CC Lecture 19

Prepared for: 7th Sem, CE, DDU

Prepared by: Niyati J. Buch

Semantic Analysis

What is Semantic Analysis?

- Source program → **Lexical Analyzer** → Token stream → **Syntax Analyzer** → Syntax tree → <u>**Semantic Analyzer**</u> → Annotated syntax tree → **Intermediate Code Generator**

- Semantic consistency that cannot be handled at the paring stage is handled in this phase.

- Parsers cannot handle context-sensitive features of the programming languages.

Static vs. Dynamic Semantics

- There are some **static semantics** of the programming languages that are checked by the semantic analyzer:
  - If variables are declared before use.
  - If types match on both sides of the assignment.
  - If parameter types and number match in the declaration and use.
- Compilers can generate the code to check **dynamic semantics** of the programming languages only at runtime:
  - Whether an overflow will occur during an arithmetic operation
  - Whether the array limits will be crossed during the execution
  - Whether recursion will cross the stack limits
  - Whether the 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

```
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 “+”

Static Semantics: Errors given by gcc Compiler?

```
#include<stdio.h>
int dot_product(int a[], int b[]) {return 1;}
int main(){
        int a[10]={1,2,3,4,5,6,7,8,9,10};
        int b[10]={1,2,3,4,5,6,7,8,9,10};
        printf("%d", dot_product(b));
        printf("%d", dot_product(a,b,a));
        int p[10];
        p=dot_product(a,b);
        printf("%d",p);
}
```

Static Semantics: Errors given by gcc Compiler

```
1.  #include<stdio.h>
2.  int dot_product(int a[], int b[])
    {return 1;}
3.  int main(){
4.          int a[10]={1,2,3,4,5,6,7,8,9,10};
5.          int b[10]={1,2,3,4,5,6,7,8,9,10};
6.          printf("%d", dot_product(b));
7.          printf("%d", dot_product(a,b,a));
8.          int p[10];
9.          p=dot_product(a,b);
10.         printf("%d",p);
11. }
```

6: error: too few arguments to function 'dot_product'

7: error: too many arguments to function 'dot_product'

9: error: assignment to expression with array type

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 the subsequent phases of compilation.
- If the declarations need not appear before the use, then the 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 context-free grammar (**CFG**) consisting of a finite set of grammar rules where
  - **N** is a set of non-terminal symbols.
  - **T** is a set of terminals where **N ∩ T = NULL.**
  - **P** is a set of rules, **P: N → (N ∪ T)***
  - **S** is the start symbol.
- and let **V = N ∪ T**.
- Every symbol **X of V** has associated with it, a set of attributes (denoted by **X:a**; **X:b**, etc.)
- Hence, the name is **attribute grammar**.

Attribute Types

- **Inherited attributes**
  - denoted by **AI(X)**

- **Synthesized attributes**
  - denoted by **AS(X)**

- 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.

Attribute Grammar

- Each attribute takes the values from a specified domain (finite or infinite) [domain is its type]
  - Typical domains of attributes are, integers, reals, characters, strings, booleans, structures, etc.
  - New domains can be constructed from the given domains by mathematical operations like cross product, map, etc.
- **Example: array**
  - a map, N → D, where, N and D are domains of natural numbers and the given objects, respectively
- **Example :structure**
  - a cross-product, A1 X A2 X ... X An, where n is the number of fields in the structure, and Ai is the domain of the ith field

Attribute Computation Rules

- A production **p ϵ P** has a set of attribute computation rules.
- 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.
- 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

- **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 under consideration.

Attribute Grammar - Example 1

- The following CFG(context free grammar)

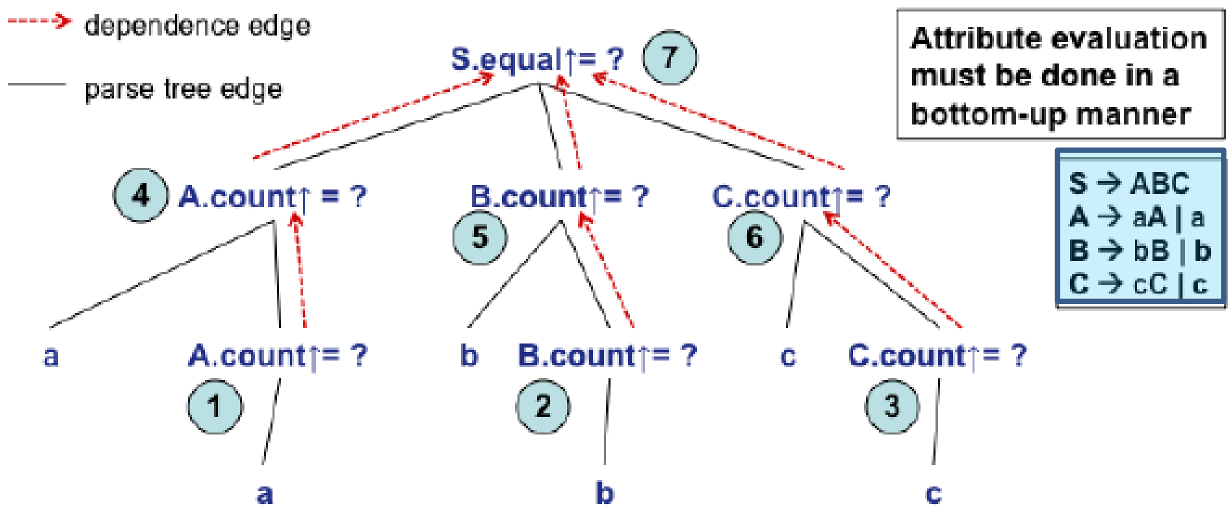
  S → ABC, A → aA|a, B→ bB|b, C →cC|c

  generates: **L(G) = {$a^m b^n c^p$ | m, n, p ≥ 1}**
- We define an AG (attribute grammar) based on this CFG to generate **L = {$a^n b^n c^n$ | n ≥ 1}**
- All the non-terminals will have only synthesized attributes

  AS(S) = {equal ↑: {T, F}}

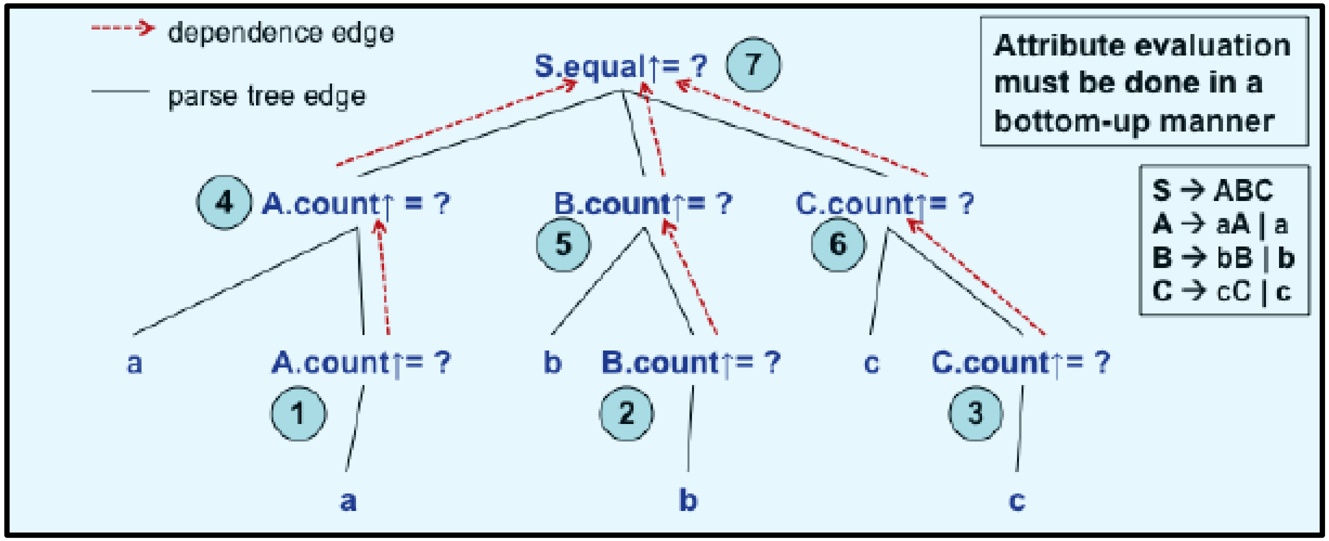
  AS(A) = AS(B) = AS(C) = {count ↑: integer}
- Up arrow means synthesized attribute
- Down arrow means inherited attribute

Attribute Grammar - Example 1



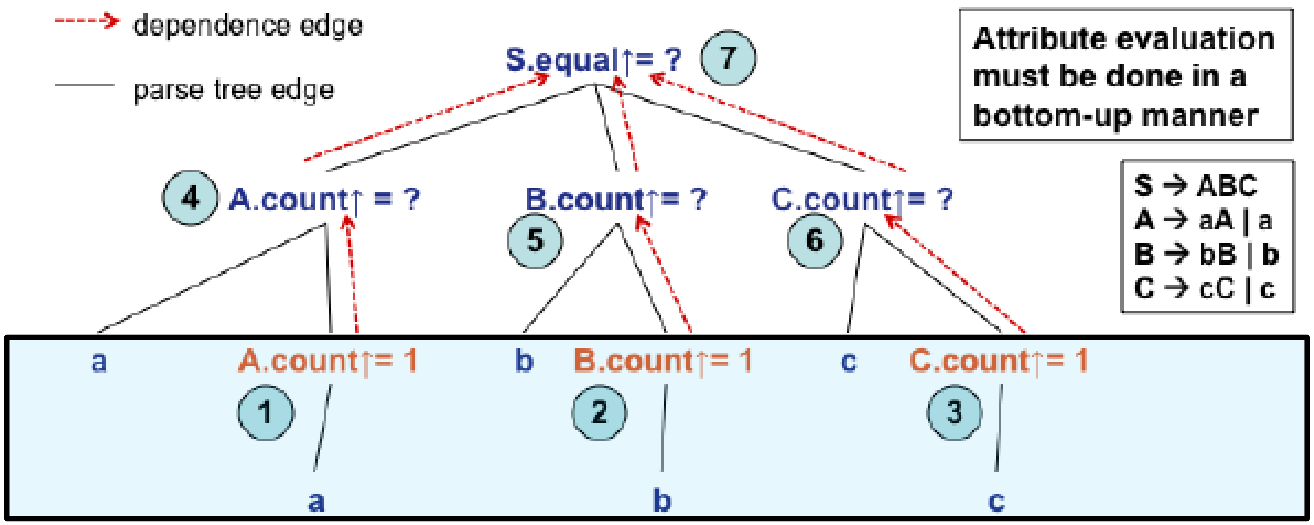
1. $S \to ABC \; \{S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \; \& \; B.count \uparrow = C.count \uparrow \; \text{then } T \text{ else } F\}$
2. $A_1 \to aA_2 \; \{A_1.count \uparrow := A_2.count \uparrow +1\}$
3. $A \to a \; \{A.count \uparrow := 1\}$
4. $B_1 \to bB_2 \; \{B_1.count \uparrow := B_2.count \uparrow +1\}$
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



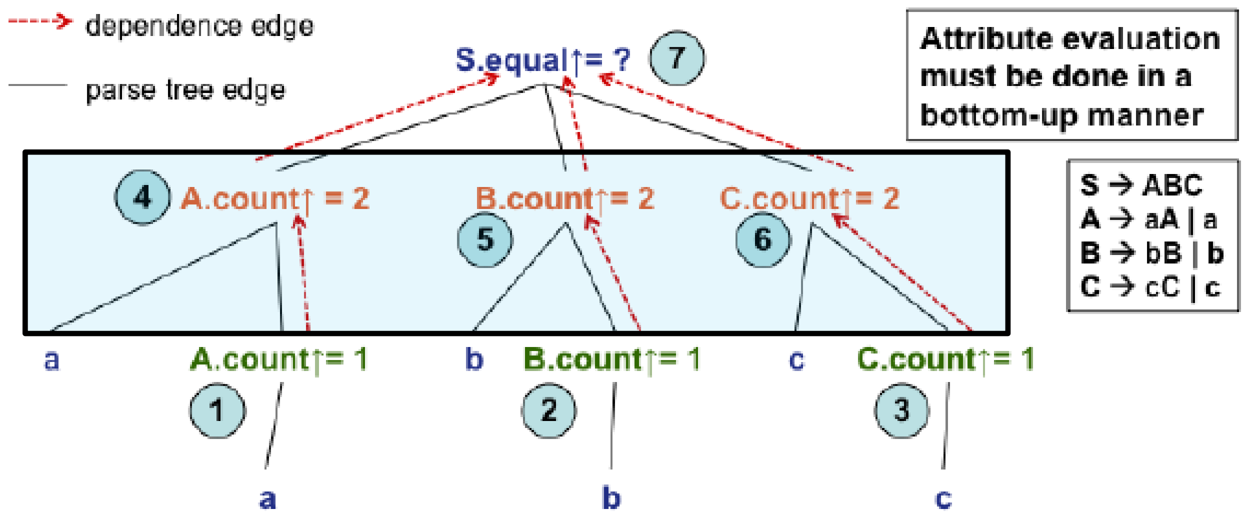
1. $S \to ABC$ $\{S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \ \text{then } T \text{ else } F\}$
2. $A_1 \to aA_2$ $\{A_1.count \uparrow := A_2.count \uparrow +1\}$
3. $A \to a$ $\{A.count \uparrow := 1\}$
4. $B_1 \to bB_2$ $\{B_1.count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



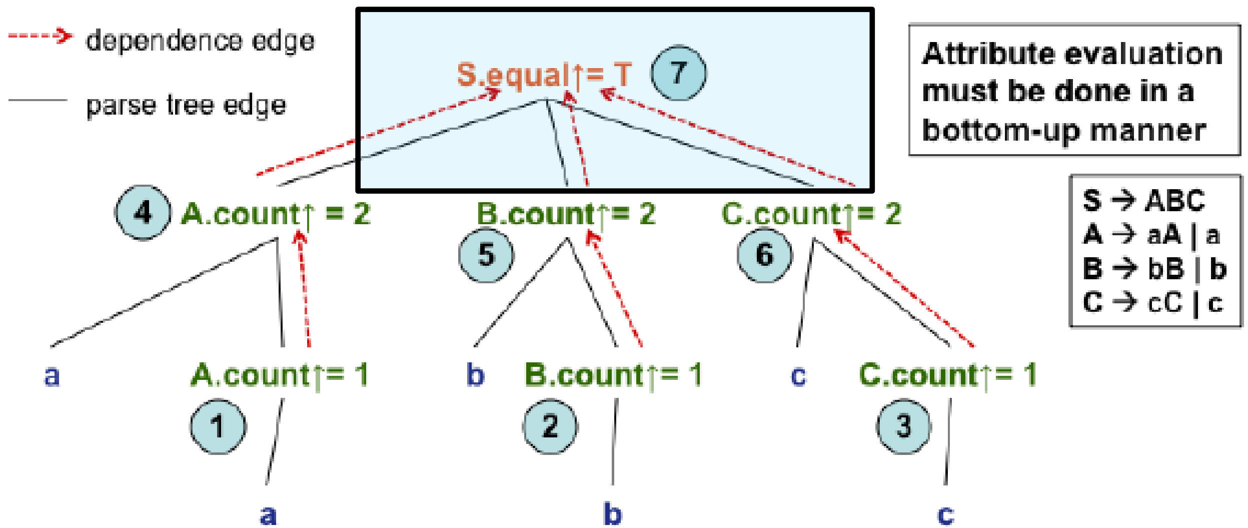
1. $S \rightarrow ABC \; \{S.equal \uparrow := \textit{if } A.count \uparrow = B.count \uparrow \; \& \; B.count \uparrow = C.count \uparrow \; \textit{then } T \textit{ else } F\}$
2. $A_1 \rightarrow aA_2 \; \{A_1.count \uparrow := A_2.count \uparrow + 1\}$
3. $A \rightarrow a \; \{A.count \uparrow := 1\}$
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



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



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. $C \rightarrow c$ $\{C.count \uparrow := 1\}$