CSE 304
Compiler Design
Syntax-Directed Translation

YOUNGMIN KWON / TONY MIONE

Overview

Associate information with programming language construct

- Attaching attributes to the grammar symbols
- Semantic rules for the production computes the attributes

S-Attributed Definitions

L-Attributed Definitions

Syntax-Directed Definition (SDD)

Generalization of Context Free Grammar

- Each grammar symbol has a set of attributes

Attributes

- Their values are computed by semantic rules (annotating, decorating)
- **Synthesized Attributes** of a node: values are computed from the attributes of children node
- **Inherited Attributes** of a node: values are computed from the sibling and parent nodes

Dependencies between attributes

- Represented by dependency graph
- Derive evaluation order from the dependency graph

Syntax-Directed Definition (SDD)

Example

	PRODUCTION	SEMANTIC RULES
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**.lexval

Terminals have synthesized attributes only

Start symbol does not have inherited attribute

Exercise: draw the parse tree for 3 * 5 + 4 n

Evaluating SDDs

When inherited and synthesized attributes are mixed, there are no guarantee that these attributes can be evaluated.

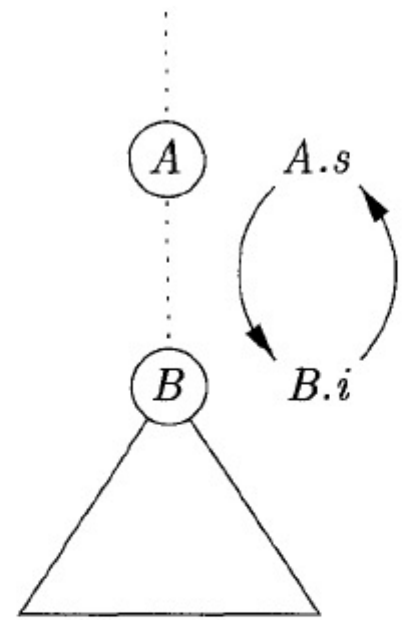
PRODUCTION

$A \rightarrow B$

SEMANTIC RULES

$A.s = B.i;$

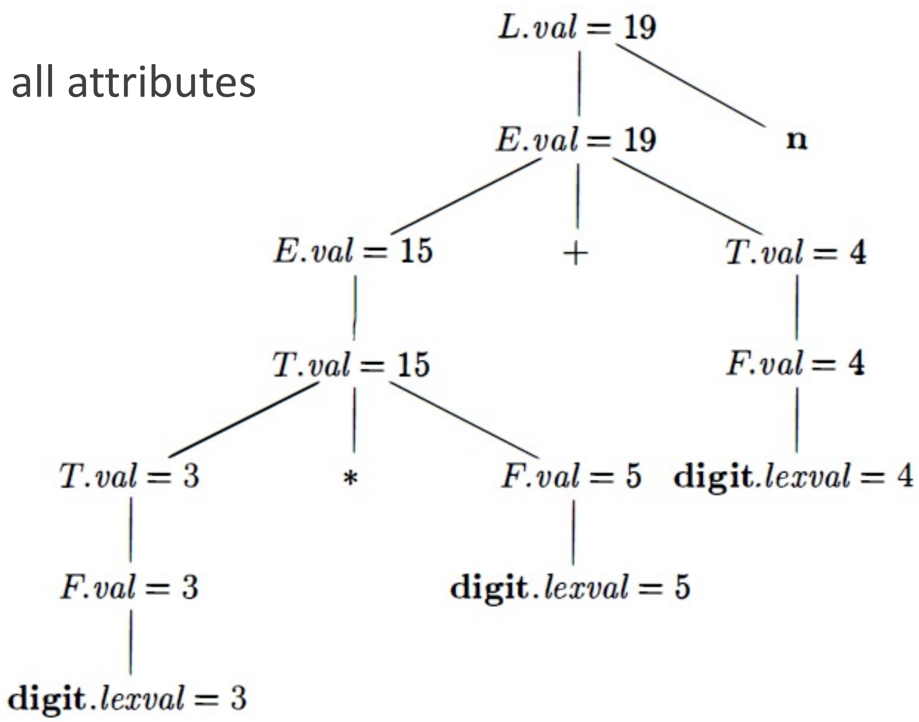
$B.i = A.s + 1$



Bottom-up Evaluation

S-attributed definition

- Syntax-directed definition that uses synthesized attributes exclusively.
- Bottom-up evaluation can annotate all attributes



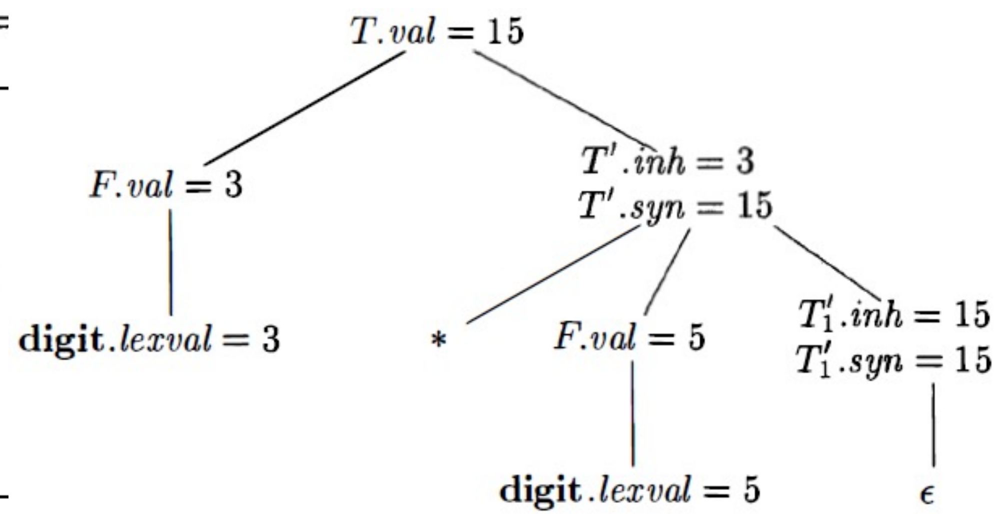
Annotated parse tree for $3 * 5 + 4$ **n**

Top-down Evaluation

Inherited attributes can give context to language construct

- E.g. Whether an Id appears on the LHS or the RHS of =
- Example below parses 1 * 2, 1 * 2 * 3, …

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$



Annotated parse tree for $3 * 5$

Dependency Graph

It can depict the interdependencies among the inherited and synthesized attributes at the node.

Determining the evaluation order of the attributes.

```
for each node n in the parse tree do
   for each attribute a of the grammar symbol at n 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)
                  associated with the production at n do
      for i := 1 to k do
         construct an edge
         from the node for ci to the node for b
```

Dependency Graph

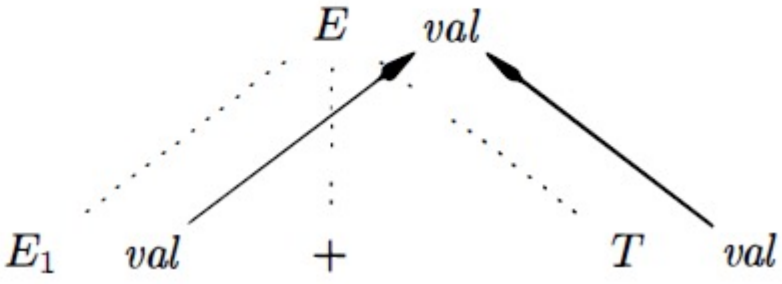
Example 1.

PRODUCTION

$E \rightarrow E_1 + T$

SEMANTIC RULE

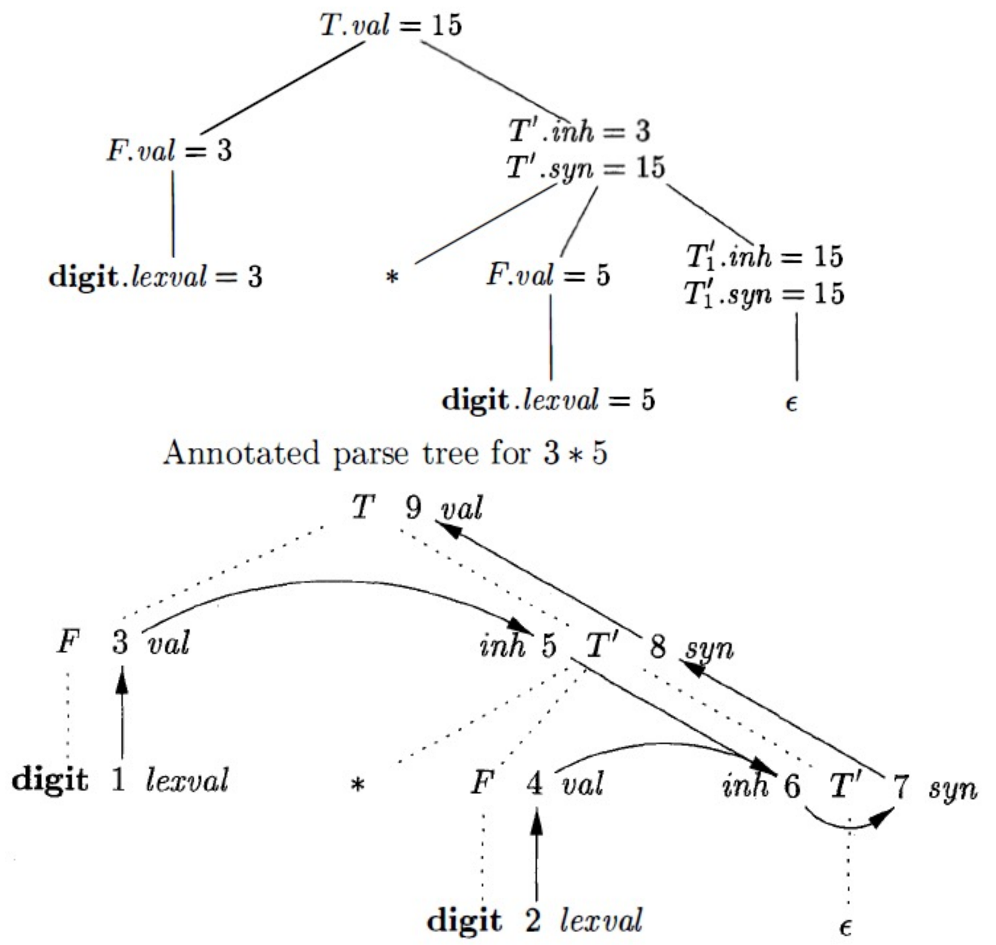
$E.val = E_1.val + T.val$



Dependency Graph

Example 2

	PRODUCTION	SEMANTIC RULES
1)	$T \to F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \textbf{digit}$	$F.val = \textbf{digit}.lexval$



Exercise: Draw a dependency graph for 2 * 3 * 4

Evaluation Order

Topological sort of a directed acyclic graph

- Any ordering $m_1, m_2, \ldots, m_k$ of the nodes of the graph such that if there is an edge m_i -> m_j, then **m_i appears before m_j** in the ordering.

Any topological sort of a dependency graph gives a valid order to evaluate attributes.

Evaluation of semantic rules in this order yields the translation.

L-Attributed Definitions

An SDD is **L-attributed**, if each **inherited** attribute of X_j in $A \rightarrow X_1\ X_2\ \ldots\ X_n$ depends only on

- The attributes of the symbols $X_1,\ X_2,\ \ldots\ X_{j-1}$
- The inherited attributes of A

Every S-attributed definition is L-attributed, because it doesn't have inherited attributes.

L-attributed definitions can be evaluated in depth-first order.

L-Attributed Definitions

Example

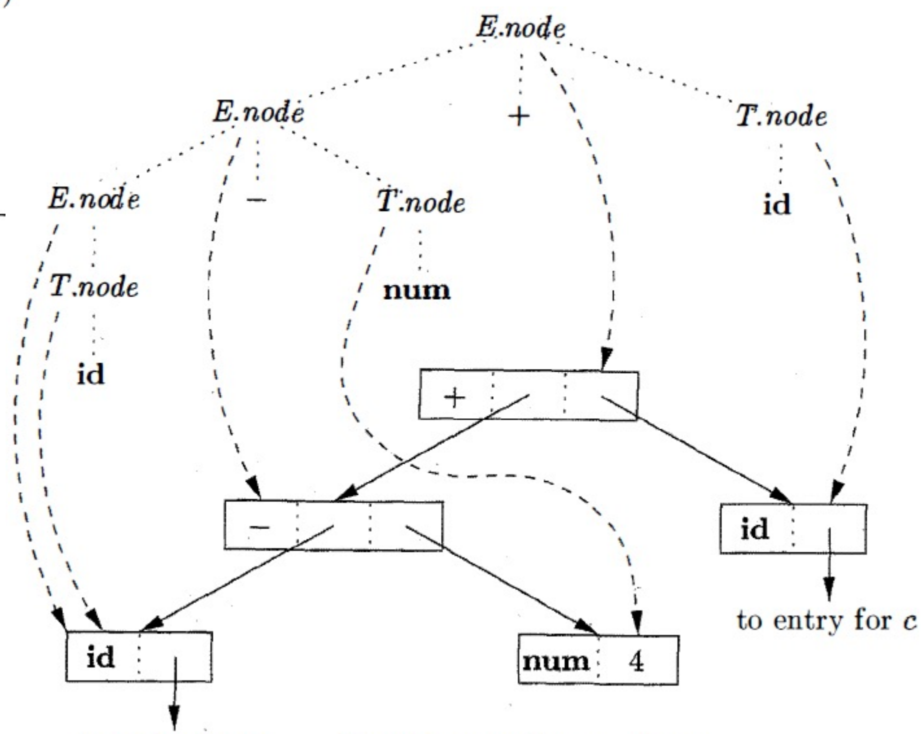
PRODUCTION	SEMANTIC RULE
$T \to F\,T'$	$T'.inh = F.val$
$T' \to *\,F\,T'_1$	$T'_1.inh = T'.inh \times F.val$

Exercise: Is this an L-attributed definition?

PRODUCTION	SEMANTIC RULES
$A \to B\ C$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

Application: Constructing a Syntax Tree

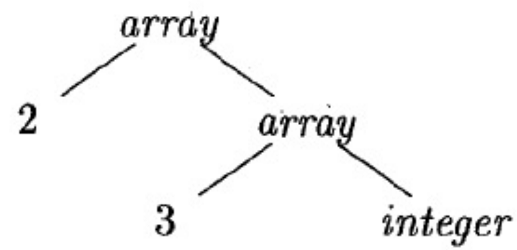
	PRODUCTION	SEMANTIC RULES
1)	$E \to E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (\ E\ )$	$T.node = E.node$
5)	$T \to \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \to \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$



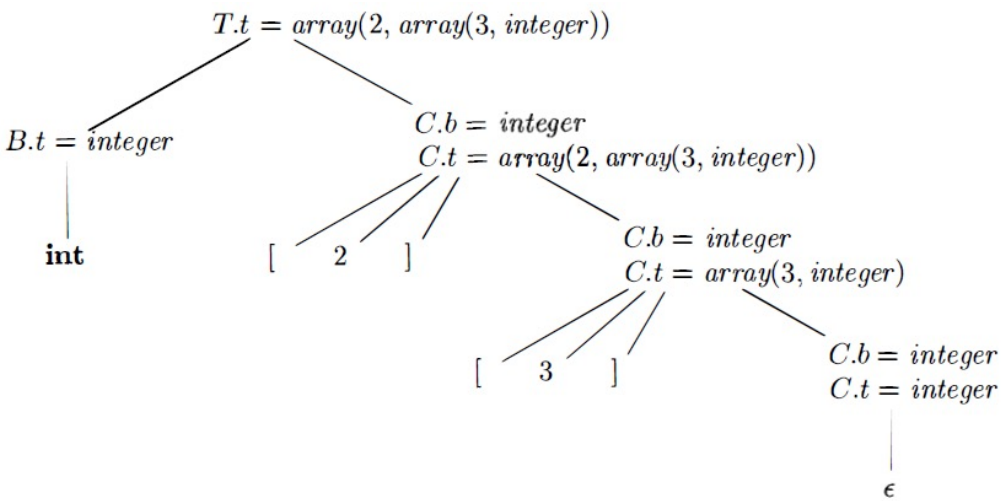
Syntax tree for $a - 4 + c$

Application: Type Expression

Production	Semantic Rules
$T \rightarrow B\ C$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow [\ \textbf{num}\ ]\ C_1$	$C.t = array(\textbf{num}.val,\ C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$



Type expression for **int**[2][3]



Top-Down Translation

L-attributed definitions will be implemented during predictive parsing.

Eliminating Left Recursion from Translation Scheme

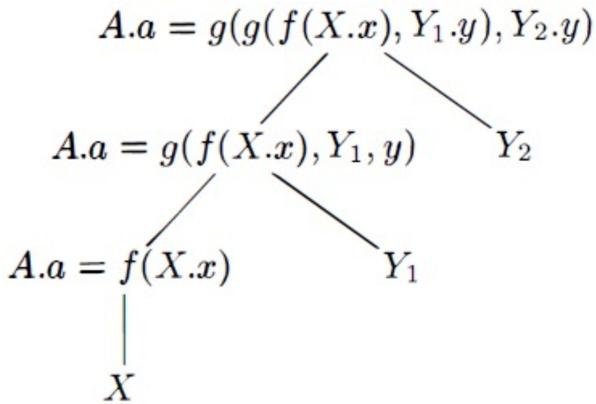
- Evaluate inherited attributes (R.i) before a use of R
- Evaluate synthesized attributes (A.a, R.s) at the end of the production

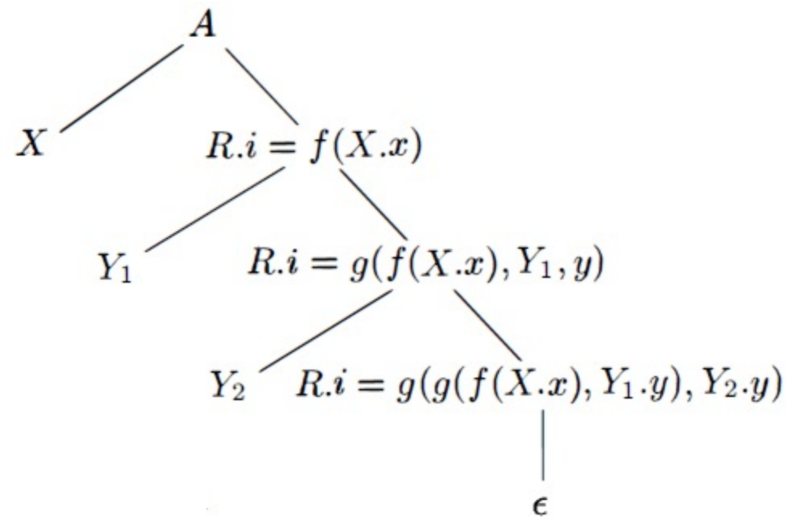
$$
\begin{aligned}
A &\rightarrow A_1\ Y\ \{A.a = g(A_1.a, Y.y)\} \\
A &\rightarrow X\ \{A.a = f(X.x)\}
\end{aligned}
$$

$$
\begin{aligned}
A &\rightarrow X\ R \\
R &\rightarrow Y\ R \mid \epsilon
\end{aligned}
$$

$$
\begin{aligned}
A &\rightarrow X\ \{R.i = f(X.x)\}\ R\ \{A.a = R.s\} \\
R &\rightarrow Y\ \{R_1.i = g(R.i, Y.y)\}\ R_1\ \{R.s = R_1.s\} \\
R &\rightarrow \epsilon\ \{R.s = R.i\}
\end{aligned}
$$

Eliminating Left Recursion from Translation Scheme





$$
\begin{aligned}
A &\rightarrow A_1\ Y\ \{A.a = g(A_1.a, Y.y)\} \\
A &\rightarrow X\ \{A.a = f(X.x)\}
\end{aligned}
$$

$$
\begin{aligned}
A &\rightarrow X\ \{R.i = f(X.x)\}\ R\ \{A.a = R.s\} \\
R &\rightarrow Y\ \{R_1.i = g(R.i, Y.y\}\ R_1\ \{R.s = R_1.s\} \\
R &\rightarrow \epsilon\ \{R.s = R.i\}
\end{aligned}
$$

Exercise: Eliminate Left Recursion from

```
A -> A1 Y { A.a = g(A1.a, Y.y) }
A -> A1 Z { A.a = h(A1.a, Z.z) }
A -> X { A.a = f(X.x) }
```

Exercise

A -> A1 Y { A.a = g(A1.a, Y.y) }

A -> A1 Z { A.a = h(A1.a, Z.z) }

A -> X { A.a = f(X.x) }

A -> X R

R -> Y R1

R -> Z R1

A -> X {R.i = f(X.x) } R {A.a = R.s}

R -> Y { R1.i = g(R.i, Y.y)} R1 {R.s = R1.s}

R -> Z { R1.i = h(R.i, Z.z) } R1 {R.s = R1.s}

Predictive Translator

For each nonterminal A, construct a function A with

- Formal parameters for the inherited attributes of A
- Returns a collection of the synthesized attributes of A

Decide what production to use based on the lookahead

Code for the production

- For a token X, save the value of X at X.x and match the token
- For a nonterminal B, do the assignment c := B(b1, … bk), where b1, …, bk are the variables for the inherited attributes of B, and c is a variable for the synthesized attribute of B
- For an action, copy the code into the parser, replace reference to attributes by their corresponding variables.

Example: Predictive Translation

```
E -> E1 + T { E.np = mknode('+', E1.np, T.np) }
E -> E1 - T { E.np = mknode('-', E1.np, T.np) }
E -> T      { E.np = T.np }
```

```
E -> T   { R.i = T.np }
     R   { E.np = R.s }
R -> +
     T   { R1.i = mknode('+', R.i, T.np) }
     R1  { R.s = R1.s}
R -> -
     T   { R1.i = mknode('-', R.i, T.np) }
     R1  { R.s = R1.s}
R -> eps { R.s =  R.i }
```

```
SN* R(SN* i) {
  char op;
  SN *Ts, *R1i, *R1s, *s;

  if (la == '+') {
    op  = la;
    match('+');
    Ts  = T();
    R1i = mknode('+', i, Ts);
    R1s = R(R1i);
    s   = R1s;
  }
  else if (la == '-')
     ...
  else
    s = i;

  return s;
}
```

Bottom-Up Translation

L-Attributed definitions will be implemented during LR-Parsing

LR parsers use a stack to hold information about parsed subtrees

- Add extra fields val in the stack to hold the values of the **synthesized attributes**.
- If the i^{th} state symbol is A, then val[i] holds the attributes associated with A.
- E.g.

  If `A -> X Y Z` is a production and
  `A.a = f(X.x, Y.y, Z.z)` is a semantic rule

  `Z.z = val[top], Y.y = val[top-1], X.x = val[top-2]`

  `A.a = f(val[top-2], val[top-1], val[top])`

Inherited Attributes in Yacc

```
declaration
    : class type idlist;
class
    : GLOBAL {$$ = 1;}
    | LOCAL  {$$ = 2;}
    ;
type
    : REAL    {$$ = 1;}
    | INTEGER {$$=2;}
    ;
idlist
    : ID         {mksymbol($0,$-1, $1)}
    | idlist ID {mksymbol($0,$-1, $2)}
    ;
```

Example: Evaluation by Parser Stack

$$
\begin{array}{llll}
L & \rightarrow & E\ \mathbf{n} & \{\ \text{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

PRODUCTION	CODE FRAGMENT
$L \rightarrow E\ \mathbf{n}$	*print* (*val* [*top*])
$E \rightarrow E_1 + T$	*val* [*ntop*] := *val* [*top* − 2] + *val* [*top*]
$E \rightarrow T$	
$T \rightarrow T_1 * F$	*val* [*ntop*] := *val* [*top* − 2] × *val* [*top*]
$T \rightarrow F$	
$F \rightarrow (\ E\ )$	*val* [*ntop*] := *val* [*top* − 1]
$F \rightarrow \mathbf{digit}$	

INPUT	*state*	*val*	PRODUCTION USED
3*5+4 n	_	_	
***5+4 n**	3	3	
***5+4 n**	*F*	3	*F* → **digit**
***5+4 n**	*T*	3	*T* → *F*
5+4 n	*T* *	3 _	
+4 n	*T* * 5	3 _ 5	
+4 n	*T* * *F*	3 _ 5	*F* → **digit**
+4 n	*T*	15	*T* → *T* * *F*
+4 n	*E*	15	*E* → *T*
4 n	*E* +	15 _	
n	*E* + 4	15 _ 4	
n	*E* + *F*	15 _ 4	*F* → **digit**
n	*E* + *T*	15 _ 4	*T* → *F*
n	*E*	19	*E* → *E* + *T*
	E **n**	19 _	
	L	19	*L* → *E* **n**

Marker Nonterminals

Nonterminals with the epsilon production.

Move embedded actions to the right ends of their productions.

```
E -> T R
E -> + T { print('+') } R
   | - T { print('-') } R
   | eps
T -> num { print(num.val) }
```

```
E -> T R
R -> + T M R | - T N R | eps
T ->  num { print(num.val) }
M ->  eps { print('+') }
N ->  eps { print('-') }
```

Marker Nonterminals

Simulating the Evaluation of Inherited Attributes

- E.g. when reducing **C->c**,

Production	Semantic Rules
S -> aAC	C.i = A.s
S -> bABC	C.i = A.s
C -> c	C.s = g(C.i)

C.i = val[top - 1] or C.i = val[top - 2]

Production	Semantic Rules
S -> aAC	C.i = A.s
S -> bABMC	M.i = A.s, C.i = M.s
C -> c	C.s = g(C.i)
M -> eps	M.s = M.i

C.i = val[top - 1]

Marker Nonterminals

When inherited attributes are not updated by copy, its value is not in the **val** stack.

Production	Semantic Rules
S -> aAC	C.i = f(A.s)

f(A.s) is not in val stack

Production	Semantic Rules
S -> aANC	N.i = A.s, C.i = N.s
N -> eps	N.s = f(N.i)

C.i = val[top -1]

Parser Stack for Inherited Attributes

Assume that every nonterminal **A** has one inherited attribute **A.i** and every grammar symbol **X** has a synthesized attribute **X.s**

For every production $A \rightarrow X_1 \ldots X_n$, replace it with $A \rightarrow M_1\ X_1 \ldots M_n\ X_n$ where $M_1 \ldots M_n$ are new markers.

- Synthesized attributes $X_j.s$ will be in **val** stack associated with X_j
- Inherited attributes $X_j.i$ appears in **val** stack but associated with M_j

Marker Nonterminals

Adding marker nonterminals doesn't introduce conflicts to LL(1) grammars

For LR(1) grammars, marker nonterminals can introduce parsing conflicts.

Questions?