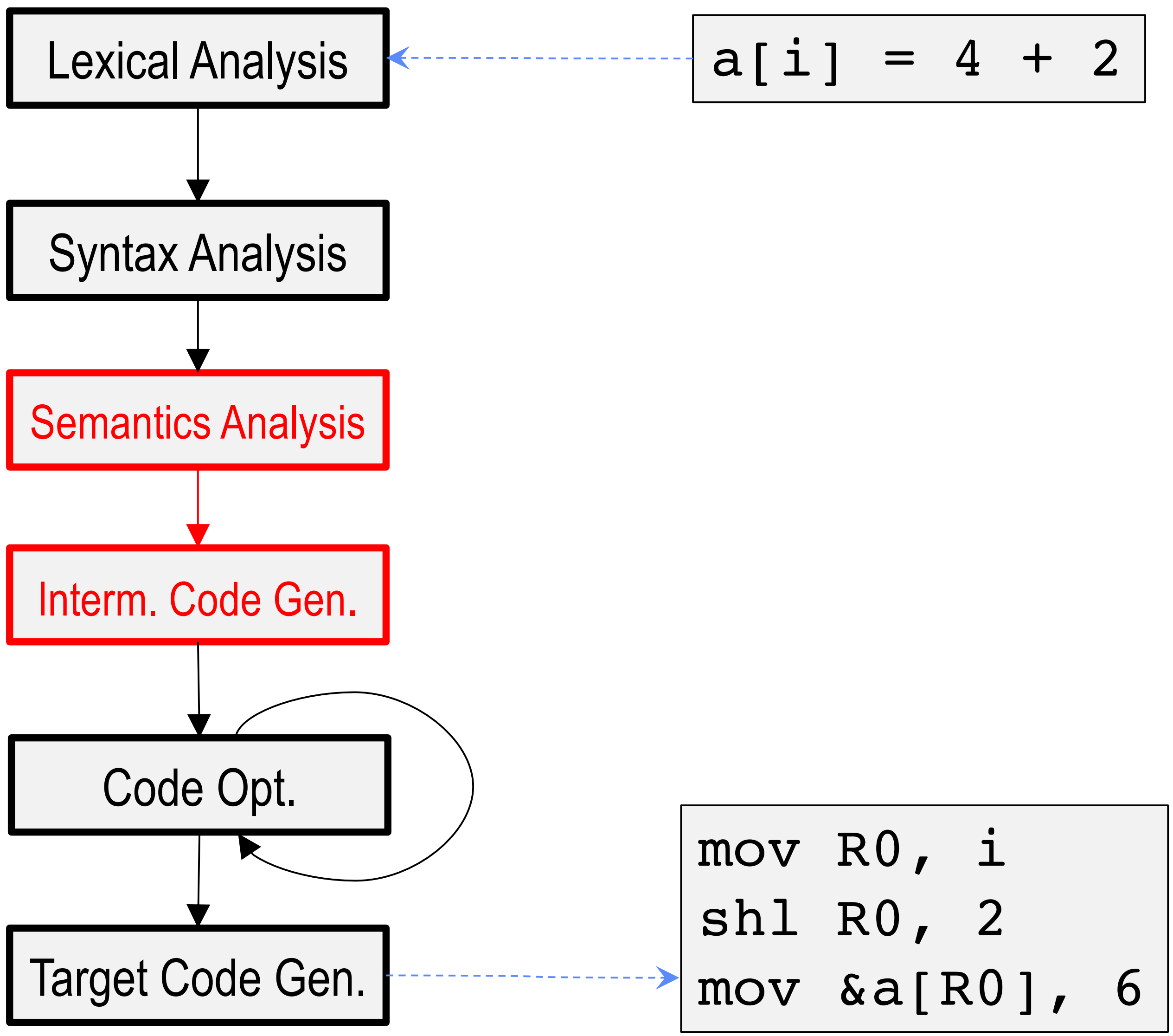
Compilation Phases



Semantic Processing: Syntax Directed Translation

- *Attributes*: Associate information with language constructs by attaching *attributes* to grammar symbols representing that construct.

  An attribute can represent anything (reasonable) that we choose, e.g. a string, number, type, memory location, code fragment etc.

- *Semantic rules*: Values for attributes are computed using *semantic rules* associated with grammar productions.

  A parse tree showing the values of attributes at each node is called an *annotated parse tree*.

Example : Attributes for an Identifier

name : character string, obtained from scanner.

scope

type :

- *integer*
- *array* :
  - no. of dimensions
  - upper and lower bounds for each dimension
  - type of elements
- *record* :
  - name and type of each field
- *function*
  - no. of parameters
  - types of parameters (in order)
  - type of returned value
  - entry point in memory
  - size of stack frame

Example : Associating Semantic Rules with Productions

Production	*Semantic Rule*
$E \longrightarrow E_1 + T$	$E.val := E_1.val \oplus T.val$
$E \longrightarrow T$	$E.val := T.val$
$T \longrightarrow T_1 * F$	$T.val := T_1.val \otimes F.val$
$T \longrightarrow F$	$T.val := F.val$
$F \longrightarrow (E)$	$F.val := E.val$
$F \longrightarrow \mathbf{intcon}$	$F.val := \mathbf{intcon}.val$

Note: The semantic rules also impose an evaluation order on the attributes.

Two-Pass vs. One-Pass Compilation

Two-Pass

1. Parse input and use semantic rules to:
   (a) process declarations into symbol table
   (b) construct syntax tree

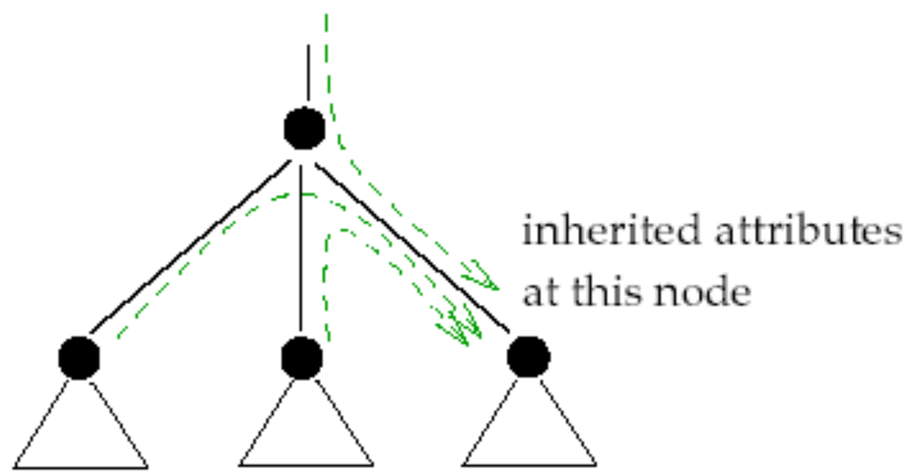
2. Traverse syntax tree:
   (a) check types
   (b) make storage allocation decisions
   (c) generate code

One-Pass

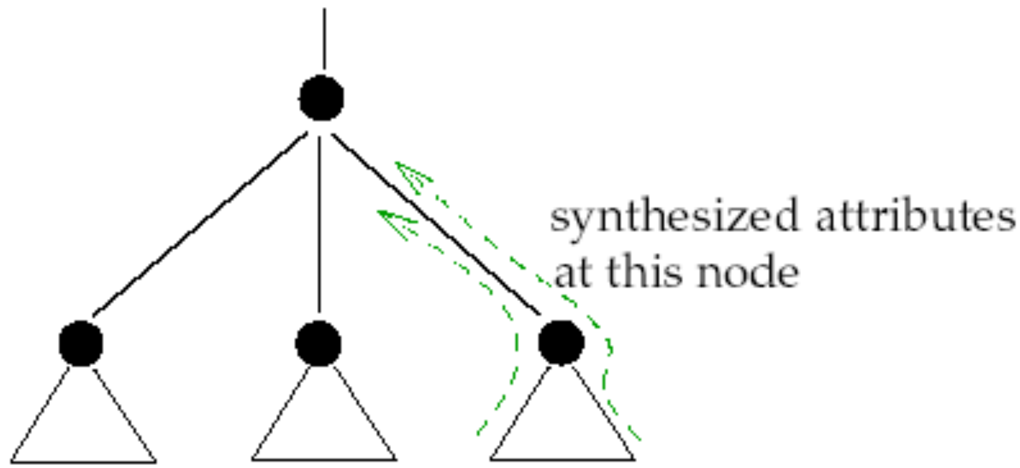
1. Parse input and use semantic rules to:
   (a) process declarations into symbol table
   (b) check types
   (c) make storage allocation decisions
   (d) generate code

Inherited and Synthesized Attributes

Inherited Attributes : An attribute at a node is *inherited* if its value is computed from attribute values at the siblings and/or parent of that node in the parse tree.



Synthesized Attributes : An attribute at a node is *synthesized* if its value is computed from the attribute values of the children of that node in the parse tree.



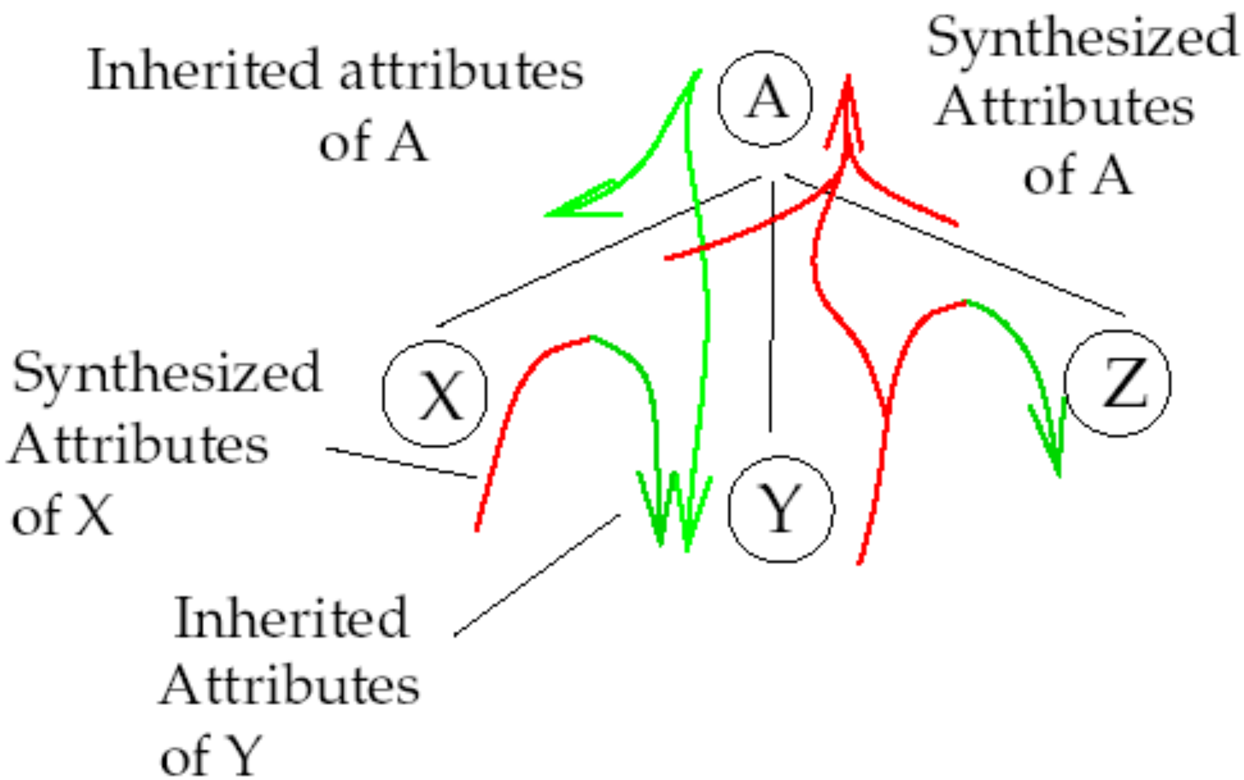
6.1.1. Attribute Grammars

Basic Idea :

- Every grammar symbol is associated with a set of attributes.
- *Semantic rules* specify how each attribute is to be computed.

The attributes of a grammar symbol are partitioned into two sets: *inherited* and *synthesized*. I.e., for any particular grammar symbol, a given attribute cannot be inherited in some places and synthesized in others.

E.g.: $A \rightarrow X\ Y\ Z$



S-Attributed Grammars

Definition : Grammar containing only synthesized attributes is called *S-attributed*.

- Synthesized attributes can be conveniently handled during bottom up parsing as it builds the parse tree bottom up.

L-Attributed Grammars

Definition : Grammar for which the attributes can always be evaluated by a depth-first L-to-R traversal of the parse tree.

- All attributes can be conveniently handled during LL(1) parsing because the parse tree is built depth-first L-to-R.

- Every S-attributed definition is L-attributed.

Example

We will develop semantic rules for constructing *symbol table* from the declarations and constructing *syntax tree* for the expression.

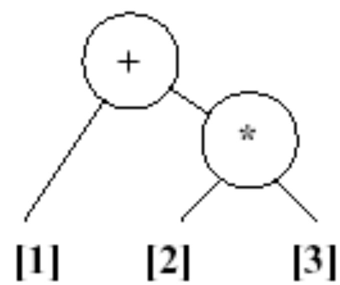
- *Inherited attribute* needed to propagate the type to each declared variable.
- *Synthesized attribute* needed to construct syntax tree for an expression from syntax trees of subexpressions.

```
int a, b, c;
a + b * c;
```

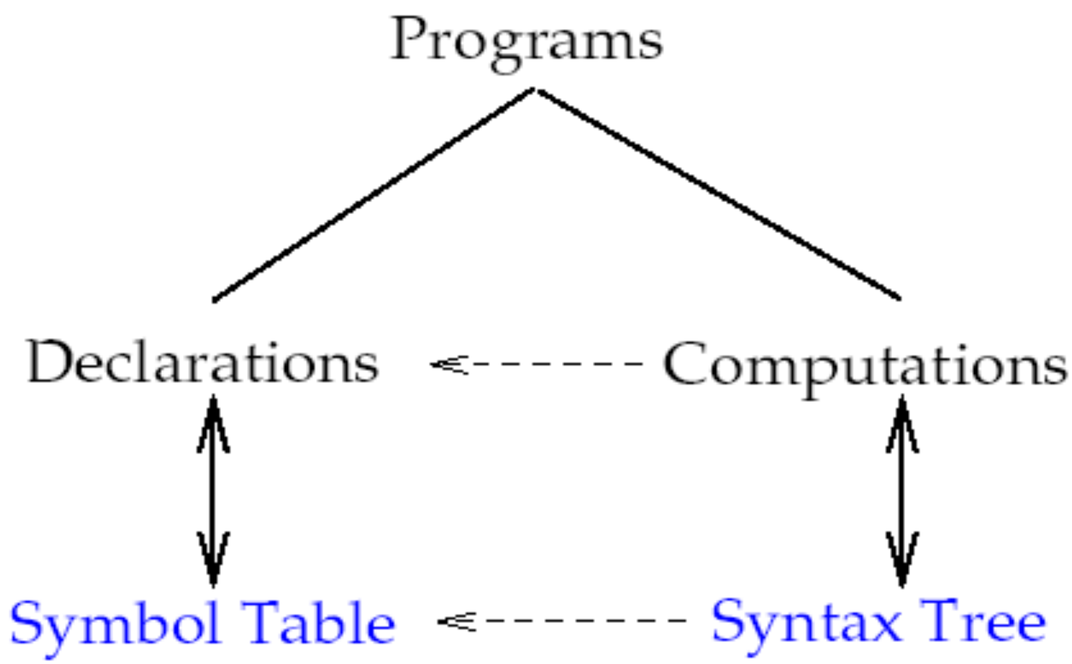
Symbol Table

Name	Type	Addr	········	
a	int	1		[1]
b	int	2		[2]
c	int	3		[3]

Syntax Tree



Syntax Trees



- A *syntax tree* is a tree that shows the syntactic structure of a program, while omitting irrelevant detail present in a parse tree.

- Each node of a syntax tree represents "what to do" at that point, i.e., a computation.

  The children of the node correspond to the objects to which that computation is applied.

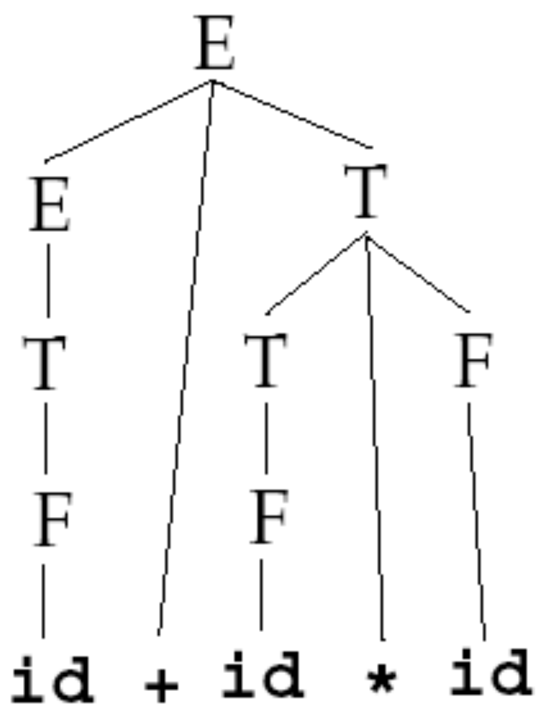
Example

Grammar :

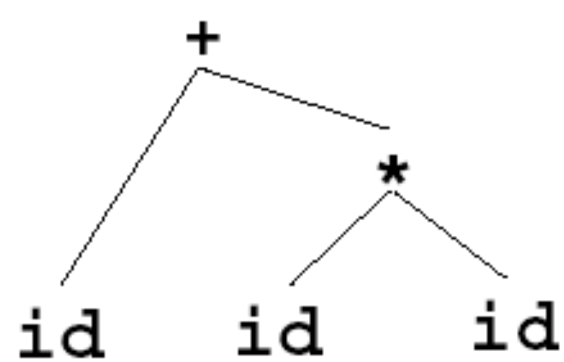
$$
\begin{aligned}
E &\rightarrow E + T \quad | \quad T \\
T &\rightarrow T * F \quad | \quad F \\
F &\rightarrow (\ E\ ) \quad | \quad \mathtt{id}
\end{aligned}
$$

Input : `id + id * id`

Parse Tree :



Syntax Tree :



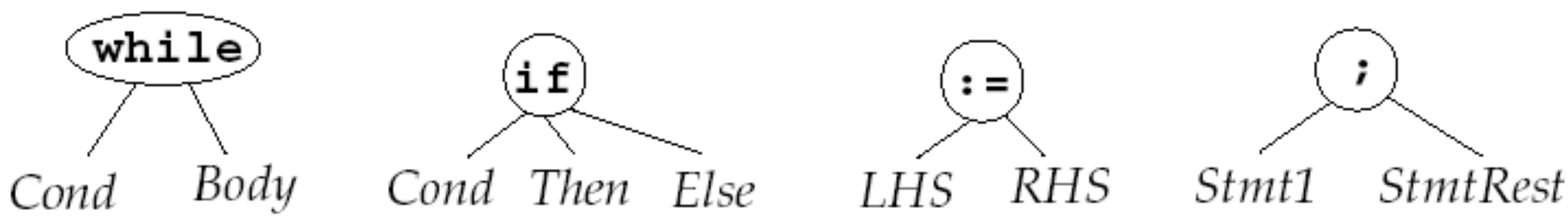
Structure of Syntax Trees

Expression :

– Leaves: identifiers or constants.
– Internal nodes labelled with operations.
– Children of a node are its operands.

Statements :

– A node's label indicates what kind of statement it is.
– The children of a node correspond to the components of the statement.



6.3. Symbol Tables

Purpose : To hold information about identifiers that is computed at one point and looked up at later points during compilation.

Example : type of a variable; entry point for a function.

Operations : `insert`, `lookup`, `delete`.

Common implementations : linked lists, hash tables.

Managing Scope Information

- When a name is looked up in a symbol table, the entry for the "appropriate" declaration of that name must be returned.

  The scope rules of the language determine which declaration is appropriate.

- Often, the appropriate declaration for a name is the "most closely nested" one. A simple implementation of this is to *push* a new symbol table when entering a new scope, and *pop* it when leaving it:
  - Implement the stack of symbol tables as a linked list of tables.
  - *lookup* : search backward starting at the innermost scope.
  - *insert*, *delete* : works on the innermost scope.

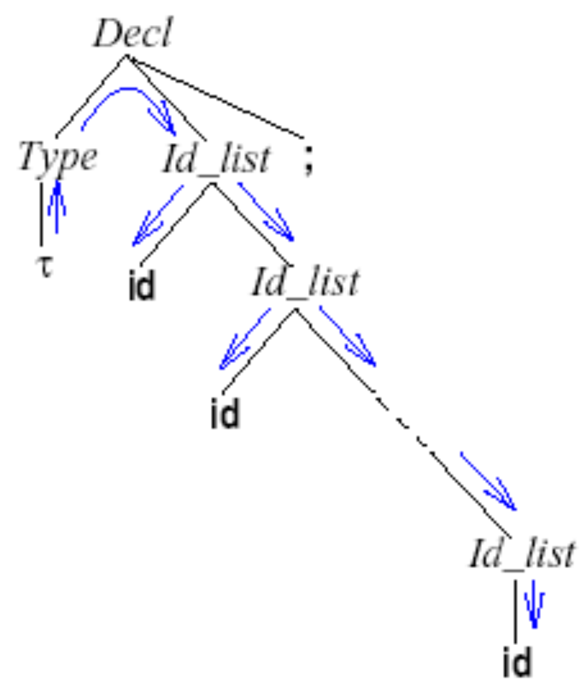
- Information may be "deleted" when leaving a scope; but it may be necessary to retain this information for use by run-time tools, e.g. debuggers.

Processing Declarations

Goal : Store information about variable names and types in symbol table.

Use of Attributes : To propagate type information to the various identifiers appearing in a declaration.

$$Decl \longrightarrow Type \quad Id\_list \ \mathbf{;}$$
$$Id\_list \longrightarrow \mathbf{id} \ \mathbf{,} \ Id\_list \mid \mathbf{id}$$
$$Type \longrightarrow int \mid real$$



Semantic Rules:

- $Type$ synthesizes the value of $tval$;
- Id_list uses $tval$ as an inherited attribute; defines type information in symbol table entries corresponding to **id**.

Production

Semantic Rule

$Decl \longrightarrow Type \quad Id\_list$ **;**

$Id\_list.tval := Type.tval$

$Id\_list \longrightarrow$ **id** **,** Id_list_1

$\mathbf{id}.type := Id\_list.tval;$

$symtab\_insert(id.name, id, type)$

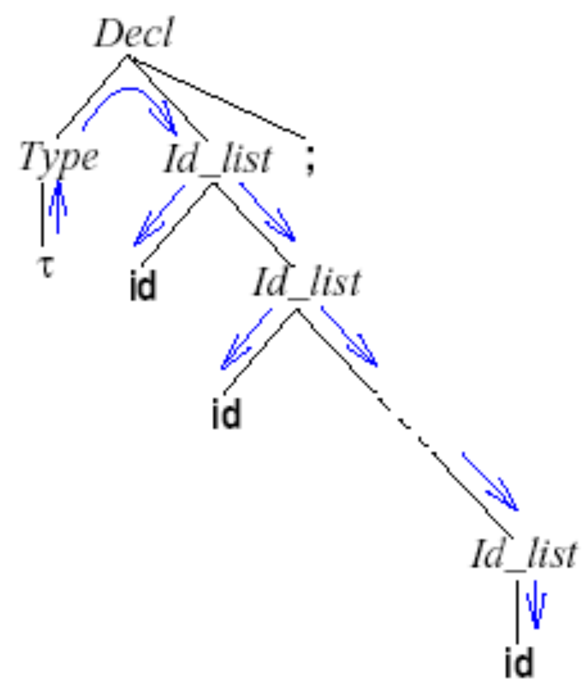
$Id\_list_1.tval := Id\_list.tval$

$Type \longrightarrow int$

$Type.tval = int$

$Type \longrightarrow real$

$Type.tval = real$



Semantic Rules for Constructing Expression Syntax Tree

Goal : Construct syntax tree for the expression; associate references to ids by entries in symbol table.

Use of Attributes : To propagate syntax trees for smaller subexpressions needed to from syntax trees for larger expressions.

Production
  Semantic Rule

$E \rightarrow E_1 + T$
  $E.tree = mktree(PLUS, E_1.tree, T.tree)$

$E \rightarrow T$
  $E.tree = T.tree$

$$T \rightarrow T \; * \; F$$
$$T.tree = mktree(TIMES, T.tree, F.tree)$$

$$T \rightarrow F$$
$$T.tree = F.tree$$

$$F \rightarrow \mathtt{id}$$
$$F.tree = mknode(idnode, symtab\_lookup(id.name))$$

$$F \rightarrow \mathtt{intconst}$$
$$F.tree = mknode(intconstnode, intconst.value)$$

Syntax-Directed Definitions vs. Translation Schemes

Syntax-directed definitions describe relationships among attributes associated with grammar symbols (so far we have only looked at these).

Syntax-directed translation schemes describe the order and timing of attribute computation.

- Embeds semantic rules into the grammar.
- Each semantic rule can only use information computed by already executed semantic rules.

Translation Scheme with Synthesized Attributes

- Synthesized attributes of a terminal are contained in the terminal symbol itself.
- Synthesized attribute associated with a *non-terminal* symbol is computed after seeing everything it derives.

$$E \rightarrow E_1 + T \; \{E.tree = mktree(PLUS, E_1.tree, T.tree)\}$$

$$E \rightarrow T \; \{E.tree = T.tree\}$$

$$T \rightarrow T * F \; \{T.tree = mktree(TIMES, T.tree, F.tree)\}$$

$$T \rightarrow F \; \{T.tree = F.tree\}$$

$$F \rightarrow \texttt{id} \; \{F.tree = mknode(idnode, symtab\_lookup(id.name))\}$$

$$F \rightarrow \texttt{intconst} \; \{F.tree = mknode(intconstnode, intconst.value)\}$$

Translation Scheme with Inherited Attributes

- Inherited attribute associated with a *non-terminal* is computed before encountering the non-terminal.

$$
\begin{aligned}
E \rightarrow & T \ \{R.itree = T.stree\} \\
& R \ \{E.stree = R.stree)\}
\end{aligned}
$$

$$
\begin{aligned}
R \rightarrow & + \ T \ \{R_1.itree = mktree("+", R.itree, T.stree)\} \\
& R_1 \ \{R.stree = R_1.stree\}
\end{aligned}
$$

$$
\begin{aligned}
R \rightarrow & - \ T \ \{R_1.itree = mktree("-", R.itree, T.stree)\} \\
& R_1 \ \{R.stree = R_1.stree\}
\end{aligned}
$$

$$R \rightarrow \epsilon \ \{R.stree = R.itree\}$$

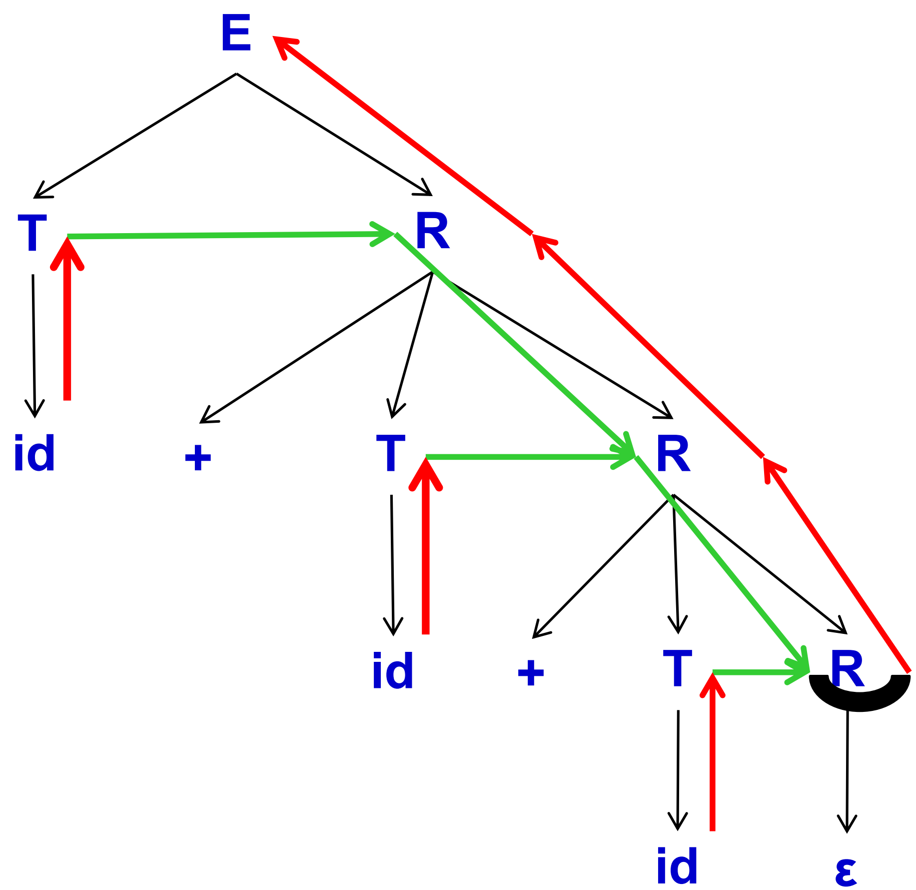
$$T \rightarrow \mathtt{id} \ \{T.stree = mknode(idnode, symtab\_lookup(id.name))\}$$

$$T \rightarrow \mathtt{intconst} \ \{T.stree = mknode(intconstnode, intconst.value)\}$$

E → T R

R → + T R | - T R | ε

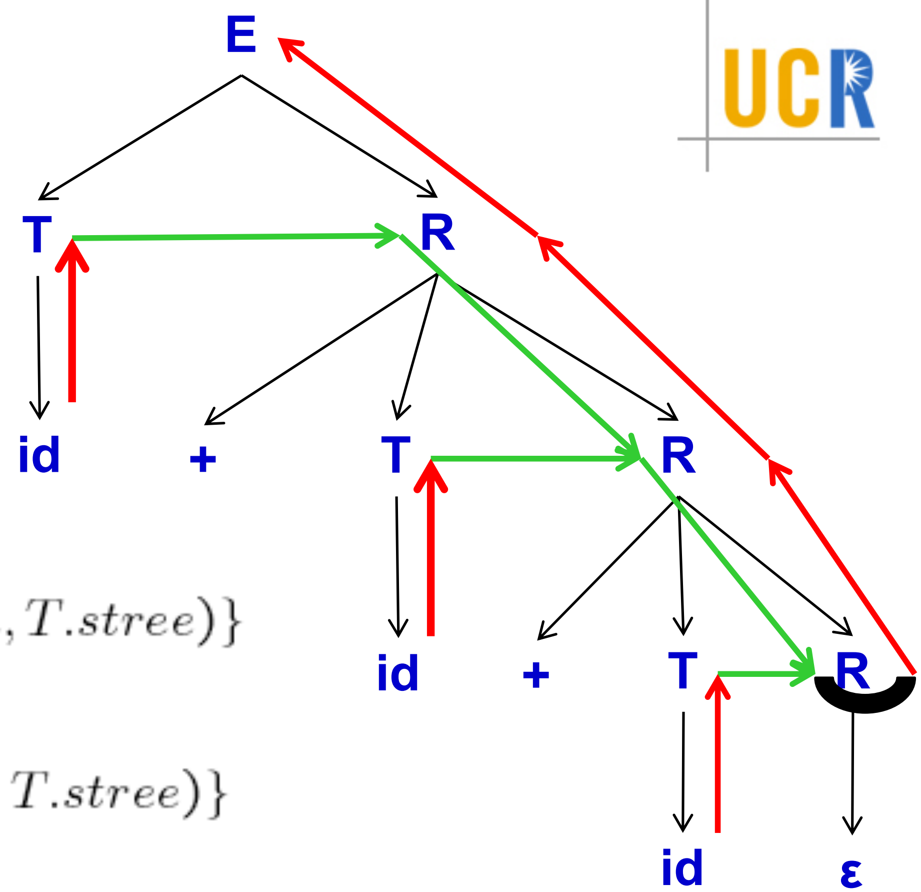
T → id | intconst



E → T R

R → + T R | - T R | ε

T → id | intconst



$$E \rightarrow T \ \{R.itree = T.stree\}$$
$$R \ \{E.stree = R.stree)\}$$

$$R \rightarrow + \ T \ \{R_1.itree = mktree("+", R.itree, T.stree)\}$$
$$R_1 \ \{R.stree = R_1.stree\}$$

$$R \rightarrow - \ T \ \{R_1.itree = mktree("-", R.itree, T.stree)\}$$
$$R_1 \ \{R.stree = R_1.stree\}$$

$$R \rightarrow \epsilon \ \{R.stree = R.itree\}$$

$$T \rightarrow \texttt{id} \ \{T.stree = mknode(idnode, symtab\_lookup(id.name))\}$$

$$T \rightarrow \texttt{intconst} \ \{T.stree = mknode(intconstnode, intconst.value)\}$$

Implementation Issues

Triggering execution of semantic actions: How can parsing actions be made to trigger execution of semantic rules?

Managing and accessing attribute values: Where should the attribute values be held and how should they be accessed?

Note: Solutions vary according to the type of parses: bottom-up vs. top-down.

Triggering Semantic Actions in a Bottom-Up Parser

- A *reduction* occurs in the parser at each point where a *synthesized* attribute is to be computed because computation of a synthesized attribute is performed at the end of the right hand side of a production.

  Example
  $E \rightarrow E_1 + T\{E.tree = mktree("+", E_1.tree, T.tree)\}$

  Reductions *trigger* execution of code corresponding to semantic rules.

- The same is not true for *inherited* attributes as semantic rules for their evaluation is embedded inside the right hand side of a production.

  Augment the grammar with *marker non-terminals* to *introduce reductions* corresponding to evalautions of inherited attributes.

Example

Before transformation:

$$
\begin{aligned}
&E \rightarrow T\ E' \\
&E' \rightarrow +\, T\ \{\texttt{print '+'}\}\ E' \mid -\, T\ \{\texttt{print '}-\texttt{'}\}\ E' \mid T \\
&T \rightarrow \mathbf{num}\ \{\texttt{print}\ \textsf{num}\texttt{.val}\}
\end{aligned}
$$

After transformation:

$$
\begin{aligned}
&E \rightarrow T\ E' \\
&E' \rightarrow +\, T\ M_1\ E' \mid -\ T\ M_2\ E' \mid T \\
&T \rightarrow \mathbf{num}\ \{\texttt{print}\ \textsf{num}\texttt{.val}\}
\end{aligned}
$$

$$
\begin{aligned}
&M_1 \rightarrow \varepsilon \quad \{\texttt{print '+'}\} \\
&M_2 \rightarrow \varepsilon \quad \{\texttt{print '}-\texttt{'}\}
\end{aligned}
$$

Managing Attributes in a Bottom-Up Parser

– A bottom-up parser maintains a semantic stack that parallels the syntax stack. Given a symbol X in the syntax stack, the attributes of X are stored in the corresponding position of the semantic stack.

– When a reduction is made, compute new *synthesized attributes* from the values currently on top of the stack.

– Computation of *inherited attributes* requires "reaching into" the semantic stack. We must ensure that the position that we must reach into is predictable.

Example with Synthesized Attribute:

$$
\begin{array}{l}
E \rightarrow E_1 + T \; \{ \\
y := semantic\_stack[top]; \\
x := semantic\_stack[top-2]; \\
z := mktree('+', x, y); \\
semantic\_stack[top-2] := z; \\
top := top - 2; \\
\}
\end{array}
$$

Example with Inherited Attribute:

$$
\begin{array}{l}
E \rightarrow T \; E' \\
E' \rightarrow OP \; T \; M \; E' \;\; | \;\; T \\
OP \rightarrow + \;\; | \, - \\
M \rightarrow \varepsilon \quad \{\texttt{print semantic\_stack[top-2]}\} \\
T \rightarrow \mathrm{num} \; \{\texttt{print } \textbf{num}\texttt{.val}\}
\end{array}
$$

Triggering Semantic Actions in a LL(1) Parser

- Unlike the bottom-up parser, there are no distinct parsing events which can be used to trigger the execution of semantic actions.

- Augment the grammar with *marker non-terminals* whose only purpose is to trigger execution of semantic actions.

  When a production rule is applied, these markers are pushed along with the rest of the symbols on to the syntax stack in reverse order.

  When a marker is popped from the syntax stack, the corresponding semantic action is executed.

Managing Attributes in a LL(1) Parser

- The syntax stack does not parallel the semantic stack – syntax stack contains what we expect to see in the future while the semantic stack contains attributes of constructs that have already been seen.

- For each production applied, reserve positions in the semantic stack to hold attributes for the left hand side non-terminal and right hand side symbols.

  Save these positions in the syntax stack to allow access to attributes.

  For more details see separate handout given in the class.