Chapter 3: Semantics

Introduction

Language implementors
- Understand how all the constructs of the language are form and their intended effect when executed.

Language users
- Determine how to encode a possible solution of a problem (program) using the reference manual of the programming language.
- Less knowledge of how to correctly define the semantics of a language.

Well-designed programming language
- Semantics should follow directly from syntax.
- Form of a statement should strongly suggest what the statement is meant to accomplish.

Definition of a programming language
- Complete: semantics and syntax are fully defined.
- A language should provides a variety of different constructs, each one with a precise definition.

Language manuals
- Definition of semantics is given in ordinary natural language.

Construct
- Syntax: a rule (or set of rules) from a BNF or other formal grammar.
- Semantics: a few paragraphs and some examples.

Natural language description
- Ambiguous in its meaning
- Different readers come away with different interpretations of the semantics of a language construct.
- A method is needed for giving a readable, precise, and concise definition of the semantics of an entire language.
Chapter 3: Semantics

Static Semantics

- BNFs cannot describe all of the syntax of programming languages.
  - Some context-specific parts are left out.
- Is there a form to generate \( L = \{a^n b^n c^n\} \) using a context-free grammar or a BNF?
- An attempt:
  
  \[
  \begin{align*}
  &<\text{string}> ::= <\text{a seq}> <\text{b seq}> <\text{c seq}> \\
  &<\text{a seq}> ::= a | <\text{a seq}> a \\
  &<\text{b seq}> ::= b | <\text{b seq}> b \\
  &<\text{c seq}> ::= c | <\text{c seq}> c
  \end{align*}
  \]

  No context-free grammar generates \( L \).

Chapter 3: Semantics

Static Semantics

- Some problems have nothing to do with "meaning" in the sense of run-time behavior
  - They are concern about the legal form of the program.
  - Static semantics refers to type checking and resolving declarations.
- Examples:
  - All variables must be declared before they are referenced
  - Ada: the name on the end of a procedure must match the procedure's name
  - Both sides of an assignment must be of the same type.

Chapter 3: Semantics

Attribute Grammars: Basic Concepts

- A context-free grammar extended to provide context-sensitivity information by appending attributes to each node of a parse tree.
- Each distinct symbol in the grammar has associated with it a finite, possibly empty, set of attributes.
  - Each attribute has a domain of possible values.
  - An attribute may be assigned values from its domain during parsing.
  - Attributes can be evaluated in assignments and conditions.

Chapter 3: Semantics

Attribute Grammars: Generalities

- Two classes of attributes:
  - Synthesized attribute
    - Gets its value from the attributes attached to its children (subtree below the node).
    - Used to pass semantic information up a parse tree.
  - Inherited attribute
    - Gets its value from the attributes attached to the parent (subtree above the node).
    - Used to pass semantic information down and across a tree.

Chapter 3: Semantics

Attribute Grammars: Parse Tree
Attribute Grammar Definition

- Associate some functions to compute the value of the attributes with each production in the grammar.
- These local definitions associated with each production of the grammar define the values of the attributes for all parse trees.
- Given the definitions and a parse tree, algorithms exist to compute the attributes of all the nodes in the three.

Attribute Grammars

- Starting with the underlying context-free grammar \( G = \langle N, T, P, S \rangle \)
- For every production \( p \in P \):
  - Number of terminal and nonterminal symbols in string \( a : n(p) \).
  - If \( a \) is the empty string, then \( n(p) = 0 \).
  - Sometimes each symbol of a production will be considered individually.
    - For all production \( p : A \rightarrow a \) or \( p_0 \rightarrow p_1, p_2, \ldots, p_n(p) \).

Attribute Grammars: attributes

- There is a set of attributes \( At(x) \subseteq At \) to every grammar symbol \( x \in N \cup T \)
  - \( At(x) \) can be seen as additional information about the symbol \( x \).
- Set
  - \( In(x) = \{ a \in At(x) \mid a \in In \} \)
  - \( Syn(x) = \{ a \in At(x) \mid a \in Syn \} \)
- Requirements:
  - \( In(S) = \emptyset \) (start symbol can inherit no information)
  - \( For all t \in T, Syn(t) = \emptyset \) (there is no structure beneath a terminal from which to synthesize information)

Attribute Grammars: rules

- Same attribute can be associated with different symbols appearing in the same grammar rule.
  - Example: \( S \rightarrow AB \), all could inherit attribute \( \text{int} \) associated to them: \( In(S) = In(A) = In(B) = \{ \text{int} \} \).
  - It is impossible to consider the set of attributes associated with all the symbols of a production without losing track of which attributes appear more than once.
  - More confusing: productions that have a nonterminal appearing more than once, as in \( S \rightarrow ASA \).

Attribute Grammars: attribute occurrences

- Attribute occurrence of a rule \( p \) is an ordered pair of attributes and natural number \( <a, j> \) representing the attribute \( a \) at position \( j \) in production \( p \).
  - Particular rule \( p \in P \) an attribute occurrence at \( j \) will be written \( p_j.a \).
  - Set of attribute occurrences for a production \( p \) is defined: \( AO(p) = \{ p_j.a \mid a \in At(p), 0 \leq j \leq n(p) \} \)
Attribute Grammars: attribute occurrences

- Set of attribute occurrences for a rule is divided into two disjoint subsets.
  - Defined occurrences for a production $p$:
    \[ \text{DO}(p) = \{ p_i.s \ (\text{ Syn}(p_i) \) $\cup \{ p_j.i \ (\text{ In}(p_j), 1 \leq j \leq n(p) \} \}
  - Used occurrences for a production $p$:
    \[ \text{UO}(p) = \{ p_i.i \ (\text{ In}(p_i)) \cup \{ p_j.s \ (\text{ Syn}(p_j), 1 \leq j \leq n(p) \} \}

In a parse tree, the set DO(p) represents the information flowing into the node of the parse tree labeled $p_0$.

Attribute Grammars: used attribute occurrences

- Used attribute occurrences (the information flowing in) are $\text{In}(S)$, $\text{Syn}(A)$, and $\text{Syn}(B)$.

Attribute Grammars: defined attribute occurrences

- Defined attribute occurrences (the information flowing out) are $\text{Syn}(S)$, $\text{In}(A)$, and $\text{In}(B)$.

Attribute Grammars: semantic function

- Semantic function $f_{p,v}$:
  - For every attribute occurrence $v \in \text{DO}(p)$
  - Defined values for attributes in $\text{DO}(p)$ in terms of the values of the attributes in $\text{UO}(p)$.
  - Produces a value for the attribute $a$ from values of the attributes of $\text{UO}(p)$.
  - There is no requirement that all the attribute occurrences of $\text{UO}(p)$ are used by $f_{p,v}$.
  - Dependency set ($D_{p,v}$) of $f_{p,v}$ is the set of attribute occurrences used (subset of $\text{UO}(p)$).
  - $D_{p,v}$ could be empty
  - Value of the attribute: computed without any other additional information. The function $f_{p,v}$ is a constant.

Attribute Grammar

- An attribute grammar as a context-free grammar with two disjoint sets of attributes (inherited and synthesized) and semantic functions for all defined attribute occurrences.
Attribute Grammar: binary digits example

Context-free grammar that generates strings of binary digits.

\[ p: B \rightarrow D \]
\[ q: B \rightarrow D B \]
\[ r: D \rightarrow 0 \]
\[ s: D \rightarrow 1 \]

Attributes:
- \( val \): accumulate the value of the binary numbers
- \( pow \) and \( pos \): keep track of the position and the power of 2.

Attribute Grammar: binary digits example

Function definitions for the eight defined attribute occurrences.

\[ p: B \rightarrow D \]
\[ B.pos := 1 \]
\[ B.val := D.val \]
\[ D.pow := 0 \]

\[ q: B_1 \rightarrow D B_2 \]
\[ B_1.pos := B_2.pos + 1 \]
\[ B_1.val := B_2.val + D.val \]
\[ D.pow := B_2.pos \]

\[ r: D \rightarrow 0 \]
\[ D.val := 0 \]

\[ s: D \rightarrow 1 \]
\[ D.val := 0 \times 2^{pos} \]

Dynamic Semantics

Semantics of a programming language is the definition of the meaning of any program that is syntactically valid.

Intuitive idea of programming meaning: “whatever happens in a (real or model) computer when the program is executed.”

A precise characterization of this idea is called operational semantics.
Dynamic Semantics

- A third way to view the semantics of a programming language is to define the meaning of each type of statement that occurs in the (abstract) syntax as a state-transforming mathematical function.
  - The meaning of a program can be expressed as a collection of functions operating on the program state.
  - This approach is called denotational semantics.

Dynamic Semantics: advantages and disadvantages

- Operational Semantics
  - Advantage of representing program meaning directly in the code of a real (or simulated) machine.
  - Potential weakness, since the definition of semantics is confined to a particular architecture (either real or abstract).
  - Virtual machine also needs a semantic description, which adds complexity and can lead to circular definitions.

Operational Semantics

- Provides a definition of program meaning by simulating the program’s behavior on a machine model that has a very simple (through not necessarily realistic) instruction set and memory organization.
- Definition of the virtual computer can be described using an existing programming language or a virtual computer (idealized computer).

Operational Semantics: process

- The process:
  - Identify a virtual machine (an idealized computer).
  - Build a translator (translates source code to the machine code of an idealized computer).
  - Build a simulator for the idealized computer.
- Operational semantics is sometimes called transformational semantics, if an existing programming language is used in place of the virtual machine.
Operational Semantics: automaton

Automaton could be used as a virtual machine:
- More complex that the simple automata models used in the study of syntax and parsing

Automaton has:
- Internal state that corresponds to the internal state of the program when it is executing;
- The state contains all the values of the variables, the executable program, and various system-defined housekeeping data structures.

---

Operational Semantics: evaluation

Advantages:
- May be simple, intuitive for small examples/
- Good if used informally.
- Useful for implementation.

Disadvantages:
- Very complex for large programs.
- Depends on programming languages of lower levels (not mathematics).

Uses:
- Vienna Definition Language (VDL) used to define PL/I (Wegner, 1972).
- Compiler work

---

Axiomatic Semantics

Programmers: confirm or prove that a program does what it is supposed to do under all circumstances

Axiomatic semantics provides a vehicle for developing proofs that a program is “correct”.

---

Axiomatic Semantics

Example: prove mathematically that the C/C++ function Max actually computes as its result the maximum of its two parameter: a and b.
- Calling this function one time will obtain an answer for a particular a and b, such as 8 and 13. But the parameters a and b define a wide range of integers, so calling it several times with all the different values to prove its correctness would be an infeasible task.
Axiomatic Semantics

- Construct a proof to prove the correctness of a program
  - The meaning of a statement is defined by the result of the logical expression that precedes and follows it.
  - Those logical expressions specify constraints on program variables.
  - The notation used to describe constraints is predicate calculus.

Axiomatic Semantics: assertions

- The logical expressions used in axiomatic semantics are called assertions.
  - **Precondition**: an assertion immediately preceding a statement that describes the constraints on the program variables at that point.
  - **Postcondition**: an assertion immediately following a statement that describes the new constraints on some variables after the execution of the statement.

Axiomatic Semantics: assertions

- Example
  \[
  \text{sum} = 2 \times x + 1 \{ \text{sum} > 1 \}
  \]
  - Preconditions and postconditions are enclosed in braces
  - Possible preconditions:
    - \( \{x > 10\} \)
    - \( \{x > 50\} \)
    - \( \{x > 1000\} \)
    - \( \{x > 0\} \)

Axiomatic Semantics: weakest precondition

- It is the least restrictive precondition that will guarantee the validity of the associated postcondition.
- Correctness proof of a program can be constructed if the weakest condition can be computed from the given postcondition.
- Construct preconditions in reverse:
  - From the postcondition of the last statement of the program generate the precondition of the previous statement.
  - This precondition is the postcondition of the previous statement, and so on.

Axiomatic Semantics: weakest precondition

- The precondition of the first statement states the condition under which the program will compute the desired results.
- Correct program: If the precondition of the first statement is implied by the input specification of the program.
- The computation of the weakest precondition can be done using:
  - **Axiom**: logical statement that is assumed to be true.
  - **Inference rule**: method of inferring the truth of one assertion on the basis of the values of other assertions.

Axiomatic Semantics: assignment statements

- Let \( x = E \) be a general assignment statement and \( Q \) its postconditions.
  - **Precondition**: \( P \Rightarrow Q \Rightarrow E \)
  - \( P \) is computed as \( Q \) with all instance of \( x \) replaced by \( E \)
- Example
  \[
  a = \frac{b}{2} - 1 \{a < 10\}
  \]
  Weakest precondition: substitute \( b/2 - 1 \) in the postcondition \( \{a < 10\} \)
  \[
  b/2 - 1 < 10
  b < 22
  \]
Axiomatic Semantics: assignment statements

• General notation of a statement: \( \{P\} S \{Q\} \)
• General notation of the assignment statement: \( \{Q\}_{x=E} x = E \{Q\} \)
• More examples:
  \[
  x = 2*y-3 \quad (x>25) \quad 2*y-3 > 25 \quad y > 14
  
  x = x+y-3 \quad (x>10) \quad x+y-3 > 10 \quad y > 13-x
  
  \]

Axiomatic Semantics: assignment statements

• An assignment with a precondition and a postcondition is a theorem.
  
• If the assignment axiom, when applied to the postcondition and the assignment statement, produces the given precondition, the theorem is proved.

• Example:
  \[
  \{x > 5\} x = x-3 \quad \{x>0\}
  \]
  
  Using the assignment axiom on
  \[
  x = x-3 \quad (x>0) \quad \{x > 3\}
  \]
  
  \[
  \{x > 5\} \implies \{x > 3\}
  \]

Axiomatic Semantics: sequences

• The weakest precondition cannot be described by an axiom (only with an inference rule)
  It depends on the particular kinds of statements in the sequence.
• Inference rule:
  • The precondition of the second statement is computed.
  • This is used as the postcondition of the first statement.
  • The precondition of the first element is the precondition of the whole sequence.

Axiomatic Semantics: sequences

• Example:
  \[
  y = 3*x+1; \quad x = y+3; \quad \{x < 10\}
  \]
  
  Precondition of last assignment statement
  \[
  y < 7
  \]
  
  Used as postcondition of the first statement
  \[
  3*x+1 < 7 \quad x < 2
  \]

Axiomatic Semantics: selection

• Inference rule:
  • Selection statement must be proven for both when the Boolean control expression is true and when it is false.
  • The obtained precondition should be used in the precondition of both the then and the else clauses.

Axiomatic Semantics: selection

• Example:
  \[
  \text{if } (x > 0) \quad y = y-1 \quad \text{else } y = y+1
  \]
  
  \[
  \{y > 0\}
  \]
  
  Axiom for assignment on the "then" clause
  \[
  y = y-1 \quad \{y > 0\} \quad y-1 > 0 \quad y > 1
  \]
  
  Same axiom to the "else" clause
  \[
  y = y+1 \quad \{y > 0\} \quad y+1 > 0 \quad y > -1
  \]
  
  But \( y > 1 \) \( \implies \) \( y > -1 \)
  Precondition of the whole statement: \( y > 1 \)
Axiomatic Semantics: evaluation

- Advantages:
  - Can be very abstract.
  - May be useful in program correctness proofs.
  - Solid theoretical foundations.
- Disadvantages:
  - Predicate transformers are hard to define.
  - Hard to give complete meaning.
  - Does not suggest implementation.
- Uses:
  - Semantics of Pascal.
  - Reasoning about correctness.

Denotational Semantics

- Most rigorous, abstract, and widely known method.
- Based on recursive function theory.
- Originally developed by Scott and Strachey (1970).
- Key idea: define a function that maps a program (a syntactic object) to its meaning (a semantic object).
  - It is difficult to create the objects and mapping functions.

Denotational vs. Operational

- Denotational semantics is similar to high-level operational semantics, except:
  - Machine is gone.
  - Language is mathematics (lambda calculus).
- Differences:
  - In operational semantics, the state changes are defined by coded algorithms for a virtual machine.
  - In denotational semantics, they are defined by rigorous mathematical functions.

Denotational Semantics: evaluation

- Advantages:
  - Compact and precise, with solid mathematical foundation.
  - Provides a rigorous way to think about programs.
  - Can be used to prove the correctness of programs.
  - Can be an aid to language design.
- Disadvantages:
  - Requires mathematical sophistication.
  - Hard for programmers to use.
- Uses:
  - Semantics for Algol 60.
  - Compiler generation and optimization.

Summary

- Each form of semantic description has its place:
  - Operational
    - Informal descriptions
    - Compiler work
  - Axiomatic
    - Reasoning about particular properties
    - Proofs of correctness
  - Denotational
    - Formal definitions
    - Probably correct implementations