Semantics

1. OBJECTIVES ................................................................. 2
2. DEFINITIONS ................................................................. 2
3. SEMANTICS RULES ......................................................... 3
4. OPERATIONAL SEMANTICS ................................................. 4
5. AXIOMATIC SEMANTICS .................................................... 8
6. FUNCTIONAL (DENOTATIONAL) SEMANTICS .............. 10
7. LANGUAGE PROCESSING ................................................... 13
1. Objectives

- BNF does not provide additional specifications about the language:
  - How BNF can check whether a variable is declared before it is referenced?
  - How to enforce the length of an identifier?
  - How do we know the maximum integer value?

2. Definitions

- It is the meaning of any syntactically valid program written in the language, e.g., expressions, statements, and program units. Note that all syntactically correct programs have valid.
- There is no single widely acceptable notation or formalism for describing semantics.
- It is hard to describe the meaning of a language:
  - Informal description: describe in English the meaning of a construct.
  - Formal description: A mathematical model using formal notation to describe each construct.

- Example:
  - The syntax of a C if statement is:

    \[
    \text{If (<expression>) <statement>}
    \]

    The meaning of this statement form is that the current value of the expression is true, the embedded statement is selected for execution.
It would be nice to have a formal description for such construct!!

3. Semantics rules

- **Definitions:**
  - These are constraints applied to syntactically correct program before execution.
  - There are two types of semantics:
    - Static semantics
    - Dynamic semantics

- **Static semantics:**
  - Rules that are enforced by the compiler at compile time.
  - Any constraints that can be checked at compile time.
  - Example: Type checking, check functions/methods formal and actual parameters, etc.

- **Dynamic semantics:**
  - Rules of a given construct that are enforced at the run-time.
  - It is the meaning of each construct of a language.

- **Approaches:**
  - There is no single widely acceptable notation or formalism for describing semantics
  - There are approaches
    - Axiomatic semantics
    - Functional (denotational) semantics
    - Operational semantics
4. Operational Semantics

- Describe the meaning of a program by executing its statements on an abstract machine. Each program statement is described by a set of operations of this machine.

- SIMPLESEM:
  - It is an abstract machine
  - It consists of the following:
    - Instructor pointer: ip
    - Memory: two types of memory: Data and Code
    - Processor

  ![Diagram of Data and Code Memory](image)

  - Memory addressing:
    - A value stored in data memory is denoted D[X] where X is the memory address (starting from 0).
    - A value stored in code memory is denoted C[X] where X is the memory address (starting from 0).

  - Instruction Format:
    - Load a memory location: set target, source

  - Examples:
- Set the memory location 5 to 10:
  Set 5, 10

- Copy the value stored in memory location 200 into memory location 2
  Set 2, D[200]

- Generate code for
  
  a = 10  
  B = 20  
  C = a + b

SIMPLESEM Code:

// Store the value of A in memory location 0
Set 0, 10

// Store the value of B in memory location 1
Set 1, 20

// Store the value of C in memory location 2
set 2,D[0]+D[1]

// Terminate your program
halt

○ Input/Output:
  ▪ Uses special registers read/write.
  ▪ Read a value into memory location 3:
    
    Set 3, read

  ▪ Output the content of memory location 20:
    
    Set write, D[20]

○ Control Flow:
- SIMPLESEM uses two statements: jump and jumpt
- Unconditional jump: set ip = 34 (go execute instruction stored at address 34)
  
  Jump 34

- Conditional jump: set ip = 65 (go execute instruction stored at address 65 if the condition is true)
  
  Jump 65, condition


  o Indirect Addressing:

    Set D[3],D[20]

    Assign the value stored at location 20 to the cell whose address is the value stored at location 3.

  o Example: A C1 program

```c
main() {
  int i,j;
  get (i,j);
  while (i != j)
    If (i > j)
      i -= j;
    else
      j -= i;
  print(i);
}
```
• Check this Example:

http://www.seas.gwu.edu/~bell/csci4223/lectures/simplesem.ppt
5. Axiomatic semantics

- It is based on formal logic (First order predicate calculus).
- **Approach:**
  - Define axioms or inference rules for each statement type in the language (to allow transformations of expressions to other expressions).
  - They are used to formally prove a property (post-condition) of the execution of a program, assuming another property (pre-condition) of the program:

\[
\{Q\} \text{ Program } \{P\}
\]

Where

- Q is a precondition (also called assertion) of the program (A precondition states the relationships and constraints among variables that are true at that point in execution), and
- P is a postcondition of the program.

- Example:
  \[
  Q=\{y=3\} \\
  x = y + 1; \\
  P=\{x>0\}
  \]

- **A Weakest Precondition:**
  - It is the least restrictive precondition that will guarantee the postcondition.

- Example:
  \[
  a = b + 1 \hspace{1cm} P=\{a > 1\}
  \]

  One possible precondition:  \(Q=\{b > 10\}\)
**Weakest precondition:** \( Q = \{ b > 0 \} \)

- **Axiom Definition or Predicate Transformer:**
  - It is a function, called \( \text{asem} \), that provides the weakest precondition \( W \) of a statement \( S \) and a postcondition \( P \):
  - Example:
    Given an assignment statement: \( S: \ x = \text{expr}; \ P \)
    The weakest precondition, denoted \( P_{x \rightarrow \text{expr}} \), of \( S \) is obtained by replacing each occurrence of \( x \) in \( P \) by \( \text{expr} \):
    \[
    \text{asem}(S, P) = P_{x \rightarrow \text{expr}}
    \]

- **Sequence**
  Given a sequence \( S: S_1; S_2; \) is:
    \[
    \begin{align*}
    \{P_1\} & \ S_1; \ \{P_2\} \\
    \{P_2\} & \ S_2; \ \{P_4\}
    \end{align*}
    \]
  The axiom of \( S \) is:
    \[
    \{P_1\} \ S_1; S_2; \ \{P_3\}
    \]

- **Conditional if Statement:**
  - Given an if statement \( S: \text{if}(B) \text{ then } L_1 \text{ else } L_2; \) is:
    \[
    \text{asem}(S, P) = (B \Rightarrow \text{asem}(L_1, P)) \text{ and } (\neg B \Rightarrow \text{asem}(L_2, P))
    \]
- In other words if we have:

\[
\{ P \} \text{ if } B \text{ then } L_1; \text{ else } L_2; \{ Q \}
\]

holds true, if and only if

\[
\{ P \text{ and } B \} L_1; \{ Q \} \text{ and } \{ P \text{ and not } B \} L_2; \{ Q \}
\]

- 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.

6. Functional (Denotational) semantics

- The most abstract semantics description method
- Originally developed by Scott and Strachey (1970)
- Based on recursive function theory:
  - It is based on the following:
    - The initial functions
    - Composition
    - Primitive Recursion
    - Minimization

- The operations of the computer are simulated by writing mathematical functions.
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:
\[ s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\} \]

Let \( \text{VARMAP} \) be a function that, when given a variable name and a state, returns the current value of the variable:
\[ \text{VARMAP}(ij, s) = v_j \]

The following denotational semantics description maps decimal numbers as strings of symbols into numeric values:

\[
\begin{align*}
\text{<dec\_num>} & \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\
 & \mid \text{<dec\_num>} (0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9)
\end{align*}
\]

\[
\begin{align*}
\text{M}_{\text{dec}}('0') &= 0, \quad \text{M}_{\text{dec}}('1') = 1, \ldots, \quad \text{M}_{\text{dec}}('9') = 9 \\
\text{M}_{\text{dec}}(\text{<dec\_num>}'0') &= 10 \times \text{M}_{\text{dec}}(\text{<dec\_num>}) \\
\text{M}_{\text{dec}}(\text{<dec\_num>}'1') &= 10 \times \text{M}_{\text{dec}}(\text{<dec\_num>}) + 1 \\
& \ldots \\
\text{M}_{\text{dec}}(\text{<dec\_num>}'9') &= 10 \times \text{M}_{\text{dec}}(\text{<dec\_num>}) + 9
\end{align*}
\]

Expressions
- We assume expressions are decimal numbers, variables, or binary expressions having one arithmetic operator and two operands, each of which can be an expression

\[
\text{Me(<expr>, s)} =
\]
case <expr> of
  <dec_num> => Mdec(<dec_num>, s)
  <var> =>
    if VARMAP(<var>, s) == undef
      then error
    else VARMAP(<var>, s)
  <binary_expr> =>
    if (Me(<binary_expr>.<left_expr>, s) == undef
         OR Me(<binary_expr>.<right_expr>, s) =
            undef)
      then error
    else
      if (<binary_expr>.<operator> == ‘+’ then
        Me(<binary_expr>.<left_expr>, s) +
        Me(<binary_expr>.<right_expr>, s)
      else Me(<binary_expr>.<left_expr>, s) *
        Me(<binary_expr>.<right_expr>, s)
  ...

- Logical Pretest Loops

  ML(while B do L, s) Δ=
  if Mb(B, s) == undef
    then error
  else if Mb(B, s) == false
    then s
  else if Msl(L, s) == error
then error
else Ml(while B do L, Msl(L, s))

- Assignment Statements
  - Maps state sets to state sets

\[
\text{Ma}(x := E, s) \Delta= \\
\text{if } \text{Me}(E, s) == \text{error}
\text{then error}
\text{else } s' = \{<i_1',v_1'>,<i_2',v_2'>,...,<i_n',v_n'>\},
\text{where for } j = 1, 2, ..., n,
\quad v_j' = \text{VARMAP}(ij, s) \text{ if } ij <> x
\quad = \text{Me}(E, s) \text{ if } ij == x
\]

- Evaluation:
  - Can be used to prove the correctness of programs
  - Provides a rigorous way to think about programs
  - Can be an aid to language design
  - Has been used in compiler generation systems
  - It is not very useful for readers trying to learn a language, or implementers trying write a compiler using traditional methods.
  - It is easy to generate a compiler of the language from the formal definition.

7. Language Processing

- Implementation Methods:
A programming language requires various system supports. Operating system provides runtime process management, file system for storing programs, I/O operations. Compiler provide source to object code translation, and runtime system provide runtime library.

A layer approach provides an abstract machine for the user.

- Pure Interpretation
  - Source code is not translated but run directly by a software interpreter.
  - Pure interpretation allows easy source level debugging implementation, provides maximum flexibility, but suffers slow execution speed and large memory requirement.
  - It is also difficult to build an interpreter because of high level constructs.

  ![Language Processing Diagram]

  Interpreter Pseudocode:
  - Read the next statement (Fetch)
  - Determine the actions to be executed (Decode)
  - Perform the actions (Execute)

- Translation
  - Source code is translated (by a compiler) into machine language and run directly by hardware.
- A compiler has a preprocessor, a lexical analyzer, a parser, optionally an optimizer, and a code generator.
- A linker links the different user and system object files into the final executable.
- The von Neumann bottle between CPU and memory limits the execution speed.

**Linker:**
- It takes all the independent object files and links them together: your program in assembly language and the assembly version of all libraries used by your program.
- The output of the linker is the **executable file** or **executable**.

**Loader:**
The loader performs the following tasks:

1. Reads the executable and create an address space large enough for the program and its data.
2. Copies the instructions and data into memory.
3. Initializes the machines registers and sets the stack pointer.
4. Load the Program Counter with the address of the entry point of the program (main method in Java or C)
5. Start the execution.

- **Hybrid Implementation Systems**
  - Source code is translated into an intermediate form and run by a software virtual machine.
  - E.g., Java compiler produces byte code that can be run on any Java virtual machine (JVM).

- **Programming Environment**
  - Programming Environment is a set of tools that help programmer to develop applications.