Lecture 14

Semantics Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

February 19, 2025

Take aways from the last class

- Task and Example of Semantics Analysis

Take aways from the last class

- Task and Example of Semantics Analysis
- Attribute Grammar Framework

Take aways from the last class

- Task and Example of Semantics Analysis
- Attribute Grammar Framework
- Attributes

Take aways from the last class

- Task and Example of Semantics Analysis
- Attribute Grammar Framework
- Attributes
- Example of propagation and evaluation of attributes.

Synthesized Attributes

- A syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition

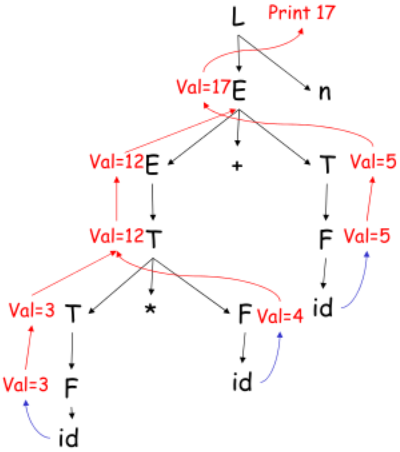
Synthesized Attributes

- A syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition
- A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes

Synthesized Attributes

- A syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition
- A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes
- $L \rightarrow En \qquad Print(E.val)$
  $E \rightarrow E + T \qquad E.val = E.val + T.val$
  $E \rightarrow T \qquad E.val = T.val$
  $T \rightarrow T * F \qquad T.val = T.val * F.val$
  $T \rightarrow F \qquad T.val = F.val$
  $F \rightarrow (E) \qquad F.val = E.val$
  $F \rightarrow digit \qquad F.val = digit.lexval$

Parse tree for 3 * 4 + 5 n



Inherited Attributes

- An inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings

Inherited Attributes

- An inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings
- Used for finding out the context in which it appears

Inherited Attributes

- An inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings
- Used for finding out the context in which it appears
- Possible to use only S-attributes but more natural to use inherited attributes

Inherited Attributes

- An inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings
- Used for finding out the context in which it appears
- Possible to use only S-attributes but more natural to use inherited attributes

  $D \rightarrow TL \qquad L.in = T.type$

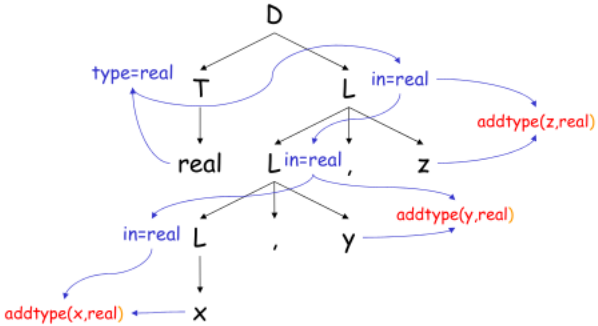
  $T \rightarrow real \qquad T.type = real$

  $T \rightarrow int \qquad T.type = int$

  $L \rightarrow L_1, id \qquad L_1.in = L.in; addtype(id.entry, L.in)$

  $L \rightarrow id \qquad addtype(id.entry, L.in)$

Parse tree for real x, y, z



Dependency Graph

- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c

Dependency Graph

- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c
- The dependencies among the nodes can be depicted by a directed graph called dependency graph

Dependency Graph

- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c
- The dependencies among the nodes can be depicted by a directed graph called dependency graph
- Algorithm:

Dependency Graph

- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c
- The dependencies among the nodes can be depicted by a directed graph called dependency graph
- Algorithm:

```
for each node n in the parse tree do
     for each attribute a of the grammar symbol do
          construct a node in the dependency graph for a
```

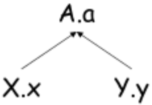
Dependency Graph

- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c
- The dependencies among the nodes can be depicted by a directed graph called dependency graph
- Algorithm:

```
for each node n in the parse tree do
     for each attribute a of the grammar symbol do
         construct a node in the dependency graph for a
for each node n in the parse tree do
    for each semantic rule b = f(c1, c2, ··· , ck) do
        for i = 1 to k do
            construct an edge from ci to b
```

Example

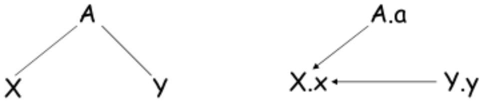
- Suppose $A.a = f(X.x, Y.y)$ is a semantic rule for $A \rightarrow XY$



Example

- Suppose $A.a = f(X.x, Y.y)$ is a semantic rule for $A \rightarrow XY$

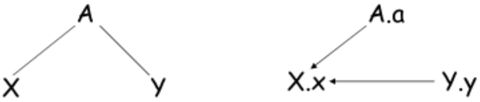
- If production $A \rightarrow XY$ has the semantic rule $X.x = g(A.a, Y.y)$



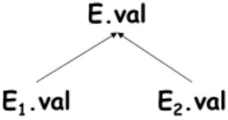
Example

- Suppose $A.a = f(X.x, Y.y)$ is a semantic rule for $A \rightarrow XY$

- If production $A \rightarrow XY$ has the semantic rule $X.x = g(A.a, Y.y)$



- Whenever following production is used in a parse tree
  $E \rightarrow E_1 + E_2 \quad E.val = E_1.val + E_2.val$



Abstract Syntax Tree

- Condensed form of parse tree

Abstract Syntax Tree

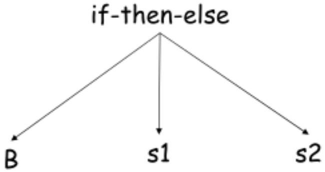
- Condensed form of parse tree
- Useful for representing language constructs.

Abstract Syntax Tree

- Condensed form of parse tree
- Useful for representing language constructs.
- The production $\texttt{S} \rightarrow \texttt{if B then s1 else s2}$ may appear as

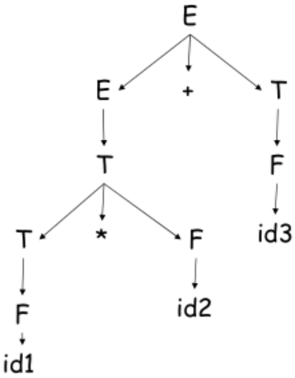
Abstract Syntax Tree

- Condensed form of parse tree
- Useful for representing language constructs.
- The production `S` $\rightarrow$ `if B then s1 else s2` may appear as



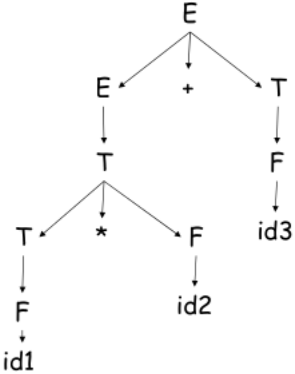
Abstract Syntax Tree

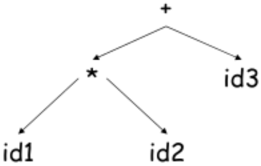
- Chain of single productions may be collapsed, and operators move to the parent nodes



Abstract Syntax Tree

- Chain of single productions may be collapsed, and operators move to the parent nodes





Constructing Abstract Syntax tree for expression

- Each node can be represented as a record

Constructing Abstract Syntax tree for expression

- Each node can be represented as a record
- **Operators:** one field for operator, remaining fields ptrs to operands
  mknode(op,left,right)

Constructing Abstract Syntax tree for expression

- Each node can be represented as a record
- **Operators:** one field for operator, remaining fields ptrs to operands
  mknode(op,left,right)
- **identifier:** one field with label id and another ptr to symbol table
  mkleaf(id,entry)

Constructing Abstract Syntax tree for expression

- Each node can be represented as a record
- **Operators:** one field for operator, remaining fields ptrs to operands
  mknode(op,left,right)
- **identifier:** one field with label id and another ptr to symbol table
  mkleaf(id,entry)
- **number:** one field with label num and another to keep the value of the number
  mkleaf(num,val)

Example

- The following sequence of function calls creates a parse tree for $a - 4 + c$

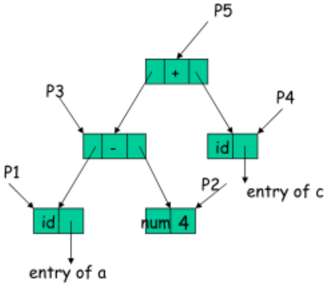
Example

- The following sequence of function calls creates a parse tree for $a - 4 + c$

  $P_1 = mkleaf(id, entry.a)$
  $P_2 = mkleaf(num, 4)$
  $P_3 = mknode(-, P_1, P_2)$
  $P_4 = mkleaf(id, entry.c)$
  $P_5 = mknode(+, P_3, P_4)$

Example

- The following sequence of function calls creates a parse tree for $a - 4 + c$
  $P_1 = mkleaf(id, entry.a)$
  $P_2 = mkleaf(num, 4)$
  $P_3 = mknode(-, P_1, P_2)$
  $P_4 = mkleaf(id, entry.c)$
  $P_5 = mknode(+, P_3, P_4)$



A syntax directed definition for constructing syntax tree

$E \rightarrow E_1 + T$	$E.ptr = mknode(+, E_1.ptr, T.ptr)$
$E \rightarrow T$	$E.ptr = T.ptr$
$T \rightarrow T_1 * F$	$T.ptr := mknode(*, T_1.ptr, F.ptr)$
$T \rightarrow F$	$T.ptr := F.ptr$
$F \rightarrow (E)$	$F.ptr := E.ptr$
$F \rightarrow id$	$F.ptr := mkleaf(id, entry.id)$
$F \rightarrow num$	$F.ptr := mkleaf(num, val)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

$P_3 = makeleaf(id, b)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

$P_3 = makeleaf(id, b)$

$P_4 = makeleaf(id, c)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

$P_3 = makeleaf(id, b)$

$P_4 = makeleaf(id, c)$

$P_5 = makenode(-, P3, P4)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

$P_3 = makeleaf(id, b)$

$P_4 = makeleaf(id, c)$

$P_5 = makenode(-, P3, P4)$

$P_6 = makenode(*, P_2, P_5)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$
$P_2 = makeleaf(id, a)$
$P_3 = makeleaf(id, b)$
$P_4 = makeleaf(id, c)$
$P_5 = makenode(-, P3, P4)$
$P_6 = makenode(*, P_2, P_5)$
$P_7 = makenode(+, P_1, P_6)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

$P_3 = makeleaf(id, b)$

$P_4 = makeleaf(id, c)$

$P_5 = makenode(-, P3, P4)$

$P_6 = makenode(*, P_2, P_5)$

$P_7 = makenode(+, P_1, P_6)$

$P_8 = makeleaf(id, b)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

$P_3 = makeleaf(id, b)$

$P_4 = makeleaf(id, c)$

$P_5 = makenode(-, P3, P4)$

$P_6 = makenode(*, P_2, P_5)$

$P_7 = makenode(+, P_1, P_6)$

$P_8 = makeleaf(id, b)$

$P_9 = makeleaf(id, c)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$
$P_2 = makeleaf(id, a)$
$P_3 = makeleaf(id, b)$
$P_4 = makeleaf(id, c)$
$P_5 = makenode(-, P3, P4)$
$P_6 = makenode(*, P_2, P_5)$
$P_7 = makenode(+, P_1, P_6)$
$P_8 = makeleaf(id, b)$
$P_9 = makeleaf(id, c)$
$P_{10} = makenode(-, P_8, P_9)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

$P_3 = makeleaf(id, b)$

$P_4 = makeleaf(id, c)$

$P_5 = makenode(-, P3, P4)$

$P_6 = makenode(*, P_2, P_5)$

$P_7 = makenode(+, P_1, P_6)$

$P_8 = makeleaf(id, b)$

$P_9 = makeleaf(id, c)$

$P_{10} = makenode(-, P_8, P_9)$

$P_{11} = makeleaf(id, d)$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

$P_3 = makeleaf(id, b)$

$P_4 = makeleaf(id, c)$

$P_5 = makenode(-, P3, P4)$

$P_6 = makenode(*, P_2, P_5)$

$P_7 = makenode(+, P_1, P_6)$

$P_8 = makeleaf(id, b)$

$P_9 = makeleaf(id, c)$

$P_{10} = makenode(-, P_8, P_9)$

$P_{11} = makeleaf(id, d)$

$P_{12} = makenode(*, P_{10}, P_{11})$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$

$P_2 = makeleaf(id, a)$

$P_3 = makeleaf(id, b)$

$P_4 = makeleaf(id, c)$

$P_5 = makenode(-, P3, P4)$

$P_6 = makenode(*, P_2, P_5)$

$P_7 = makenode(+, P_1, P_6)$

$P_8 = makeleaf(id, b)$

$P_9 = makeleaf(id, c)$

$P_{10} = makenode(-, P_8, P_9)$

$P_{11} = makeleaf(id, d)$

$P_{12} = makenode(*, P_{10}, P_{11})$

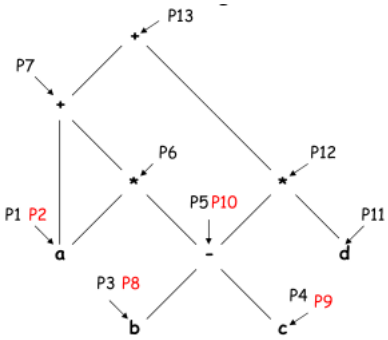
$P_{13} = makenode(+, P_7, P_{12})$

DAG for Expressions

Expression a + a * (b – c) + (b - c) * d

make a leaf or node if not present, otherwise return pointer to the existing node

$P_1 = makeleaf(id, a)$
$P_2 = makeleaf(id, a)$
$P_3 = makeleaf(id, b)$
$P_4 = makeleaf(id, c)$
$P_5 = makenode(-, P3, P4)$
$P_6 = makenode(*, P_2, P_5)$
$P_7 = makenode(+, P_1, P_6)$
$P_8 = makeleaf(id, b)$
$P_9 = makeleaf(id, c)$
$P_{10} = makenode(-, P_8, P_9)$
$P_{11} = makeleaf(id, d)$
$P_{12} = makenode(*, P_{10}, P_{11})$
$P_{13} = makenode(+, P_7, P_{12})$



Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing

Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack

Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also

Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also
- The current top of stack is indicated by `ptr` top

Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also
- The current top of stack is indicated by `ptr` top
- Suppose semantic rule $A.a = f(X.x, Y.y, Z.z)$ is associated with production $A \rightarrow XYZ$

Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also
- The current top of stack is indicated by `ptr` top
- Suppose semantic rule $A.a = f(X.x, Y.y, Z.z)$ is associated with production $A \rightarrow XYZ$
- Before reducing XYZ to A, value of Z is in $val(top)$, value of Y is in $val(top-1)$ and value of X is in $val(top-2)$

Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also
- The current top of stack is indicated by `ptr` top
- Suppose semantic rule $A.a = f(X.x, Y.y, Z.z)$ is associated with production $A \rightarrow XYZ$
- Before reducing XYZ to A, value of Z is in $val(top)$, value of Y is in $val(top-1)$ and value of X is in $val(top-2)$
- If symbol has no attribute then the entry is undefined

Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also
- The current top of stack is indicated by `ptr` top
- Suppose semantic rule $A.a = f(X.x, Y.y, Z.z)$ is associated with production $A \rightarrow XYZ$
- Before reducing XYZ to A, value of Z is in $val(top)$, value of Y is in $val(top - 1)$ and value of X is in $val(top - 2)$
- If symbol has no attribute then the entry is undefined
- After the reduction, top is decremented by 2 and state covering A is put in val(top)

Example: Calculator

$L \to En$	$print(val(top))$
$E \to E + T$	$val(ntop) = val(top - 2) + val(top)$
$E \to T$	
$T \to T * F$	$val(ntop) = val(top - 2) * val(top)$
$T \to F$	
$F \to (E)$	$val(ntop) = val(top - 1)$
$F \to digit$	

Calculator

Input	State	Val	Production
3*5+4n			
*5+4n	digit	3	
*5+4n	F	3	F→digit
*5+4n	T	3	T→ F
5+4n	T*	3*	
+4n	T*digit	3*5	
+4n	T*F	3*5	F → digit
+4n	T	15	T→ T*F
+4n	E	15	E→ T
4n	E+	15-	
n	E+digit	15-4	
n	E+F	15-4	F→ digit
n	E+T	15-4	T→ F
n	E	19	E→ E+T

L-attributed definitions

- A syntax directed definition is L-attributed if each inherited attribute of $X_j (1 \leq j \leq n)$ at the right hand side of $A \rightarrow X_1 X_2 \cdots X_n$ depends only on

L-attributed definitions

- A syntax directed definition is L-attributed if each inherited attribute of $X_j (1 \leq j \leq n)$ at the right hand side of $A \rightarrow X_1 X_2 \cdots X_n$ depends only on
  - Attributes of symbols $X_1 X_2 \cdots X_{j-1}$ and

L-attributed definitions

- A syntax directed definition is L-attributed if each inherited attribute of $X_j (1 \leq j \leq n)$ at the right hand side of $A \rightarrow X_1 X_2 \cdots X_n$ depends only on
  - Attributes of symbols $X_1 X_2 \cdots X_{j-1}$ and
  - Inherited attribute of A

L-attributed definitions

- A syntax directed definition is L-attributed if each inherited attribute of $X_j (1 \leq j \leq n)$ at the right hand side of $A \rightarrow X_1 X_2 \cdots X_n$ depends only on
  - Attributes of symbols $X_1 X_2 \cdots X_{j-1}$ and
  - Inherited attribute of A
- $A \rightarrow LM$ L.i = f1(A.i)
  M.i = f2(L.s)
  A.s = f3(M.s)

L-attributed definitions

- A syntax directed definition is L-attributed if each inherited attribute of $X_j (1 \leq j \leq n)$ at the right hand side of $A \rightarrow X_1 X_2 \cdots X_n$ depends only on
  - Attributes of symbols $X_1 X_2 \cdots X_{j-1}$ and
  - Inherited attribute of A
- $A \rightarrow LM$ L.i = f1(A.i)
  M.i = f2(L.s)
  A.s = f3(M.s)
- $A \rightarrow QR$ R.i = f4(A.i)
  Qi = f5(R.s)
  A.s = f6(Q.s)

L-attributed definitions

- A syntax directed definition is L-attributed if each inherited attribute of $X_j (1 \leq j \leq n)$ at the right hand side of $A \rightarrow X_1 X_2 \cdots X_n$ depends only on
  - Attributes of symbols $X_1 X_2 \cdots X_{j-1}$ and
  - Inherited attribute of A
- $A \rightarrow LM$ L.i = f1(A.i)
  M.i = f2(L.s)
  A.s = f3(M.s)
- $A \rightarrow QR$ R.i = f4(A.i)
  Q.i = f5(R.s)
  A.s = f6(Q.s)

Translation Scheme

- A CFG where semantic actions occur within the rhs of production

Translation Scheme

- A CFG where semantic actions occur within the rhs of production
- A translation scheme to map infix to postfix

Translation Scheme

- A CFG where semantic actions occur within the rhs of production
- A translation scheme to map infix to postfix

  $E \rightarrow TR$

  $R \rightarrow addop\ T \quad print(addop)R|\epsilon$

  $T \rightarrow num \quad print(num)$

Parse tree for 9-5+2