and list, and the * operator functions on the base data types when multiplied by an int type value.

In Java, the print method operator functions on any object, just as it does in Python.

In a language like C, it is not possible to overload operators beyond their basic functionality, or to overload functions at all. Functions that apply to multiple data types must use generic pointers, such as a void *, or a union structure to pass around information. Responsibility for handling types correctly is the responsibility of the programmer. Creating new data types like a Vector or a Matrix requires making functions to implement standard mathematical operators.

In C++ and Java it is possible to overload methods. A class can contain numerous functions with the same identifier, so long as they have different arguments. Two identically named functions cannot differ only in their return type. They must be differentiable by their argument list. Ada is the only language to permit overloading based on return type.

The difficulty with differentiating functions by only return type is twofold.

- A function can be called without making use of the return value. If the function is called outside of an assignment or its value is not being passed into another function, then there is no way to determine from the code which of the functions is meant to be executed.
- Even if a function is on the right-hand side of an assignment, the variable on the left side may not be of the same type as the return value of the function. While this is not an issue in a strongly typed language like Ada, it is an issue in Java and C where automatic casts are common and do not generally raise errors or warnings.

C++ also allows operators to be overloaded, as does Ada. Therefore, if you create a Vector class, for example, you can define + and * to do the right thing, whether it is adding a scalar or another Vector. Java, on the other hand, does not permit operator overloading.

Java does allow variables and functions to share the same identifier. A class can have a field variable with the name *blah* and a method with the name *blah* and it can always differentiate the two because a function must have parentheses attached to the end of the identifier. C and Python cannot allow a class field and a method to share the same name because in those languages a function name can be used as a right-hand side expression without any parentheses.

## 3.5 Lifetime

The lifetime of a variable is the time during program execution when the variable is bound to a real memory location. Once the lifetime of a variable ends the value the variable held is lost. Even if the variable is recreated–such as in a function call–it may be bound to a different memory location.

Fortran and Cobol, as the first two significant programming languages, use static allocation and all variables had a lifetime that was the complete execution time. Variables were assigned a memory location when the program was loaded and continued to use the same memory address throughout execution. Therefore, even local variables in functions retained their value between function calls. Dynamic memory management had to be handled by the programmer.

Algol first proposed the (very radical) idea of dynamic allocation of memory, and thus introduced the concept of variable lifetimes less than program execution. Today, most programming languages link lifetime and scope. Variables do not generally retain their value once they go out of scope because they lose their binding to a specific memory address.

Different languages have different methods of allow the programming to specify lifetime. In some languages–e.g. C and Pascal–variables declared at the compilation unit have global scope. These are variables declared outside of any function, allocated memory space when the program starts, and retain the same memory location until termination.

While Pascal allowed only a single compilation unit, C permits many compilation units (files). Therefore, C has a mechanism for giving other compilation units access to global variations. The extern statement is a way of advertising the existing of a global variable declared in one file to other files. Generally, a programmer puts extern statements into an include file.

file1.c

```
#include "file.1h"

int gblA = 0;
int gblB = 0;
```

file1.h

```
extern int gblA;
extern int gblB;
```

file2.c

```
#include "file1.h"

int func() {
  return(gblA + gblB);
}
```

C also permits the programmer to use the `static` keyword to give a local variable within a function a global lifetime. Declaring a local variable static means it keeps its value between calls to the function. Its scope remains the same–it can be used only within the function–but its lifetime is the extent of the program.

Note that declaring a global variable `static` in C affects the scope of the variable, not its lifetime, which is still the length of the program. In particular, it limits the scope of the global variable to the compilation unit (the file). Other compilation units cannot use the variable. Limiting the scope of global variables to a compilation unit is good programming practice if the variable is not necessary in other files. It avoids collisions between global variables defined in different compilation units with the same name.

Java also makes use of the `static` keyword to specify that a class field should have a global lifetime. Static class fields in Java are bound to the class, not to individual objects of the class. All class objects refer to the same binding of a static variable. C++ and Python have similar mechanisms for creating class fields with a single binding and program lifetime.

# 4 Types

Types determine how a sequence of bits is to be interpreted by the compiler. A type defines not only a data storage format but also the set of operations applicable to the data. In some cases, such as floating point representations, the format and operations are defined as a standard so that the data type is common across all platforms. In other cases, such as the int or long representations, the data type is platform and compiler dependent.

In a programming language, variables must have a type associated with them (assembly does not count as a programming language). Without type information, the compiler or interpreter does not know which machine level instructions to execute on the data. Variable types, however, do not have to be assigned at compile time or assigned explicitly.

- A language is statically typed if the variable types are determined or specified at compile time.
- A language is dynamically typed if the variable types are determined or specified at run time.

A type error is when a program executes an operation on a data type for which the operation is undefined. This includes incorrect implementations of correct operations. Not all languages attempt to catch type errors. C permits a wide range of really bad things to occur through both casting and the union data structure. C and C++, therefore, are not strongly typed. Ada and Java, on the other hand, are strongly typed languages and are designed to catch all type errors at compile or run time.

Type errors can have significant consequences, because they affect very low level operations that algorithms depend on being correct. The benefit of a strongly typed language is that the compiler or run-time engine tends to find type errors due to programmer logic, resulting in more reliable programs. The drawback of a strongly typed language is that it makes it more difficult to implement certain algorithms or manipulate or view memory directly.

## 4.1 Basic types

Most programming language data types are built on a set of more or less standard types. The types are standard largely because the computational hardware is designed to support them. The instruction sets of most CPUs include functions for handling integer data types and floating point data types. Integer data types are a natural internal representation because they have a trivial mapping into a binary representation, and manipulation on binary numbers is well defined.

Floating point data types, however, have required a significant amount of effort to define and implement and are defined in the IEEE 754 standard. The standard is not arbitrary, in the sense that it was well thought-out and includes a number of features that make it efficient and useful, but the mapping from floating point numbers to a binary representation is not natural or obvious. A common standard for both the format and the algorithms that manipulate floating point numbers is essential for making code portable across computers and programming languages.

Some CPUs, and most older CISC style CPUs also support string operations such as copying, concatenation, and comparison. The IA-32 data set has a number of different string manipulation operators at the hardware level. Some languages have strings as a basic data type (Python), while some incorporate them as built-in objects (Java). C does not have a built-in string definition, only a common standard for string representation (a character array with a null terminator) that is built into the standard C library. Note that the hardware operators do not necessarily correspond to the programming language representations.

In a somewhat odd twist, the assumption that integer representations and manipulations are natural has led to the odd fact that integer formats and manipulations are not, in fact, common across computers across all languages. This has led to many type errors when the results of integer manipulations are different on different platforms. C, for example, lets the compiler define the format for the `int` and `long` data types based on the machine hardware. While this has the benefit that it optimizes memory usage and integer arithmetic for the hardware, it also means that code that works on one machine may not work on another.

Many programming languages eliminate this problem by defining a specific integer format as part of the language. Java and Python, for example, are portable because their type formats are defined in the language specification.

In almost any language, it is possible to build data types of arbitrary form and format, creating appropriate operators for each type. The specifics of the language determine the level of difficulty of implementing a specific type.

### 4.1.1 Formats

Programming languages do not always fully define the semantics of data types. Java, because it has control over the architecture, defines data type sizes explicitly. Ada, C, and Python do not. Note that Python, because it is built on C, is implementation dependent. However, since Python chose to use the longest data types for representing basic types, there is less likelihood of a problem.

The main reason for defining a data type as part of a language specification is portability of code. The main reason for not defining it is optimization to the particular architecture. A data type that has no hardware support may be convenient, but it will also be slower.

| Type | C/C++ | Java | Ada | Python |
|---|---|---|---|---|
| byte | - | 1 | - | - |
| char | 1 bytes | 2 bytes | 1 byte | - |
| short | 2 bytes | 2 bytes | 2 bytes | - |
| int | short $\leq$ int $\leq$ long | 4 bytes | $\geq$ 2 bytes | C long |
| long | $\geq$ 4 bytes and $\geq$ int | 8 bytes | $\geq$ 4 bytes and $\geq$ int | unlimited precision |
| float | IEEE 754: 4 bytes | IEEE 754: 4 bytes | IEEE 754: 4 bytes | C double |
| double | IEEE 754: $\geq$ 8 bytes | IEEE 754: 8 bytes | IEEE 754: $\geq$ 8 bytes | C double |

Note that the organization of the bits and bytes within the data type is always machine dependent. A machine is big-endian if the first byte of a multi-byte integer type corresponds to the most significant bits. A machine is little-endian if the first byte of a multi-byte integer type corresponds to the least significant bits. In most (if not all) CPUs, bit ordering is such that the first bit is the most significant.

## 4.2 Implicit conversions

Implicit conversions between types are a common occurrence in most programs and programming languages. Most languages have a set of rules for implicit conversions that may occur without producing a warning, even strongly typed languages. The ANSI standard for C, for example, defines the set of implicit conversions for numeric types in expressions with binary operators. A subset of the rules are given below.

- If either operand is long double $\rightarrow$ convert the other to long double
- else if either operand is double $\rightarrow$ convert the other to double
- else if either operand is float $\rightarrow$ convert the other to float
- else if either operand is unsigned long int $\rightarrow$ convert the other to unsigned long int
- else if the operands are long int and unsigned int and long int can represent unsigned int $\rightarrow$ convert the unsigned int to long int
- else if the operands are long int and unsigned int and long cannot represent unsigned int $\rightarrow$ convert both operands to unsigned long int
- else if one operand is long int $\rightarrow$ convert the other to long int
- else if one operand is unsigned int $\rightarrow$ convert the other to unsigned int

Java, which has both Objects and primitive data types defines 11 broad categories of conversions (Java Language Specification, 3rd Ed). These provide a reasonable description of most conversions in most languages, as well as some peculiar to Java.

- Identity conversions - convert a type to the same type
- Widening primitive conversions - conversion of a numeric data type to another numeric data type of equal or greater information content*. The magnitude of the number is always preserved, but some conversions lose precision.
  - byte $\rightarrow$ short, int, long, float, or double
  - short $\rightarrow$ int, long, float, or double
  - char $\rightarrow$ int, long, float, or double
  - int $\rightarrow$ long, float, or double (note that int to float is not necessarily exact)
  - long $\rightarrow$ float, or double (note that long to float or double is not necessarily exact)
  - float $\rightarrow$ double
- Narrowing primitive conversions - conversion of a numeric data type to another numeric data type of less information content. The magnitude of the number is not guaranteed to be maintained.
- Widening reference conversions - a widening reference conversion from type S to type T if S is a subtype of T. A subtype can be an extension of a type or an implementation of an interface. A run-time check of widening reference conversions is not necessary, as the subtype implements the supertype.

- Narrowing reference conversions - a conversion from a type S to a subtype T, provided that S is a supertype of T. Narrowing reference conversions also include casting a class type C to an interface type K that C does not implement. Narrowing reference conversions require a run-time check to determine if the reference value is a legitimate value of the new type.
- Boxing conversions - converts a primitive type (boolean, byte, char, short, int, long, float, double) to its matching reference type. Boxing conversions do not lose information.
- Unboxing conversions - converts a reference type to its primitive type.
- Unchecked conversions - given a generic type G with n formal type parameters, an unchecked conversion converts the raw type G to any parameterized form of G. Such a conversion is permitted largely because of legacy code written before the introduction of generic types.
- Capture conversions - converts a parameterized generic type G$< T_1, T_2, \ldots, T_n >$ to a type G$< S_1, S_2, \ldots, S_n >$ by converting each argument $T_i$ to its $S_i$ counterpart using a set of rules that move wildcard symbols in $T_i$ to actual types in $S_i$.
- String conversions - uses the $toString$ method to convert any type to a String.
- Value set conversions - permits choices in the implementation of the Java programming language by permitting internal representations of float and double that contain more information than the language standard types. The value set conversions provide guidance in the case that the value of the internal representation cannot be represented in the formal representation.

Java defines five contexts in which the compiler can make implicit conversions. The rules are different for different contexts.

- Assignment conversion - converts the output type of the rhs expression to the type of the lhs variable.
- Method invocation conversion - converts each argument of a method call to the appropriate type.
- Casting conversion - converts the type of the expression to the explicitly specified type of the cast.
- String conversion - converts the type to a String.
- Numeric promotion - converts the operands of a numeric operator to a common type so the operation is executable. The rules for numeric promotion in Java are simpler than those of C.
  - If any of the operands is of a reference type, execute an unboxing conversion, then:
  - if either operand is of type double $\rightarrow$ convert the other to double
  - else if either operand is of type float $\rightarrow$ convert the other to float
  - else if either operand is of type long $\rightarrow$ convert the other to long
  - else if either operand is of type int $\rightarrow$ convert the other to int

## 4.3 Pointers, Arrays, and Strings

Pointers are a data type that enables explicit memory addressing. A pointer holds the address of a variable of a certain type. In a strongly typed language, the pointer must have the same base type as the variable it addresses. In a weakly typed language, the pointer does not have to be the same type as the variable it addresses.

A pointer holds only an address; it does not hold data, although the address itself can be treated as data in some situations. The **dereference** operator applies only to pointers. The dereference operator provides the value of the variable at the address specified by the pointer.

All languages have pointers. Not all languages call them pointers or give the user explicit operators for managing them.

In Java, all variables that are of type Object or an extension thereof are pointers. Only variables of the primitive types are not pointers. Java uses the word reference instead of pointer, however, and the references are strongly typed. Dereferencing in Java is automatic. The code below requires no explicit dereferencing and prints out the value 10, despite the fact that all the variables involved are pointers.

```
public class deref {
  public static void main(String argv[]) {
    Integer a = new Integer(4);
    Integer b = new Integer(6);
    Integer c;

    c = a + b;
    System.out.println("c is: "+c);
  }
}
```

Code with exactly the same functionality in C would look like the following. Note that each variable, including `c` must be explicitly allocated in this situation and the memory must be explicitly freed at the end of the program. Likewise, all dereferences must be explicit. Without the dereferences, the program would not work correctly.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int *a = malloc(sizeof(int));
  int *b = malloc(sizeof(int));
  int *c = malloc(sizeof(int));

  *a = 4;
  *b = 6;
  *c = *a + *b;
  printf("c is %d\n", *c);

  free(a);
  free(b);
  free(c);

  return(0);
}
```

Arrays are chunks of contiguous memory with a virtual grid of a particular type size laid over it. The array can be an array of base types or an array of pointers to other types.

- In a statically typed language, the elements of an array must all be the same type, from the programmer's point of view.
- In a dynamically typed languages, the elements of an array can be different types, from the programmer's point of view, but they are all the same type–pointers–from the interpreter's point of view.

Arrays and pointers are linked, because the address of the first element in the array and the size of each individual element are the critical pieces of information required to access the array properly. Therefore, if you have the address of a variable of a particular type, you have all of the information required to access an array of the same type.

In C, pointers and arrays are explicitly defined to be the same thing.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int i;
  int *x = malloc(sizeof(int) * 6);

  x[0] = 1;
  *(x+1) = 2;

  i = 2;
  x[i] = 3;

  i++;
  i[x] = 4;

  i++;
  *(x + i) = 5;

  x += 5;
  *x = 6;
  x -= 5;

  printf("%d %d %d %d %d %d\n", x[0], x[1], x[2], x[3], x[4], x[5]);

  free( x );

  return(0);
}
```

Pointer arithmetic is unlike standard arithmetic in that the pointer's value is incremented by units of the type size. In the example above, if the size of an int is 4 bytes, then the expression `x += 5` increments `x` by 20. In C, the effects of all pointer arithmetic are calculated at compile time, as are all array dereferences such as `x[4]`.

For strongly typed languages, and languages, such as Java, that cause exceptions for out of bounds array accesses, the compiler must build in memory space to store information about the size and type of the array. Compilers generally make space for this information, apparently called a *dope vector*, in memory space just before the start of the array. The dope vector might store the size of the array elements, the number of elements, the type of the elements, and the starting address of the data. Therefore, the dope vector contains all of the information required at run-time to identify the type, data location, and number of elements in the array.

Arrays can be explicitly multi-dimensional (Ada, Pascal, Matlab, Fortran) or built by the programmer out of one-dimensional arrays (Java, C, C++). The latter permits non-rectangular multi-dimensional arrays.

Languages that permit explicit multi-dimensional arrays must still store them in a one-dimensional memory. Most modern languages store multi-dimensional information in row-major order, place each row one after another in 1-D memory. Fortran is an exception to this and stores multi-dimensional arrays in column-major order. Knowing the ordering of data in memory is critical to achieving good performance given the heavy use of caching in modern processors. The initial value of the elements of an array is a language dependent issue. In C, memory allocated for an array contains whatever was there prior to the allocation. The programmer must explicitly fill the array with data. In Java, arrays of primitive numeric types are initialized to zero, objects to null, and booleans to false.

The index of the first element of an array is also language dependent. Fortran and Matlab, for example, use 1-based indexing, which means the first element in the array is 1. Python, Java, and C/C++ use 0-based indexing. One trick often used when converting Fortran code to C is to allocate an array and then move the array's value (the starting address) to the memory location before the start of the array. Note that this would not work properly with a language that uses a dope vector.

Strings are fundamentally arrays of characters. However, they are essential enough to programming that most modern languages handle them as a special case. C and C++ do not, however, and the convention in those languages is that strings are null-terminated. Therefore, every string must end with a 0 valued character ('\0'). The entire C standard library is predicated on null-terminated strings. The lack of a null terminator is a primary cause of segmentation and security faults.

Java, Perl, and Python are examples of languages with an explicit String type and a large library of string operations.

## 4.4 Structures

Structures were first implemented in Cobol and PL/I. Pascal and C style languages make heavy use of structures to organize collections of data. A structure is a user defined collection of data. Each individual piece of data has a type and a name. Once defined, the programmer can create a variable of the type of the structure. To get at the individual parts of the structure requires a structure reference operator, usually defined as a dot (.).

The expression `a.b` tells the compiler to access the value of the field `b` in the structure referenced by a. The same notation is generally used to access the fields of classes, which are structures on steroids.

One issue with structures is that the modern design of computer buses and caching works best if memory accesses are on N-byte borders (N = 4, for example). A structure with a short and an int in it, for example, might have a size of 8 bytes, as would a struct with two shorts and an int. On the other hand, a struct with three chars in it has a size three. The exact rules may be implementation and architecture dependent.

## 4.5 Functions

In many languages, variables can take on the value of a function reference (C, C++, Python, Lisp). Function references are useful in many situations. For example, an algorithm may want to let the user insert an arbitrary function into one step of its algorithm. C permits a special syntax to specify the type of a variable that will hold a function. It also contains the void * type, which can hold function references.

C does not (explicitly) allow functions to be manufactured dynamically as part of the language, however, which differentiates the function type from other data types. Some languages permit the programmer to manufacture functions on the fly, including PHP, Python, and Lisp either through built-in library calls (PHP) or through lambda function notation (Python and Lisp).

## 4.6 Variants

A variant is a data structure that gives the programmer explicit control over how the compiler is to interpret bits contained in the structure. A variant is not a true polymorphic type, like variables in Python, because it does not dynamically track the type of data contained in memory. The programmer can write one data type to the structure and then access the bits as a different type (intentionally or unintentionally). In most cases this results in a type error, as in the code snippet below.

```
typedef union {
  int ivalue;
  float fvalue;
} utype;

...

utype x;
float b;

x.ivalue = 4;
b = x.fvalue + 6.0;
```

## 4.7 Polymorphism

Writing efficient algorithms can be challenging. Sometimes algorithms require a specific data type, such as floats or doubles for most scientific computing. Sometimes, however, algorithms are applicable across a broad range of types, such as sorting or searching. If a type is comparable, for example, a standard sorting algorithm can sort it. Writing multiple copies of an algorithm, one for each type, is both inefficient and likely to cause errors if a change is made to one version but not the others. Polymorphic types and algorithms attempt to reduce the burden on the programmer by permitting a single piece of code to function for multiple data types.

Programming languages implement many different methods for polymorphic types. Languages such as PHP and Python have polymorphic variables, so if comparison is defined between all elements of an array a single sorting algorithm will execute properly. Statically typed languages have to get more complex in order to permit functions that operate generically.

- C uses the void * data type to hold generic pointers, and function pointers to give the user the ability to pass in a comparison function for the given data type. The algorithm knows nothing about the data type except the size of each element and it calls the user function to execute the comparison. The C library function sort is an example of a polymorphic function that uses void pointers and a user defined function.

- OOP languages have the concept of inheritance. A parent class can define a method such as Compare and child classes can overwrite it as necessary. An algorithm written for the parent class can also handle the child classes. There are two variants of inheritance designed specifically for writing polymorphic algorithms.
  - Pure virtual parent classes: the parent class defines the return type and arguments of the method, but not the implementation, forcing the child classes to write their own implementation in order to inherit the parent. C++ implements this concept.
  - Multiple inheritance: a child class can inherit from multiple parent classes, which means a pure virtual parent class can require only those functions necessary for a specific algorithm, making it act like an interface. If a child class cannot extend more than one parent, it makes inheritance a difficult mechanism to use for writing generic algorithms, as not everything makes sense to inherit from a single parent. C++ implements multiple inheritance, Java does not.
- Interfaces: An interface is almost identical to a pure virtual class. It defines the input and output types of a set of functions but not their implementation. A class can implement an interface by writing its own version of the specified functions. The programmer cannot create an object of the interface type, but a variable with the type of the interface can hold a reference to any object that implements the interface, making it a polymorphic type. Algorithms written for an interface use the interface data type, which guarantees that their parameters implement the necessary methods. Java implements interfaces.
- Templates: A template is code written using a placeholder for the data type. At compile time, the compiler copies the data type definition or algorithm from the template and executes a search and replace from the placeholder to the actual data type. Templates can be defined for any data type or for subsets of types. Unlike interfaces, templates do not guarantee that the input types implement all of the required functionality, but the compiler can catch most of those errors. One difficulty with templates is that the object code generated by the compiler does not match up cleanly with the source code. A single call to a template function in the source code, for example, maps to a copy of the template code, which does not correspond to any particular line number in the source code. This can make debugging templates difficult. C++, Ada, and Java all implement variations of templates, but Java and C++ take very different approaches.

The implementation of templates across languages is not the same. Java uses a method called type erasure, which removes all generic notations and inserts casts as appropriate into the code. Erasure means that a the generic type placeholder gets replaced by its parent furthest up the inheritance tree for the purpose of compilation. When compiling code that uses a generic type, like `MyClass<String>`, all of the generic type specifiers inside the brackets get erased by the compiler. Once compilation is complete, the byte code contains no references to the original generic type. The sole purpose of generics is to permit compile time type checking, not to generate different code for different types. Generics in Java do not permit actions that require the compiler to know the type of the operation at compile time. Actions such as allocating a new object or array, for example, require knowledge of the specific data type.

C++, on the other hand, implements templates as a macro substitution process prior to compilation. If a template is called with a new type, the compiler copies the template code, replacing the placeholders with the proper type, and then compiles the new code in addition to any existing code submitted to the compiler.

In effect, for each type used with a template it is adding a virtual file to the list of files to compile. The compiler can catch type errors if, for example, an operation in the template is not defined for the template instantiation type. Run-time errors, however, do not have an associated location in an actual file, so a debugger is unable to specify a line number for the fault. Because C++ templates use macro substitution, however, the compiler knows exactly what types are in use at compile time.

One result of this difference is that in Java you cannot allocate a new instance or array of a generic type within your code. For example, the following example will not compile. The reason is that Java compiles generic code only once, inserting casts into the code, as necessary to convert data into the appropriate type. In the case of allocating single objects, without knowing the type, the compiler does not know how much memory to allocate for the individual instance. In the case of arrays, it might seem that the compiler should be able to create an array of references. Unfortunately, permitting allocation of generic types permits run time type errors that the compiler cannot catch. Therefore, all generic array allocation is forbidden.

```
public class MyStuff<E> {
  private E mything;
  private E[] mydata;

  public MyStuff() {
    mything = new E();  // illegal, size of E is not known at compile time
    mydata = new E[ 6 ]; // illegal, E is not known at compile time
    mydata = (E[])new Object[5]; // legal, E is replaced by Object at compile time
  }
}
```

In the Collections classes, Java uses the oldest trick in the OO book to create the arrays of pointers required to hold the collections.

```
public class MyCollection<E> {
   private E[] myArray;

  public MyCollection() {
    myArray = (E[]) new Object[ DEFAULT_[SIZE] ];
  }
}
```

Because C++ does macro substitution, and the compiler knows the actual type of each variable at compile time, C++ templates do not have as many constraints on what is possible. The following, for example, works fine.

```
template <class T>
class Holder {
public:
  T *x;

  Holder( const T &thing ) {
    x = new T( thing ); // requires a constructor with a const T argument
  }

  T &get() {
    return *x;
  }
};
```

# 5 Semantics

Operational semantics are a behavioral specification of the meaning of a program. Given an architecture and a compiler, the meaning of a program is the output produced by that architecture/compiler pair. While it sets a clear standard for meaning, operational semantics are dependent on an architecture and the meaning is implicit in the design of the system, not explicit.

Axiomatic semantics are based on a formal specification of each statement in the language as an axiom. The formal specification enables a programmer to rigorously prove the behavior of a program through systematic application of the axioms and logic.

Denotational semantics represent operations in the abstract syntax as mathematical transformations of the computer's state. The meaning of a program is how it affects the state of the computer. The computer's state is an abstract concept, but representable on any *Turing complete* architecture. A Turing complete architecture requires that the computer be able to make assignments, execute a sequence of instructions, execute a conditional statement, and loop.

Semantics are critical to the design of a programming language, because the meaning of a language is not completely captured by an abstract parse tree, as the following two sections demonstrate.

## 5.1 Expressions

Operator precedence can be handled by syntax and the design of the abstract parse tree. However, the structure of the parse tree only guarantees the order of operation of the mathematical operators. It does not guarantee the order or timing of the execution of the sub-trees of any particular operator. A left-to-right pass to generate the machine code can produce different results than a right-to-left pass if the base expressions are functions with side effects.

For example, the pre- and post- increment operators create semantic ambiguity in C-like languages when there are more than one inside an expression. Consider the following code snippet, which is perfectly legal C.

```
int *a = malloc(sizeof(int) * 3);
int b = 3;
int c = 4;

a[0] = 1;
a[1] = 2;
a[2] = 3;

*a =  *(a++) * b + *(++a)*c;
```

The order of operation is not completely defined by the syntax for the pre- and post- increment operators. The order of tree traversal when generating the code, and the rules for when the pre- and post- increment operators execute determine the semantics of the expression. Some of the options include the following.

- Execute all pre-increment operators prior to all other accesses, picking an order L-R or R-L in the expression.
- Execute pre-increment operators when the node is reached during tree traversal.
- Execute all post-increment operators after the expression has been calculated and assigned.

- Execute post-increment operators when the node is reached during tree traversal.
- Execute post-increment operators after the value has been computed, but before assignment.

It turns out that the C language specification does not define the behavior of the program where there is more than one pre- or post- increment operator in applied to a single variable within a single expression, so the behavior ends up being compiler specific. The semantic meaning of the program is undefined. gcc 4.0.1 on a Macbook Pro produces the number 11 and puts it in the second location in the memory space malloc'd for a (index 1). Therefore, the rules for pre- and post- increment must be:

- Execute the pre-increment operator when the node is reached during tree traversal, but before accessing the value referenced by the variable.
- Execute the post-increment operator after the assignment is complete.

Note, this is yet another reason not to use pointer arithmetic to manipulate memory.

### 5.1.1 Lazy evaluation

Evaluating expressions with boolean operators (and, or, not) is another area where the value of an expression is not well-defined without a semantic definition. Consider the expression below.

```
if( a || (b && c && d) )
  /* do something */;
```

The values a, b, c, and d may all be functions, possibly even functions that require lots of resources to compute. Note that, if `a` is true, then the rest of the expression is irrelevant to the truth of the entire statement. Only if `a` is false, does the rest of the statement require evaluation. The expression `(b && c && d)` also permits lazy evaluation, because if any of the individual items are false, the entire statement is false. Therefore, the most efficient method of evaluating the expression is to treat it as a set of nested if-statements.

```
if(b) {
  if(c) {
    if(d) {
      /* do something */
    }
  }
}
```

Most languages implement lazy evaluation of boolean expressions to take advantage of the efficiencies. Lazy evaluation means that the order of operations produces semantically different results. For example, consider the expression `if( ptr != NULL && *ptr == a )`. If the pointer is null, then the computer does not evaluate the right side of the expression and the value of the expression is false. If we reversed the order of the sub-expressions, however, then a null pointer would result in an undefined value for the expression because dereferencing a null pointer has an undefined result.

## 5.2 Program State and Denotational Semantics

Program state is an abstract representation of the state of the computer during a program's execution. In terms of the vocabulary we have used so far, the program's state consists of all variables with active bindings, and the value of the memory locations to which they are bound. Note that a variable with an active binding may not necessarily be in scope or visible. For example, a static local variable in a function has scope only within the function, but a global lifetime, which means its binding is active from the start of the program to its end.

Statements such as conditionals, while they may not affect the program state as defined above, affect which statement the computer will execute next. Arguably, we should include the address of the next statement to be executed–the program counter [PC]–in a computer's state, along with other implied variables such as the stack pointer [SP] and frame pointer [FP]. From a hardware point of view, the PC, SP, and the flags register are the primary requirements of computer state, with the contents of registers and memory constituting the rest.

Denotational semantics describe the meaning of each token in an abstract syntax as a transformation on the program state. The notation generally used is given below, where $M$ represents *meaning*.

$$\begin{aligned} M &: Program \rightarrow State \\ M &: Statement \times State \rightarrow State \\ M &: Expression \times State \rightarrow Value \end{aligned} \tag{16}$$

The meaning of a Program is defined by a function that generates an initial state. The meaning of a Statement is a function applied to the current state to generate a new state. The meaning of an Expression is a function applied to the current state to generate a value, which may then be used within a statement.

### 5.2.1 Program

The abstract syntax of the Program says that it consists of a declaration section `decpart` and a body section `body`.

> The meaning of a program is defined to be the meaning of its `body` when given an initial state consisting of the variables of the `decpart`, each initialized to the undef value corresponding to its declared type. (Tucker and Noonan, 2007)

If we have an abstract syntax tree for a program, we can calculate the meaning of the program by replacing the abstract operators with functions that specify how the operator modifies the program state.

The Program node in the abstract syntax is important to the program's meaning because it initializes the state of the computer. One way of thinking of the program is as a function that takes in the initial state and the program body and produces a final state.

$$State\ MProgram(\text{Program.body}, InitialState(\text{Program.decpart})) \tag{17}$$

The function M will traverse through the program body, represented as an abstract syntax tree, executing the transformations defined by the sequence of statements. The return value is the State of the computer at the end of the traversal.

The $InitialState$ function must be clearly defined for a language in order for programs to have meaning. Different languages implement it differently. C, for example, does not initialize the memory space used by the program to any particular value. Therefore, the $InitialState$ function will generate the bindings for all variables in the given declaration section, but the memory contents of all variables are initially undefined.

Java, on the other hand, initializes all variables at binding. All primitive values have an initial value that is the equivalent of zero, and all reference types have an initial value of null. The difference between Java and C is significant, as the same code in one language will have a different result in the other. Consider the following code.

```
// main function declaration
int sum;

for(int i=0;i<10;i++) {
  sum += 5;
}
// print out sum
```

In C, the value of sum after the code is executed is undefined, because the initial value of sum is undefined. In Java, the value of sum is defined, because the language's $InitialState$ function sets the initial value of int type variables to 0 when they are bound. Automatic initialization, while it causes the above code to be correct, and well-defined, can also create confusion. Someone who does not understand the semantics of Java will not necessarily understand the meaning of the code. Adding the assignment `sum = 0;` just before the for loop removes any confusion about the meaning of the code, and the code will then have the same meaning in both C and Java.

Note that just because the meaning of the code snippet in C is undefined does not mean that the code will not return the correct value. Zero is a common set of bits in computer memory, so it may be the case that much of the time the code functions as expected. This is a common pitfall, and can be a difficult bug to detect as many debuggers initialize memory to specific values even when the semantics of the language do not require it.

## 5.3 Statement Semantics

Statement semantics are all functions applied to the State that return a new State. As an example, Clite contains five types of statements: Skip, Assignment, Conditional, Loop, and Block. The Skip statement is simply an empty statement, and is used in the abstract syntax representation when, for example, an else clause or a for loop contains no statements in its body.

The Skip transformation function is simply the identity: the output state is the input state.

$$M : Skip \times State_i \rightarrow State_i \tag{18}$$

```
State MSkip( Skip s, State state) { return state; }
```

### 5.3.1 Assignments

Assignments are a good example of why semantics are critical to the design of a programming language. Assignments are usually treated by languages as statements, not expressions. The difference is that, if an assignment is an expression, then it needs to evaluate to a value and it can be used anywhere an expression is allows, such as in a conditional test or as an r-value.

Java, for example, requires that all expressions used in a control flow test (if, while, for) evaluate to a boolean value. C/C++, on the other hand, treat assignments as expressions. Therefore, assignments can be used in C in any of the following ways.

- In control flow statements `if( a = b )` evaluates to whatever value is assigned to `a`
- In assignments `a = b = c` evaluates c, assigns that to b, then assigns that to a.
- In expressions `c = 3 + (a = b)` assigns the value of b to a and then adds 3 to that value and assigns it to c.

Assignment semantics also define what data is being moved and stored by the computer. In C, the assignment takes the value generated by the right side and puts a copy of it into the memory location bound to the variable on the left side. This is defined as copy semantics. Aliasing is still an issue in C, because the value on the right side can be a pointer to a memory location.

In Java, assignment semantics have a slightly different interpretation. For primitive types, copy semantics apply. For all Object types, however, Java uses reference semantics. The data type Object is not bound to the memory location where the Object's data is located, but to a memory location that holds a reference to the Object. Therefore, the assignment copies only the reference to the Object.

In truth, I'm not convinced there is a real difference. The difference is really one of the semantic meaning of the types, not the semantic meaning of the assignment. If all types that are derivatives of Object are references, and a reference is a memory location with an address, then the assignment statement is following only copy semantics. However, if Object types are something much more complex, then the assignment has more meaning than just copying bits.

As an example, Python keeps track of how many references to an object exist so it can manage garbage collection. Therefore, an assignment has to do more than copy bits, it also has to increment the number of references to the object. This is a semantic difference in the meaning of an assignment.

It is possible to imagine a language with both copy semantics and reference semantics. If, for example, in Python the syntax `a := b` made a copy of the object `b` and gave `a` a reference to the copy, then `:=` would implement a different form of copy semantics. Note that copying arbitrary objects is challenging.

Assignments transform the state of the program by changing the value stored in the memory of the receiving variable. An assignment must make use of the expression transformation in order to determine the value of the right side. An assignment does not necessarily change the state of the computer–the same value could be assigned back to the variable–but it has the potential to change the computer's state. Recall that the abstract syntax represents an assignment with a `target` field and a `source` field.

$$M : \mathit{Assignment} \times \mathit{State} \rightarrow \mathit{State} \tag{19}$$

```
def MAssignment( Assignment s, State state ):
  return Put( a.target, MExpression( a.source, state ) )
```

Depending upon the language, an expression may or may not have side-effects. In most languages, an expression can have side-effects: actions that change the state of the machine while the expression is being evaluated. The Put function must return a new state, modified by assigning the value of the expression to the target. Therefore, it must take in not only the target of the assignment and the value of the expression, but also the state that exists immediately prior to the assignment of the value to the target. The MExpression function must, therefore, return both a value and a state. In Python we could easily accomplish this by returning a duple of (value, state).

The Put function implements an operation called an overriding union. An overriding union is defined on a hashmap, or dictionary. Given two sets of ordered pairs, X and Y, the result of an overriding union of X with Y is all of the ordered pairs in Y plus all of the pairs in X whose first members do not match any ordered pair in Y. Another way to phrase it is that any ordered pair in Y whose first member matches a pair in X replaces the pair in X in the output. Any ordered pair in Y not already in X is added to the result.

In a language that requires declaration of all variables before use, there will never be any ordered pairs in Y (expression) that do not exist in X (prior state), sort of. Object or memory allocation effectively creates a new temporary variable and returns a reference to the variable. In interpreted languages such as Python, Perl, or PHP, an assignment statement may add a new variable to the existing state of the computer. The overriding union is able to both replace old values with new and add entries as necessary.

### 5.3.2 Conditional

The abstract syntax of a conditional in Clite contains an expression and two statements.

$$Conditional = Expression \text{ test; } Statement \text{ thenbranch, elsebranch} \tag{20}$$

Conditional expressions in most languages follow the same abstract syntax, whatever their concrete syntax. The meaning of a conditional is dependent upon the value of the `test` expression. If the `test` expression is true, then the meaning of the conditional is the meaning of the `thenbranch`. If the `test` expression is false, then the meaning of the conditional is the meaning of the `elsebranch`.

Mathematically, we can express this as the following.

$$M(Conditional\ c, State\ state) = \begin{cases} M(c.\text{thenbranch}, state) & M(c.\text{test}, state) = True \\ M(c.\text{elsebranch}, state) & \text{otherwise} \end{cases} \tag{21}$$

Functionally, we can represent it as the following function in Python.

```
def MConditional( c, state ):
  if MExpression( c.test, state ) == True:
    MStatement( c.thenbranch, state )
  else
    MStatement( c.elsebranch, state )
```

Note that this representation of conditionals assumes there are no side effects that occur in the evaluation of the test expression. In order to properly account for potential side-effects, `MExpression( c.test, state)` must return both a state and its evaluation of the test expression. The function must then pass the intermediate state representation to the appropriate branch. The denotational representation can avoid using an explicit temporary variable by using nested functions, with the first function evaluating the expression and passing the (condition, state) pair to the actual conditional function.

```
# MExpression returns a duple with (result, newstate)
def MExpressionTest( c, state ):
  MConditional( c, MExpression( c.test, state ) )

def MConditional( c, exp ):
  if exp[0] == True:
    MStatement(c.thenbranch, exp[1])
  else:
    MStatement(c.elsebranch, exp[1])
```

### 5.3.3 Loop

Different types of loops have different kinds of semantics. The standard while loop is the only type of loop required for a Turing complete language. We can construct any other type of loop from a while loop and assignment statements. The abstract syntax of the while loop in Clite has two parts, an expression and a body.

$$Loop = Expression \text{ test; } Statement \text{ body} \tag{22}$$

If the value returned by `test` is false, then the loop does not change the state of the computer. If the value returned is true, then the function returns a call to itself with the state modified by the body. In this case we use recursion to represent the meaning of iteration.

$$M(Loop\; lp, State\; state) = \begin{cases} state & M(lp.\text{test}, state) = False \\ M(lp, M(lp.\text{body}, state)) & \text{otherwise} \end{cases} \tag{23}$$

Functionally, we can represent it as the following function in Python, if we assume the loop condition expression has no side effects. If expressions can have side-effects, then it is necessary to use nested functions, as with the conditional statement above.

```
def MLoop( lp, state ):
  if MExpression( lp.test, state ) == True:
    return MLoop( lp, MStatement( lp.body, state ) )
  else
    return state
```

Consider the for loop semantics in Java, which does not exist in Clite. The abstract syntax for a for loop requires a number of different fields: an initial statement, an expression, the body statement, and the post statement. Note that the initial statement in Java or C++ can be more than just an assignment. It can be a statement or even a declaration and an initialization. Since declarations can occur anywhere in Java, we must let them be a part of Statement.

$$For = Statement \text{ initial; } Expression \text{ test; } Statement \text{ body; } Statement \text{ post} \tag{24}$$

The meaning of the for loop is a combination of things.

- The initial statement modifies the state prior to evaluation of the test expression.
- The body and post statements modify the state only if the test expression is true.
- The post statement modifies the state after the application of the body to the state.

The for loop syntax in C/Java is special, because commas are used to differentiate different statements in the initial and post sections. However, the semantic interpretation is to treat them as blocks, in which case they are Statements. Because the initial section executes only once, we have to use two functions to represent the meaning of the for loop.

$$M(For\ f, State\ state) = M^*(f, M(f.initial,\ state)) \tag{25}$$

$$M^*(For\ f, State\ state) = \begin{cases} state & M(f.test, state) = False \\ M^*(f, M(f.post, M(f.body, state))) & \text{otherwise} \end{cases} \tag{26}$$

The functional version also requires two functions because, as in the mathematical representation, the initial body executes only once.

```
def MFor( f, state ):
  return MForS( f, MStatement( f.initial, state ) )

def MForS( f, state ):
  if MExpression( f.test, state ) == False
    return state
  else:
    return MForS( f, MStatement( f.post, MStatement( f.body, state ) ) )
```

There are several issues that come up in the context of the for loop. In CLite, the program can declare variables only in the declaration section. However, in C++ and Java the program can include declarations in the initial section of the for loop. Therefore, declarations become Statements (as they are in dynamically typed languages like PHP).

The above functional representation also does not permit expressions to modify the state. As with the conditional statement, this requires an additional level of nested functions in-between the MFor and MForS functions that evaluates the loop expression and returns a boolean value and an updated state.

```
def MFor( f, state ):
  return MForSExpr( f, MStatement( f.initial, state ) )

def MForSExpr(f, state):
  return MForS(f, MExpression(f.test, state))

def MForS( f, expr ):
  if expr[0] == False
    return expr[1]
  else:
    return MForSExpr( f, MStatement( f.post, MStatement( f.body, expr[1] ) ) )
```

One aspect of semantics we have not yet covered is scope. How is scope handled by denotational and functional semantic methods? In particular, how do we get variables out of the computer's state when they leave scope and end their lifetime?

In some languages, scope blocks include while loops, for loops, conditionals, and blocks. Java and C, for example, give variables declared in the initial section of a for loop scope only within the loop. In other languages, such as PHP and python, variables declared in the initial section of a for loop have scope in the block encompassing the for loop.

Consider the following two representations of for loops. The first represents C++ and Java for loops. The second represents PHP. The only difference is the existence of an outer block that encompasses the initial statement, but that difference affects the scope of the loop variable.

```
// original code
for(int i=0;i<5;i++) {
  // body of loop
}

{ // equivalent C++ and Java semantics
  int i;
  i = 0;
  while( i < 5 ) {
    // body of loop
    i++;
  }
}

// equivalent PHP semantics
int i;
i = 0;
while(i < 5) {
  // body of loop
  i++; }
```

How do we handle scope in functional semantics?

- When the system enters a new scope, it must create a new symbol table as part of the state.
- When the system leaves scope, it must delete the current symbol table from the state.

We can implement this in a functional manner by maintaining a stack and using push and pop operations on the state. The Push function needs to push the given state on the stack and return a copy of the pushed state. The Pop function needs to remove and return the top symbol table from the stack. In a for loop, the functional representation needs to push a new symbol table onto the state prior to executing the initial section and pop the symbol table table off the state when the loop stops executing.

```
def MFor( f, state ):
  return MForS( f, MStatement( f.initial, MPush(state) ) )

def MForS( f, state ):
  if MExpression( f.test, state ) == False
    return MPop(state)
  else:
    return MForS( f, MStatement( f.post, MStatement( f.body, state ) ) )
```

CLite doesn't require modifying the state table because it permits variable declarations only in the header section, not in blocks.

Note that static variables declared within functions must be created with the initial program state in order to exhibit global lifetime during program execution. Since the compiler will permit no references to these variables outside of their scope, scope is enforced at compile time, not run time. In a language such as C, it should be possible to locate the static variable locations in memory and access them outside of the function in which they are defined.

Note that in languages that mix dynamic and static scoping, such as Python, the context of functions must be maintained in between function calls. We saw this in one of the example programs where a syntactically nested function maintains a link to its parent's symbol table, even when the function is called outside of the parent's syntactic scope. This requires that symbol tables be maintained in a graph structure, based on how they are defined in the code, with parent pointers from children in the graph to their syntactic parent.

### 5.3.4 Block

Most languages support a lexical mechanism for specifying a set of statements as a single entity. In C a block is represented by a sequence of statements between curly brackets. In Python, a block is a set of statements with different tabbing that come immediately after a statement ending in a colon. In Ada and VHDL, a block is a set of statements between a `begin` and an `end` keyword.

Blocks usually define some kind of scope, within which certain variables have binding or are visible.

Blocks also have a semantic meaning that specifies the order in which the statements within the block are executed. In most programming languages, the ordering of statements within the block specifies the order in which a computer executes each statement. Each statement must fully complete, and its results on the computer's state must be fully visible before the next statement begins execution. Note that this may not actually correspond to the way in which a program is executed on the physical hardware, but pipelined and superscalar architectures generally support this model of execution.

The implication of sequential semantics is that the order in which two statements are written is relevant to the final program state. Clearly, there are many cases where the order of operation is irrelevant–such as writing independent values to two different variables–and superscalar architectures take advantage of this fact. However, in most programming languages you cannot arbitrarily change the statement order and expect that the two programs result in the same computer state at the end of a block.

We can encode the block statement using a functional definition, assuming a block contains a list of statements. The recursive functional definition says that the result of a block is the result of applying the first statement in the block to the current state and then executing the remaining statements in the block.

```
def MBlock( block, state ):
    return MBlock( Block( block.stmt[1:] ), MStatement( block.stmt[0], state ) )
```

Not all programming languages support sequential semantics (what?!). Consider the following program in VHDL that computes the Boolean expression $Q \leftarrow D + A(B + \bar{C})$. The program is functionally correct.

```
entity logictree is
  port( A, B, C, D: in std_logic;
             Q: out std_logic );
end logictree;

architecture example of logictree is
  signal I1, I2: std_logic;

begin
  Q <= I2 or D;
  I2 <= A and I1;
  I1 <= B or (not C);
end example;
```

The three assignment statements in the body of the architecture form a block. If the block statement followed standard sequential semantics, then the output value `Q` would be undefined because `I2` is undefined when it is used in the expression assigned to `Q`. Signal assignments in VHDL, however, do not follow sequential semantics. Instead, they follow concurrent semantics. All three signal assignments are executing simultaneously and continuously because, in this case, they represent wires in a circuit, not memory locations. Another way to explain the semantics is that each signal assignment executes whenever the r-value changes. Therefore, the order in which the three statements appear in the code is irrelevant. The program's behavior is identical for all orderings of statements within the block.

The `process` statement in VHDL does implement a form of sequential semantics for a block of code, in the sense that the ordering of operations can matter to program functionality. However, later signal assignments within a `process` statement do not see the full effects of earlier signal assignments. While this may not make much sense from the point of view of a typical programming language, VHDL is intended to describe the behavior of physical circuits consisting of wires, logic gates, and flip-flops (1 bit storage units). Wires do not behave like memory locations in a computer, but VHDL permits us to attach identifiers to a wire and specify which wires connect to which other wires.

## 5.4 I/O Semantics

I/O is confused and confusing.

- Some languages have I/O mechanisms built into the language syntax (PHP, Prolog, and Python)
- Many languages use a built-in library of functions to handle all I/O (C, Fortran, Lisp)
- OO languages generally use standard classes to handle I/O (Java, C++)

Since all I/O affects the computer's state through manipulation of memory, we can reduce I/O to a set of statements if we dig into the functions and classes that implement it. In a language like Python, the semantics of `print` are actually similar to assignment. Python first evaluates each of the comma-separated list of expressions and then calls the `__str__` method on each one–if it is not already a string–to generate a string type. It then assigns the concatenation of the strings to a buffer that sends it to the standard output.

In the absence of side-effects, we could view the print statement as a skip statement. The program cannot access the output buffer directly, so its contents don't affect the meaning of the program when viewed strictly as the set of bound variables and their contents. Even with side-effects, we can represent the print statement as a set of expression evaluations.

Of course, there is a big difference between the meaning of a program that prints something and one that doesn't. From the point of view of the program, however, equating printing text output to writing to a buffer area is sufficient. All of the processes that go on after that point are independent of the program itself.

- The operating system and the user's environment determine where the buffer's contents will appear.
- Commands to the OS when the program is executed can determine the buffer's destination.
- If the contents are destined for the terminal, the OS and the user's preferences determine things like the font, font size, and other information used to transform the buffer's characters into graphics.

File I/O is different in that the programmer has access to certain variables that affect the I/O process.

- The input buffer itself–many languages permit a program to peek at an input buffer before reading it.
- The output buffer–in some languages, a program can read the contents of a buffer it just wrote.
- The current buffer location–in most languages, a program can query and set the location in the buffer from which the next read or write operation begins.
- Status flags–most languages have a set of status flags associated with I/O buffers. In most cases these are read-only.

If we think of I/O using a buffer, or stream model, then we can represent I/O semantically as operations on variables. The buffer is a variable that an open command adds to the computer's state. The open command would also add the status flags and current buffer location. Write commands move data to the buffer, and read commands move data from the buffer.

The challenge in formally defining these actions is handling unknown input, input errors, and output errors (like the disk is full). In cases of errors, the meaning of a program may end up being undefined. However, if the program is able to handle errors robustly, the meaning may be well-defined, just not necessarily what the user intended.

## 5.5 Exception Semantics

Early programming languages did not handle exceptions gracefully. Fortran and C, for example, have no built-in exception handling mechanisms. The programmer is responsible for handling all errors. The C standard libraries do provide a method of handling exceptions that interfaces with the operating system, but it is not part of the language syntax or semantics. C also has the convention that a function that executes properly should return a 0 if it is not using the function's return value for something else. C functions with a void return type tend to be rare because of this, while they are common in Java programming.

More recent programming languages provide extensive exception handling (Ada, C++, Java, Python). While exception handling is not the default action of a program in these languages, they all have syntactic and semantic elements that support exception handling. A formal description of exception handling semantics is beyond the scope of this course.

The issue with errors is that the operation or function in which the error occurs generally cannot handle the error. A division operator cannot handle a divide by zero error. A function that is supposed to open a file cannot handle the error that occurs when there is no file by the given name. A function that is supposed to execute an operation on an image cannot handle the error that occurs when there is insufficient memory for it to allocate temporary working space.

In all of these cases, information about the error has to propagate up to the next level of scope, if not further.

Error communication strategies:

- Have each function/method return a specific value when it encounters an error. The calling function/method must test the return value of the function and take appropriate action. Many library functions in C/C++ and Java use this method.
- Have each function/method set the value of a variable when it encounters an error. The calling function/method must be able to access this variable and test it after the function call to take appropriate action. Many library functions in C/C++ use this method, especially those interfacing with the operating system and file I/O. Sometimes functions will both return an error value and set an error variable, usually using the latter to communicate which error occurred in the function.
- A function/method that may potentially encounter an error should set up an exception handler that gets called if an exception occurs. In some cases, exception handlers allow the program to continue operation, in other cases program execution halts when the exception handler completes.
- A function/method that encounters an error should throw an exception. The calling function has the responsibility of catching the exception and either handling it or passing it on to a higher context.

### 5.5.1 Try/Catch Approach

The try/catch approach encapsulates code that could cause an exception in a try block. If an exception occurs, execution in the try block halts and restarts with the first statement in the catch block. The try/catch approach usually also incorporates a `throw` command. A throw command permits the programmer to throw an exception deliberately, usually with a programmer defined exception type.

**Example: C++**

```
#include <cstdio>

int main(int argc, char *argv[]) {

  for(;;) {
    int q;

    try {
      printf("Enter a number 0..9: ");
      int k = scanf("%d", &q );

      if(k == 0) {
        scanf("%*s");
        throw "value is not a number";
      }

      if( q < 0 || q > 9 ) {
        throw q;
      }
    }
    catch( const char *s ) {
      printf("Error: %s\n", s);
      continue;
    }
    catch( int v ) {
      printf("Error: number %d is out of range (0..9)\n", v);
      continue;
    }

    if( q == 0 )
      break;
  }

  return(0);
}
```

In the case of C++, the semantics are simple to describe in words. Execution begins with the code in the try block. If no exception occurs, execution skips the catch blocks. If an exception occurs, execution continues with the first statement in the appropriate catch block, if one exists. If there is no catch block for a given exception type, the program aborts.

The catch blocks are effectively overloaded functions called by the throw statement, except the catch blocks do not return to the next statement in the try block as a function would. The catch blocks must be differentiable by their parameter list. C++ includes a set of standard exceptions defined in `std::exception` that statements like new would throw. The list also includes a catchall exception. The benefit of using try/catch blocks is that the function in which the exception occurs can catch the problem.

Note that any function can throw an exception at any time. If the function call is within a try/catch block of a parent function, then the parent function will catch the exception if there is a matching catch block. You can even design a hierarchical system whereby the local function catches some exceptions and passes others on to the parent.

**Example: Java**

Java uses a similar syntax to C++, but the semantics are somewhat different. One of the major differences is that exceptions in Java are Objects and are part of an Exception hierarchy. Therefore, when an exception occurs, Java is creating a new object and adding it to the symbol table.

```
import java.io.IOException;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class trycatch {

  public static void main( String argv[] ) throws IOException {
    byte b[] = new byte[256];
    int number;

    while(true) {
      try {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in) );
        System.out.print("Enter number: ");
        number = Integer.parseInt(in.readLine());
        break;
      }
      catch(NumberFormatException e) {
        System.out.println("Illegal number");
      }
      finally {
        System.out.println("In the finally block");
      }
    }

    System.out.printf("number is %d\n", number);
  }
}
```

A function that could throw an exception it does not catch must declare that property to calling functions by including a `throws` clause in its preamble. If a function X uses an IO call, for example, and wants the parent to handle exceptions, then X must declare that it throws an IOException, as shown in the example above. The method itself handles problems with the number format, but passes off other issues to the parent context (the JVM, in this case).

Like C++, Java permits multiple catch blocks differentiated by their argument. The try/catch structure can also include a block labeled with `finally`, which has no arguments. Code in the `finally` section is always executed whether or not the code throws an exception. It is executed even if one of the try/catch blocks calls break or return.

**Example: Python**

Python has a similar semantic structure to C++ and Java, but a slightly different syntax.

```
a = 0
while a == 0:

    try:
        s = raw_input('enter a number: ')
        val = int(s)

    except ValueError:
        print 'not a valid number'
        continue

    if val == 7:
        a = 1

print 'terminating'
```

As with the other cases, if an exception occurs if the try statement, execution moves to the except block if the exception matches one of the exceptions listed. The syntax permits multiple `except` cases. An except block can also list multiple exceptions inside a tuple, e.g. `(RuntimeError, TypeError, NameError, ValueError)`. Python includes many standard error types generated by built-in functions. An except block with no arguments catches all exceptions not explicitly caught by other except cases.

Python also permits an `else` block at the end of the try/except sequence. Code in the `else` block executes after the code in the try block if no exception occurs.

The equivalent of a `throw` in Python is the `raise` keyword. The raise statement takes the exception class as an argument. A raise statement with no argument inside an except block re-raises the exception that caused the block to be executed and passes the buck to any encompassing try/except blocks.

Exceptions can be arbitrary user-defined classes. Usually they are simple.

Python, like Java, also includes a `finally` clause. Code in the `finally` block executes no matter what else occurs in the exception structure. The code in the `finally` block executes whether the try structure exits on no exception, a handled exception, an unhandled exception, a return, a break, or a continue. In the case of an unhandled exception, Python re-raises the exception once the `finally` block completes.

Python has one other convenience feature that falls in the category of "language features that avoid oopses". Consider the following example.

```
with open("myfile.txt") as fp:
    for line in fp:
        print line
```

The `with` statement puts the result of the call to open, which is a file object, into the variable `fp`. The file class has a predefined cleanup action in its class definition. The `with` statement simply calls the cleanup method of the class when the block terminates.

**Example: C**

C includes a simple exception handling mechanism that relies on the operating system to pass control flow to an exception handling function if a run time error occurs. The signal package includes a function

```
signal( int sig, sig_t func )
```

that enables the programmer to attach a function pointer to either a standard OS signal or a user defined signal. After setting up the handler, if a signal occurs, the OS checks the signal handler table and routes execution to the user function. The user function has access to the program's memory space and global variables, but there is no guarantee they have not been corrupted by the event that caused the signal. In general, signal handlers shut down the program gracefully and exit, if possible.

The signal function also permits the user to specify that certain signals should be ignored, allowing program execution to continue where the signal left off. Some signals are not ignorable and cause program termination.

If the signal handler simply returns instead of exiting the program, then the main program will continue where it left off. Unfortunately, because of the parallel processes occurring in I/O situations, the exact state of the program upon return is not necessarily well defined. If the signal occurred during an I/O operation, the operation may be restarted. Therefore, it is possible to get in an infinite loop if restarting the operation causes another segmentation fault.

Semantically, the C signal handling is different from the prior 3 cases. In C++, Java, and Python the program does not attempt to restart execution where it left off. Instead, execution moves to the catch block, giving the program a well-defined behavior after an exception occurs. Therefore, exceptions are a reasonable method of handling IO errors, for example.

In C, execution order does not skip, but tries to pick up where the signal occurred. In the case of signals from outside (e.g. cntl-c) ignoring the error and returning from the handler should have no effect since the signal did not occur because of the program's own behavior. In the case of signals generated by the program itself, since return from a signal handler can lead to re-execution of an I/O operation, the meaning of the program can be undefined.

Note that, because execution picks up at the machine instruction where the problem occurred, that it may not be possible to correct the problem. If the issue is the value of a variable, for example, that variable's memory location may have already been copied into a function, converted to a different type, or read into a register and put on the bus. Changing the value of a variable after any of these actions will have no impact on the program's ability to continue.

The following example demonstrates using a signal handler in C to exit a loop upon receiving a cntl-c on the terminal. While the code causes the program to exit, the terminal displays extra characters in the output that depend upon when the user types cntl-c, which makes the meaning of the program undefined in a technical sense. Replacing the `return` in the handler with an `exit(0)` terminates the program cleanly.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

int quit = 0;

void handler( int signal ) {
  printf("\nCaught signal %d\n", signal);
  quit = 1;
  return; // replace with exit(-1) to terminate
}

int main(int argc, char *argv[]) {
  signal( SIGINT, handler );

  while(!quit) {
    printf("blah\n");
  }
  printf("\nCleaning up\n");

  return(0);
}
```

The following shows the problem with trying to continue execution from the location where the fault occurred. This program enters an infinite loop that generates a continuous stream of bus errors.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void seghandler( int signal ) {
  printf("\nCaught seg fault %d\n", signal );
  return; // replace with exit(-1) to terminate
}

void bushandler( int signal ) {
  printf("\nCaught bus error %d\n", signal );
  return; // replace with exit(-1) to terminate
}

int main(int argc, char *argv[]) {
  float *x = NULL;

  signal( SIGSEGV, seghandler );
  signal( SIGBUS, bushandler );

  printf("%.2f\n", x[0]);

  return(0);
}
```

### 5.5.2 Assertions

Assertions are a formal language structure designed to replace informal error testing.

For example, a common task in programming is to evaluate the inputs to a function for correctness. Statements of the form

```
if( c < 0 || c > CMax) {
  fprintf( stderr, "input is out of bounds [0, \%d] with value \%d\n", CMax, c);
  return(-1);
}
```

are extremely common. The example demonstrates one of the conventions in C, which is to return 0 upon successful completion of the function in cases where the function's return value is not intentional. Negative numbers are traditionally used to represent errors that might occur during execution.

Given the ability to throw exceptions in C++, the programmer might use a `throw` statement instead of returning an error condition in the example above, to enable discontinuous program execution within a hierarchical error tree.

An assertion is a formal language structure designed to simplify error checking and ensure that a certain condition is true at a certain place in program execution. The `assert` structure arose from a desire to match programs with formal representations of program semantics that require pre- and post-conditions on the state of the computer.

In Java, the `assert` statement has the following forms. In the first case, the assertion is simply a test. If the test fails, the assertion throws an AssertionError exception. In the second case, if the test fails, the assertion throws an Assertion Error exception and passes to it the string.

```
assert (c < 0 || c > CMax);
assert (c < 0 || c > CMax) : "input is out of bounds";
```

A programmer can choose to let the AssertionError cause program execution to halt or catch the Assertion-Error in a try/catch structure. In the latter case, the message to the AssertionError does not get printed to the terminal as the exception is handled quietly.

By default, Java does not enable assertions. At run time, you can enable assertions by using the flag `-ea` when executing the JVM.

In Python, the `assert` statement works identically to Java. Note that the second expression need not be a string. It can by any generic expression with no side-effects.

```
assert c < 0 || c > CMax
assert c < 0 || c > CMax : "input is out of bounds"
```

In C++, `assert()` is a macro that evaluates its argument and halts execution if the expression is false. The macro prints out both the expression and the source code line on which the assertion failed.

The macro is defined in the include file `assert.h`. A programmer can turn off assertions by putting the statement

```
#define NDEBUG
```

before the inclusion of the assert.h file. Assertions make the debugging stage easier, but can easily be turned off for optimized compilation.

# 6 Functions

Functions are a key capability of programming languages. They enable a number of good things.

- Parameterization of functional blocks, which helps avoid code repetition.
- Encapsulation of code, which helps avoid code repetition.
- Modular, top-down design, which facilitates building large software systems.
- Unit testing of code modules, which facilitates robust software systems.
- Portability and re-use

Note that some people claim functions are the ultimate reusable unit, not objects.

## 6.1 Terminology

**Procedure / Subroutine**: A call/return block of code that does not return a value, but may take arguments and modify their value.

**Function**: A call/return block of code that returns a value. Functions may take arguments and modify their value.

**Method**: A function or subroutine that is part of a class.

**Argument**: An expression used when calling a function or subroutine.

**Parameter**: The identifier declared in a function or subroutine definition that will connect to the corresponding argument in a function call.

For the remainder of these notes, I'll use function to mean function, procedure, or subroutine.

There are a number of methods for connecting arguments and parameters. This is a critical part of the semantics of a language, as it greatly affects the meaning of a program.

- Pass by value: the computer evaluates the argument expression and places a copy of its value into the memory address referenced by the parameter.
- Pass by reference: the computer evaluates the argument expression and places a reference to the result into the memory address referenced by the parameter.
- Pass by value-result: the computer evaluates the argument expression to a memory location and then copies its value to the memory address referenced by the parameter. When the function returns, the computer copies the value of the parameter back to the location holding the argument.
- Pass by name: the compiler textually substitutes the argument expression for the parameter within the function call.

Pass by name creates code that is difficult to understand, and which has different meanings depending upon its arguments. Algol 68 used pass by name, but none of its successor languages have.

Ada is one language that specifies usage of each parameter and actually lets the compiler decide the mechanism of implementation. The compiler is free to make variables of type in-out use pass by reference or pass by value-result. VHDL is another language that requires the programmer to specify parameter usage.

## 6.2 Function Optimization

Specifying parameter usage is one method of giving the compiler more information it can use to optimize code. For example, if function is using a pointer or reference only to access data, not change it, then the compiler is free to do things like pre-load the required data and throw it out when the function completes.

C++ includes several features intended to improve compiler optimization. For example, the `const` keyword gives the programmer the ability to specify parameter, field, or variable usage in a function, class, or declaration, respectively.

The `const` keyword applied to a parameter means the parameter will not be modified within the function, even if the parameter is a reference. The following example shows how we could avoid copying an object by passing it as a const reference parameter. Since the function uses the object fields only as r-values, the compiler knows that the function does not modify `a` or `b`.

```
int sum( const MyObject &a, const MyObject &b ) {
  return(a.x + b.x);
}
```

What if we want to call methods on the two MyObject variables? The compiler has to be able to enforce the const modifier on the two objects.

To handle this situation, C++ permits methods to declare themselves as `const`, which means the function has no side-effects on the object itself. The ability to declare methods as `const` is essential for the `const` modifier for reference parameters to be meaningful for objects. As in the example below, if the programmer calls a method on an object that is a const reference parameter, the compiler must be able to guarantee that the method call does not change the object. Without the ability to declare methods as `const`, the programmer cannot call any object methods if the object is passed as a const reference. In the example below, the `get` method is specified as `const` so the compiler does not complain when it gets called in the `sum` function. If the `sum` function had to access the field directly, as in the above example, it would defeat the purpose of data encapsulation in objects.

```
class MyObject {
  int x;

public:
  int get(void) const {
    return( x );
  }
};

int sum( const MyObject &a, const MyObject &b ) {
  return(a.get() + b.get());
}
```

Another modifier used by C++ to enable compiler optimization is the keyword `inline`, which can be used when defining class methods in the class definition. The `inline` keyword is only a suggestion to the compiler, not a requirement. If the compiler decides to make the function inline, then, whenever the function is called, the compiler would not actually generate a function call. Instead, it would substitute the code from the function into the code from the calling program, making sure that the overall behavior was identical to an actual function call. With respect to formal semantics, the `inline` keyword should have no effect. However, it can have a significant effect on running time if the cost of the function call is large relative to the function's body.

## 6.3 Function call implementation

Function calls require the computer to transfer data from the parent context to the function context, then transfer execution control to the function. When the function is done executing, it must transfer data back to the parent context and then transfer execution control. The standard approach to implementing these capabilities is to use an activation record. The parent context builds part of the activation record, the function builds the rest. The function has access to the whole activation record, enabling information transfer between the two contexts.

Languages other than older versions of Fortran–which did not permit recursion–must dynamically create activation records. Generally, the computer uses the system or program stack to hold activation records, and explicit support for them exists in almost all hardware instruction sets, even in RISC machines.

Activation records, also called stack frames, need to hold the following types of information.

- Return address
- Return value
- Arguments
- Parameters
- Local variables
- A pointer to the prior activation record
- A pointer to the function's static context (e.g. class context or base context)
- Saved registers

Which context saves which pieces? It depends partly upon convention, partly upon which context has access to the information when. The return address is the only piece actually required to occur at the level of hardware.

**Parent context**:

- Push space for the return value
- Push on the arguments
- Push on the local stack frame (activation record) pointer
- Push on the static context reference
- Call the function (pushes on return address)

**Function context**:

- Set up its own stack frame pointer (all information is relative to this location)
- Push on space for local variables
- Push on any registers the function will modify
- Do its thing
- Copy the return value to its location on the stack

- Restore registers
- Remove local variables
- Return (pops return address)

**Parent context**:

- Pop the static context
- Restore the stack frame pointer
- Pop the arguments
- Pop and store the return value

Note that responsibility for several of the pieces is based on convention. The function is generally responsible for saving any registers it modifies. In the example above I gave the parent context the responsibility of handling the stack frame, but that also could be treated by the function as a register that needs saving.

## 6.4 Function semantics

Implementing a function is similar to our discussion of for loop implementation and handling scope inside a block. Somehow we have to add the local variables and parameters to the computer's state when entering the function and remove them on exit. Mathematically, we can express a void function with the following semantic expression, where C is the parent call data, F is the function data, and S is state.

$$\begin{aligned} M(C,F,S) = RemoveActivationRecord(&F.params,\\ &F.locals,\\ &M(F.body,\\ &\ ByValue(F.params,\\ &\ \ C.args,\\ &\ \ AddActivationRecord(F.locals, F.params, S)))) \end{aligned} \tag{27}$$

The function is the result of removing the local variables from the result of applying the body to the state after adding the local variables and connecting the parameters with the function arguments.

Another way to view it is to push the current state, making a new blank symbol table, add the parameters connected with the function arguments to the new table, execute the body of the function on the state, and then remove the symbol table with the local variables and parameters from the function.

```
def MFunc( c, f, state ):
  return MPop( MStatement( f.body, MByValue( f.params, c.args, MPush( state ))))
```

For a function to return a value, we have to create a space for it in the parent symbol table so it will persist after the function call.

```
def MFunc( c, f, state ):
  return MPop( MStatement( f.body,
                           MByValue( f.params, c.args,
                                     MPush( MAddIdentifier( f.retval, state )))))
```

# 7 Memory Management

Memory organization is a matter of convention. The memory of a program consists of four types of information.

- The program code
- Statically allocated variables with global lifetime (everything, in Fortran)
- Program stack, used for local variables and function calls
- Program heap, used for dynamically allocated variables

Generally, the stack is in high memory and grows down towards the heap. The heap is in low memory and grows up towards the stack. Static variables may be above the stack or below the heap.

In modern CPUs and operating systems with virtual memory, each program may have its own complete virtual memory space. Virtual memory allows many different memory spaces to co-exist in the same physical memory space. The CPU contains registers that specify which virtual memory table it should use to discover the physical address of a virtual memory address.

Most interpreted or virtual machine languages (e.g. PHP, Python, Java) have a fixed amount of system stack space, usually defined in a parameter file. When an interpreted language hits the stack space limit the interpreter generally throws an exception and terminates.

Compiled languages depend upon the operating system to determine when a stack overflow has occurred. Older operating systems without adequate memory management would often simply fail as the stack grew over important code, causing an unrecoverable system-wide crash. Modern operating systems and memory systems give each program its own virtual stack space; when a program overflows its own stack the OS can isolate the program and terminate it.

Deep, or incorrectly written recursion is the primary reason for stack overflow errors. Typical program execution rarely executes function calls more than 10-deep. The SPARC CPU took advantage of this characteristic to design a hardware stack on the CPU that could handle some number of function calls-typically seven or eight–without needing to write data to the stack in main memory. While the actual implementation turned out to be a poor design decision, the analysis of programs that led to the design is valid: most programs rarely use deep recursion.

The heap, on the other hand, is heavily used by most programs that have dynamic memory needs. Because the heap is the general purpose dynamic memory for a program, it must efficiently support requests for memory blocks of all sizes and be able to effectively recover freed memory. The data structures required for the heap are significantly more complex than the program or system stack.

## 7.1 Dynamic allocation

Even languages without automatic memory management must use information not available to the programmer to manage memory. Consider the memory allocation functions `malloc` and `free`, which are part of the standard C library.

- `malloc` must obtain an unused section of the heap of the requested size and return a pointer to the beginning of that area (growing up).
- `free` must mark the area of the heap used by the given pointer as unused.

The `malloc` function must be able to identify which areas of the heap are in use, which are not, and mark an unused area as ”in use”. The `free` function must be able to mark the area used by the pointer as ”unused” given only a copy of the value of the pointer variable.

What information about each allocated area do these functions require to do their job? It turns out that they don’t need much.

- Starting address of the used memory
- Size of the used memory

However, these functions also need a data structure in which to hold information about currently allocated memory and some idea of where unused memory is located. They could use an array to hold the information, but given that memory is often allocated and deleted, an array would quickly become inefficient. Instead, a dictionary or hashmap, with the pointer address as the index, offers a more useful data structure. Adding or deleting a pointer from the hashmap is then a constant time operation.

Identifying where free memory is located is more challenging. `malloc` shouldn’t spend time searching from the start of the heap for a space big enough to hold the request, although that would be efficient in terms of memory usage. Instead, it might be easier to maintain a free space pointer to the first memory location of the free space beyond the last memory request. Each time the program makes a memory request, the new memory comes from the beginning of the free space and the free space pointer moves forward by the requested amount.

Given virtual memory, the huge size of virtual memory, and the fact that paging efficiently maps virtual memory to physical memory, such a strategy is not actually that bad, at least until a program is allocating and deallocating huge blocks of memory repeatedly. In that case, the memory management functions may want to keep track of freed memory behind the free space pointer.

The glibc `malloc` uses a data structure called a chunk to hold memory information. Each used or unused block has a header section and then a memory space of some size. Chunks in use have information about their size and possibly the type of the chunk (how it is being used). All of the unused chunks form a doubly-linked list, also grouped into bins of similar size, where the next/prev pointers are stored in a header section of the chunk. The last chunk in the list is effectively the free space pointer mentioned above. A call to `free` puts the chunk into the list of unused memory in the proper bin according to its size.

There are many other `malloc` implementations, and implementations within the kernel must be tightly integrated with the OS and architecture’s virtual memory systems. What should be clear, however, is that a call to `malloc` can be an expensive operation. Hence, Fortran’s use of static variable allocation is a contributor to its speed.

## 7.2 Handles

Handles were/are a mechanism for dynamic memory allocation that permit the operating system to compress the heap during run-time. A Handle is a double pointer to an arbitrary chunk of memory. For example, the following would ask for a Handle to a chunk of memory large enough to hold 500 floats.

```
myhandle = Handle( sizeof(float) * 500 );
```

The Handle call is actually an operating system call. The OS maintains a list of pointers to which it can assign arbitrary memory addresses. The Handle call allocates the requested space in the program's memory space, assigns the address of the space to one of its pointers, and then returns the address of one of those pointers. Therefore, the Handle itself is a double pointer. The Handle points to one of the OS pointers, which points to a chunk of memory.

Why the indirection? Since the OS has control over the list of pointers, it can arbitrarily decide to move around the memory referenced by the handle. It can halt the program's execution, move the chunk of memory from one place to another–e.g. compact the heap–and then update the pointer. Since the program has only the address of the pointer, any double dereference still works fine.

The programmer still has to be careful about memory accesses. For example, the following code could potentially have problems.

```
Handle hnd = new Handle( sizeof(int) * 100 );

int *ptr = *hnd;
for(i=0;i<100;i++) {
  ptr[i] = i * 2;
}
```

If the OS does not compact memory during the for loop, then the above code will run as expected. However, if the OS decides to compact the heap during the for loop execution, then the `ptr` will no longer be pointing to the array of 100 ints. If the program used a double-dereference in the inner loop `(*hnd)[i]` then the program would be fine, as the assignment is an atomic instruction. A better method, however, is to use another OS call to lock the Handle prior to the for loop and unlock it afterwards, which prevents the OS from moving the memory associated with that Handle.

```
Handle hnd = new Handle( sizeof(int) * 100 );

HLock( hnd );
int *ptr = *hnd;
for(i=0;i<100;i++) {
  ptr[i] = i * 2;
}
HUnlock( hnd );
```

Handles were a key component of the Macintosh OS up through at least system 7, and are still supported in legacy code. Prior to moving to a Unix base and a paged virtual memory system, the MacOS relied on Handles to enable heap compaction by the OS.

## 7.3 Garbage Collection

A programming language requires garbage collection if it does not require the programmer to explicitly free memory. Java, for example, requires the programmer to allocate objects, but has no programmer controlled mechanism for freeing the allocated memory. Instead, a separate process must run in the background to clean up objects that are no longer in use.

How does a garbage collection system know when an object is no longer in use? When should garbage collection occur? There are many different strategies in use. The following are three general strategies. Specific garbage collection systems may be hybrids.

- Reference Counting: The heap consists of a set of nodes, where each node has a counter that specifies the number of references to that node. All unused nodes are in a single linked list. Assignments involving references must increment the counter of the node specified by the r-value and possibly decrement the counter of a different node. Any time the program deletes a reference to a node, or the reference's lifetime ends, the program has to decrement the reference counter of the node.

  If a counter becomes zero, because of either an assignment or a deletion, then the program must traverse any children it references and see if their counters are now also zero. Any nodes with zero counters go back to the free list.

  Reference counting adds overhead to assignment statements, but does not require time in large chunks to clean up memory. It never has to go over every node in the heap, for example, but only those that are in use. As such, the interpeter/garbage collection thread could organize the heap as in the glibc example above, using reference counting to determine when to put a chunk back into the free list.

  The weakness of reference counting is that it cannot detect rings of nodes that are no longer accessible to the program. In a circular data structure, each node will always have at least one reference to it.

- Mark-Sweep: As with reference counting, the heap consists of a set of nodes. All nodes begin on the free list. Instead of a counter, each node has a single boolean that is initially zero. The mark-sweep algorithm runs only when the heap is full (there are no more nodes on the free list).

  The algorithm is simple: starting with each reference in the active symbol table, follow that reference through memory, setting each visited node's boolean marker to 1. Once the reference traversal is complete, run through the entire heap and put any node who's marker is 0 back on the free list. The mark-sweep algorithm has the benefit of running only when the heap is full (which is a rare occurrence for many programs) and it clears out all free memory.

- Copy collection: The heap is divided into two equal parts in copy collection. The program actively uses only one half at any given time. When the current half of the heap runs out of space, the garbage collection system makes one pass through the symbol table variables, moving all active memory into the other half of the heap and packing them tightly. Once complete, the active memory half switches and the program resumes. Nodes without any references in the active symbol table get left behind in the old half of memory. Copy collection is expensive in terms of memory, but it is faster than mark-sweep since it traverses the heap only once and does not have to do anything explicit to free orphaned memory.

Actual implementations of garbage collection may combine the three strategies. Python, for example, uses a variation of reference counting.

# 8 Concurrent Programming

Concurrent programming is the use of multiple **threads** to execute a single task. A thread is a sequence of instructions that has its own stack and CPU state. Concurrent threads share at least one variable of their state. Concurrent programming is probably the most challenging programming paradigm, because the individual instructions of different threads are not synchronized. Therefore, many bad things can occur if concurrent programs are not written properly.

Two of the most common problems are **deadlock** and **race conditions**.

- Deadlock occurs when a ring of threads comes to a point in the program where each needs a resource from the next thread in the ring in order to continue. No thread can provide the resource because each thread is waiting for another thread.
- A race condition occurs when the meaning of a program depends upon the order in which each thread accesses a shared variable. For example, consider two threads, each executing the following code.

  ```
  1 int nextfree = endOfBuffer;
  2 endOfBuffer++;
  3 buffer[nextfree] = someValue;
  ```

  If thread A executes the first two instructions in sequence and then thread B executes the first two instructions in sequence, each will assign their value to the appropriate buffer location. However, if thread A executes instruction 1 and then thread B executes instruction 1, then thread A executes instructions 2 and 3 and then thread B executes instructions 2 and 3, both threads will write to the same location and thread B's value will overwrite thread A's value. Because the threads are running concurrently, possibly on different processors, it is impossible to explicitly synchronize execution at the instruction level.

Most modern languages, and many old ones, provide support for concurrent programming in standard libraries. The programmer is responsible for spawning the threads, synchronizing them, and terminating them. The language itself does not provide explicit support for concurrent programs.

## 8.1 Synchronization

Clearly, if multiple threads depend on common variables, the programmer must have some method of synchronizing access to the variables. Sections of a program where a thread must access (read or write) a shared variable are called critical sections. If a thread has to execute multiple atomic instructions on a shared variable it must lock out other threads until it is done with its critical section. Even the expression `a = a + 1` could be dangerous if the compiler separates the operation into a read operation on `a` followed by an add operation on `a`. Some RISC computers, for example, would require that sequence of instructions. Since operations are atomic only at the level of machine instructions, a thread has to be able to lock `a` until both the access and modification are complete.

There are many different strategies used by programmers to manage critical sections. One of the oldest, and most common methods is to use a **semaphore** (Dijkstra, 1968). A semaphore uses two functions $P(s)$ and $V(s)$–both of which must be atomic–to control access to a shared resource. The variable $s$ is the semaphore signal variable. Control of a single resource requires two semaphores, just as access to a one-lane construction area or single train track requires two separate signals.

The functionality of the P(s) and V(s) functions are as follows.

- P(s): if $s > 0$ then assign `s = s -1`, else block the thread that called P.
- V(s): if a thread T is blocked on the semaphore `s`, then wake up T, otherwise assign $s = s + 1$.

Consider the two programs below. The two semaphore signals are `empty` and `full`. The first program is a producer of data, and the second program is a consumer of data.

```
// initial state
Semaphore empty = 1;
Semaphore full = 0;
Thing commonBuffer;
```

**Thread A**

```
while( true ) {
  Thing value = f();
  P( empty );
  commonBuffer = value;
  V( full );
}
```

**Thread B**

```
while( true ) {
  P( full );
  Thing value = commonBuffer;
  V( empty );
  g( value );
}
```

If both thread A and B are running, then either one could reach their call to P(s) first. If thread B reaches its call, then P(full) blocks thread B from continuing and it hands off control to thread A. When thread A reaches P(empty) it sets empty to 0 and continues into its critical section. When it reaches V(full), it increments full to 1 (thread B is not blocked on the full semaphore, just the empty semaphore). Thread A then begins its second loop. When it hits P(empty), however, it blocks and control passes to thread B, which decrements full and executes its critical section. Then thread B executes V(empty), which sets empty to 1. When thread B begins its loop again, it blocks on P(full), which is 0 and control passes to thread A.

The key to semaphores is that each process increments only one semaphore and decrements only one semaphore. Therefore, each process depends upon the other process to execute the opposite operation.

Most languages that permit parallelism use system classes or library calls to access semaphores provided by the operating system. There are bus operations (e.g. PCI bus) that support multi-processor semaphores and other signaling mechanisms.

## 8.2 Threads in C

As noted above, a thread is a sequence of operations running independently, but sharing state with another independently running sequence of operations. In C/C++ the pthread library provides support for concurrent programming. The pthread library includes support for creating, synchronizing, and deleting threads. Threads have access to the same global program state as the parent program, and the parent program can pass in pointers to any other variables in its scope.

For example, the following code shows how to split up a computation in separate threads. In this case, the threads do not need to communicate, but the parent program halts until each thread is finished with its work. This type of synchronization is a form of synchronization called a **barrier**, which means that no thread continues until all threads reach this point in the program. When all threads have reached the barrier, the parent thread could have more work for the child threads to do after collecting their results.

```c
/* example of using threads to split up the work */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

typedef struct {
  double *array;
  int N;
  double sum;
} threadInfo;

void *fsum( void *ud ) {
  threadInfo *ti = (threadInfo *)ud;
  int i;

  for(i=0;i<ti->N;i++) {
    ti->sum += ti->array[i];
  }
}

int main(int argc, char *argv[]) {
  const int size = 1000000;
  int i, mux;
  double *bigarray = malloc( sizeof(double) * size );
  double sum;
  threadInfo ti[4];
  pthread_t thread[4];

  // make a big array
  for(i=0;i<size;i++) {
    bigarray[i] = drand48();
  }
  // split up the work
  for(i=0;i<4;i++) {
    ti[i].N = size/4;
    ti[i].array = &( bigarray[i*(size/4)] );
    ti[i].sum = 0.0;
  }
  // call the threads
  for(i=0;i<4;i++) {
    pthread_create(&(thread[i]), NULL, fsum, &(ti[i]) );
  }
  // join the threads
  for(i=0;i<4;i++) {
    pthread_join( thread[i], NULL );
  }
  // sum the results
  sum = ti[0].sum + ti[1].sum + ti[2].sum + ti[3].sum;
  printf("sum %.2f\n", sum/size );

  return(0);
}
```

The pthread library also contains support for a form of semaphore called a mutex lock. A mutex lock is a variable that represents a key. The key controls access to a critical resource. Each thread must lock the resource before modifying it using the call `pthread_mutex_lock` and release it when finished using `pthread_mutex_unlock`. A mutex is itself a semaphore with the value 0 or 1, and you can use a mutex lock to implement a multi-valued semaphore.

In the example below, the main program spawns four processes that are all supposed to increment a common counter variable.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void *fcounter(void *);
void *fbad(void *);
int counter = 0;

typedef struct {
  pthread_mutex_t mutex;
  int id;
} threadInfo;

int main(int argc, char *argv[]) {
  int i;
  pthread_mutex_t mutex;
  pthread_t thread[4];
  threadInfo ti[4];

  // init the shared mutex lock
  pthread_mutex_init( &(mutex), NULL );

  // spawn four threads
  for(i=0;i<4;i++) {
    ti[i].id = i;
    ti[i].mutex = mutex;
    pthread_create( &(thread[i]), NULL, fcounter, &(ti[i]) );
  }

  // wait until all four are done (join)
  for(i=0;i<4;i++) {
    pthread_join( thread[i], NULL );
  }

  // free the mutex lock
  pthread_mutex_destroy( &(mutex) );

  return(0);
}
```

If we implement the fcounter function as below, then the number printed out by the function will be unique, as each thread increments the counter before printing. The mutex lock guarantees that no other thread will use the counter for reading or writing while the mutex variable is locked.

```
void *fcounter( void *userdata ) {
  int i;
  threadInfo *ti = (threadInfo *)userdata;

  for(i=0;i<3;i++) {
    pthread_mutex_lock( &(ti->mutex) );
    counter++;
    printf("%d counter is %d\n", ti->id, counter );
    pthread_mutex_unlock( &(ti->mutex) );
    usleep( rand() % 10 );
  }
  return( NULL );
}
```

On the other hand, if we use the function below, fbad, then the value printed out by the function may be the same as a value printed by another function, since the threads will interleave the increment and the print statement.

```
void *fbad( void *userdata ) {
  int i;
  threadInfo *ti = (threadInfo *)userdata;

  for(i=0;i<3;i++) {
    counter++;
    usleep( rand() % 10 );
    printf("%d counter is %d\n", ti->id, counter );
  }
  return( NULL );
}
```

The course web site also contains code showing a producer-consumer example using mutex locks, similar to the Java threads example in the textbook.

## 8.3 Threads in Java/Python

Java and Python uses classes to implement threads. The class encapsulates a program that will run independent of the parent program. The parent program can start the thread, stop the thread, wait for the thread to complete, and communicate with the thread using methods on the thread class.

The functionality of the thread class methods is similar to the functionality of the C thread API (not surprisingly). The following two examples demonstrate using two threads to sum the values in an array. The first example is in Java, the second in Python.

It's important to note that threads in Python and Java do not necessarily have the same performance gain as threads in C. For both languages, the interpreter runs in its own thread, perhaps in multiple threads (e.g. garbage collection in Java). Therefore, on a dual processor system there may be no performance gain by using multiple threads in a dynamic language. Instead, their primary purpose on one and two cpu systems is to enable asynchronous processes, such as user input response routines.

```java
import java.lang.Thread;
import java.util.Random;

public class dualsum extends Thread {
    int start;
    int extent;
    double bigarray[];
    double sum;

    public dualsum( int istart, int iextent, double ibigarray[] ) {
        start = istart;
        extent = iextent;
        bigarray = ibigarray;
        sum = 0.0;
    }

    public void run() { // called when the parent program calls start
        int end = start + extent;
        for(int i = start;i<end;i++) {
            sum += bigarray[i];
        }
    }

    public double sum() { // methods can access/set variables
        return sum;
    }

    public static void main(String argv[]) {
        final int size = 100;
        double ba[] = new double[ size ];
        Random gen = new Random();

        for(int i=0;i<size;i++) {
            ba[i] = gen.nextDouble();
        }
        // split the problem into two parts for two threads
        dualsum ds1 = new dualsum( 0, size/2, ba );
        dualsum ds2 = new dualsum( size/2, size/2, ba );

        ds1.start(); // Thread class, calls the run method
        ds2.start();
        try {
            ds1.join(); // Thread class, waits until the run method is complete
            ds2.join();
        }
        catch(InterruptedException e ) {
            return;
        }
        double sum = ds1.sum() + ds2.sum() ;
        System.out.printf("avg %.2f\n", sum/size);
    }
}
```

Figure 3: Threading in Java

```
import threading
import time

# inherit the Thread class
class dualsum(threading.Thread):

    def __init__(self, istart, iextent, ilist): # set up the task
        threading.Thread.__init__(self)

        self.begin = istart
        self.extent = iextent
        self.list = ilist
        self.sum = 0.0

    def run(self): # called by the parent start method
        for j in range( 0, 10000 ):
            for i in range( self.begin, self.begin+self.extent ):
                self.sum += self.list[i]

    def getsum(self):
        return self.sum

def main():

    size = 100
    biglist = []
    for i in range( 0, size ):
        biglist.append( i )

    # set up the problem as two separate parts
    ds1 = dualsum( 0, size/2, biglist )
    ds2 = dualsum( size/2, size/2, biglist )

    # call the parent Thread start method to execute the run method
    ds1.start()
    ds2.start()

    # wait for the run methods to complete
    ds1.join()
    ds2.join()

    print "dual sum is ", ds1.getsum() + ds2.getsum()

if __name__ == "__main__":
    main()
```

Figure 4: Threading in Python

## 8.4 Explicitly Concurrent Languages

There are programming languages that are explicitly parallel, or that provide support for concurrent programming as part of the language definition. An example of one such language is SA-C, which was designed to facilitate writing image processing operations for execution on a Field Programmable Gate Array [FPGA]. An FPGA is a configurable digital logic device that can implement massively parallel operations such as a gradient operation on a all pixels of an image. Consider the following example.

```
int16[:,:] main (uint8 Image[:,:]) {
  int16 H[3,3] = {{-1, -1, -1}, {0, 0, 0}, {1, 1, 1}};
  int16 V[3, 3] = {{-1, 0, 1}, {-1, 0, 1}, {-1, 0, 1}};

  int16 M[:, :] =
    for window W[3, 3] in Image {
      int16 dfdy, int16 dfdx =
        for h in H dot w in W dot v in V
          return( sum(h * w), sum(v * w) );

      int16 magnitude = sqrt(dfdy*dfdy + dfdx*dfdx);

    } return( array( magnitude ) );
} return( M );
```

SA-C is a functional programming languages. Variables represent a fixed value, not memory locations. The scope of a variable is defined by when it is assigned a value. All variables in SA-C are arrays, including scalars. Function calls, loops, and conditionals are expressions, not statements; all expressions return values.

In the example above, the parameter for the function is a 2D Image. The variables H and V are defined as 3x3 arrays with the given values. The main body of the function is a single assignment statement.

The right side of the assignment is a for loop over all 3x3 windows within the Image. The body of the loop consists of two assignments. The first is the result of a nested loop over the entries in H, W, and V. The `dot` keyword means that the traversal through H, W and V is in unison, and the arrays must all be the same shape. The return value for the loop is the sum over all entries in H, W, and V of the products of H and W and V and W. The code implements a simple dot product of each overlaid pair of windows.

The second assignment calculates the square root of the sum of squares of the prior computation. The expression `array(magnitude)` tells the compiler to collect all of the magnitude values in an array of appropriate size and return that as the value to be assigned to M. The return value of the function is M.

The result of compiling the SA-C code is an intermediate representation as VHDL. A programmer can simulate the VHDL code on an chip model and subsequently compile and download it to chip hardware.

Another explicitly parallel programming language is Erlang, which has been used in cell phone networks to manage telephone exchanges. Like SA-C, Erlang is also a single-assignment language. However, programs in Erlang are more similar to Prolog, as it uses pattern matching to determine the meaning of an expression.

### 8.4.1 VHDL

VHSIC Hardware Description Language [VHDL] is a language used to define and describe the behavior of digital circuits. Unlike most other programming languages, VHDL is explicitly parallel. It must be able to describe parallel computations because VHDL programs often describe the behavior of thousands, if not millions of gates, each gate acting independently as a function of its inputs and outputs.

VHDL programs have two parts.

- `entity` - the entity section is like a prototype in C, or an interface in Java. It contains a `port` statement that describes the name of the circuit and its inputs and outputs. All inputs and outputs are signals, which you can think of as wires. Each input or output must be labeled as to its usage (`IN, OUT, INOUT`) and type.
- `architecture` - the architecture is the circuit definition. The actual code that describes the circuit goes here. There are many ways to describe circuits in VHDL. A **netlist** definition instantiates gates and specifies which wires connect which inputs and outputs. A **dataflow** representation describes the flow of data through the circuit. A **behavioral** description uses more complex control structures to specify the functionality of the circuit over time.

**Types**

There are built-in types for VHDL, but they are not commonly used. Instead, most programs use a set of types defined by a IEEE standard library. The base type is `std_logic`, which represents one bit. The `std_logic` type uses the characters `'0'` and `'1'` to represent binary values. In addition, the `std_logic` type can take on values such as `'X'`, which is undefined, or `'H'` or `'L'`, which correspond to weak high and low values, which are important in some circuit designs.

VHDL has a flexible array mechanism that permits arbitrary range indexing with specification of the indexing direction: left or right with increasing index value. The basic IEEE array type is the `std_logic_vector`, which is an array of type `std_logic`. The following example shows an 8 bit register, indexed from 7 to 0, with 7 being the index of the leftmost bit, or the most significant bit in the array.

```
signal register:  std_logic_vector( 7 downto 0 );
```

The IEEE library also defines vector types for which addition and subtraction are defined. These types are `signed` and `unsigned`, and they require a range when defined, as below. The range of the indices determines the number of bits required by the signal.

```
signal counter:  unsigned( 7 downto 0 );
```

Note that, because VHDL represents hardware, the programmer must specify the number of bits used in the type definition. An `unsigned` type does not have a pre-defined precision. If you need 64 bits of information, then you define the variable to have a range covering 64 bits. If you just need 3 bits, then you define the variable to have 3 bits.

Why not use the built-in VHDL types? The primary reason is that the IEEE types represent a standard that clearly states the hardware behavior of each type in the IEEE library. Simulators, therefore, ensure that the IEEE types work according to the standard, and the standard ensures that if you use your VHDL code to synthesize a real circuit, the behavior in simulation matches the behavior of the actual circuit.

If you don't care about creating hardware, there are a number of VHDL data types and control structures you can use.

## Statements

Statements in VHDL all run continuously, in parallel. There are three major types of statements: assignments, port maps, and processes.

An assignment statement is, in effect, a wire connecting two nodes of the circuit. Any change in the right side of an assignment immediately affects the left side.

There are several variations of assignment statement, however, including conditional signal assignments that allow an exterior signal to control which value gets routed to the left-hand symbol.

```
-- connects the value of B to the value of A with a wire
A <= B;

-- connects the value of B to the value of A with a wire and a time delay
A <= B after 5 ns;

-- selects B or (not B) to connect to A depending upon signal S
A <= B when S = '0' else (not B);

-- combines a switch and delay
A <= B after 5 ns when S = '0' else (not B) after 5 ns
```

The `port map` statement is similar to creating an instance of a particular class. A port map creates an instance of an entity and connects the inputs and outputs of the entity with local signals. The syntax of a port map is as follows.

```
<label> : <entity> port map ( <argument list> );

-- example
A0: andgate port map (I1 => A, I2 => B, Q => C);
```

In the example code, the label of the instance, which must be unique within scope, is `A0`, the entity, defined elsewhere, is called `andgate`, and it has three arguments. The => symbol connects fhe formal parameter names `I1, I2, Q` to their local signals `A, B, C`. Once port mapped, the andgate entity becomes part of the VHDL program and runs in parallel to other statements. Any time the inputs A or B change, the andgate entity executes and updates the value of C.

Just as you can create many instances of a class, so you can create many instances of an entity using a port map. If you need ten andgate circuits, you can port map the entity ten times, linking them together using local signals. Each instance requires a unique label.

The `process` statement is somewhat more similar to a procedure or block statement. The purpose of a process is to permit the programmer to write a functional description of the circuit using more traditional control flow techniques.

Inside a process statement you can use structures like for loops and if-then-else branching. You can also use variables, as opposed to signals, but the use of variables is not recommended as it is difficult to compile code using variables into hardware.

When the process executes, the statements in the process get interpreted in serial order, which permits control flow statements to act as usual. However, despite the apparently normality of control flow and signal assignments inside a process, they are still not like a regular programming language.

The primary difference is how VHDL handles signal assignments inside a process. At the beginning of the process, each signal used in the process has a value. That value does not change as the system executes the behavior defined in the process. Instead, any assignments calculate the value of the right hand side based on the current value of the signals and then schedule that value for assignment when the process completes. One of the issues that arises is that a signal should receive a new value only once within one traversal of a process block. There may be many assignment statements that give a new value to a signal, but only one of those should actually execute given the control flow.

Note the similarity of the recommendation that a signal be assigned a new value only once within a process block and the same requirement in single-assignment C. In both cases, parallel computation makes multiple assignments to the same value difficult to resolve.

Unlike SA-C, however, VHDL does not forbid multiple assignments to a signal. One of the attributes of the IEEE std_logic package types is that a subset of them contain resolution tables that specify the proper value when a signal receives two inputs simultaneously. In hardware, this enables bus-style hardware where any one of a number of signals can pull the bus low, but in the absence of a low signal the bus floats high. As noted earlier, the `std_logic` type has a number of possible values besides 1 and 0. An `'H'` and a 0, for example, resolve to 0 because the `'H'` is a weak high.

Consider the process example below. First, note that a process must have a sensitivity list. The sensitivity list indicates which signals should cause the process to execute. The instructions inside the process only execute when a signaling variable changes. The process statement, therefore, is how you implement variables that change state only when an event occurs, such as a clock signal changing. Looking at the process itself, the if-statement is sensitive to the reset and clk signals, so they are in the sensitivity list.

Second, note that, while there are two signal assignments for each of A and B, at most one will execute in a single pass through the process. If the reset signal is high, then the variables asynchronously receive the specified values (the clock is irrelevant). On the other hand, if the reset signal is not high, then the second condition specifies the rising edge of the clock: the clk signal is high and the clk signal changing caused the process to execute.

Finally, making the signal assignments in the rising edge case works because the assignments schedule the assignment to take place, they do not actually modify the value of A or B during the process execution. Instead, all updates take place simultaneously at the end of the process.

```
library ieee;
use ieee.std_logic_1164.all;

entity swap is
  port( reset, clk: in std_logic;
        Q: out std_logic );
end swap;

architecture test of swap is

  signal A : std_logic := '0';           -- register A
  signal B : std_logic := '1';           -- register B

begin  -- test

  main: process (reset, clk)
  begin  -- process main
    if reset = '1' then
      A <= '0';
      B <= '1';
    elsif clk = '1' and clk'event then
      A <= B;                            -- swap the values
      B <= A;                            -- works b/c swaps occurs simultaneously
    end if;
  end process main;

  Q <= A; -- runs simultaneously with the process

end test;
```

Inside a process, you can use for loops and while loops, which are most appropriately used to reduce the number of lines required to describe a process. For loops and while loops can use variables, which are not signals and do not have a physical analog. If you want to ensure your VHDL code can compile to an actual circuit, the loop variables should not occur on the right side of an assignment except as index variables. Loop variables do follow standard rules of assignment within a process (the assignment occurs at the statement in which it is made). Variables also require a different syntax `j := j + 1` rather than `A <= B`.

To use variables properly, it is important to understand their role in the circuit description. A variable is not a wire or a bit of storage. It is not available to the circuit as an actual physical value. The role of a variable is to enable the programmer to describe the circuit more efficiently, such as looping over a sequence of wires that all behave similarly. A for loop, therefore, will almost always resolve into a set of copies of a circuit, which is important to remember, since chip size is a fixed quantity.

## Examples

The following is an example of a process that incorporates an if-statement, signal assignments, and a case statement. The process describes a circuit that outputs different values depending upon the value of state, which must be a two-bit std_logic_vector.

```
process( reset, clock ) begin
  if reset = '1' then
    A <= '0';
    B <= '1';
  elsif clock = '1' and clock'event then
    case state is
    when "00" =>
      A <= '0';
      B <= not B;
    when "01" =>
      A <= '1';
      B <= not B;
    when "10" =>
      A <= B;
      B <= A;
    when others =>
      A <= '1';
      B <= '1';
    end case;
  end if;
end process;
```

A testbench is a VHDL program designed to test other VHDL entities. The example below is a testbench program for the swap entity described above. Note how the output of the `S` instance becomes the clock for the `T` instance. The delay form of the signal assignment generates the original clock and reset inputs. Since the testbench has no inputs or outputs, its entity statement is empty.

```
library ieee;
use ieee.std_logic_1164.all;

entity testswap is
end testswap;

architecture test of testswap is
  component swap
  port( reset, clk: in std_logic;
        Q: out std_logic );
  end component;

  signal clk, reset : std_logic;
  signal Q, R : std_logic;
begin  -- test
  clk <= '0', '1' after 50 ns, '0' after 100 ns, '1' after 150 ns, '0' after 200 ns,
       '1' after 250 ns, '0' after 300 ns;
  reset <= '1', '0' after 10 ns;

  S: swap port map ( reset, clk, Q );   -- these circuits are executing in parallel
  T: swap port map ( reset, Q, R );
end test;
```

# 9 Language Paradigms

## 9.1 Functional languages

A functional language approaches programming from a more mathematical point of view than an imperative or OO language. Functional languages try to avoid the concept of state and instead focus on variables as representative of values rather than memory addresses.

The analysis of program semantics has already shown us that we can represent the meaning of a program using a functional representation. Functional languages enable a programmer to write directly in functional form. While it is not always obvious how to write a particular algorithm in functional form–just as it is not always obvious how to write some tasks in imperative form–functional and imperative languages are equally powerful in their capabilities.

The basis for functional languages is lambda calculus. A lambda expression, or lambda function is a nameless mathematical expression, where the mathematical expression becomes an object itself. As with typical substitution in mathematical expressions, variables in the expression can substitute for other variable names or for actual values.

Functional programming languages give the programmer a set of built-in functions, support various data types, and enable the definition of new functions. Loops are usually implemented recursively–as in our functional representation of loop semantics–and much of the language is built around lists.

Unlike pure mathematics, functional programming languages do permit state information and generally include an assignment statement. However, most functional language programmers try to avoid assignments, if at all possible.

Lambda expressions are not limited to functional programming languages. Python, for example, permits lambda expressions. In the example below, the lambda expression is assigned to a parameter of the function and then applied to the elements of a list.

```
# An example of using lambda functions in python

# imperative version that applies f to the elements of a
def applyFunction( f, a ):
    result = []
    for item in a:
        result.append( f(item) )

    return result

org = [1, 2, 3, 4, 5]

res1 = applyFunction( lambda x: 2*x, org )
print res1

res2 = applyFunction( lambda x: x*x, org )
print res2

expr = raw_input("Enter an expression of x: ");
res3 = applyFunction( eval('lambda x:' + expr), org )
print res3
```

The third case uses the eval function to permit the user to enter an arbitrary expression of x to apply to the list. Lambda functions permit the programmer to dynamically create functions of one or more variables and pass them around as objects.

Note that functional programming is not limited to functional languages. It is quite possible to write functional code in imperative languages that permit function return values. The example below is an identical implementation of the applyFunction imperative version above. The functional version does not use a loop, but instead uses recursion to apply the function to the elements of the list.

```
# functional version
def applyFunctionRec(f, a, result = []):
    if a == []:
        return result
    else:
        return applyFunctionRec( f, a[1:], result + [f(a[0])] );

org = [1, 2, 3, 4, 5]

res1 = applyFunctionRec( lambda x: 2*x, org )
print res1

res2 = applyFunctionRec( lambda x: x*x, org )
print res2
```

### 9.1.1 Lisp/Scheme

Lisp and Scheme are some of the original functional languages, with Scheme being a derivative of Lisp. Expressions/statements in Scheme follow Cambridge-prefix notation with the function name given first, followed by the arguments, and the entire expression in parentheses.

```
(+ 8 5) : evaluates as 13
(* 4 6) : evaluates as 24
(- 8 3) : evaluates as 5
(+ 1 2 3 4 5) : evaluates as 15
```

The argument to any function can also be a function. We can define variables in Scheme using the define function, which puts a symbol into the global symbol table. Note, define is also how we can define new functions.

```
(define myvar 100)
```

Besides basic data types, the workhorse data type of functional languages is the list. Scheme has built in support for list creation and processing. The following shows some examples of list creation and manipulation. The traditional list manipulators are the `car` and `cdr` operators. The `car` of a list is the first element of the list. The `cdr` of a list is all of the list but the first element. The `cdr` of a list is always a list, or the empty list `()` if there are no remaining elements.

**Example Scheme**

```
: use a quote to specify that the list is not to be evaluated, but is a constant
(define fib '(1 1 2 3 5 8 13) )

(car fib) : evaluates to 1
(cdr fib) : evaluates to (1 2 3 5 8 13)
(car (cdr fib) ): evaluates to 1
(car (cdr (cdr fib ) ) ) : evaluates to 2
(caddr fib) : same as previous, evaluates to 2
(cdr (cdr fib) ): evaluates to (2 3 5 8 13)
(cddr fib): same as previous, evaluates to (2 3 5 8 13)
```

The `cons` and `append` functions enable building up lists from pieces. Append expects two list as arguments and returns their concatenation. Cons expects a value and a list and puts the value as the head of the list. Putting a non-list as the second argument to `cons` creates a non-list data structure called a dotted pair, which is generally not used in Lisp programming.

```
(append '(3 4) '(5 6) ) : evaluates to (3 4 5 6)
(cons 4 '(5 6) ) : evaluates to (4 5 6)
(cons '(3 4) '(5 6) ) : evaluates to ( (3 4) 5 6 )
```

Conditional control flow in Scheme uses if-then-else or a case statement. The function below, for example, defines a new symbol `backward` that reverses the order of a list.

```
(define (backward a)
  (if (null? a) () (append (backward (cdr a)) (list (car a)))))
```

The case syntax is similar to other languages. The first argument to case is the condition to test, the remaining arguments are the cases. In the example below, if the result of the test expression is 1, 2, or 3 then the first case executes. If the result is 10, 11, or 12, the second case executes. Otherwise, the else case executes. An else case is not required.

```
> (display (case (+ 7 5)
            [(1 2 3) "weird\n"]
            [(10 11 12) "odd\n"]
            [else "whatever\n"]))
odd
```

There are several forms of iteration available in scheme, including for-loops, do-loops, and maps over lists. Python's for loop construct is identical to that of Scheme. The loop variable iterates of the list provided. As shown in the example below, the default `for` expression returns void as its result.

```
>  (display (for ([i '(0 1 2 3 4)])
             (+ i i)))
#<void>

>  (display (for/list ([i '(0 1 2 3 4)])
              (+ i i)))
(0 2 4 6 8)
```

There are many other forms of the for loop that return different kinds of values. For example, the `for/list` function show above returns a list that is the concatenation of the values produced by the last expression in the body during each iteration.

The do function is more like a for loop in C. In the example below, the variable i is the loop variable. It receives the initial value 0 and the step-expression describes how the loop variable changes each iteration. The second argument is the termination condition; the loop will run so long as the termination condition is false. The remaining arguments form the body of the loop.

```
>  (do ([i 0 (+ i 1)])
     ((> i 5))
     (display i))
012345
```

A map applies an expression to each element of a list and returns the new list. Lambda expressions are used within map expressions to define the operation applied to the list elements. They work identically to Python.

```
>  (define fib '(1 1 2 3 5 8 13))
>  (display (map (lambda (x)
                   (* x x))
                 fib))
(1 1 4 9 25 64 169)
```

Scheme also includes functions for creating randomly-accessible data structures, such as arrays. In Scheme, the symbol vector creates a vector of items. The vector-ref function enables direct indexing into the structure. Therefore, we can write a simple recursive binary search on vector types, as shown below.

```
(define fibv (vector 1 1 2 3 5 8 13))
(define (binsearch v key lower upper)
  (let ( (mid (quotient (+ lower upper) 2) ) )
    (if (> lower upper) '()
        (if (= key (vector-ref v mid) key) (list (vector-ref v mid))
            (if (> key (vector-ref v mid)) (binsearch v key (+ mid 1) upper)
                (binsearch v key lower (- mid 1)))))))
(binsearch fibv 5 0 (- (vector-length fibv) 1))
(binsearch fibv 4 0 (- (vector-length fibv) 1))
```

Given the ability to assign values, execute conditional expressions, and loop, Scheme is clearly computationally equivalent to imperative languages. What is important to realize, however, is that functional languages are not intended to be written as imperative languages. They are optimized for tail recursion and functional solutions to problems. Many Scheme programs are a single expression, which often contain recursion.

The semantic meaning of Scheme programs is almost trivial to explain as a functional expression, since the Scheme program itself is a functional representation, even to the point of having functions like let and define that specifically add variables to the symbol table and define their scope.

Like other languages we have studied, Scheme has many built-in data types, mechanisms for catching exceptions, creating multiple threads, and reading and writing to the terminal, files, and ports. Scheme handles garbage collection automatically, although there are a set of functions the programmer can use to let the garbage collector know when a variable is no longer needed and to manually force garbage collection.

## 9.2 Logic Languages

Logic languages grew out of the AI research field in the 1970s. One of the main results of symbolic AI was an understanding of how to execute searches within a state space. A state space is a graph, where nodes in the graph represent locations in a virtual space. These locations can be real-valued or symbolic. The connections between nodes may be defined by their relative location in the state space, or they may be defined by a series of rules. In either case, given a specific location in state space, there are some number of other nodes (including zero) that are connected to that location. When executing a search, the nodes may exist in a database, or they may be generated algorithmically as the search moves through the space.

Search is the process of identifying A) if two nodes are connected in the graph, and B) the path connecting the two nodes. The primary concerns of search are efficiency and optimality. Efficiency is concerned with the total time and/or memory required to answer part A. Optimality is concerned with whether the path connecting the two nodes is the shortest path. Different search strategies result in different tradeoffs between time efficiency, memory efficiency, and optimality.

A logic language is an interface to a search engine. It is a syntax for specifying nodes and connections in a state space and then making queries to a search engine about the state space. Like other programming languages, it has naming conventions, types, assignments, expressions and statements. Prolog is a Turing complete language, so it is possible to write interpreters, compilers, and anything else in Prolog.

The unique characteristic of logic languages is that the user's program is generally a specification of facts, followed by a series of queries. The program generally does not look like a sequence of operations to be executed by the computer. Each query specifies a starting location in the state space and one or more goal nodes. The search engine then identifies if the start node and one or more goals are connected.

Prolog is the primary surviving logic language. Prolog uses propositional and predicate logic to specify facts (nodes) and relationships (connections). For example,

```
hobbit(frodo)
```

is a statement that the predicate (property) `hobbit` applies to the atom (symbol) `frodo`. A Horn clause is a statement that connects a single predicate to a set of conditions for that predicate to apply. For example, a Hobbit hole is dry, comfortable, and underground, which is expressed by the following Horn clause. The way to read the clause is that the hobbitHole predicate applies to `X` if the predicates dry, comfortable, and underground also apply to `X`.

```
hobbitHole(X) :- dry(X), comfortable(X), underground(X)
```

A Horn clause defines a new predicate as a function of other predicates. The predicate on the left applies to the variable `X` only if all of the predicates on the right also apply to `X`. By itself, a Horn clause is not meaningful. However, when combined with factual predicates, it defines relationships in a state space.

Following the rules of logic, a Horn clause of the form `p(A) :- q(A)` is equivalent to `p(A) or (not q(A)`, which is a Boolean statement of facts, rather than an if-then relationship. Given a Boolean statement of facts, we can use search to identify if we can satisfy the expression, and whether there are multiple possible solutions.

**Resolution** is the process of identifying when the head (the left side) of Horn clause A exists as one of the terms (the right side) of a different Horn clause B and substituting the terms of A into the list of terms for B. Resolution is the transitive property of logic. In the example below, if we have the first two Horn clauses, then the third one is also true.

```
likesfood(X) :- hobbit(X)
likessecondbreakfast(X) :- likesfood(X)

likessecondbreakfast(X) :- hobbit(X)
```

A simple predicate of the form `comfortable(bagend)` is the head of a Horn clause with no terms, which means it is a fact. Resolution gives us the ability to start searching a database of facts by connected together Horn clauses. As resolution proceeds, it finds atoms, like `bagend` it can bind to variables. Binding a variable with a value is called **instantiation**. The process of identifying the set of values for a set of variables in a Horn clause that make it true is called **unification**.

For example, consider the following set of facts and the predicate `hobbitHole` defined as a Horn clause.

```
dry(ashes).
dry(desert).
dry(bagend).
dry(num2bagshotrow).
dry(prancingpony).

comfortable(chair).
comfortable(sweatshirt).
comfortable(bagend).
comfortable(num2bagshotrow).
comfortable(prancingpony).

underground(sewer).
underground(goblinhole).
underground(bagend).
underground(num2bagshotrow).

hobbitHole(X) :- dry(X), comfortable(X), underground(X).
```

If we were to make a query about what atoms satisfied the expression `hobbitHole(X)`, the search engine would use resolution to identify possible substitutions for the terms in the hobbitHole clause. Both `bagend` and `num2bagshotrow` satisfy all three of the terms.

### 9.2.1 Prolog Syntax

Prolog programs consist of a series of terms. Each term is a statement of facts or a relationship between terms. Terms can contain, or consist of constants, variables, and structures. Constants, or atoms, are names beginning with lower-case letters or strings enclosed in quotes. A variable is a name that begins with a capital letter. A structure is a predicate followed immediately by zero or more terms in parentheses.

A Horn clause that relates predicates has the following syntax, which states that term0 is true only if the terms on the right side are true. The following are both Horn clauses. The first is a statement of fact that is always true. The second has a head term that is true if all of the tail (right side) terms are true. The code above shows specific examples.

```
term0.
term0 :- term1, term2, ...
```

All facts, rules, and queries in Prolog must end with a period, and a Prolog program file must end with an end-of-line character.

When searching for a solution, the search engine searches rules in the order in which they appear in the program file. When resolving the terms of a Horn clause, the search engine proceeds from left to right. By default, Prolog stops and returns the first value that satisfies the query. It is also possible to tell Prolog to continue searching from where it left off, forcing it to backtrack and find other possible solutions until it has exhausted the state space.

When writing Prolog, it is important to keep in mind that you are directing a search engine. If you want it to explore a set of alternative solutions, you have to give it enough direction that it can find those alternatives at the appropriate time in the search. This observation is most important when you are specifying that a particular solution should not be equal to a specific value. A **not equals** relationship means that the search engine has found a solution that is not equal to a particular value. The expression, on its own, does not give the engine any particular direction as to where to search for those potential solutions.

The following example shows how one could use Prolog to solve a complex puzzle problem. The puzzle is from logic-puzzles.org: `http://www.logic-puzzles.org/pdf/E353ZJ.pdf`.

1. The person whose flight departs at 6:30am is not Emilio.
2. The person who bought the earrings collects Pieridae.
3. The person who bought the belt doesn't collect Lycaenidae and is not Alec.
4. Alec will depart earlier than the person who bought the purse.
5. Serenity didn't buy the shoes and doesn't collect Lycaenidae.
6. The lepidopterist who collects Pieridae is Emilio.
7. The person whose flight departs at 9:30am is not Cameron.
8. Of the person who bought the earrings and the lepidopterist who collects Riodinidae, one will depart at 9:30am and the other will depart at 12 noon.
9. The lepidopterist who collects Pieridae will depart earlier than the person who bought the shoes.
10. Either the person who bought the earrings or the person who bought the purse is Emilio.
11. The person who bought the purse doesn't collect Nymphalidae.

The following Prolog program solves the puzzle. Note that the first solution it finds is invalid, because two people are collecting the same butterfly. The second solution is valid. To ensure only valid solutions, you would need to add additional rules that specify that solution can use each value only once.

### Example: Logic Puzzle Solution

In order to give Prolog direction on where to search, note that all of the not-equals relationships come after all the positive specifications for each variable. On the first line, for example, the code specifies that A is a person before specifying that A is not emilio. This gives Prolog a definite set of options to search.

```
person(alec).
person(cameron).
person(emilio).
person(serenity).

butterfly( lycaenidae ).
butterfly( nymphalidae ).
butterfly( pieridae ).
butterfly( riodinidae ).

thing( belt ).
thing( earrings ).
thing( purse ).
thing( shoes ).

ftime(630).
ftime(930).
ftime(1200).
ftime(1830).

earlier(A, B) :- A < B.

times( [ flightTime( 630, _, _, _ ),
flightTime( 930, _, _, _ ),
flightTime( 1200, _, _, _ ),
flightTime( 1830, _, _, _ ) ] ).

group( X ) :- times( X ),
member( flightTime( 630, A, B, C ), X ), person(A), butterfly( B ), thing( C), A \= emilio,
member( flightTime( D, E, pieridae, earrings ), X ), person(E), ftime(D),
member( flightTime( F, G, H, belt ), X ), person(G), butterfly( H ), ftime(F), G \= alec, H \= lycaenidae,
member( flightTime( I, alec, J, K ), X ), butterfly( J ), thing( K), ftime(I),
member( flightTime( L, M, N, purse ), X ), earlier(I, L), person(M), butterfly(N), ftime(L), M \= alec,
member( flightTime( O, serenity, P, Q), X ), butterfly(P), thing(Q), ftime(O), P \= shoes, Q \= lycaenidae,
member( flightTime( R, emilio, pieridae, S ), X), thing(S), ftime(R),
member( flightTime( 930, T, U, V), X), person(T), butterfly(U), thing(V), T \= cameron,
member( flightTime( W, Z, Y, purse), X), person(Z), butterfly(Y), ftime(W), Y \= nymphalidae,
member( flightTime( AA, emilio, BB, CC ), X), butterfly(BB), thing(CC), ftime(AA), CC \= belt, CC \= shoes,
member( flightTime( DD, EE, pieridae, FF), X), person(EE), thing(FF), ftime(DD),
member( flightTime( GG, HH, II, shoes), X ), earlier( DD, GG ), person(HH), butterfly(II), ftime(GG),
member( flightTime( JJ, KK, LL, earrings ), X ), person(KK), butterfly(LL), ftime(JJ), JJ \= 630, JJ \= 1830,
member( flightTime( MM, NN, riodinidae, OO), X), person(NN), thing(OO), ftime(MM), MM \= 630, MM \= 1830.
```

Loading and running the puzzle produces the following result.

```
| ?- ['puzzle2.pl'].
compiling /Users/maxwell/courses/cs333/F10/lectures/src/puzzle2.pl for byte code...
/Users/maxwell/courses/cs333/F10/lectures/src/puzzle2.pl compiled, 41 lines read - 9355 bytes written, 24 ms

(2 ms) yes
| ?- group(X).

X = [flightTime(630,cameron,nymphalidae,belt),flightTime(930,alec,pieridae,earrings),
flightTime(1200,serenity,riodinidae,shoes),flightTime(1830,emilio,pieridae,purse)] ? ;

X = [flightTime(630,cameron,nymphalidae,belt),flightTime(930,emilio,pieridae,earrings),
flightTime(1200,alec,riodinidae,shoes),flightTime(1830,serenity,lycaenidae,purse)] ?
```