Principles of Programming Languages (23CS508)

UNIT-1
Preliminary Concepts

Background

- Frankly, we didn't have the vaguest idea how the thing [FORTRAN language and compiler] would work out in detail. …We struck out simply to optimize the object program, the running time, because most people at that time believed you couldn't do that kind of thing. They believed that machined-coded programs would be so inefficient that it would be impractical for many applications.
- John Backus, unexpected successes are common – the browser is another example of an unexpected success

1.1 Reasons for Studying Concepts of Programming Languages- CO1

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Overall advancement of computing

1.2 Programming Domains – CO1

- Scientific applications
  - Large number of floating point computations
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated
  - LISP
- Systems programming
  - Need efficiency because of continuous use
  - C
- Web Software
  - Eclectic collection of languages: markup (**e.g.**, XHTML), scripting (**e.g.**, PHP), general-purpose (**e.g.**, Java)

1.2 Language Evaluation Criteria – CO1, CO2

- Readability : the ease with which programs can be read and understood
- Writability : the ease with which a language can be used to create programs
- Reliability : conformance to specifications (i.e., performs to its specifications)
- Cost : the ultimate total cost

Readability

- Overall simplicity
  - A manageable set of features and constructs
  - Few feature multiplicity (means of doing the same operation)
  - Minimal operator overloading
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal
- Control statements
  - The presence of well-known control structures (**e.g.**, while statement)
- Data types and structures
  - The presence of adequate facilities for defining data structures
- Syntax considerations
  - Identifier forms: flexible composition
  - Special words and methods of forming compound statements
  - Form and meaning: self-descriptive constructs, meaningful keywords

Writability

- Simplicity and Orthogonality
  - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
  - The ability to define and use complex structures or operations in ways that allow details to be ignored
- Expressivity
  - A set of relatively convenient ways of specifying operations
  - Example: the inclusion of for statement in many modern languages

Reliability

- Type checking
  - Testing for type errors
- Exception handling
  - Intercept run-time errors and take corrective measures
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
  - A language that does not support “natural” ways of expressing an algorithm will necessarily use “unnatural” approaches, and hence reduced reliability

Cost

- Training programmers to use language
- Writing programs (closeness to particular applications)
- Compiling programs
- Executing programs

- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

Others

- Portability
  - The ease with which programs can be moved from one implementation to another
- Generality
  - The applicability to a wide range of applications
- Well-definedness
  - The completeness and precision of the language's official definition

1.3 Influences on Language Design - CO3

- Computer Architecture
  - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- Programming Methodologies
  - New software development methodologies (**e.g.**, object-oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
  - Data and programs stored in memory
  - Memory is separate from CPU
  - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
- Variables model memory cells
- Assignment statements model piping
- Iteration is efficient

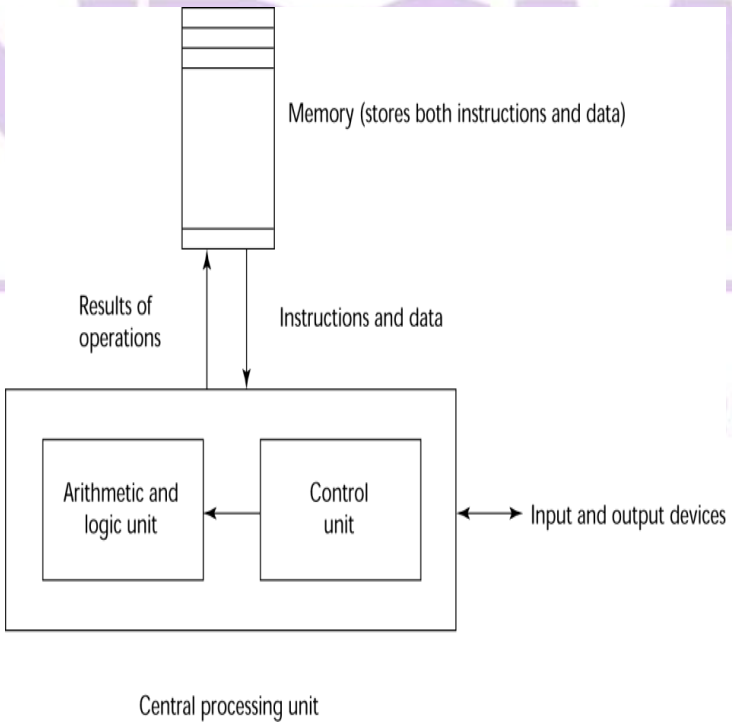


Figure 1.1 The von Neumann Computer Architecture

Programming Methodologies

- 1950s and early 1960s: Simple applications; worry about machine efficiency
- Late 1960s: People efficiency became important; readability, better control structures
  - structured programming
  - top-down design and step-wise refinement
- Late 1970s: Process-oriented to data-oriented
  - data abstraction
- Middle 1980s: Object-oriented programming
  - Data abstraction + inheritance + polymorphism

1.4 Language Categories – CO1

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Examples: C, Pascal
- Functional
  - Main means of making computations is by applying functions to given parameters
  - Examples: LISP, Scheme
- Logic
  - Rule-based (rules are specified in no particular order)
  - Example: Prolog
- Object-oriented
  - Data abstraction, inheritance, late binding
  - Examples: Java, C++
- Markup
  - New; not a programming per se, but used to specify the layout of information in Web documents
  - Examples: XHTML, XML

Language Design Trade-Offs

- Reliability vs. cost of execution
  - Conflicting criteria
  - Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs
- Readability vs. writability
  - Another conflicting criteria
  - Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
  - Another conflicting criteria
  - Example: C++ pointers are powerful and very flexible but not reliably used

1.5 Implementation Methods -CO2

- Compilation
  - Programs are translated into machine language
- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters

Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code
  - code generation: machine code is generated

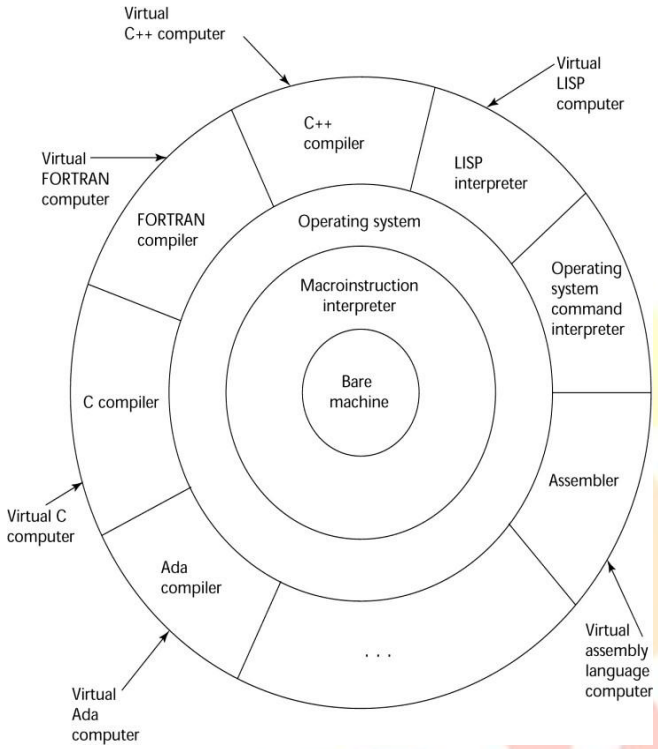


Figure 1.2 Layered View of Computer: The operating system and language implementation are layered over Machine interface of a computer

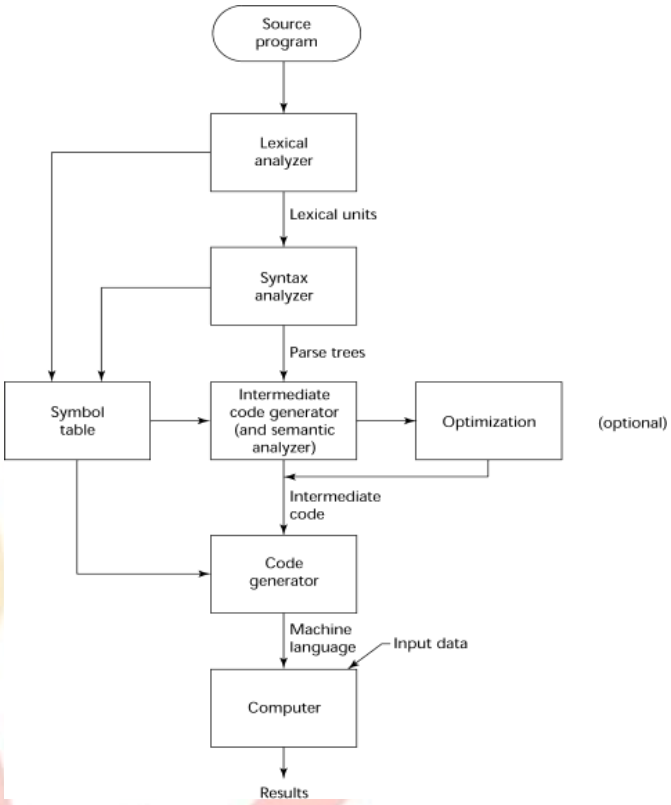


Figure 1.3 The Compilation Process

Additional Compilation Terminologies

- Load module (executable image): the user and system code together
- Linking and loading: the process of collecting system program and linking them to user program

Execution of Machine Code

- Fetch-execute-cycle (on a von Neumann architecture)

initialize the program counter
repeat forever
fetch the instruction pointed by the counter
increment the counter
decode the instruction
execute the instruction
end repeat

Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed a lot faster than the above connection speed; the connection speed thus results in a *bottleneck*
- Known as von Neumann bottleneck; it is the primary limiting factor in the speed of computers

Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Becoming rare on high-level languages

- Significantly comeback with some latest web scripting languages (**e.g.**, JavaScript)

Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
  - Perl programs are partially compiled to detect errors before interpretation
  - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a runtime system (together, these are called *Java Virtual Machine*)

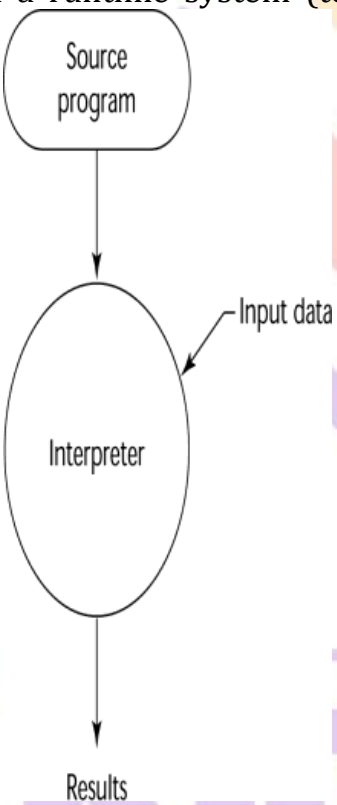


Figure 1.4 Pure Interpreataion

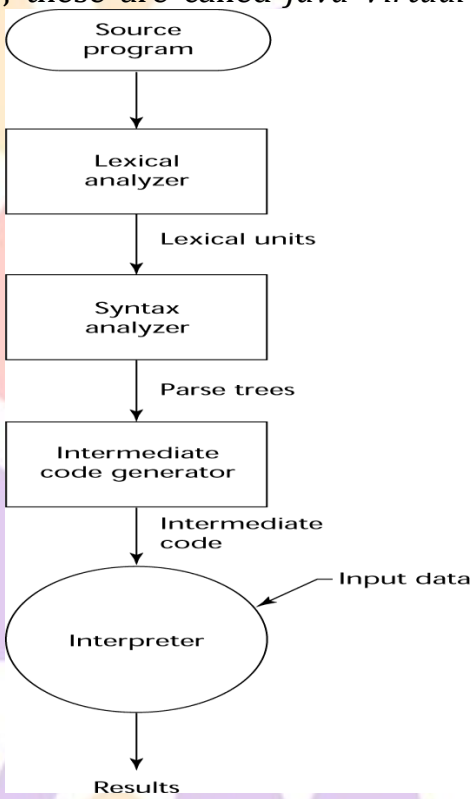


Figure 1.5 Hybrid Implementation

Just-in-Time Implementation Systems

- Initially translate programs to an intermediate language
- Then compile intermediate language into machine code
- Machine code version is kept for subsequent calls
- JIT systems are widely used for Java programs
- .NET languages are implemented with a JIT system

1.6 Preprocessors - CO2

- Preprocessor macros (instructions) are commonly used to specify that code from another file is to be included
- A preprocessor processes a program immediately before the program is compiled to expand embedded preprocessor macros
- A well-known example: C preprocessor
  - expands #include, #define, and similar macros

Syntax and Semantics

Introduction

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
  - Users of a language definition
  - Other language designers
  - Implementers
  - Programmers (the users of the language)

1.7 The General Problem of Describing Syntax – CO1

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (**e.g.**, *, sum, begin)
- A *token* is a category of lexemes (**e.g.**, identifier)
- **Languages Recognizers**
  - A recognition device reads input strings of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler
- **Languages Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator

1.8 Formal Methods of Describing Syntax – CO1,CO2

- Backus-Naur Form and Context-Free Grammars
  - Most widely known method for describing programming language syntax
- Extended BNF
  - Improves readability and writability of BNF
- Grammars and Recognizers

Backus-Naur Form and Context-Free Grammars

- Context-Free Grammars
- Developed by Noam Chomsky in the mid-1950s
- Language generators, meant to describe the syntax of natural languages
- Define a class of languages called context-free languages

Backus-Naur Form (BNF)

- Backus-Naur Form (1959)
  - Invented by John Backus to describe ALGOL 58
  - BNF is equivalent to context-free grammars
  - BNF is a *metalanguage* used to describe another language
  - In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols*)

BNF Fundamentals

- Non-terminals: BNF abstractions
- Terminals: lexemes and tokens
- Grammar: a collection of rules
  - Examples of BNF rules:

<ident_list> → identifier | identifer, <ident_list>
*<if_stmt> → **if** <logic_expr> **then** <stmt>*

BNF Rules

- A rule has a left-hand side (LHS) and a right-hand side (RHS), and consists of *terminal* and *nonterminal* symbols
- A grammar is a finite nonempty set of rules
- An abstraction (or nonterminal symbol) can have more than one RHS

<stmt> → <single_stmt>
*| **begin** <stmt_list> **end***

Describing Lists

- Syntactic lists are described using recursion

<ident_list> → ident
| ident, <ident_list>

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const

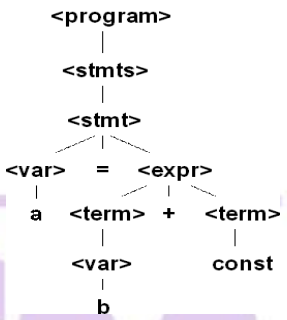
Parse Tree

A hierarchical representation of a derivation

An example derivation

<program>⇨<stmts>
⇨<stmt>
⇨<var>=<expr>
⇨a=<expr>
⇨a=<term>+<term>
⇨a=<var>+<term>
⇨a=b+<term>
⇨a=b+const

Figure 2.1 Parse Tree



Derivation

- Every string of symbols in the derivation is a sentential form
- A sentence is a sentential form that has only terminal symbols
- A leftmost derivation is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Ambiguity in Grammars

- A grammar is *ambiguous* iff it generates a sentential form that has two or more distinct parse trees

An Unambiguous Expression Grammar

If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

<expr> → <expr> - <term>|<term>
<term> → <term> / const|const

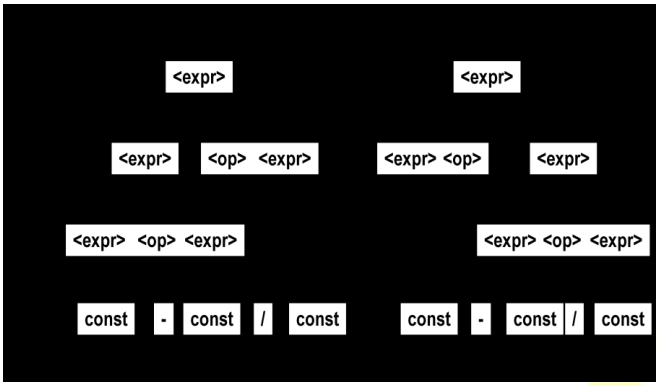


Figure 2.2 An Ambiguous Expression Grammar

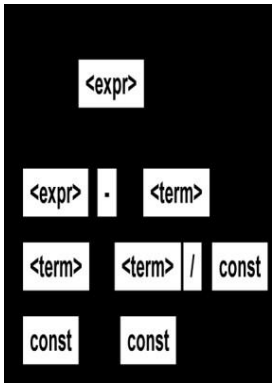


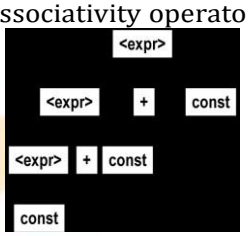
Figure 2.3 An Unambiguous Expression Grammar

Associativity of Operators

Operator associativity can also be indicated by a grammar

<expr> → <expr> + <expr> | *const* (ambiguous)
<expr> → <expr> + const | *const* (unambiguous)

Figure 2.4 Parse Tree for Associativity operator



1.9 Extended Backus-Naur Form (EBNF) – CO2

- Optional parts are placed in brackets ([ ])
  <proc_call> → ident [(<expr_list>)]
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
  <term> → <term> (+|-) const
- Repetitions (0 or more) are placed inside braces ({ })
  <ident> → letter {letter|digit}

BNF and EBNF

- BNF

  <expr> → <expr> + <term>
  | <expr> - <term>
  | <term>
  *<term> → <term> * <factor>*
  | <term> / <factor>
  | <factor>

- EBNF

  <expr> → <term> {(+ | -) <term>}
  <term> → <factor> {(| /) <factor>}*

1.10 Attribute Grammars – CO1, CO4

- Context-free grammars (CFGs) cannot describe all of the syntax of programming languages
- Additions to CFGs to carry some semantic info along parse trees
- Primary value of attribute grammars (AGs):
  - Static semantics specification
  - Compiler design (static semantics checking)

Definition

- An attribute grammar is a context-free grammar G = (S, N, T, P) with the following additions:
  - For each grammar symbol *x* there is a set *A(x)* of attribute values

- Each rule has a set of functions that define certain attributes of the nonterminals in the rule
- Each rule has a (possibly empty) set of predicates to check for attribute consistency
- Let X0 X1 … Xn be a rule
- Functions of the form S(X0) = f(A(X1), … , A(Xn)) define *synthesized attributes*
- Functions of the form I(Xj) = f(A(X0), … , A(Xn)), for i <= j <= n, define *inherited attributes*
- Initially, there are *intrinsic attributes* on the leaves

Example

- Syntax

  <assign> → <var> = <expr>
  <expr> → <var> + <var> | <var>
  <var> → A | B | C
- actual_type: synthesized for *<var>* and *<expr>*
- expected_type: inherited for *<expr>*
- Syntax rule :*<expr> → <var>[1] + <var>[2]*
  Semantic rules :*<expr>.actual_type → <var>[1].actual_type*
  Predicate :*<var>[1].actual_type == <var>[2].actual_type*
  <expr>.expected_type == <expr>.actual_type
- Syntax rule :*<var> → id*
  Semantic rule :*<var>.actual_type ← lookup (<var>.string)*
- How are attribute values computed?
  - If all attributes were inherited, the tree could be decorated in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.

    <expr>.expected_type ← inherited from parent
    <var>[1].actual_type ← lookup (A)
    <var>[2].actual_type ← lookup (B)
    <var>[1].actual_type =? <var>[2].actual_type
    <expr>.actual_type ← <var>[1].actual_type
    <expr>.actual_type =? <expr>.expected_type

Describing the Meanings of Programs:Dynamic Semantics

- There is no single widely acceptable notation or formalism for describing semantics
- Operational Semantics
  - Describe the meaning of a program by executing its statements on a machine, either simulated or actual. The change in the state of the machine (memory, registers, etc.) defines the meaning of the statement
- To use operational semantics for a high-level language, a virtual machine is needed
- A hardware pure interpreter would be too expensive
- A software pure interpreter also has problems:
  - The detailed characteristics of the particular computer would make actions difficult to understand
  - Such a semantic definition would be machine- dependent

Operational Semantics

- A better alternative: A complete computer simulation
- The process:
  - Build a translator (translates source code to the machine code of an idealized computer)
  - Build a simulator for the idealized computer
- Evaluation of operational semantics:
  - Good if used informally (language manuals, etc.)
  - Extremely complex if used formally (**e.g.**, VDL), it was used for describing semantics of PL/I.
- Axiomatic Semantics
  - Based on formal logic (predicate calculus)
  - Original purpose: formal program verification
  - Approach: Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions)
  - The expressions are called assertions

Axiomatic Semantics

- An assertion before a statement (a precondition) states the relationships and constraints among variables that are true at that point in execution
- An assertion following a statement is a postcondition
- A weakest precondition is the least restrictive precondition that will guarantee the postcondition
- Pre-post form: {P} statement {Q}
- An example: a = b + 1 {a > 1}
- One possible precondition: {b > 10}
- Weakest precondition: {b > 0}
- Program proof process: The postcondition for the whole program is the desired result. Work back through the program to the first statement. If the precondition on the first statement is the same as the program spec, the program is correct.
- An axiom for assignment statements
  (x = E):
  {Qx->E} x = E {Q}
- An inference rule for sequences
  - For a sequence S1;S2:
  - {P1} S1 {P2}
  - {P2} S2 {P3}
- An inference rule for logical pretest loops
  For the loop construct:
  {P} while B do S end {Q}
  Characteristics of the loop invariant
  I must meet the following conditions:
  - P => I (the loop invariant must be true initially)
  - {I} B {I} (evaluation of the Boolean must not change the validity of I)
  - {I and B} S {I} (I is not changed by executing the body of the loop)
  - (I and (not B)) => Q (if I is true and B is false, Q is implied)
  - The loop terminates (this can be difficult to prove)
- The loop invariant I is a weakened version of the loop postcondition, and it is also a precondition.

- I must be weak enough to be satisfied prior to the beginning of the loop, but when combined with the loop exit condition, it must be strong enough to force the truth of the postcondition.

Evaluation of Axiomatic Semantics:

- Developing axioms or inference rules for all of the statements in a language is difficult
- It is a good tool for correctness proofs, and an excellent framework for reasoning about programs, but it is not as useful for language users and compiler writers
- Its usefulness in describing the meaning of a programming language is limited for language users or compiler writers

Denotational Semantics

- Based on recursive function theory
- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)
- The process of building a denotational spec for a language (not necessarily easy):
- Define a mathematical object for each language entity
- Define a function that maps instances of the language entities onto instances of the corresponding mathematical objects
- The meaning of language constructs are defined by only the values of the program's variables
- The difference between denotational and operational semantics: In operational semantics, the state changes are defined by coded algorithms; in denotational semantics, they are defined by rigorous mathematical functions
- The state of a program is the values of all its current variables

  s = {<i1, v1>, <i2, v2>, ..., <in, vn>}
- Let VARMAP be a function that, when given a variable name and a state, returns the current value of the variable

  VARMAP(ij, s) = vj

- Decimal Numbers
  - The following denotational semantics description maps decimal numbers as strings of symbols into numeric values

    <dec_num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

    ⇨ | <dec_num> (0 | 1 | 2 | 3 | 4 |5 | 6 | 7 | 8 | 9)

    Mdec('0') = 0, Mdec ('1') = 1, ..., Mdec ('9') = 9

    Mdec (<dec_num> '0') = 10 * Mdec (<dec_num>)

    Mdec (<dec_num> '1') = 10 * Mdec (<dec_num>) + 1

    ...

    Mdec (<dec_num> '9') = 10 * Mdec (<dec_num>) + 9

Expressions

- Map expressions onto Z ∪ {error}
- We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression
- Assignment Statements
  - Maps state sets to state sets
- Logical Pretest Loops
  - Maps state sets to state sets