Semantic Analysis with Attribute Grammars Part 1

Y.N. Srikant

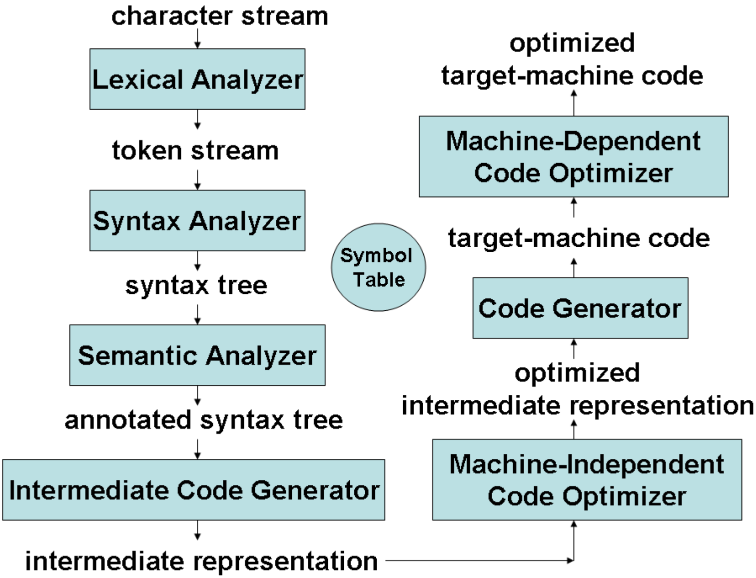
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction
- Attribute grammars
- Attributed translation grammars
- Semantic analysis with attributed translation grammars

Compiler Overview



Semantic Analysis

- Semantic consistency that cannot be handled at the parsing stage is handled here
- Parsers cannot handle context-sensitive features of programming languages
- These are *static semantics* of programming languages and can be checked by the semantic analyzer
  - Variables are declared before use
  - Types match on both sides of assignments
  - Parameter types and number match in declaration and use
- Compilers can only generate code to check *dynamic semantics* of programming languages at runtime
  - whether an overflow will occur during an aritmetic operation
  - whether array limits will be crossed during execution
  - whether recursion will cross stack limits
  - whether heap memory will be insufficient

Static Semantics

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Samples of static semantic checks in *main*

- Types of *p* and return type of *dot_prod* match
- Number and type of the parameters of *dot_prod* are the same in both its declaration and use
- *p* is declared before use, same for *a* and *b*

Static Semantics: Errors given by gcc Compiler

```
int dot_product(int a[], int b[]) {...}

1 main(){int a[10]={1,2,3,4,5,6,7,8,9,10};
2 int b[10]={1,2,3,4,5,6,7,8,9,10};
3 printf("%d", dot_product(b));
4 printf("%d", dot_product(a,b,a));
5 int p[10]; p=dotproduct(a,b); printf("%d",p);}

In function ‘main’:
error in 3: too few arguments to fn ‘dot_product’
error in 4: too many arguments to fn ‘dot_product’
error in 5: incompatible types in assignment
warning in 5: format ‘%d’ expects type ‘int’, but
              argument 2 has type ‘int *’
```

Static Semantics

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Samples of static semantic checks in *dot_prod*

- *d* and *i* are declared before use
- Type of *d* matches the return type of *dot_prod*
- Type of *d* matches the result type of "$*$"
- Elements of arrays *x* and *y* are compatible with "$*$"

Dynamic Semantics

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Samples of dynamic semantic checks in *dot_prod*

- Value of *i* does not exceed the declared range of arrays *x* and *y* (both lower and upper)
- There are no overflows during the operations of “$*$” and “$+$” in `d += x[i]*y[i]`

Dynamic Semantics

```
int fact(int n){
    if (n==0) return 1;
    else return (n*fact(n-1));
}
main(){int p; p = fact(10); }
```

Samples of dynamic semantic checks in *fact*

- Program stack does not overflow due to recursion
- There is no overflow due to “$*$” in `n*fact(n-1)`

Semantic Analysis

- Type information is stored in the symbol table or the syntax tree
  - Types of variables, function parameters, array dimensions, etc.
  - Used not only for semantic validation but also for subsequent phases of compilation
- If declarations need not appear before use (as in C++), semantic analysis needs more than one pass
- Static semantics of PL can be specified using attribute grammars
- Semantic analyzers can be generated semi-automatically from attribute grammars
- Attribute grammars are extensions of context-free grammars

Attribute Grammars

- Let $G = (N, T, P, S)$ be a CFG and let $V = N \cup T$.
- Every symbol X of V has associated with it a set of *attributes* (denoted by $X.a$, $X.b$, etc.)
- Two types of attributes: *inherited* (denoted by $AI(X)$)and *synthesized* (denoted by $AS(X)$)
- Each attribute takes values from a specified domain (finite or infinite), which is its *type*
  - Typical domains of attributes are, integers, reals, characters, strings, booleans, structures, etc.
  - New domains can be constructed from given domains by mathematical operations such as *cross product, map*, etc.
  - *array*: a map, $\mathcal{N} \rightarrow \mathcal{D}$, where, $\mathcal{N}$ and $\mathcal{D}$ are domains of natural numbers and the given objects, respectively
  - *structure*: a cross-product, $A_1 \times A_2 \times \ldots \times A_n$, where n is the number of fields in the structure, and A_i is the domain of the i^{th} field

Attribute Computation Rules

- A production $p \in P$ has a set of attribute computation rules (functions)
- Rules are provided for the computation of
  - Synthesized attributes of the LHS non-terminal of p
  - Inherited attributes of the RHS non-terminals of p
- These rules can use attributes of symbols from the production p only
  - Rules are strictly local to the production p (no side effects)
- Restrictions on the rules define different types of attribute grammars
  - L-attribute grammars, S-attribute grammars, ordered attribute grammars, absolutely non-circular attribute grammars, circular attribute grammars, etc.

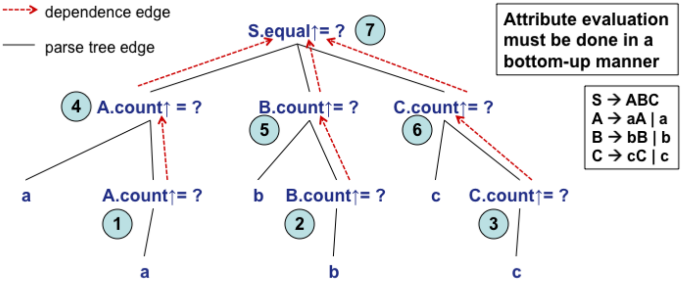
Synthesized and Inherited Attributes

- An attribute cannot be both synthesized and inherited, but a symbol can have both types of attributes
- Attributes of symbols are evaluated over a parse tree by making passes over the parse tree
- Synthesized attributes are computed in a bottom-up fashion from the leaves upwards
  - Always synthesized from the attribute values of the children of the node
  - Leaf nodes (terminals) have synthesized attributes initialized by the lexical analyzer and cannot be modified
  - An AG with only synthesized attributes is an *S-attributed grammar (SAG)*
  - YACC permits only SAGs
- Inherited attributes flow down from the parent or siblings to the node in question

Attribute Grammar - Example 1

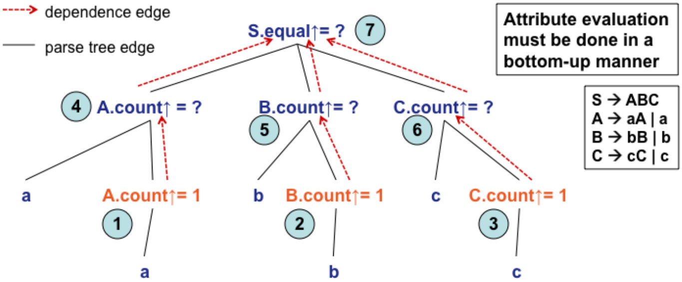
- The following CFG
  $S \rightarrow A\ B\ C,\ A \rightarrow aA \mid a,\ B \rightarrow bB \mid b,\ C \rightarrow cC \mid c$
  generates: $L(G) = \{a^m b^n c^p \mid m, n, p \geq 1\}$
- We define an AG (attribute grammar) based on this CFG to generate $L = \{a^n b^n c^n \mid n \geq 1\}$
- All the non-terminals will have only synthesized attributes
  - $AS(S) = \{equal \uparrow\colon\ \{T, F\}\}$
  - $AS(A) = AS(B) = AS(C) = \{count \uparrow\colon\ integer\}$

Attribute Grammar - Example 1 (contd.)

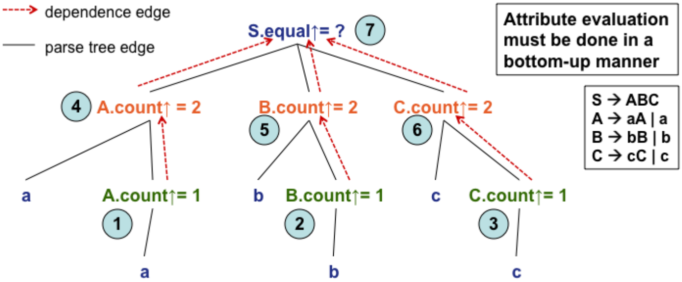


1. $S \to ABC$ $\{S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \text{ then } T \text{ else } F\}$
2. $A_1 \to aA_2$ $\{A_1.count \uparrow := A_2.count \uparrow + 1\}$
3. $A \to a$ $\{A.count \uparrow := 1\}$
4. $B_1 \to bB_2$ $\{B_1.count \uparrow := B_2.count \uparrow + 1\}$
5. $B \to b$ $\{B.count \uparrow := 1\}$
6. $C_1 \to cC_2$ $\{C_1.count \uparrow := C_2.count \uparrow + 1\}$
7. $C \to c$ $\{C.count \uparrow := 1\}$

Attribute Grammar - Example 1 (contd.)

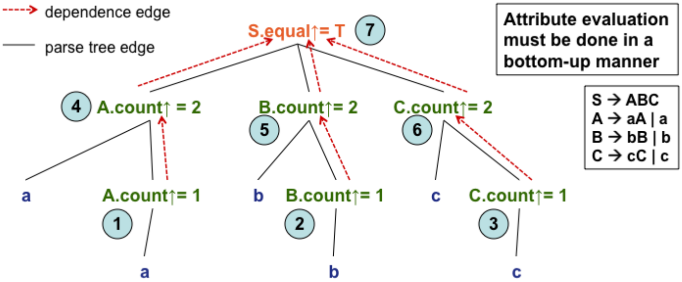


1. $S \rightarrow ABC$ $\{S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \ \text{ then } T \text{ else } F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count \uparrow := A_2.count \uparrow +1\}$
3. $A \rightarrow a$ $\{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count \uparrow := B_2.count \uparrow +1\}$
5. $B \rightarrow b$ $\{B.count \uparrow := 1\}$
6. $C_1 \rightarrow cC_2$ $\{C_1.count \uparrow := C_2.count \uparrow +1\}$
7. $C \rightarrow c$ $\{C.count \uparrow := 1\}$



1. $S \rightarrow ABC$ $\{S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \ \text{then } T \text{ else } F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count \uparrow := A_2.count \uparrow + 1\}$
3. $A \rightarrow a$ $\{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count \uparrow := B_2.count \uparrow + 1\}$
5. $B \rightarrow b$ $\{B.count \uparrow := 1\}$
6. $C_1 \rightarrow cC_2$ $\{C_1.count \uparrow := C_2.count \uparrow + 1\}$
7. $C \rightarrow c$ $\{C.count \uparrow := 1\}$

Attribute Grammar - Example 1 (contd.)



1. $S \rightarrow ABC$ $\{S.equal\uparrow := \text{if } A.count\uparrow = B.count\uparrow \ \& \ B.count\uparrow = C.count\uparrow \ \text{then } T \text{ else } F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count\uparrow := A_2.count\uparrow + 1\}$
3. $A \rightarrow a$ $\{A.count\uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count\uparrow := B_2.count\uparrow + 1\}$
5. $B \rightarrow b$ $\{B.count\uparrow := 1\}$
6. $C_1 \rightarrow cC_2$ $\{C_1.count\uparrow := C_2.count\uparrow + 1\}$
7. $C \rightarrow c$ $\{C.count\uparrow := 1\}$

Attribute Dependence Graph

- Let T be a parse tree generated by the CFG of an AG, G.
- The *attribute dependence graph* (dependence graph for short) for T is the directed graph, $DG(T) = (V, E)$, where

  $V = \{b | b$ is an attribute instance of some tree node$\}$, and

  $E = \{(b, c) | b, c \in V$, b and c are attributes of grammar symbols in the same production p of B, and the value of b is used for computing the value of c in an attribute computation rule associated with production $p\}$

Attribute Dependence Graph

- An AG G is *non-circular*, iff for all trees T derived from G, $DG(T)$ is acyclic
    - Non-circularity is very expensive to determine (exponential in the size of the grammar)
    - Therefore, our interest will be in subclasses of AGs whose non-circularity can be determined efficiently
- Assigning consistent values to the attribute instances in DG(T) is *attribute evaluation*

Attribute Evaluation Strategy

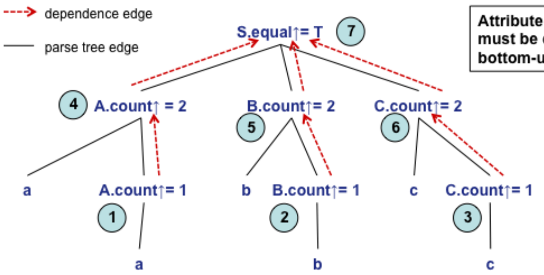
- Construct the parse tree
- Construct the dependence graph
- Perform topological sort on the dependence graph and obtain an evaluation order
- Evaluate attributes according to this order using the corresponding attribute evaluation rules attached to the respective productions
- Multiple attributes at a node in the *parse tree* may result in that node to be visited multiple number of times
  - Each visit resulting in the evaluation of at least one attribute

Attribute Evaluation Algorithm

Input: A parse tree T with unevaluated attribute instances
Output: T with consistent attribute values

```
{ Let (V, E) = DG(T);
  Let W = {b | b ∈ V & indegree(b) = 0};
  while W ≠ φ do
    { remove some b from W;
      value(b) := value defined by appropriate attribute
                  computation rule;
      for all (b, c) ∈ E do
       { indegree(c) := indegree(c) − 1;
         if indegree(c) = 0 then W := W ∪ {c};
       }
    }
}
```

Dependence Graph for Example 1



Attribute evaluation must be done in a bottom-up manner

S → ABC
A → aA | a
B → bB | b
C → cC | c

1,2,3,4,5,6,7 and 2,3,6,5,1,4,7 are two possible evaluation orders. 1,4,2,5,3,6,7 can be used with LR-parsing. The right-most derivation is below (its reverse is LR-parsing order)

S => ABC => ABcC => ABcc => AbBcc => Abbcc => aAbbcc => aabbcc

1. A.count = 1 {A → a, {A.count := 1}}
4. A.count = 2 {A_1 → aA_2, {A_1.count := A_2.count + 1}}
2. B.count = 1 {B → b, {B.count :=1}}
5. B.count = 2 {B_1 → bB_2, {B_1.count := B_2.count + 1}}
3. C.count = 1 {C → c, {C.count :=1}}
6. C.count = 2 {C_1 → cC_2, {C_1.count := C_2.count + 1}}
7. S.equal = 1 {S → ABC, {S.equal := *if* A.count = B.count & B.count = C.count *then* T else F}}

Attribute Grammar - Example 2

- AG for the evaluation of a real number from its bit-string representation
  Example: 110.101 = 6.625
- $N \rightarrow L.R,\ L \rightarrow BL \mid B,\ R \rightarrow BR \mid B,\ B \rightarrow 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value\uparrow: real\}$,
  $AS(L) = \{length\uparrow: integer,\ value\uparrow: real\}$
  1. $N \rightarrow L.R$ $\{N.value\uparrow := L.value\uparrow + R.value\uparrow\}$
  2. $L \rightarrow B$ $\{L.value\uparrow := B.value\uparrow;\ L.length\uparrow := 1\}$
  3. $L_1 \rightarrow BL_2$ $\{L_1.length\uparrow := L_2.length\uparrow + 1;$
     $L_1.value\uparrow := B.value\uparrow * 2^{L_2.length\uparrow} + L_2.value\uparrow\}$
  4. $R \rightarrow B$ $\{R.value\uparrow := B.value\uparrow /2\}$
  5. $R_1 \rightarrow BR_2$ $\{R_1.value\uparrow := (B.value\uparrow + R_2.value\uparrow)/2\}$
  6. $B \rightarrow 0$ $\{B.value\uparrow := 0\}$
  7. $B \rightarrow 1$ $\{B.value\uparrow := 1\}$