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.

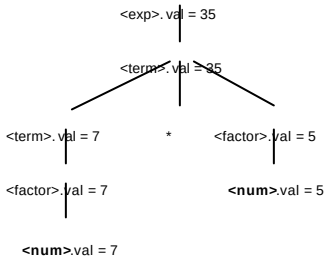
Attributes

- 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: Evaluating arithmetic expressions

<exp> ::= <exp> + <term>
<exp> ::= <exp> – <term>
<exp> ::= <term>
<term> ::= <term> * <factor>
<term> ::= <term> div <factor>
<term> ::= <factor>
<factor> ::= (<exp>)
<factor> ::= num

Example: 7*5

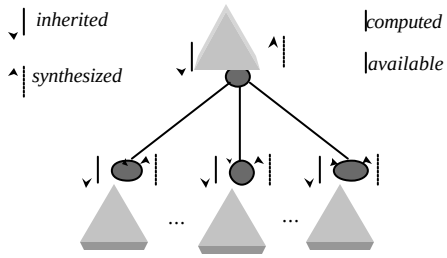


- **val** is the value of the digit
- At the root of the parse tree:
  - **<exp>.val** has value **35**
- At the bottom-left:
  - **<num>.val** has value **7**
- Attributes for terminal symbols:
  - come with the symbol
  - the value of the token
- Attribute values for nonterminals:
  - Defined by semantic rules
    - Attached to productions
- *Decorated parse tree*
  - Attributes attached to the nodes

Attributes

- Syntax symbols can return values (sort of output parameters)
  - Digits can return its numeric value
    - digit <?val>
- Nonterminal symbols can have also input attributes.
  - Parameters that are passed from the "calling" production.
    - number <?base, ?val>
      - base: number base (e.g. 10 or 2 or 16)
      - val: returned value of the number

Information Flow



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	*phrase*
0	zero
10	ten
19	nineteen
20	twenty
31	thirty one

  - Translations are based on each digit
    - 31: 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 *<a,j>* 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.*

Used Attribute Occurrences

Rule: *S? AB*

	S	A	B
synthesized	**Syn(S)**	**Syn(A)**	**Syn(B)**
inherited	**In(S)**	**In(A)**	**In(B)**

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

Defined Attribute Occurrences

Rule: *S? AB*

	S	A	B
synthesized	**Syn(S)**	**Syn(A)**	**Syn(B)**
inherited	**In(S)**	**In(A)**	**In(B)**

- 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.

	Defined	Used
Rule 1	…	…
Rule 2	…	…

Function definitions

Example 1: Semantics

<N> ::= <D>	N.trans := spell(D.val)
<N> ::= <D> <S>	S.in ::= D.val
	N.trans ::= S.trans
<S> ::= <D>	S.val := **if** D.val = 0 **then** *decade*(S.in)
	else if S.in ≤ 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

Decimal value of a binary number

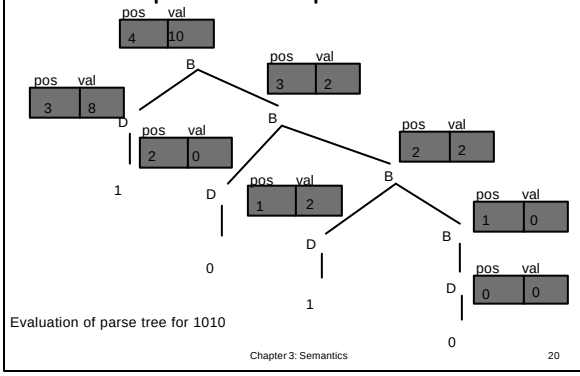
<binary> ::= <digit>
<binary> ::= <digit> <binary>
<digit> ::= 0
<digit> ::= 1

<B> ::= <D>
<B> ::= <D> <B>
<D> ::= 0
<D> ::= 1

Example 2: Semantics

<B> ::= <D>	B.pos := 1
	B.val := D.val
	D.pow := 0
$<B_1>$ ::= <D> $<B_2>$	B_1.pos := B_2.pos + 1
	B_1.val := B_2.val + D.val
	D.pow := B_2.pos
<D> ::= 0	D.val := 0
<D> ::= 1	D.val := $2^{D.pow}$

Example 2: Sample Parse Tree



Evaluation of parse tree for 1010

Example 3: Syntax

Simple Assignment Statements

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

<A> ::= <V> = <E>
<E> ::= <V> + <V>
<E> ::= <V>
<V> ::= X | Y | Z

Example 3: Semantics

<A> ::= <V> = <E>	E.exp := V.act
<E> ::= <V> + <V>	E.act = if (V_1.act = int) and V_2.act := int) then int else real
<E> ::= <V>	E.act := E.exp
<V> ::= X | Y | Z	V.act = …

Variables can be either real or integer.
Both sides of an assignment different: type = real
Same type on both sides of an assignment

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.

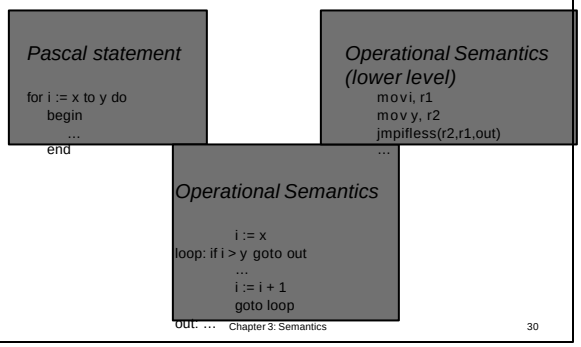
Operational Semantics

- 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.

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.

Example



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 = { <x,1>, <y,2>, <z,3> }
    - After y = 2 * z + 3 σ = { <x,1>, <y,9>, <z,3> }
    - After w = 4 σ = { <x,1>, <y,9>, <z,3>, <w,4> }
- State transformation of these type of assignments can be represented by a function called *overriding union U*
  - σ_1 = { <x,1>, <y,2>, <z,3> }
  - σ_2 = { <y,9>, <w,4> }
  - $\sigma_1 \cup \sigma_2$ = { <x,1>, <y,9>, <z,3>, <w,4> }

Notation

- *Execution rule*:

$$\frac{\textit{premise}}{\textit{conclusion}}$$

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

Examples

- Addition of two expressions

$$\frac{s(e_1) \Rightarrow v_1 \quad s(e_1) \Rightarrow v_1}{s(e_1 + e_2) \Rightarrow v_1 + v_2}$$

- Assignment statement (*s.target = s.source)*

$$\frac{s(s.source) \Rightarrow v}{s(s.target = s.source) \Rightarrow s \cup \{ \langle s.target, v \rangle \}}$$

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

$$\frac{s(x) \Rightarrow 5 \quad s(1) \Rightarrow 1}{s(x+1) \Rightarrow 6}$$

$$s(x = x+1) \Rightarrow \{\ldots, \langle x,5 \rangle, \ldots\} \cup \{ \langle x,6 \rangle \}$$

Examples

- Conditionals (s = if (s.text) s.then else s.else)

$$\frac{s(s.test) \Rightarrow true \quad s(s.then) \Rightarrow s_1}{s(if(s.test)s.then\ else\ s.else) \Rightarrow s_1}$$

$$\frac{s(s.test) \Rightarrow false \quad s(s.else) \Rightarrow s_1}{s(if(s.test)s.then\ else\ s.else) \Rightarrow s_1}$$

Examples

- Loops (s = while (s.test) s.body)

$$\frac{s(s.test) \Rightarrow true \quad s(s.body) \Rightarrow s_1 \quad s_1(while(s.test)s.body) \Rightarrow s_1}{s(while\ (s.text)\ s.body) \Rightarrow s_1}$$

$$\frac{s(s.test) \Rightarrow false}{s(\ while\ (s.text)\ s.body) \Rightarrow s}$$

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 al 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.

Assertions

- Example

```
sum = 2 * x + 1 { sum > 1 }
```

  - Preconditions and postconditions are enclosed in braces
  - Possible preconditions:

```
{ x > 10 }
{ x > 50 }
{ x > 1000 }
{ x > 0 }
```

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.
  - 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.

Assignment Statements

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

```
a = b/2-1 {a<10}
```

Weakest precondition: substitute `b/2-1` in the postcondition `{a<10}`

```
b/2-1 < 10
b < 22
```


Assignment Statements: examples

- General notation of a statement: *{P} S {Q}*
- More examples:
  - `x = 4*y+5 { x>13 }`
  - `X = y-3*6 { x>-5 }`
  - `X = 2*y+3*x { x>10}`

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 {x>0}
```

Using the assignment axiom on

```
x = x-3 {x>0}
{x > 3}
```

`{x > 5}` *implies* `{x > 3}`

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.
- 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.

Sequences: examples

- Example:

```
y = 3*x+1;
x = y+3;
{x < 10}
```

Precondition of last assignment statement

```
y < 7
```

Used as postcondition of the first statement

```
3*x+1 < 7
x < 2
```

- Other example:

a = 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} Þ {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.

Chapter 3

Denotational 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.*

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