Chapter 3
Semantics

Topics
- Introduction
- Static Semantics
- Attribute Grammars
- Dynamic Semantics
- Operational Semantics
- Axiomatic Semantics
- Denotational 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.

Introduction

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.

Introduction

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.

Introduction

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.
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:

\[
\begin{align*}
\texttt{string} &::= \texttt{a seq} \texttt{b seq} \texttt{c seq} \\
\texttt{a seq} &::= a | \texttt{a seq} a \\
\texttt{b seq} &::= b | \texttt{b seq} b \\
\texttt{c seq} &::= c | \texttt{c seq} c
\end{align*}
\]

\( L' = \{a^k b^m c^n | k = 1, m = 1, n = 1\} \)

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

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.

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.
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 tree.

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 \( p \in P : A \rightarrow a \) or \( p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n \)

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
  - \( In(x) = \{ a \in At(x) \ | \ a \in In \} \)
  - \( Syn(x) = \{ a \in At(x) \ | \ a \in Syn \} \)
- Requirements:
  - \( In(S)=\emptyset \) (start symbol can inherit no information)
  - \( \forall x \in T, \ Syn(x)=\emptyset \) (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 \( int \) associated to them: \( In(S)=In(A)=In(B)=\{ int \} \).
  - 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: \( AO(p) = \{ p_j.a \ | \ a\in At(p), \ 0 \leq j \leq n(p) \} \)
Chapter 3: Semantics

Attribute Grammars: attribute occurrences

Set of attribute occurrences for a rule is divided into two disjoint subsets.

- Defined occurrences for a production \( p \):
  \[
  DO(p) = \{ p_j.s | s \in \text{Syn}(p_j), 1 \leq j \leq n(p) \}
  \]
  In a parse tree, the set \( DO(p) \) represents the information flowing into the node of the parse tree labeled \( p \).

- Used occurrences for a production \( p \):
  \[
  UO(p) = \{ p_0.i | i \in \text{In}(p_0) \} \cup \{ p_j.s | s \in \text{Syn}(p_j), 1 \leq j \leq n(p) \}
  \]
  In a parse tree, the set \( UO(p) \) represents the information flowing out 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 \( \text{Syn}(B) \).<br>

\[
\begin{array}{ccc}
S & A & B \\
\text{Syn}(S) & \text{Syn}(A) & \text{Syn}(B) \\
\text{In}(S) & \text{In}(A) & \text{In}(B) \\
\end{array}
\]

Attribute Grammars: defined attribute occurrences

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

\[
\begin{array}{ccc}
S & A & B \\
\text{Syn}(S) & \text{Syn}(A) & \text{Syn}(B) \\
\text{In}(S) & \text{In}(A) & \text{In}(B) \\
\end{array}
\]

Attribute Grammars: semantic function

Semantic function \( f_{p,v} \):

- For every attribute occurrence \( v \in DO(p) \)
- Defined values for attributes in \( DO(p) \) in terms of the values of the attributes in \( UO(p) \).
- Produces a value for the attribute \( a \) from values of the attributes of \( UO(p) \).
- There is no requirement that all the attribute occurrences of \( UO(p) \) are used by \( f_{p,v} \).
- Dependency set \( (D_{p,v}) \) of \( f_{p,v} \) is the set of attribute occurrences used (subset of \( UO(p) \))
- \( D_{p,v} \) 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.
  
  \[
  \begin{align*}
  p & : B & \rightarrow & D \\
  q & : B & \rightarrow & D\ B \\
  r & : D & \rightarrow & 0 \\
  s & : D & \rightarrow & 1
  \end{align*}
  \]

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

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>synthesized</td>
<td>pos</td>
<td>val</td>
</tr>
<tr>
<td>inherited</td>
<td></td>
<td>pow</td>
</tr>
</tbody>
</table>

Attribute Grammar: binary digits example

- Compute the defined and the used occurrences for each production
- The defined occurrences is the set of synthesized attributes of the LHS plus the set of inherited attributes of all the grammar symbols of the RHS.

<table>
<thead>
<tr>
<th></th>
<th>Defined</th>
<th>Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.pos, B.val, D.pow</td>
<td>D.val</td>
<td></td>
</tr>
<tr>
<td>pos, B.val, D.pow</td>
<td>B.pos, B.val, D.val</td>
<td></td>
</tr>
<tr>
<td>D.val</td>
<td>D.pow</td>
<td></td>
</tr>
<tr>
<td>Dual</td>
<td>Dual</td>
<td></td>
</tr>
</tbody>
</table>

Attribute Grammar: binary digits example

- Function definitions for the eight defined attribute occurrences.

  \[
  \begin{align*}
  p & : B & \rightarrow & D \\
  & & B.pos & := 1 \\
  & & B.val & := D.val \\
  & & D.pow & := 0 \\
  q & : B & \rightarrow & D\ B \\
  & & B.pos & := B_2.pos + 1 \\
  & & B_1.val & := B_2.val + D_1.val \\
  & & D.pow & := B_2.pos \\
  r & : D & \rightarrow & 0 \\
  s & : D & \rightarrow & 1 \\
  & & D.val & := 0 \\
  & & D.val & := 2^D.pow
  \end{align*}
  \]

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

- Another way to view programming meaning is to start with a formal specification of what a program is supposed to do, and then rigorously prove that the program does that by using a systematic series of logical steps.
  - This approach evokes the idea of axiomatic 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

- Axiomatic semantics is useful in the exploration of formal properties of programs.
  - Programmers who must write provably correct programs from a precise set of specification are particularly well-served by this semantic style.
- Denotational semantics is valuable because its functional style brings the semantic definition of a language to a high level of mathematical precision.
  - Language designers obtain a functional definition of the meaning of each language construct that is independent of any particular machine architecture.

Operational Semantics

- Provides a definition of program meaning by simulating the program’s behavior on a machine model that has a very simple (though 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).
- 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: process

- Change in the state of the machine (memory, registers, etc) defines the meaning of the statement.
- The operational semantics of a high-level language can be described using a virtual computer.
  - A pure hardware interpreter is too expensive.
  - A pure software interpreter has also problems:
    - Machine-dependent
    - Difficult to understand
  - A better alternative: a complete computer simulation.
Chapter 3: Semantics

Operational Semantics: automaton

- Automaton could be used as a virtual machine:
  - More complex than 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: process

Example:

Pascal statement
for i := x to y do
begin
  ...
end

Operational Semantics

for i := x to y do
begin
  ...
end

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 specifies 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
  \[
  \text{sum} = 2 \times x + 1 \begin{cases} \text{sum} > 1 \end{cases}
  \]
  - 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: 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
  \[
  a = b/2-1 \begin{cases} a<10 \end{cases}
  \]
  Weakest precondition: substitute \( b/2-1 \) in the postcondition \( a<10 \)
  \[
  b/2-1 < 10 \Rightarrow b < 22
  \]
Chapter 3: Semantics

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:
  \[x = 2\times y - 3 \quad (x>25) \quad 2\times y - 3 > 25 \]
  \[y > 14 \]
  \[x = x+y - 3 \quad (x>10) \quad x+y - 3 > 10 \]
  \[y > 13-x \]

Chapter 3: Semantics

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
    \[x = x-3 \quad (x>0)\]
    \[\{x > 3\}\]
    \[\{x > 5\} \text{ implies } (x > 3)\]

Chapter 3: Semantics

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.

Chapter 3: Semantics

Axiomatic Semantics: sequences

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

Chapter 3: Semantics

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.

Chapter 3: Semantics

Axiomatic Semantics: selection

• Example:
  **if** \( (x > 0)\)
  \[y = y-1\]
  **else** \[y = y+1\]
  \[\{y > 0\}\]
  **Axiom for assignment on the "then" clause**
  \[y = y-1 \quad \{y > 0\}\]
  \[y-1 > 0\]
  \[y > 1\]
  **Same axiom to the "else" clause**
  \[y = y+1 \quad \{y > 0\}\]
  \[y+1 > 0\]
  \[y > -1\]
  **But** \[\{y > -1\}\]
  **Precondition of the whole statement:** \[y > 1\]
Chapter 3: Semantics

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 Strachery (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

Chapter 3

Attribute Grammars
Meaning

What is the semantics or meaning of the expression: 2+3
- Its value: 5
- Its type (type checker): int
- A string (infix-to-postfix translator): + 2 3

The semantics of a construct can be any quantity or set of quantities associated with the construct.

Attribute Grammars

Formalism for specifying semantics based on context-free grammars (BNF).

Used to solve some typical problems:
- Type checking and type inference
- Compatibility between procedure definition and call.

Associate attributes with terminals and nonterminals.

Associate semantic functions with productions.
- Used to compute attribute values.

Example: Evaluating arithmetic expressions

\[
\begin{align*}
\text{<exp> ::= <exp> + <term>} \\
\text{<exp> ::= <exp> – <term>} \\
\text{<exp> ::= <term>} \\
\text{<term> ::= <term> * <factor>} \\
\text{<term> ::= <term> div <factor>} \\
\text{<term> ::= <factor>} \\
\text{<factor> ::= ( <exp> )} \\
\text{<factor> ::= num}
\end{align*}
\]

Attributes

A quantity associated with a construct.
- \( X.a \) for attribute \( a \) of \( X \) (\( X \) is either a nonterminal or a terminal).

Attributes have values:
- Each occurrence of an attribute of an attribute in a parse tree has a value.

Grammar symbols can have any number of attributes.

Example: 7*5

\[
\begin{align*}
\text{<exp>.val = 35} \\
\text{<term>.val = 7} \\
\text{<factor>.val = 5} \\
\text{<num>.val = 7}
\end{align*}
\]

Attributes

Syntax symbols can return values (sort of output parameters)
- Digits can return its numeric value
  \( \text{digit < [val]} \)

Nonterminal symbols can have also input attributes.
- Parameters that are passed from the "calling" production.
  \( \text{number < [base, [val]} \)
  - base: number base (e.g. 10 or 2 or 16)
  - val: returned value of the number
Chapter 3: Semantics

Information Flow

- Inherited
- Synthesized
- Computed
- Available

Synthesized Attributes
- The values is computed from the values of attributes of the children.
- Pass information up the parse tree (bottom-up propagation).
- S-attribute grammar uses only synthesized attributes
- Example:
  - Value of expressions
  - Types of expressions

Inherited Attributes
- The values is computed from the values of attributes of the siblings and parent.
- Pass information down the parse tree (top-down propagation) or from left siblings to the right siblings
- Example:
  - Type information
  - Where does a variable occur? LHS or RHS

Example 1
- Translating decimal numbers between 0 and 99 into their English phrases.
  - Number | Phase
    - 0 | zero
    - 10 | ten
    - 19 | nineteen
    - 20 | twenty
    - 31 | thirty one
  - Translations are based on each digit
    - \( 1 \): thirty, the translation of 3 on the left, and one, the translation of 1 on the right.
  - Exceptions:
    - 30 is thirty, not thirty zero
    - 19: is nineteen, not ten nine

Example 1: Syntax

```
<number> ::= <digit>
<number> ::= <digit> <set_digit>
<set_digit> ::= <digit>
<digit> ::= 0|1|2|3|4|5|6|7|8|9

<N> ::= <D>
<N> ::= <D> <S>
<S> ::= <D>
<D> ::= 0|1|2|3|4|5|6|7|8|9
```

Attribute Occurrences
- Same attribute can be associated with different symbols appearing in the same grammar rule.
- Attribute occurrence of a rule \( p \) is an ordered pair of attributes and natural number \( \langle a, j \rangle \) representing the attribute \( a \) at position \( j \) in production \( p \)
- Two disjoint subsets:
  - Defined occurrences for a production:
    - The information flowing into a node of the parse tree.
  - Used occurrences for a production
    - The information flowing out a node of the parse tree.
Chapter 3: Semantics

Rule: \( S \rightarrow AB \)
- Set of inherited attributes of all the grammar symbols on the LHS plus the set of synthesized attributes of the RHS.

Defined Attribute Occurrences

<table>
<thead>
<tr>
<th>Rule: ( S \rightarrow AB )</th>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>synthesized</td>
<td>Syn(S)</td>
<td>Syn(A)</td>
<td>Syn(B)</td>
</tr>
<tr>
<td>inherited</td>
<td>In(S)</td>
<td>In(A)</td>
<td>In(B)</td>
</tr>
</tbody>
</table>

• Set of synthesized attributes of all the grammar symbols on the LHS plus the set of inherited attributes of the RHS.

Semantic Function

Define a semantic function for every defined occurrence in terms of the values of used occurrences.

Example 1: Semantics

<N> ::= <D> N.trans := spell(D.val)
<N> ::= <D> <S> S.in ::= D.val
<S> ::= <D> S.val :=
  if D.val = 0 then decade(S.in)
  else if S.in \leq 1 then spell(10*S.in + D.val)
  else decade(P.in) || spell(D.val)
<D> ::= 0 <D>.val := 0
...<D> ::= 9 <D>.val := 9

Functions spell and decade:
- spell(1) = one, spell(2) = two, ..., spell(19) = nineteen
- decade(0) = zero, decade(1) = ten, ..., decade(9) = ninety

Example 2: Syntax

<B> ::= <D> B.pos := 1
<B> ::= <D> B B.pos := B.pos + 1
<B> ::= <D> <B> B.val := B.val + D.val
<B> ::= 0 B.val := 0
<B> ::= 1 B.val := 2

Example 2: Semantics

<B> ::= <D> B.pos := 1
<B> ::= <D> B B.pos := B.pos + 1
<B> ::= <D> <B> B.val := B.val + D.val
<B> ::= 0 B.val := 0
<B> ::= 1 B.val := 2
Chapter 3: Semantics

Example 2: Sample Parse Tree

Example 3: Syntax

Simple Assignment Statements

<assign> ::= <var> = <expr>
<expr> ::= <var> + <var>
<expr> ::= <var>
<var> ::= X | Y | Z

Example 3: Semantics

<A> ::= <V> = <E>
E.exp := V.act
<E> ::= <V> + <V>
E.act = if (V1.act = int) and
       V2.act := int) then int
       else real
<E> ::= <V>
E.act := E.exp
<V> ::= X | Y | Z
V.act = ...

Attribute Grammars: Summary

An attribute grammar is a context-free grammar with two disjoint sets of attributes (inherited and synthesized) and semantic functions for all defined attribute occurrences.

Attribute Grammar: Process

1. EBNF
2. Attributes
   - Identify the parameters of the syntax symbols.
     - Output attributes (synthesized) yield results.
     - Input attributes (inherited) provide context.
3. Semantic functions

Chapter 3

Operational Semantics
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.

Operational 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).

Change in the state of the machine (memory, registers, etc) defines the meaning of the statement.

Example

Pascal statement

for i := x to y do
begin
  ...
end

Operational Semantics

i := x
loop: if i > y goto out
  ...
  i := i + 1
  goto loop

Out: ...

Operational Semantics (lower-level)

movi, r1
mov y, r2
jmpifless (r2, r1, out)
  ...

Notation

State of a program σ:

A set of pairs <v,val> that represent all active variables and their current assigned values at some stage during the program’s execution.

σ = { <x1, 1>, <y2, 2>, <z3, 3> }

After y = 2 * z + 3

σ = { <x1, 1>, <y, 9>, <z, 3> }

After w = 4

σ = { <x1, 1>, <y, 9>, <z, 3>, <w, 4> }

State transformation of these type of assignments can be represented by a function called overriding union U

σ1 = { <x1, 1>, <y, 2>, <z, 3> } + σ2 = { <y, 9>, <w, 4> }

σ1 U σ2 = { <x1, 1>, <y, 9>, <z, 3>, <w, 4> }
Chapter 3: Semantics

Notation

**Execution rule:**

\[
\frac{\text{premise}}{\text{conclusion}}
\]

"If the premise is true, then the conclusion is true"

Examples

**Addition of two expressions**

\[
\begin{align*}
\sigma(e_1) &= v_1 \\
\sigma(e_2) &= v_2 \\
\sigma(e_1 + e_2) &= v_1 + v_2
\end{align*}
\]

**Assignment statement**

\[
\sigma(s) = o \left( s_{\text{source}} \right) \]

- Suppose: assignment \( x = x + 1 \), current state \( x=5 \)

\[
\begin{align*}
\sigma(x) &= 5 \\
\sigma(1) &= 1
\end{align*}
\]

\[
\sigma(x+1) \Rightarrow 6
\]

\[
\sigma(x = x+1) \Rightarrow \{ ... , <x,5>, ... \} \cup \{ <x,6> \}
\]

Examples

**Conditionals**

\[
\begin{align*}
\sigma(s_{\text{test}}) &\Rightarrow \text{true} \Rightarrow \sigma(s_{\text{then}}) \Rightarrow \sigma_j \\
\sigma(\text{if}(s_{\text{test}})s_{\text{then}} \text{ else } s_{\text{else}}) &\Rightarrow \sigma_j
\end{align*}
\]

\[
\begin{align*}
\sigma(s_{\text{test}}) &\Rightarrow \text{false} \Rightarrow \sigma(s_{\text{else}}) \Rightarrow \sigma_j \\
\sigma(\text{if}(s_{\text{test}})s_{\text{then}} \text{ else } s_{\text{else}}) &\Rightarrow \sigma_j
\end{align*}
\]

Examples

**Loops**

\[
\begin{align*}
\sigma(s_{\text{test}}) &\Rightarrow \text{true} \Rightarrow \sigma(s_{\text{body}}) \Rightarrow \sigma_j \\
\sigma(\text{while}(s_{\text{test}})s_{\text{body}}) &\Rightarrow \sigma_j
\end{align*}
\]

\[
\begin{align*}
\sigma(s_{\text{test}}) &\Rightarrow \text{false} \Rightarrow \sigma
\end{align*}
\]

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

Chapter 3

Axiomatic Semantics
**Dynamic Semantics**

Another way to view programming meaning is to start with a formal specification of what a program is supposed to do, and then rigorously prove that the program does that by using a systematic series of logical steps.

- This approach evokes the idea of **axiomatic semantics**.

**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.

**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.

**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.
Weakest Precondition

- The precondition of the first statement states the condition under which the program will compute the desired results.
- 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.

Assignment Statements

- Let $x=E$ be a general assignment statement and $Q$ its postconditions.
- 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:
- $P=Q_{x=E}$
- $P$ is computed as $Q$ with all instance of $x$ replaced by $E$

Assignment Statements: examples

- More examples:
  - $x = 4*y+5 \{ x>13 \}$
  - $x = y-3*6 \{ x>-5 \}$
  - $x = 2*y+3*x \{ x>10 \}$

Sequences

- The weakest precondition for a sequence cannot be described by an axiom (only with an inference rule)
- It depends on the particular kinds of statements in the sequence.
- 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.

Sequences: examples

- Example:
  - $y = 3*x+1$
  - $x = y+3$
  - $x < 10$
  - Precondition of last assignment statement $3*x+1 < 7$
  - $y < 7$
  - Used as postcondition of the first statement
  - $x < 2$
  - $x = 3(2^b+a)$
  - $b = 2^a - 1$
  - $b > 5$
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.

Selection: example

- Example:
  ```
  if (x > 0)
  y = y-1
  else y = y+1
  {y > 0}
  Axiom for assignment on the "then" clause
  y = y-1 {y > 0}
  y-1 > 0
  y > 1
  Same axiom to the "else" clause
  y = y+1 {y > 0}
  y+1 > 0
  y > -1
  But {y > 1} => {y > -1}
  Precondition of the whole statement: {y > 1}
  ```

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.

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.

Chapter 3

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