Syntax-Directed Translation

ALSU Textbook Chapter 5.1–5.4, 4.8, 4.9

Tsan-sheng Hsu

tshsu@iis.sinica.edu.tw

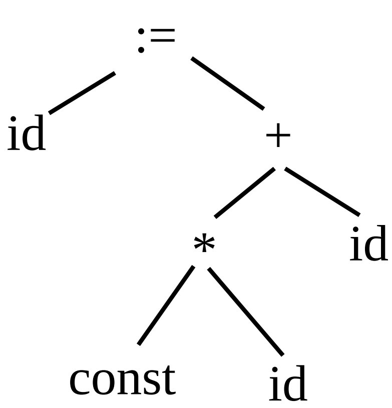
http://www.iis.sinica.edu.tw/~tshsu

What is syntax-directed translation?

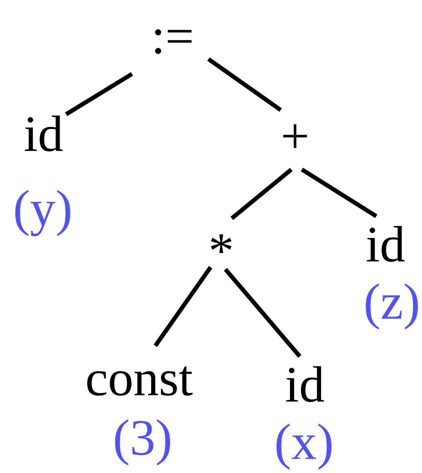
- **Definition:**
  - **The compilation process is driven by the syntax.**
  - **The semantic routines perform interpretation based on the syntax structure.**
  - **Attaching attributes to the grammar symbols.**
  - **Values for attributes are computed by semantic actions associated with the grammar productions.**

Example: Syntax-directed translation

- **Example in a parse tree:**
  - **Annotate the parse tree by attaching semantic attributes to the nodes of the parse tree.**
  - **Generate code by visiting nodes in the parse tree in a given order.**
  - **Input:** $y := 3 * x + z$



parse tree



annotated parse tree

Syntax-directed definitions

- **Each grammar symbol is associated with a set of attributes.**
  - **Synthesized attribute : value computed from its children or associated with the meaning of the tokens.**
  - **Inherited attribute : value computed from parent and/or siblings.**
  - **General attribute : value can be depended on the attributes of any nodes.**

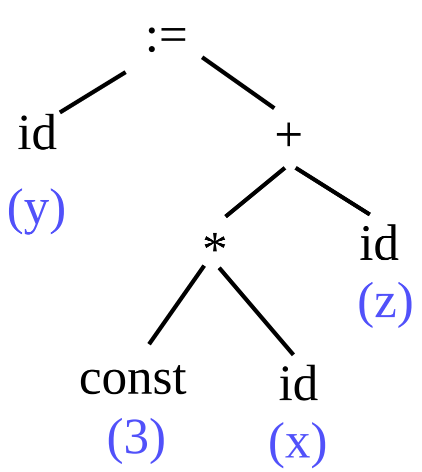
Format for writing syntax-directed definitions

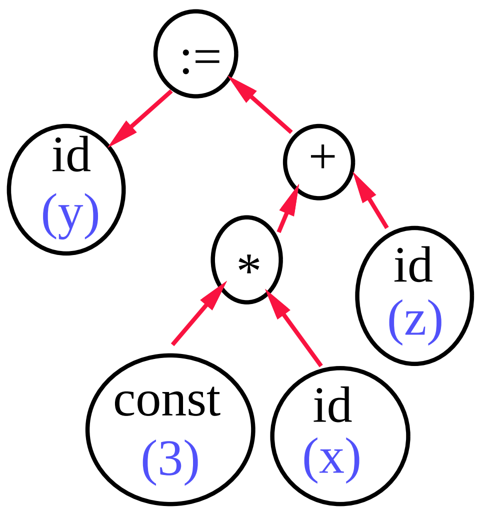
Production	Semantic actions
$L \rightarrow E$	print($E.val$)
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow digit$	$F.val := digit.lexval$

- $E.val$ **is one of the attributes of** E**.**
- **To avoid confusion, recursively defined nonterminals are numbered on the RHS.**
- **Semantic actions are performed when this production is "used".**

Order of evaluation (1/2)

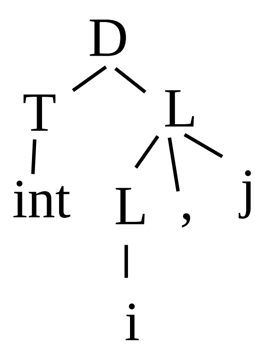
- **Order of evaluating attributes is important.**
- **General rule for ordering:**
  - **Dependency graph :**
    - ▷ ***If attribute b needs attributes a and c, then a and c must be evaluated before b.***
    - ▷ ***Represented as a directed graph without cycles.***
    - ▷ ***Topologically order nodes in the dependency graph as n_1, n_2, ..., n_k such that there is no path from n_i to n_j with $i > j$.***

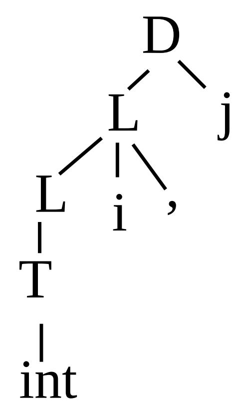




Order of evaluation (2/2)

- **It is always possible to rewrite syntax-directed definitions using only synthesized attributes, but the one with inherited attributes is easier to understand.**
  - **Use inherited attributes to keep track of the type of a list of variable declarations.**
    - ***Example:*** $int\ i, j$
  - **Grammar 1: using inherited attributes**
    - $D \rightarrow T L$
    - $T \rightarrow int \mid char$
    - $L \rightarrow L, id \mid id$
  - **Grammar 2: using only synthesized attributes**
    - $D \rightarrow L\ id$
    - $L \rightarrow L\ id, \mid T$
    - $T \rightarrow int \mid char$





Attribute grammars

- **Attribute grammar: a grammar with syntax-directed definitions and having no side effects.**
  - **Side effect: change values of others not related to the return values of functions themselves.**
- **Tradeoffs:**
  - **Synthesized attributes are easy to compute, but are sometimes difficult to be used to express semantics.**
    - ▷ *S-attributes.*
  - **Inherited and general attributes are difficult to compute, but are sometimes easy to express the semantics.**
  - **The dependence graph for computing some inherited and general attributes may contain cycles and thus not be computable.**
  - **A restricted form of inherited attributes is invented.**
    - ▷ *L-attributes.*

S-attributed definition

- **Definition: a syntax-directed definition that uses synthesized attributed only.**
  - **A parse tree can be represented using a directed graph.**
  - **A post-order traverse of the parse tree can properly evaluate grammars with S-attributed definitions.**
  - **Goes naturally with LR parsers.**
- **Example of an S-attributed definition: $3 * 5 + 4$ return**

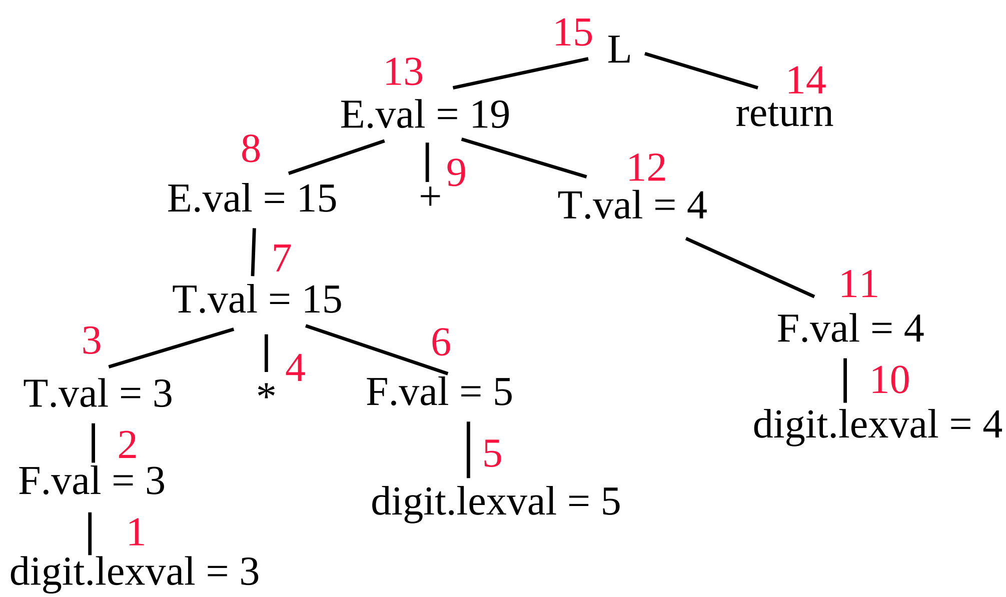
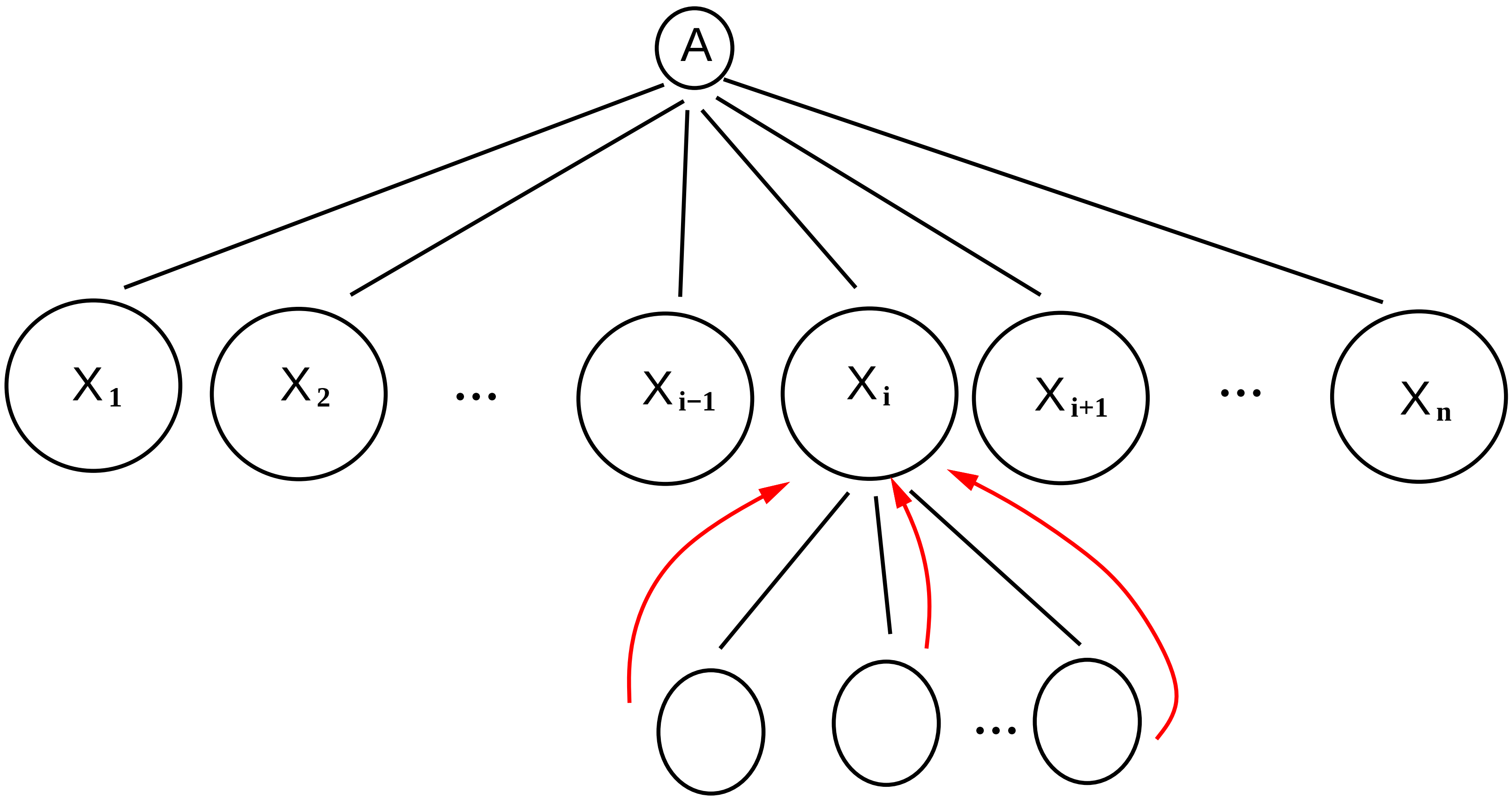


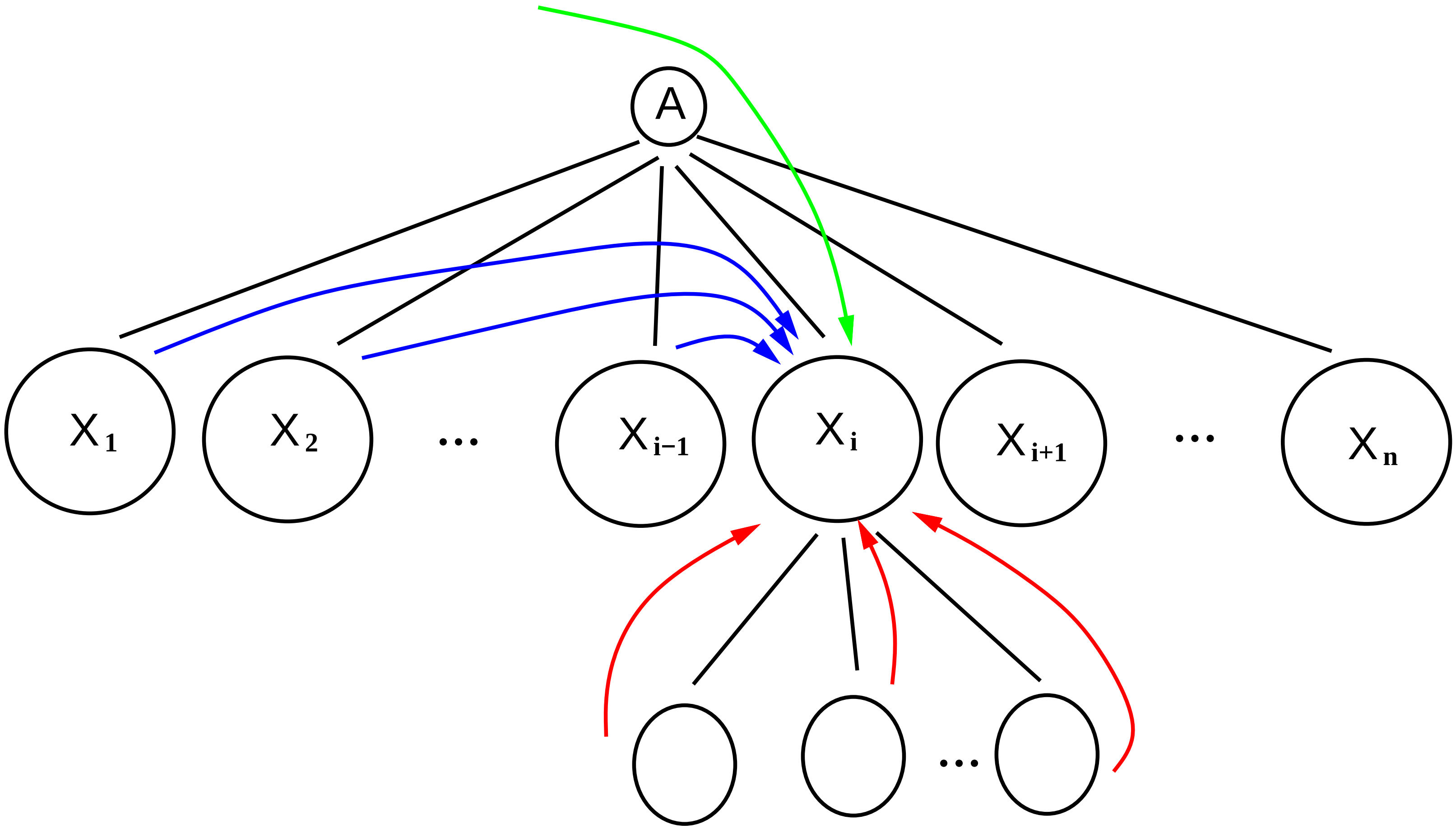
Illustration: S-attributed definition



L-attributed definitions

- **Each grammar symbol can have many attributes. However, each attribute must be either**
  - **a synthesized attribute, or**
  - **an inherited attribute with the following constraints.**
    Assume there is a production $A \rightarrow X_1 X_2 \cdots X_n$ and the inherited attribute is associated with X_i. Then this inherited attribute depends only on
    - ***the inherited attributes of its parent node A;***
    - ***either inherited or synthesized attributes from its elder siblings $X_1, X_2, \ldots, X_{i-1}$;***
    - ***inherited or synthesized attributed associated from itself X_i, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i.***
- **Every S-attributed definition is an L-attributed definition.**

Illustration: L-attributed definition



Evaluations of L-attributed definitions

- **For grammars with L-attributed definitions, special evaluation algorithms must be designed.**
- **L-attributes are always computable.**
  - **Similar arguments as the one used in discussing Algorithm 4.19 for removing left recursion.**
- **Evaluation of L-attributed grammars.**
  - **Goes together naturally with LL parsers.**
    - ▷ ***Parse tree generate by recursive descent parsing corresponds naturally to a top-down tree traversal using DFS by visiting the sibling nodes from left to right.***
- **High level ideas for tree traversal.**
  - **Visit a node v first.**
    - ▷ ***Compute inherited attributes for v if they do not depend on synthesized attributes of v.***
  - **Recursively visit each children of v one by one from left to right.**
  - **Visit the node v again.**
    - ▷ ***Compute synthesized attributes for v.***
    - ▷ ***Compute inherited attributes for v if they depend on synthesized attributes of v.***

Format for writing L-attributed definitions

- $D \to T\ \{L.in := T.type\}\ L$
- $T \to int\ \{T.type := integer\}$
- $T \to real\ \{T.type := real\}$
- $L \to\ \{L_1.in := L.in\}\ L_1, id\ \{addtype(id.entry, L.in)\}$
- $L \to id\ \{addtype(id.entry, L.in)\}$

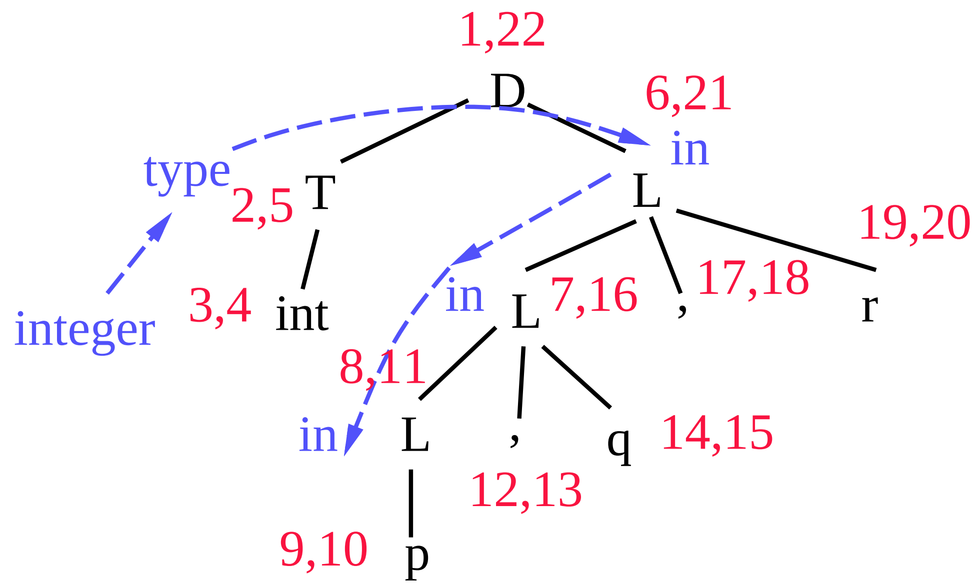
- **Some semantic actions can be inserted between symbols on the RHS of a production.**
  - $A \to B\ \{action\}\ C$
  - **When A expands to B and C, after finishes expanding B, performs $action$, then expands C.**

Example: L-attributed definitions

- $D \to T$ $\{L.in := T.type\}$ L
- $T \to int$ $\{T.type := integer\}$
- $T \to real$ $\{T.type := real\}$
- $L \to$ $\{L_1.in := L.in\}$ L_1, id $\{addtype(id.entry, L.in)\}$
- $L \to id$ $\{addtype(id.entry, L.in)\}$

Parsing and dependency graph:

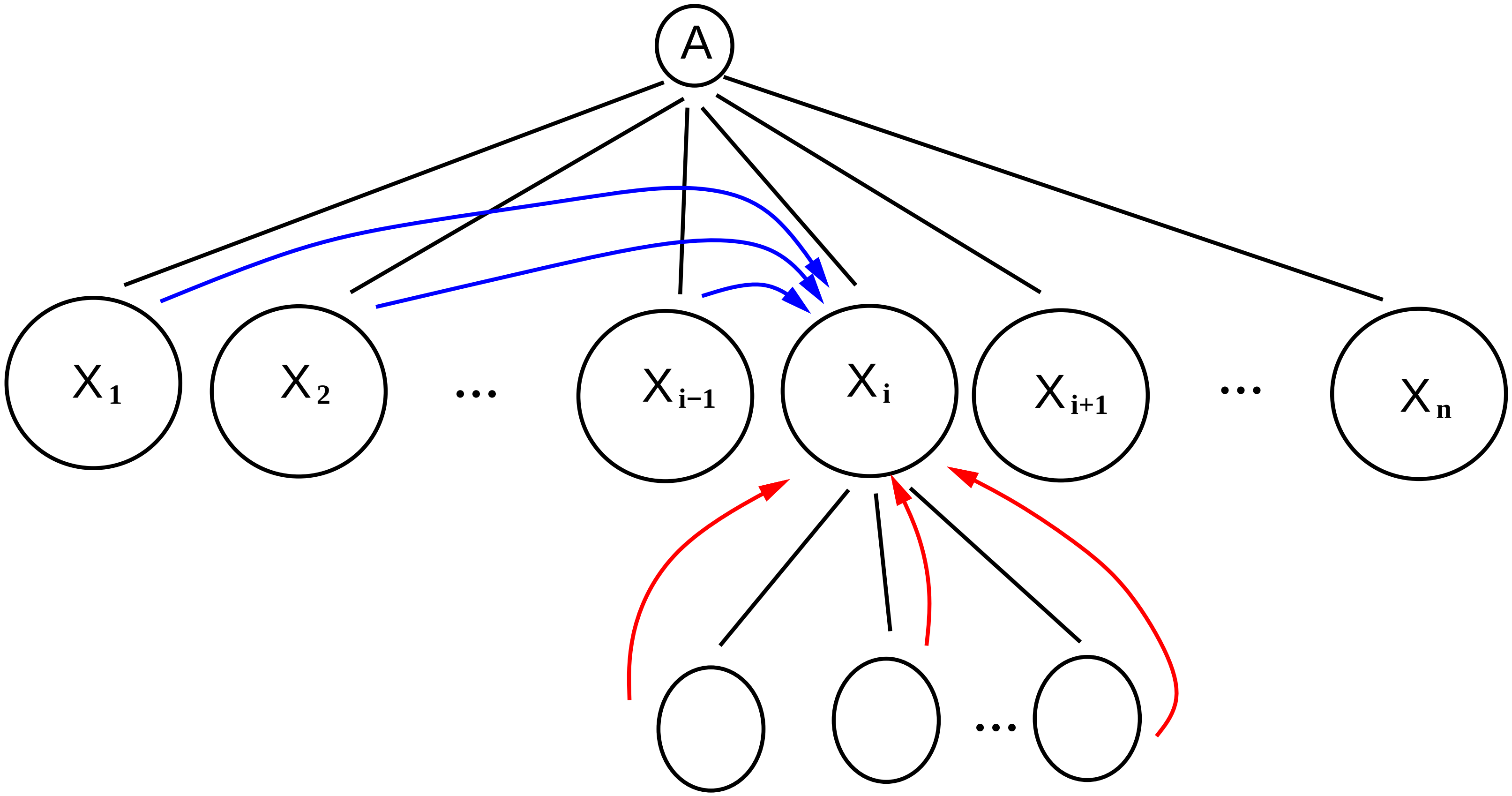
STACK	input	production used
	int p, q, r	
D	**int** p, q, r	
$L\ T$	**int** p, q, r	$D \to TL$
$L\ int$	**int** p, q, r	$T \to int$
L	p, q, r	
$id\ ,\ L$	p, q, r	$L \to L, id$
$id\ ,\ id\ ,\ L$	p, q, r	$L \to L, id$
$id\ ,\ id\ ,\ id$	p, q, r	$L \to id$
$id\ ,\ id$	q, r	
id	q	



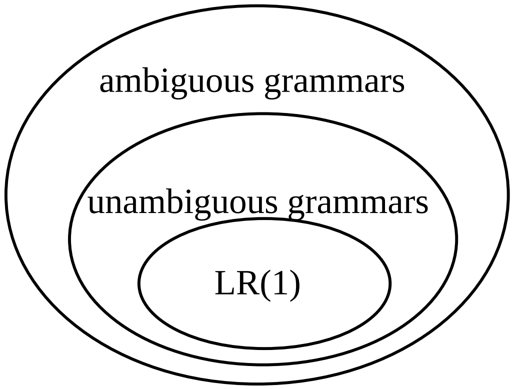
Problems with L-attributed definitions

- **Comparisons:**
  - L-attributed definitions go naturally with LL parsers.
  - S-attributed definitions go naturally with LR parsers.
  - L-attributed definitions are more flexible than S-attributed definitions.
  - LR parsers are more powerful than LL parsers.
- **Some cases of L-attributed definitions cannot be in-cooperated into LR parsers.**
  - Assume the next handle to take care is $A \rightarrow X_1 X_2 \cdots X_i \cdots X_k$, and $X_1, \ldots, X_i$ is already on the top of the STACK.
  - Attribute values of $X_1, \ldots, X_{i-1}$ can be found on the STACK at this moment.
  - No information about A can be found anywhere at this moment.
  - Thus the attribute values of X_i cannot be depended on the value of A.
- **L^--attributed definitions:**
  - Same as L-attributed definitions, but **do not** depend on
    - ▷ *the inherited attributes of parent nodes, or*
    - ▷ *any attributes associated with itself.*
  - Can be handled by LR parsers.

Illustration: L^--attributed definition



Using ambiguous grammars



- **Ambiguous grammars often provide a shorter, more natural specification than their equivalent unambiguous grammars.**
- **Sometimes need ambiguous grammars to specify important language constructs.**
  - **Example: declare a variable before its usage.**

```
var xyz : integer
begin
    ...
    xyz := 3;
    ...
```

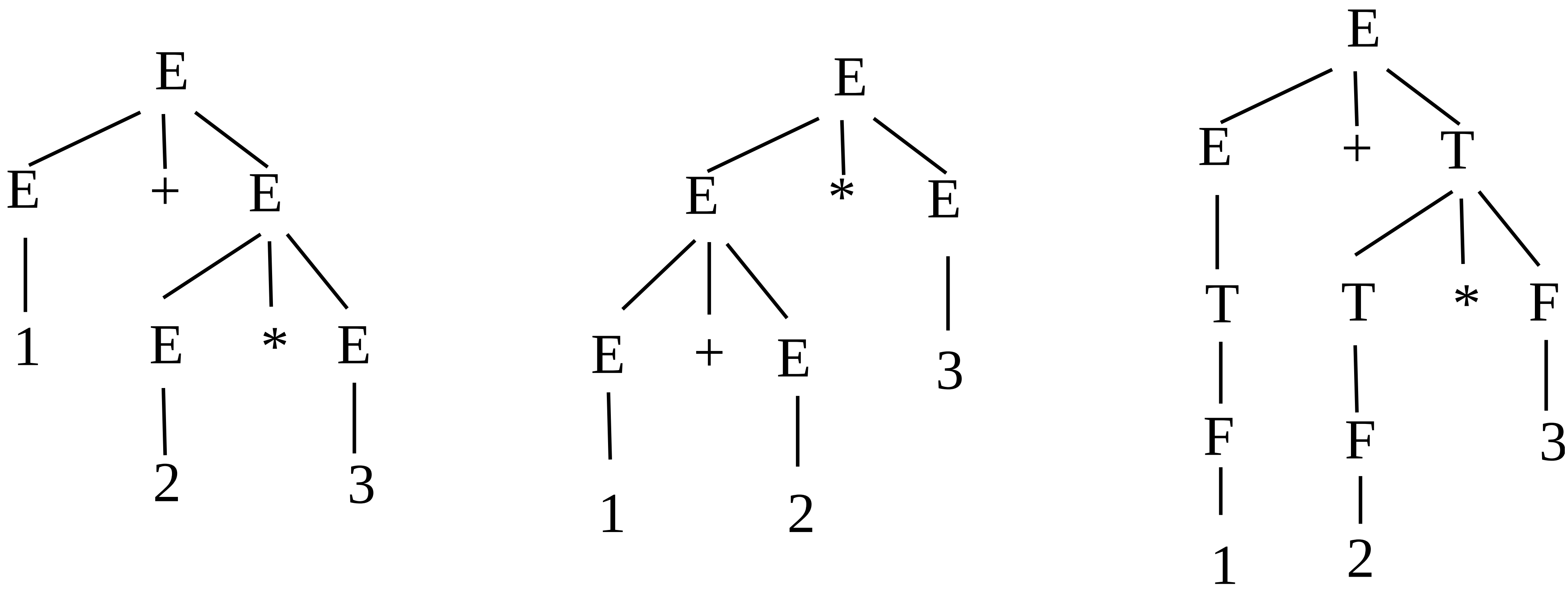
- **Use symbol tables to create "side effects."**

Ambiguity from precedence and associativity

- **Precedence and associativity are important language constructs.**
- **Example:**
  - G_1:
    - $E \rightarrow E + E \mid E * E \mid (E) \mid id$
    - ***Ambiguous, but easy to understand and maintain!***
  - G_2:
    - $E \rightarrow E + T \mid T$
    - $T \rightarrow T * F \mid F$
    - $F \rightarrow (E) \mid id$
    - ***Unambiguous, but difficult to understand and maintain!***

Illustration: using ambiguous grammars

Input: 1+2*3



Parse tree#1: G_1 Parse tree#2: G_1 Parse tree: G_2

Deal with precedence and associativity

- **When parsing the following input for G_1: $id + id * id$.**
  - **Assume the input parsed so far is $id + id$.**
  - **We now see "*".**
  - **We can either shift or perform "reduce by $E \rightarrow E + E$".**
  - **When there is a conflict, say in $LALR(1)$ parsing, we use precedence and associativity information to resolve conflicts.**
    - ▷ ***Here we need to shift because of seeing a higher precedence operator.***
- **Need a mechanism to let user specify what to do when a conflict is seen based on the viable prefix on the STACK so far and the token currently encountered.**

Ambiguity from dangling-else

- **Grammar:**
  - **Statement** $\rightarrow$ **Other_Statement**
    $\mid$ if **Condition** $then$ **Statement**
    $\mid$ if **Condition** $then$ **Statement** $else$ **Statement**
- **When seeing**

$$\underline{if\ \textbf{C}\ then\ \textbf{S}}\ else\ \textbf{S}$$

  - **there is a shift/reduce conflict,**
  - **we always favor a shift.**
  - **Intuition: favor a longer match.**
- **Need a mechanism to let user specify the default conflict-handling rule when there is a shift/reduce conflict.**

Special cases

- **Ambiguity from special-case productions:**
  - **Sometime a very rare happened special case causes ambiguity.**
  - **It is too costly to revise the grammar. We can resolve the conflicts by using special rules.**
  - **Example:**
    - ▷ $E \rightarrow E\ sub\ E\ sup\ E$
    - ▷ $E \rightarrow E\ sub\ E$
    - ▷ $E \rightarrow E\ sup\ E$
    - ▷ $E \rightarrow \{E\} \mid character$
  - **Meanings:**
    - ▷ $W\ sub\ U$: W_U.
    - ▷ $W\ sup\ U$: W^U.
    - ▷ $W\ sub\ U\ sup\ V$ is W_U^V, not ${W_U}^V$.
  - **Resolve by semantic and special rules.**
  - **Pick the right one when there is a reduce/reduce conflict.**
    - ▷ *Reduce the production listed earlier.*
  - **Need a mechanism to let user specify the default conflict-handling rule when there is a reduce/reduce conflict.**

Implementation

- **Passing of synthesized attributes is best.**
  - **Without using global variables.**
- **Cannot use information from its younger siblings because of the limitation of LR parsing.**
  - **During parsing, the STACK contains information about the elder siblings.**
- **It is difficult and usually impossible to pass information from its parent node.**
  - **May be possible to use the state information to pass some information.**
- **Some possible choices:**
  - **Build a parse tree first, then evaluate its semantics.**
  - **Parse and evaluate the semantic actions on the fly.**
- **YACC, an $LALR(1)$ parser generator, can be used to implement L^{-}-attributed definitions.**
  - **Use top of STACK information to pass synthesized attributes.**
  - **Use global variables and internal STACK information to pass the inherited values from its elder siblings.**
  - **Cannot process inherited values from its parent.**

YACC

- Yet Another Compiler Compiler [Johnson 1975]:
  - A UNIX utility for generating $LALR(1)$ parsing tables.
  - Convert your YACC code into C programs.
  - file.y $\longrightarrow$ [yacc file.y] $\longrightarrow$ y.tab.c
  - y.tab.c $\longrightarrow$ [cc y.tab.c -ly -ll] $\longrightarrow$ a.out
- Format:
  - declarations
  - %%
  - grammars and semantic actions.
  - %%
  - supporting C-routines.
- Libraries:
  - Assume the lexical analyzer routine is $yylex()$.
    - ▷ *Need to include the scanner routines.*
  - There is a parser routine $yyparse()$ generated in $y.tab.c$.
  - Default $main$ routines both in LEX and YACC libraries.
    - ▷ *Need to search YACC library first.*

YACC code example (1/2)

```
%{
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#define YYSTYPE int /* integer type for YACC stack */

%}

%token NUMBER ERROR '(' ')'
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
```

YACC code example (2/2)

```
lines   :  lines expr '\n'        {printf("%d\n", $2);}
        |  lines '\n'
        |  /* empty, i.e., epsilon */
        |  lines error '\n' {yyerror("Please reenter:");yyerrok;}
        ;
expr    :  expr '+' expr    { $$ = $1 + $3; }
        |  expr '-' expr    { $$ = $1 - $3; }
        |  expr '*' expr    { $$ = $1 * $3; }
        |  expr '/' expr    { $$ = $1 / $3; }
        |  '(' expr ')'     { $$ = $2; }
        |  '-' expr  %prec UMINUS  { $$ = - $2; }
        |  NUMBER      { $$ = atoi(yytext);}
        ;

%%
#include "lex.yy.c"
```

Included LEX program

```
%{
%}
Digit        [0-9]
IntLit       {Digit}+
%%
[ \t] {/* skip white spaces */}
[\n] {return('\n');}
{IntLit}                                {return(NUMBER);}
"+"                                     {return('+');}
"-"                                     {return('-');}
"*"                                     {return('*');}
"/"                                     {return('/');}
"("                                     {return('(');}
")"                                     {return(')');}
.         {printf("error token <%s>\n",yytext); return(ERROR);}
%%
```

YACC: Declarations

- **System used and C language declarations.**
  - *%{ $\cdots$ %} to enclose C declarations.*
  - *Type of attributes associated with each grammar symbol on the STACK: YYSTYPE declaration.*
  - *This area will not be translated by YACC.*
- **Tokens with associativity and precedence assignments.**
  - *In increasing precedence from top to the bottom.*
  - *%left, %right or %token (non-associativity): e.g., dot products of vectors has no associativity.*
- **Other declarations.**
  - *%type*
  - *%union*
  - $\cdots$

YACC: Productions and semantic actions

- **Format: for productions P with a common LHS**
  - ▷ *<common LHS of P>: <RHS_1 of P> { semantic actions # 1}*
  - ▷ *|<RHS_2 of P> { semantic actions # 2}*
  - ▷ $\cdots$
- **The semantic actions are performed, i.e., C routines are executed, when this production is reduced.**
- **Special symbols and usages.**
  - **Accessing attributes associated with grammar symbols:**
    - ▷ *$$: the return value of this production if it is reduced.*
    - ▷ *$$i$: the returned value of the ith symbol in the RHS of the production.*
  - **%prec declaration.**
- **When there are ambiguities:**
  - **reduce/reduce conflict: favor the one listed first.**
  - **shift/reduce conflict: favor shift, i.e., longer match.**
  - **Q: How to implement this?**

YACC: Error handling

- **Example:** `lines: error '\n' {...}`
  - *When there is an error, skip until newline is seen.*
- $error$: **special nonterminal.**
  - *A production with $error$ is "inserted" or "processed" only when it is in the reject state.*
  - *It matches any sequence on the STACK as if the handle "$error \rightarrow \cdots$" is seen.*
  - *Use a special token to immediately follow $error$ for the purpose of skipping until something special is seen.*
  - *Q: How to implement this?*
- **Use $error$ to implement statement terminators in language designs.**
  - *The token after $error$ is a synchronizing token for panic mode recovery.*
  - *Difficult to implement statement separators using $error$.*
- $yyerrok$: **a macro to reset error flags and make $error$ invisible again.**
- $yyerror(string)$: **pre-defined routine for printing error messages.**

In-production actions

- **Actions can be inserted in the middle of a production, each such action is treated as a nonterminal.**
  - **Example:**

    ```
    expr    :  expr {actions} '+' expr {$$ = $1 + $4; }
    ```

    is translated into

    ```
    expr   :  expr $ACT '+' expr {$$ = $1 + $4;}
    $ACT   : {actions}
    ```

    ▷ ***Split a production into two.***

    ▷ ***Create a nonterminal $ACT and an ϵ-production.***
- **Avoid in-production actions.**
  - **An ϵ-production, e.g., $A \to \epsilon$, can easily generate conflicts.**

    ▷ ***A reduce by "$A \to \cdot$" for states including this item.***
- **Split the production yourself.**

  ▷ ***May generate some conflicts.***

  ▷ ***May be difficult to specify precedence and associativity.***

  ▷ ***May change the parse tree and thus the semantic.***

```
expr    :  exprhead exptail {$$ = $1 + $2;}
exphead :  expr { perform some semantic actions; $$ = $1;}
exptail : '+'  expr {$$ = $2;}
```

Some useful YACC programming styles

- **Keep the RHS of a production short, but not too short.**
  - **Better to have 3 to 4 symbols.**
- **Language issues.**
  - **Avoiding using names starting with "$".**
    - ▷ *YACC auto-generated variable names.*
  - **Watch out C-language rules.**
    - ▷ *goto*
  - **Some C-language reserved words are used by YACC.**
    - ▷ *union*
  - **Some YACC pre-defined routines are macros, not procedures.**
    - ▷ *yyerrok*
- **Rewrite the productions with L-attributed definitions to productions with S-attributed definitions.**
  - **Grammar 1: Array $\rightarrow$ id [Elist]**
  - **Grammar 2:**
    - ▷ *Array* $\rightarrow$ *Aelist]*
    - ▷ *Aelist* $\rightarrow$ *Aelist, id* $|$ *Ahead*
    - ▷ *Ahead* $\rightarrow$ *id [id*

Limitations of syntax-directed translation

- **Limitation of syntax-directed definitions: Without using global data to create side effects, some of the semantic actions cannot be performed.**
- **Examples:**
  - **Checking whether a variable is defined before its usage.**
  - **Checking the type and storage address of a variable.**
  - **Checking whether a variable is used or not.**
  - **Need to use a symbol table : global data to create controlled side effects of semantic actions.**
- **Common approaches in using global variables:**
  - **A program with too many global variables is difficult to understand and maintain.**
  - **Restrict the usage of global variables to essential ones and use them as objects.**
    - ▷ *Symbol table.*
    - ▷ *Labels for GOTO's.*
    - ▷ *Forwarded declarations.*
  - **Tradeoff between ease of coding and ease of maintaining.**