CSC 340

YACC and Syntax Directed Translation

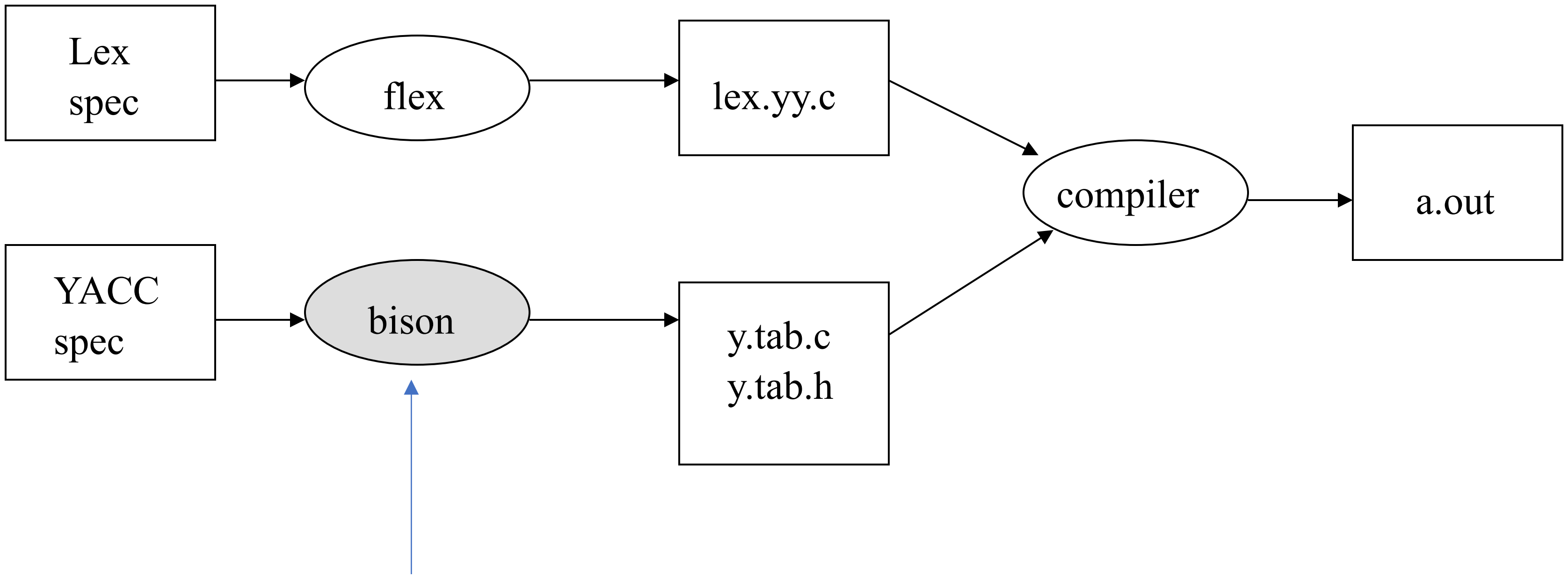
Part 1: Introduction to YACC

YACC

- parser generator used to facilitate the construction of the front end of a compiler.
- We will use the LALR parser generator **YACC**
- **YACC** stands for "**y**et **a**nother **c**ompiler-**c**pmpiler"
  - reflecting the popularity of parser generators in the early 1970s when the first version of **YACC** was created by S. C. Johnson.
  - **YACC** is available as a command on the UNIX system, and has been used to help implement many production compilers.
  - Bison: Gnu version by Corbett and Stallman(1985)

YACC – Yet Another Compiler Compiler

C/C++ tools



More powerful open source replacement of YACC

YACC Specifications

```
Declarations
%%
Translation rules
%%
Supporting C/C++ code
```

- Similar structure to Lex
- Syntax Analyzer (parser) generator

YACC Declarations

Includes:

- Optional C/C++/Java code (%{ … %}) – copied directly into y.tab.c
- YACC definitions (%token, %start, …) – used to provide additional information
  - **%token** – interface to lex, declares names representing tokens
  - Use the %token directive
    - All terminal symbols should be declared through `%token`.
    - Every name not defined in the declarations section is assumed to represent a nonterminal symbol
    - %token DIGIT
  - **%start** – start symbol, the parser is designed to recognize the start symbol, which represents the largest, most general structure described by the grammar rules
    - By default, the start symbol is taken to be the left hand side of the first grammar rule in the rules section
  - Others: %type, %left, %right, %union …

YACC Rules

- Each rule contains LHS and RHS, separated by a colon and end by a semicolon.
- White spaces or tabs are allowed.
- Ex:

```
statement: name EUQALSIGN expression
              | expression ;

expression: number PLUSSIGN number
             | number MINUSSIGN number ;
```

- Actions may be associated with rules and are executed when the associated production is reduced.

YACC Actions

- Actions are C/C++/Java code.
- Actions can include references to attributes associated with terminals and non-terminals in the productions(Semantic Routines.)
- Actions may be put inside a rule – action performed when symbol is pushed on stack

```
expression : simple_expression
 | simple_expression {somefunc($1);} relop
simple_expression;
```

- Safest (i.e. most predictable) place to put action is at end of rule.

The Lexer

- the parser is the higher level routine, and calls the lexer `yylex()` when it needs a new token
- Yacc produces y.tab.h by %token definitions.
- The lex input file must contains y.tab.h
- For each token that lex recognized, a number is returned (from *yylex()* function.)

Integration with Flex (C/C++)

- `yyparse()` calls `yylex()` when it needs a new token. YACC handles the interface details

In the Lexer	In the Parser
`return(TOKEN)`	`%token TOKEN` TOKEN used in productions
`return('c')`	`'c'` used in productions

- `yylval` is used to return attribute information

Building YACC parsers

- For `input.l` and `input.y`

- In `input.l` spec, need to `#include “input.tab.h”`

```
bison -d input.y  # make input.tab.h input.tab.c  OR  yacc -d input.y
flex input.l      # make lex.yy.c
gcc input.tab.c lex.yy.c -ly -ll  # compile
./a<test.txt  # run
```

the order matters

Basic Lex/YACC example

Lex (sample.l)

```
%{
#include “sample.tab.h”
%}
%%
[a-zA-Z]+         {return( NAME );}
[0-9]{3}”-”[0-9]{4}
                  {return( NUMBER );}
[ \n\t]           {;}
%%
```

YACC (sample.y)

```
%token NAME NUMBER
%%
file : file line
     | line
     ;
line : NAME NUMBER
     ;
%%
```

Associated Flex specification

```
%{
#include "expr.tab.h"
%}
%%
\*        {return( '*' );}
\+        {return( '+' );}
\(        {return( '(' );}
\)        {return( ')' );}
[0-9]+    {return( NUMBER );}
.         {;}
%%
```

```
%token NUMBER
%%
line    : expr
        ;
expr    : expr '+' term
        | term
        ;
term    : term '*' factor
        | factor
        ;
factor  : '(' expr ')'
        | NUMBER
        ;
%%
```

Notes: Debugging YACC conflicts: shift/reduce

- Sometimes you get shift/reduce errors if you run YACC on an incomplete program.
  - Don’t stress about these too much UNTIL you are done with the grammar.
- If you get shift/reduce errors, YACC can generate information for you (`y.output`) if you tell it to (`-v`)

Example: IF stmts

```
%token IF_T THEN_T ELSE_T STMT_T
%%
if_stmt    : IF_T condition THEN_T stmt
           | IF_T condition THEN_T stmt ELSE_T stmt
           ;

condition  : '(' ')'
           ;
stmt       : STMT_T
           | if_stmt
           ;
%%
```

This input produces a shift/reduce error

In `y.output` file:

```
7: shift/reduce conflict (shift 10, red'n 1) on ELSE_T

state 7
        if_stmt :  IF_T condition THEN_T stmt_    (1)
        if_stmt :  IF_T condition THEN_T stmt_ELSE_T stmt

        ELSE_T  shift 10
        .  reduce 1
```

Precedence/Associativity in YACC

- precedence and associativity is a major source ambiguity →shift/reduce conflict in YACC.
- You can specify precedence and associativity in YACC, making your grammar simpler.
- Associativity: `%left`, `%right`, `%nonassoc`
- Precedence given order of specifications”

```
%left PLUS MINUS
%left MULT DIV
%nonassoc UMINUS
```

Precedence/Associativity in YACC

```
%left PLUS MINUS
%left MULT DIV
%nonassoc UMINUS
…
%%
…
expression : expression PLUS expression
           | expression MINUS expression
…
```

Error

- When an error occurs yyerror() is called
    - yyerror()is built-in

```
yyerror(const char *msg) { printf("%s\n", msg);}
```

- You may want to redefine it to give more information such as:

```
yyerror(const char *s) {
printf("%d: %s at '%s'\n",yylineno,s,yytext); }
```

  - You may have to define and/or set yylineno

Error State / Error Recovery

- Only one reserved symbol, error.
  - This is a special symbol that can be used for error recovery
- Example:

```
while: WHILE cond statements END;
| WHILE error ; {printf(“Invalid While\n”);}
```

- Placement of error token is difficult to get right,
- try putting it before a statement terminal, i.e. ';'

- Yacc uses a form of error productions

  A→error α

```
%%
line : lines expr '\n'
| lines '\n'
| /* empty */
| error '\n' {yyerror("reenter previous line:");
yyerrok; } ;
```

- yyerrok: resets the parser to normal mode of operation

Part 2: Syntax Directed Translation

Syntax Directed Translation

- Syntax Directed Translations
- Syntax Directed Definitions
- Implementing Syntax Directed Definitions
  - Dependency Graphs
  - S-Attributed Definitions
  - L-Attributed Definitions

Syntax Directed Translation

- **Semantic Analysis** computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a *Representation Formalism* and an *Implementation Mechanism*.
- As representation formalism this lecture illustrates what are called ***Syntax Directed Translations*.**

Syntax Directed Translation

- The **Principle of Syntax Directed Translation** states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By **Syntax Directed Translations** we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
- We associate **Attributes** to the grammar symbols representing the language constructs.
- Values for attributes are computed by **Semantic Rules** associated with grammar productions.

Syntax Directed Translation

- Evaluation of Semantic Rules may:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages;
  - etc.
- • There are two notations for attaching semantic rules:
  1. **Syntax Directed Definitions.** High-level specification hiding many
- implementation details (also called **Attribute Grammars**).
- 2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions

- **Syntax Directed Definitions** are a generalization of context-free grammars in which:

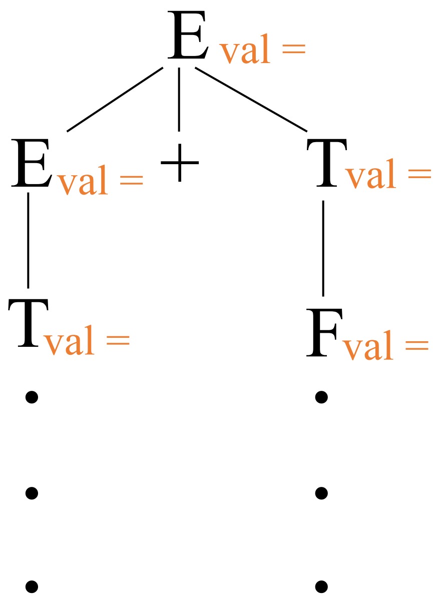
1. Grammar symbols have an associated set of **Attributes**;
2. Productions are associated with **Semantic Rules** for computing the values of attributes.

- Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g., X.a indicates the attribute a of the grammar symbol X).

Example Attribute Grammar

Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = value(num)$
$F \rightarrow (E)$	$F.val = E.val$

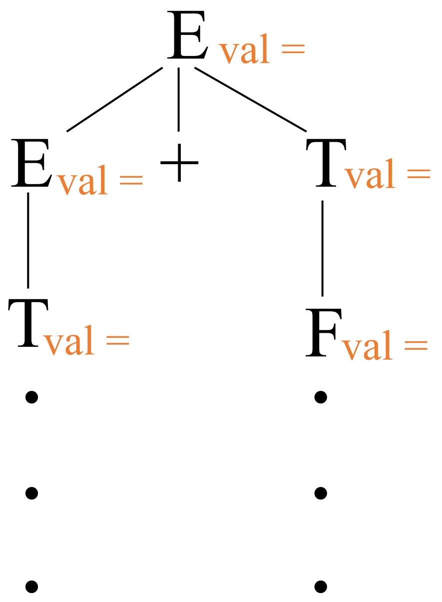
attributes can be associated with nodes in the parse tree



Example Attribute Grammar

Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = value(num)$
$F \rightarrow (E)$	$F.val = E.val$

Rule = compute the value of the attribute 'val' at the parent by adding together the value of the attributes at two of the children



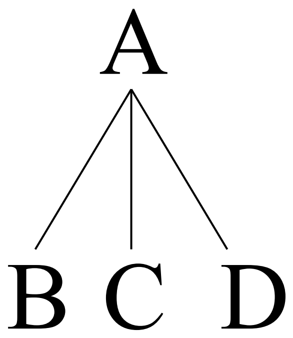
Syntax Directed Definitions (Cont.)

- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.
- We distinguish between two kinds of attributes:
  1. **Synthesized Attributes.** They are computed from the values of the attributes of the children nodes.
- 2. **Inherited Attributes.** They are computed from the values of the attributes of both the siblings and the parent nodes.

Synthesized Attributes

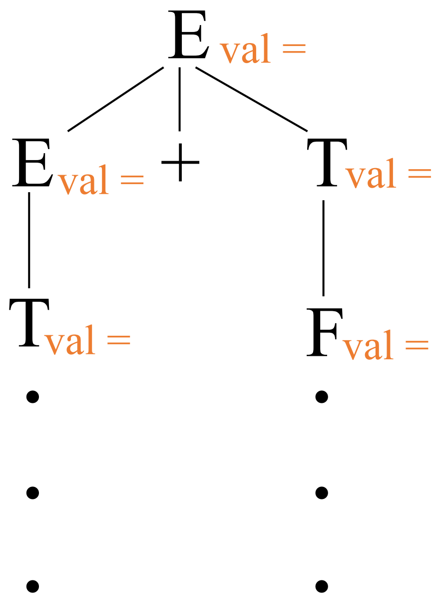
- **Synthesized attributes:** the value of a synthesized attribute for a node is computed using only information associated with the node and the node's children (or the lexical analyzer for leaf nodes).
- **Example**:

Production	**Semantic Rules**
A → B C D	A.a := B.b + C.e



Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = value(num)$
$F \rightarrow (E)$	$F.val = E.val$

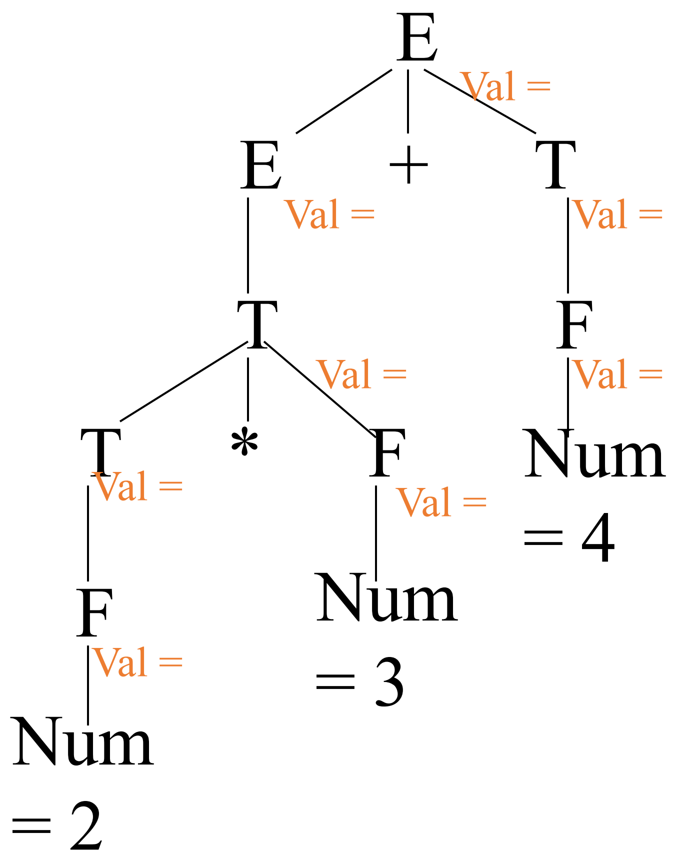


- A set of rules that only uses synthesized attributes is called **S-attributed**

Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val

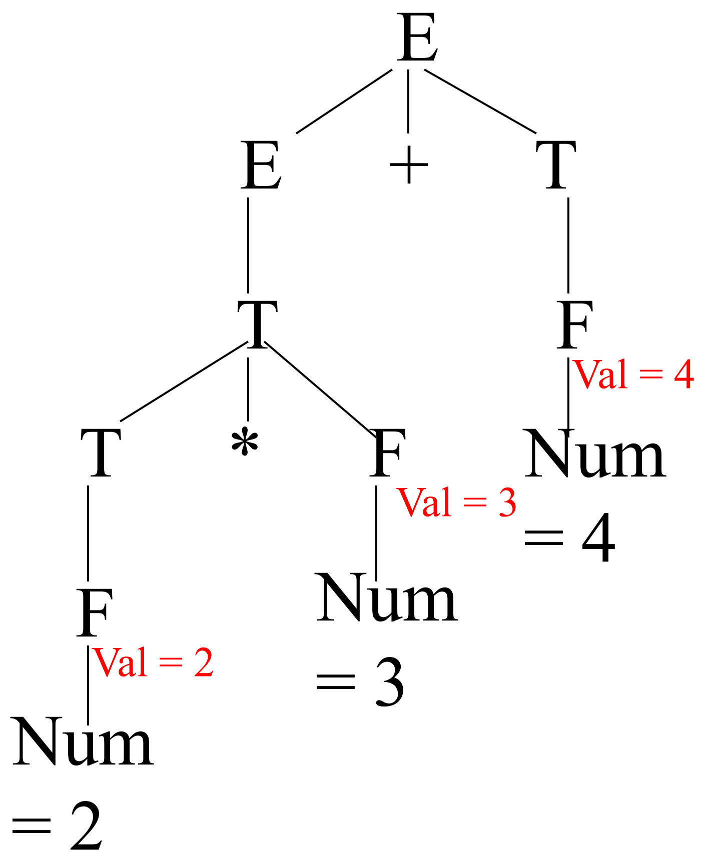
Input: 2 * 3 + 4



Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = value(num)$
$F \rightarrow (E)$	$F.val = E.val$

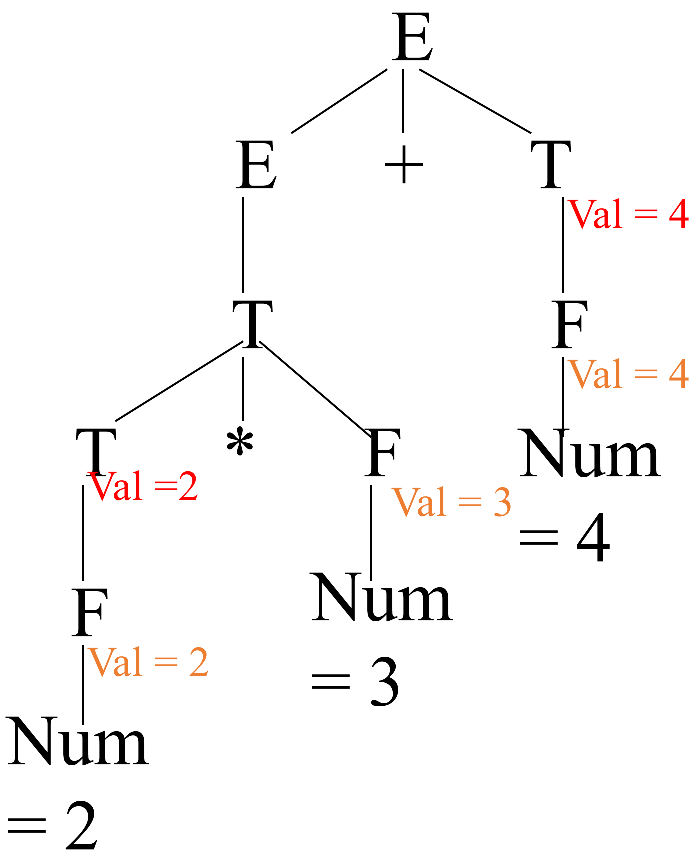
Input: 2 * 3 + 4



Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = value(num)$
$F \rightarrow (E)$	$F.val = E.val$

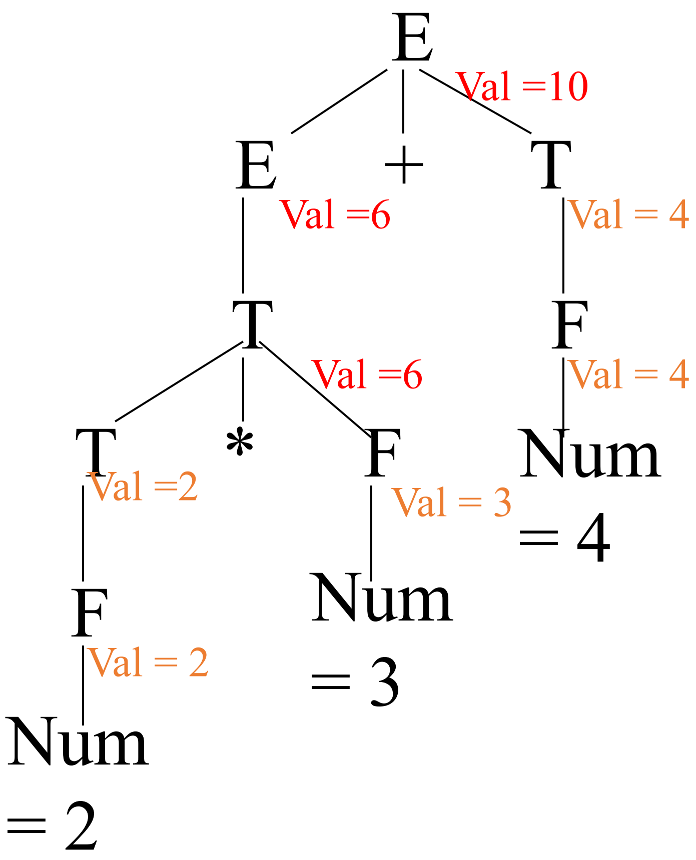
Input: 2 * 3 + 4



Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = value(num)$
$F \rightarrow (E)$	$F.val = E.val$

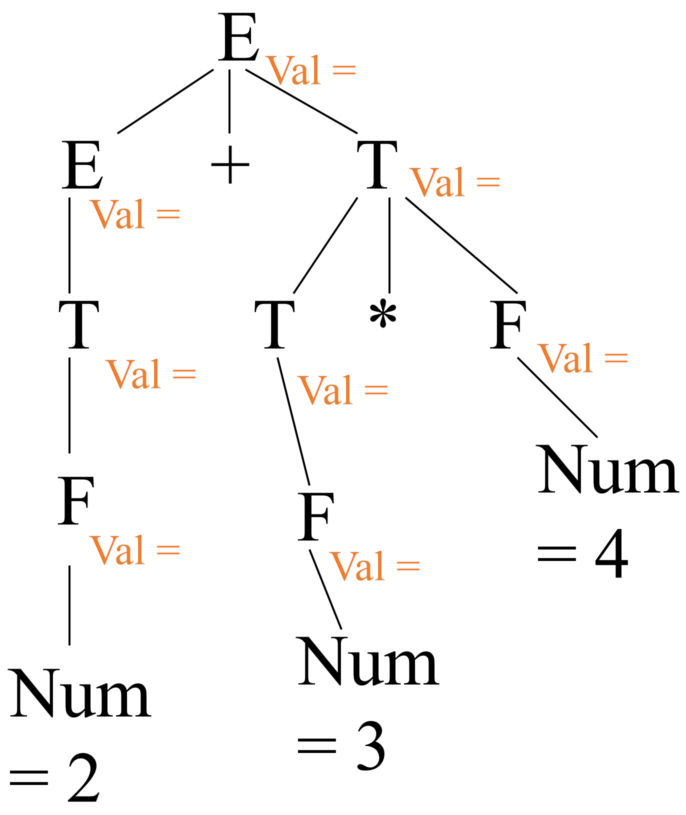
Input: 2 * 3 + 4



Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = value(num)$
$F \rightarrow (E)$	$F.val = E.val$

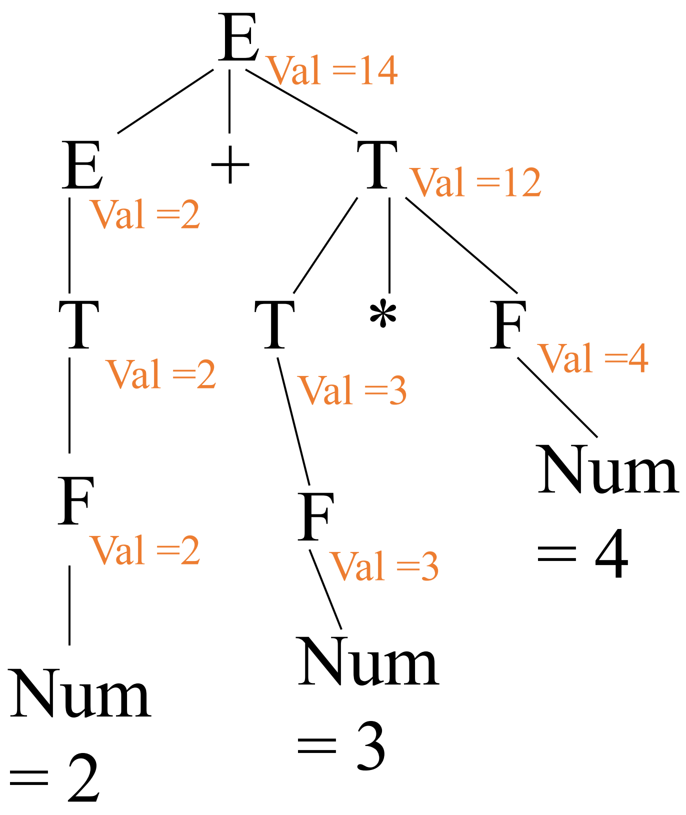
Input: 2 + 4 * 3



Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = value(num)$
$F \rightarrow (E)$	$F.val = E.val$

Input: 2 + 4 * 3

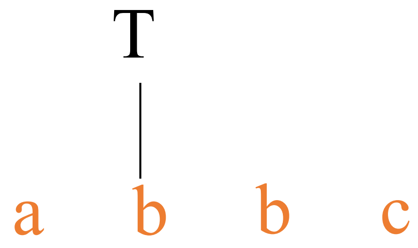


Synthesized Attributes and LR Parsing

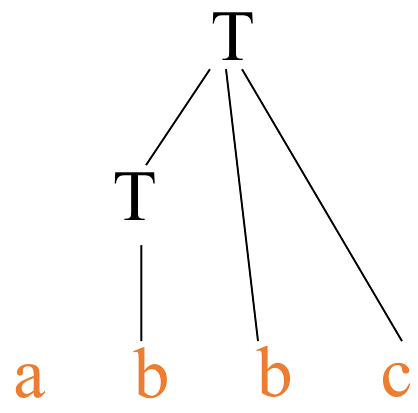
- Synthesized attributes have natural fit with LR parsing
- Attribute values can be stored on stack with their associated symbol
- When reducing by production $A \rightarrow \alpha$, both α and the value of α's attributes will be on the top of the LR parse stack

Synthesized Attributes and LR Parsing

- Example Stack: $0[attr],a1[attr],T2[attr],b5[attr],c8[attr]



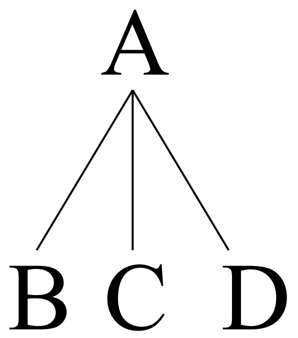
- Stack after T → T b c: $0[attr],a1[attr],T2[attr']



Inherited Attributes

- **Inherited Attributes** are useful for expressing the dependence of a construct on the context in which it appears.
- Example:

Production	**Semantic Rules**
A → B C D	$B\boxed{.b} := A.a + C.b$

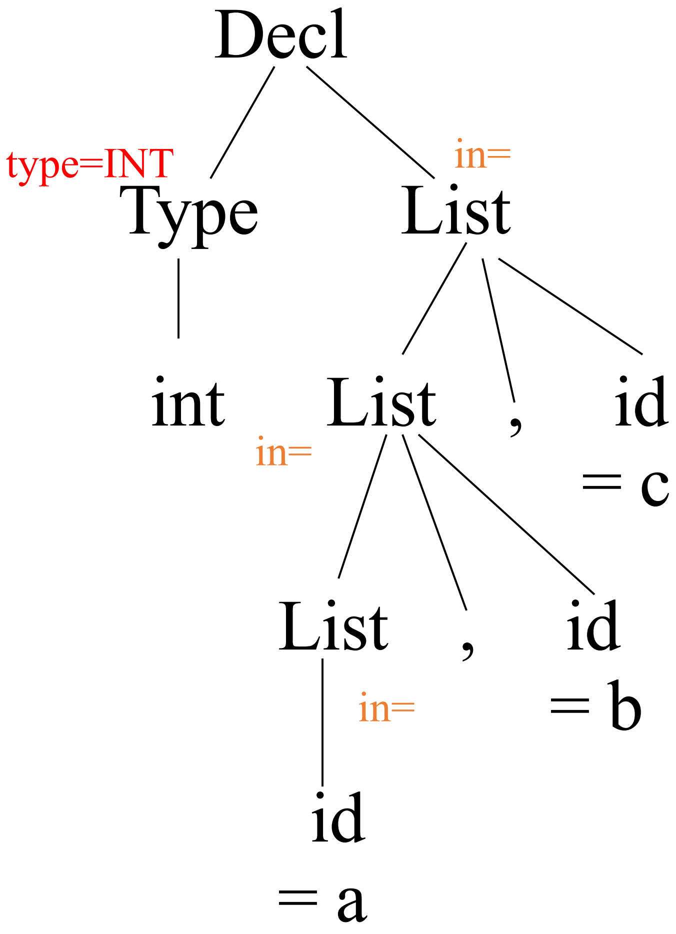


Inherited Attributes – Determining types

Productions	**Semantic Actions**
Decl → Type List	List.in = Type.type
Type → int	Type.type = INT
List → List_1, id	List_1.in = List.in, addtype(id.entry,List.in)
List → id	addtype(id.entry,List.in)

Inherited Attributes – Example

Productions	**Semantic Actions**
Decl → Type List	List.in = Type.type
Type → int	Type.type = INT
List → $List_1$, id	$List_1$.in = List.in, addtype(id.entry.List.in)
List → id	addtype(id.entry,List.in)

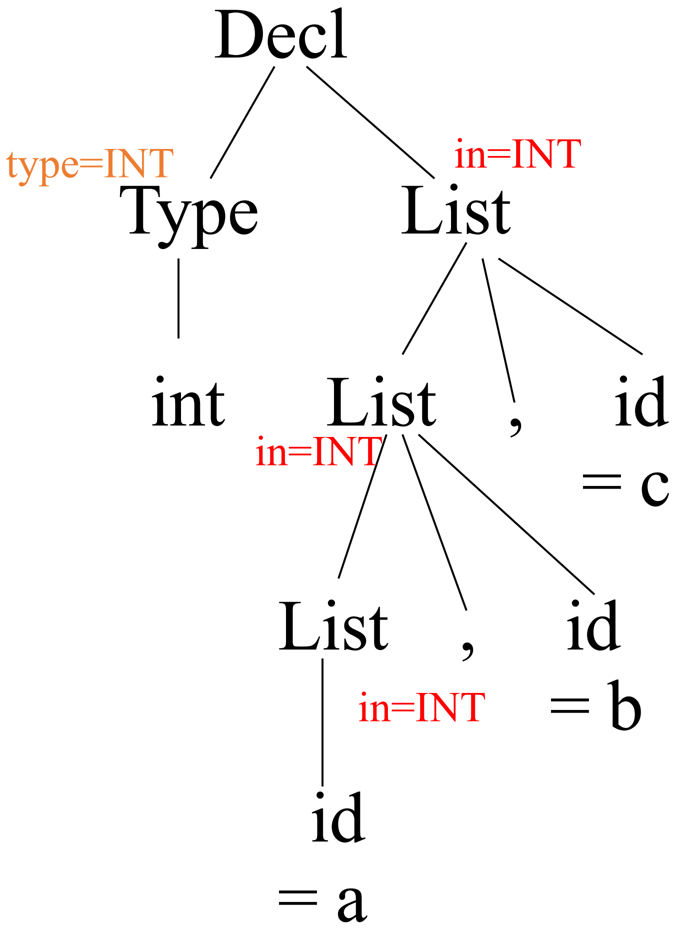


Input: int a,b,c

Inherited Attributes – Example

Productions	**Semantic Actions**
Decl → Type List	List.in = Type.type
Type → int	Type.type = INT
List → $List_1$, id	$List_1$.in = List.in, addtype(id.entry.List.in)
List → id	addtype(id.entry,List.in)

Input: int a,b,c



Evaluation Order. Inherited attributes cannot be evaluated by a simple PreOrder traversal of the parse-tree:
– Unlike synthesized attributes, the order in which the inherited attributes of the children are computed is important!!! Indeed:
∗ Inherited attributes of the children can depend from both left and right siblings!

Attribute Dependency

- An attribute *b* **depends** on an attribute *c* if a valid value of *c* must be available in order to find the value of *b*.
- The relationship among attributes defines a **dependency graph** for attribute evaluation.
- are the most general technique used to evaluate syntax directed definitions with both synthesized and inherited attributes.

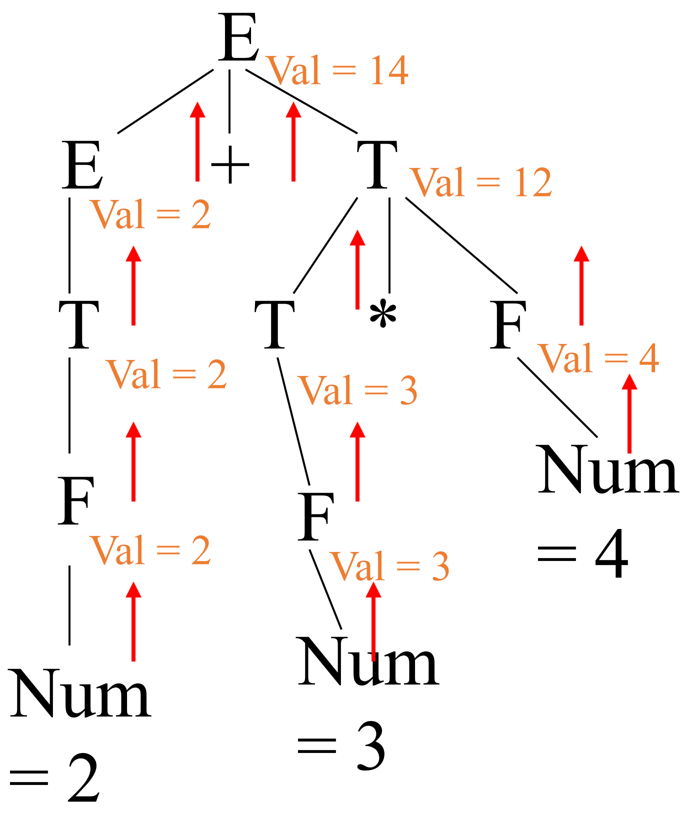
Dependency Graph

- A Dependency Graph shows the interdependencies among the attributes of the various nodes of a parse-tree.
  - There is a node for each attribute;
  - If attribute b depends on an attribute c there is a link from the node for c to the node for b

    (b ← c).
- Dependencies matter when considering syntax directed translation in the context of a parsing technique.

Attribute Dependencies

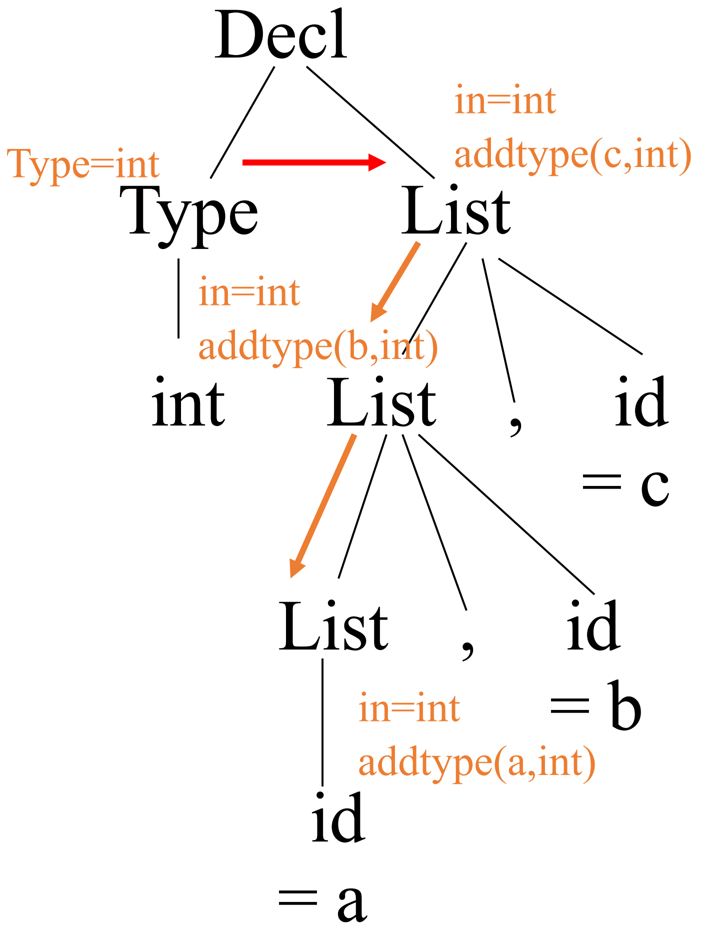
Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow num$	$F.val = value(num)$
$F \rightarrow (E)$	$F.val = E.val$



Synthesized attributes – dependencies always up the tree

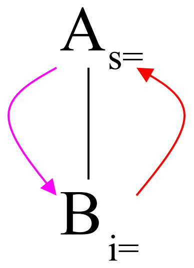
Attribute Dependencies

Productions	**Semantic Actions**
Decl → Type List	List.in = Type.type
Type → int	Type.type = INT
Type → real	T.type = REAL
List → List_1, id	List_1.in = List.in, addtype(id.entry.List.in)
List → id	addtype(id.entry,List.in)



Attribute Dependencies

Productions	**Semantic Actions**
A → B	A.s = B.i B.i = A.s + 1



Circular dependences are a problem

S-Attributed Definitions

- An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.
- **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.

L-Attributed definition

- **L-Attributed Definitions** contain both synthesized and inherited attributes but do not need to build a dependency graph to evaluate them.
- **Definition.** A syntax directed definition is *L-Attributed* if each *inherited attribute* of X_j in a production $A \rightarrow X_1 \ldots X_j \ldots X_n$, depends only on:
  1. The attributes of the symbols to the **left** (this is what L in *L-Attributed* stands for) of X_j, i.e., $X_1 X_2 \ldots X_{j-1}$, and
  2. The *inherited* attributes of A.
- can be evaluated by a mixing PostOrder (synthesized) and PreOrder (inherited) traversal.

Part 3: Back to YACC

Attributes in YACC

- You can associate attributes with symbols (terminals and non-terminals) on right side of productions.
- Elements of a production referred to using ‘$’ notation. Left side is $$. Right side elements are numbered sequentially starting at $1.

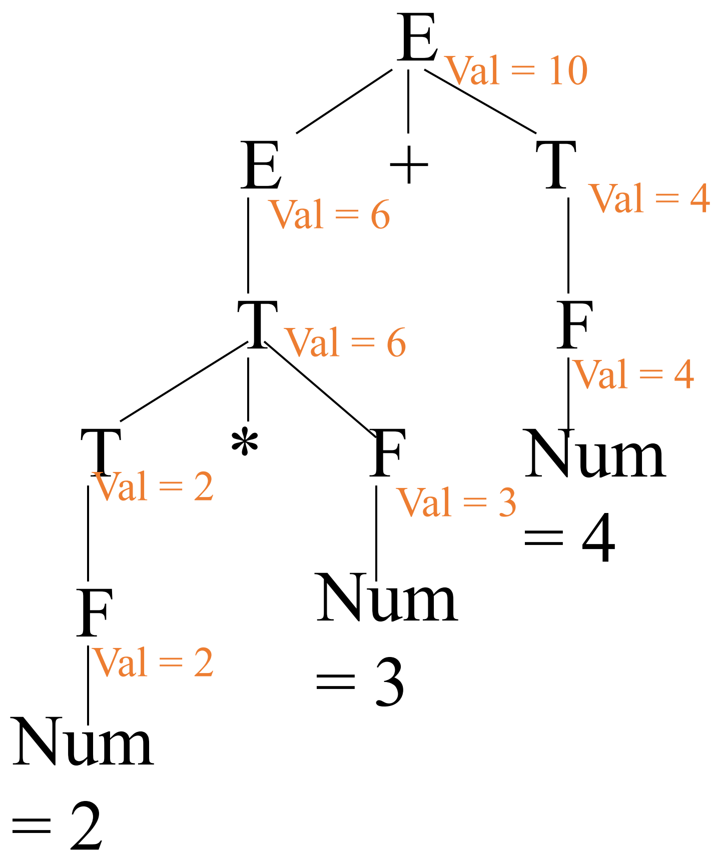
  For A : B C D,

  A is $$, B is $1, C is $2, D is $3.
- Default attribute type is int.
- Default action is $$ = $1;

Back to Expression Grammar

Production	**Semantic Actions**
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val

Input: 2 * 3 + 4



Expression Grammar in YACC

```
%token NUMBER CR                    /* CR: carriage return */
%%
lines  : lines line
       | line
       ;

line   : expr CR                    {printf(“Value = %d”,$1); }
       ;

expr   : expr ‘+’ term              { $$ = $1 + $3; }
       | term                       { $$ = $1;  /* default – can omit */}
       ;

term   : term ‘*’ factor            { $$ = $1 * $3; }
       | factor
       ;

factor: ‘(’ expr ‘)’                { $$ = $2; }
       | NUMBER
       ;
%%
```

Associated Lex Specification

```
%%
\+          {return( '+' ); }
\*          {return( '*' ); }
\(          {return( '(' ); }
\)          {return( ')' ); }
[0-9]+      {yylval = atoi(yytext); return( NUMBER ); }
[\n]        {return( CR ); }
[ \t]       {;}
%%
```

Non-integer Attributes in YACC

- `yylval` assumed to be integer if you take no other action.
- First, types defined in YACC definitions section.

```
%union {
  type1 name1;
  type2 name2;
  …
}
```

Non-integer Attributes in YACC

- Next, define what tokens and non-terminals will have these types:

```
%token <name> token
%type  <name> non-terminal
```

- In the YACC spec, the `$n` symbol will have the type of the given token/non-terminal. If type is a record, field names must be used (i.e. `$n.field`).
- In Lex spec, use `yylval.name` in the assignment for a token with attribute information.
- Careful, default action `($$ = $1;)` can cause type errors to arise.

Example 2 with floating point

```
%union { double f_value; }
%token <f_value>NUMBER
%type  <f_value> expr term factor
%%
expr   : expr '+' term    { $$ = $1 + $3; }
       | term             /* Default action here is $$ = $1 */
       ;
term   : term '*' factor  { $$ = $1 * $3; }
       | factor
       ;
factor: '(' expr ')'      { $$ = $2; }
       | NUMBER
       ;
%%
#include "lex.yy.c"
```

Associated Lex Specification

```
%%
\*              {return( '*' ); }
\+              {return( '+' ); }
\(              {return( '(' ); }
\)              {return( ')' ); }
[0-9]*"."[0-9]+ {yylval.f_value = atof(yytext); return( NUMBER );}
%%
```

Rules for Implementing L-Attributed SDD's.

- If we have an L-Attibuted Syntax-Directed Definition we must enforce the following restrictions:

  1. An inherited attribute for a symbol in the right-hand side of a production must be computed in an action <u>before</u> the symbol;

  2. A synthesized attribute for the non terminal on the left-hand side can only be computed when all the attributes it references have been computed:

  The action is usually put at the end of the production.

A : B {action1} C {action2} D {action3};

- Actions can be embedded in productions. This changes the numbering ($1,$2,…)
- Embedding actions in productions are not always guaranteed to work. However, productions can always be rewritten to change embedded actions into end actions.

```
A     : new_B  new_C  D  {action3 }   ;
new_B : B                {action1 }   ;
new_C : C                {action2 }   ;
```

- Embedded actions are executed when all symbols to the left are on the stack.

When type is a record:

- Field names must be used: `$n.field` has the type of the given field.
- In Lex, `yylval` uses the complete name:

  `yylval.typename.fieldname`
- If type is pointer to a record, `->` is used (as in C/C++).

Example with records

Production	**Semantic Actions**
seq → seq_1 instr	seq.x = seq_1.x + instr.dx seq.y = seq_1.y + instr.dy
seq → BEGIN	seq.x = 0, seq.y = 0
instr → N	instr.dx = 0, instr.dy = 1
instr → S	instr.dx = 0, instr.dy = -1
instr → E	instr.dx = 1, instr.dy = 0
instr → W	instr.dx = -1, instr.dy = 0

Example in YACC

```
%union{
    struct s1  {int x ; int y } pos;
    struct s2  {int dx; int dy} offset;
}
%type <pos>    seq
%type <offset> instr
%%
seq    : seq instr {$$.x = $1.x + $2.dx; $$.y = $1.y + $2.dy; }
       | BEGIN     {$$.x = 0; $$.y = 0; }
       ;
instr  : N         {$$.dx = 0; $$.dy = 1;}
       | S         {$$.dx = 0; $$.dy = -1;}
       …  ;
%%
```

Attribute oriented YACC error messages

```
%union{
    struct s1  {int x ; int y } pos;
    struct s2  {int dx; int dy} offset;
}
%type <pos>    seq
%type <offset> instr
%%
seq    : seq instr {$$.x = $1.x + $2.dx; $$.y = $1.y + $2.dy; }
       | BEGIN     {$$.x = 0; $$.y = 0; }
       ;
instr  : N          Missing action
       | S         {$$.dx = 0; $$.dy = -1;}
       …   ;
%%
```

yacc example2.y
"example2.y", line 13: fatal: default action causes potential type clash