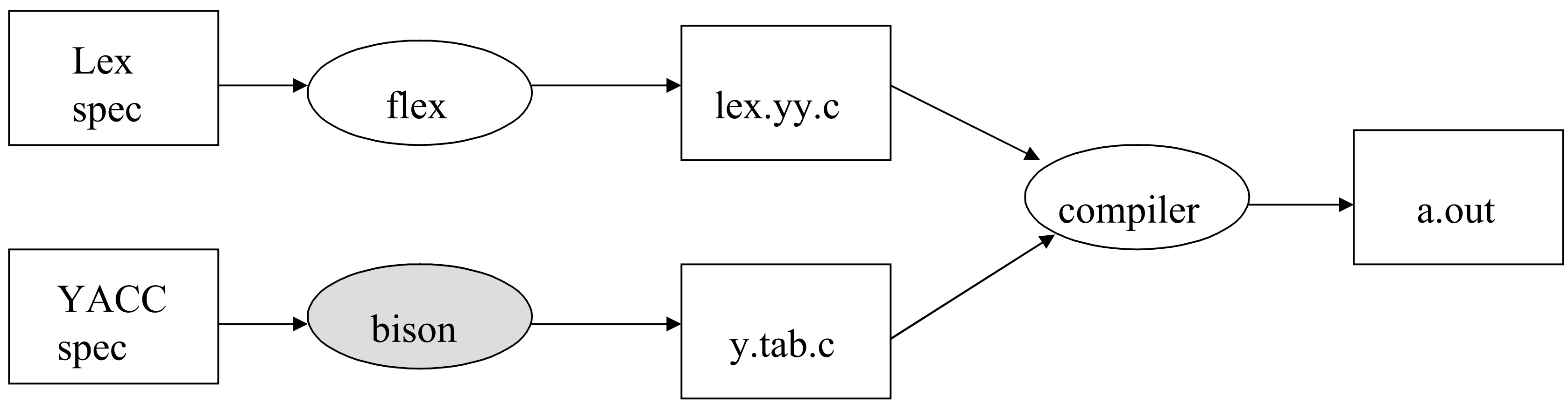
Lecture 6: YACC and Syntax Directed Translation

CS 540

George Mason University

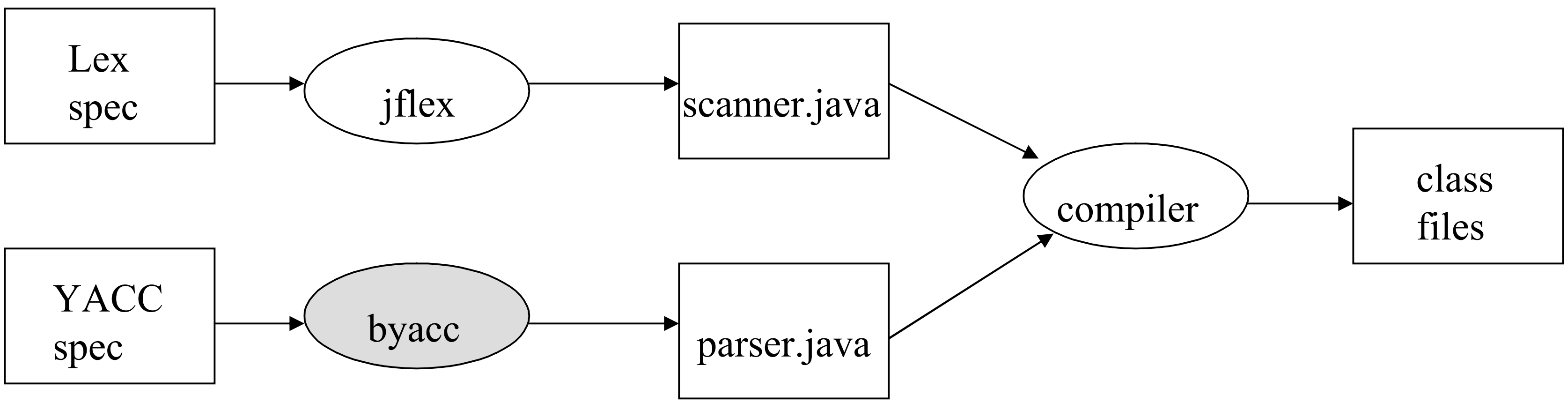
Part 1: Introduction to YACC

YACC – Yet Another Compiler Compiler



C/C++ tools

YACC – Yet Another Compiler Compiler



Java tools

YACC Specifications

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

Similar structure to Lex

YACC Declarations Section

- Includes:
  - Optional C/C++/Java code (%{ … %}) – copied directly into y.tab.c or parser.java
  - YACC definitions (%token, %start, …) – used to provide additional information
    - %token – interface to lex
    - %start – start symbol
    - Others: %type, %left, %right, %union …

YACC Rules

- A rule captures all of the productions for a single non-terminal.
  - Left_side : production 1

```
Left_side : production 1
          |   production 2
          …
          |   production n
          ;
```

- 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.
- Actions may be put inside a rule – action performed when symbol is pushed on stack
- Safest (i.e. most predictable) place to put action is at end of rule.

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

Integration with Jflex (Java)

In the Lexer:	In the Parser:
return Parser.TOKEN	%token TOKEN TOKEN used in productions
{return (int) yycharat(0);}	‘c’ used in productions

Building YACC parsers

For input.l and input.y

- In input.l spec, need to **`#include "input.tab.h"`**
- ```
  flex input.l
  bison -d input.y
  gcc input.tab.c lex.yy.c -ly -ll
  ```

the order matters
  ```

Basic Lex/YACC example

```
%{
#include "sample.tab.h"
%}
%%
[a-zA-Z]+   {return(NAME);}
[0-9]{3}"-"[0-9]{4}

          {return(NUMBER); }
[ \n\t]               ;
%%
```

Lex (sample.l)

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

YACC (sample.y)

Associated Lex Specification (flex)

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

Associated Flex specification

```
%{
#include “expr.tab.h”
%}
%%
\*              {return(‘*’); }
\+              {return(‘+’); }
\(              {return(‘(‘); }
\)              {return(‘)’); }
[0-9]+                {return(NUMBER);}
.               ;
%%
```

byacc Specification

```
%{
import java.io.*;
%}
%token PLUS TIMES INT CR RPAREN LPAREN
%%
lines : lines line | line ;
line : expr CR ;
expr : expr PLUS term | term ;
term : term TIMES factor | factor ;
factor: LPAREN expr RPAREN | INT ;
%%
private scanner lexer;
private int yylex() {
   int retVal = -1;
   try { retVal = lexer.yylex(); }
   catch (IOException e) { System.err.println("IO Error:" + e); }
   return retVal;
}
public void yyerror (String error) {
   System.err.println("Error : " + error + " at line " +
   lexer.getLine());
   System.err.println("String rejected");
}
public Parser (Reader r) { lexer = new scanner (r, this); }
public static void main (String [] args) throws IOException {
 Parser yyparser = new Parser(new FileReader(args[0]));
   yyparser.yyparse();
}
```

Associated jflex specification

```
%%
%class scanner
%unicode
%byaccj
%{
private Parser yyparser;
public scanner (java.io.Reader r, Parser yyparser) {
      this (r); this.yyparser = yyparser; }
public int getLine() { return yyline; }
%}
%%
"+"    {return Parser.PLUS;}
"*"    {return Parser.TIMES;}
"("    {return Parser.LPAREN;}
")"    {return Parser.RPAREN;}
[\n]   {return Parser.CR;}
[0-9]+        {return Parser.INT;}
[ \t]  {;}
```

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

- Forgetting about precedence and associativity is a major source of 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
```

- P. 62-64 in Lex/YACC book

Precedence/Associativity in YACC

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

Part 2: Syntax Directed Translation

Syntax Directed Translation

Syntax = form, Semantics = meaning

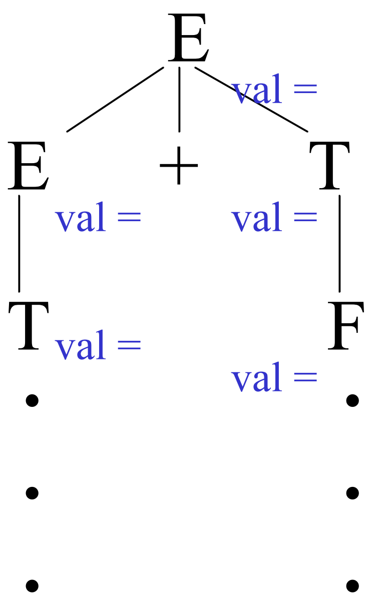
- Use the syntax to derive semantic information.
- Attribute grammar:
  - Context free grammar augmented by a set of rules that specify a computation
  - Also referred to using the more general term: Syntax Directed Definition (SDD)
- Evaluation of attributes grammars – can we fit with parsing?

Attributes

- Associate ***attributes*** with parse tree nodes (internal and leaf).
- Rules (semantic actions) describe how to compute value of attributes in tree (possibly using other attributes in the tree)
- Two types of attributes based on how value is calculated: Synthesized & Inherited

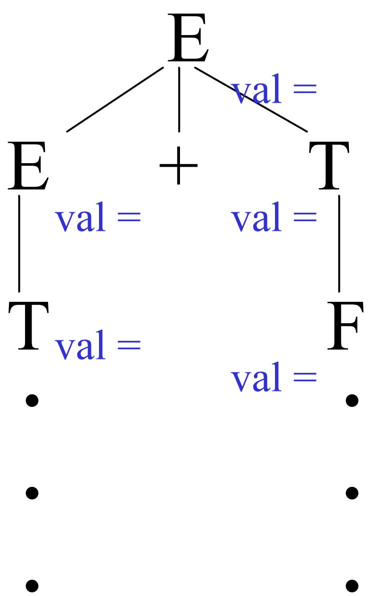
Example Attribute Grammar

attributes can be associated with nodes in 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

Example Attribute Grammar

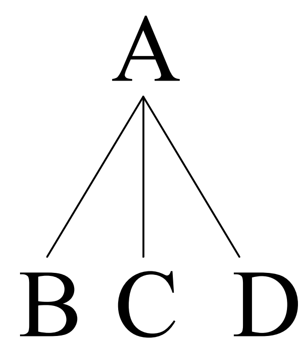


Production	Semantic Actions
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val

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

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).

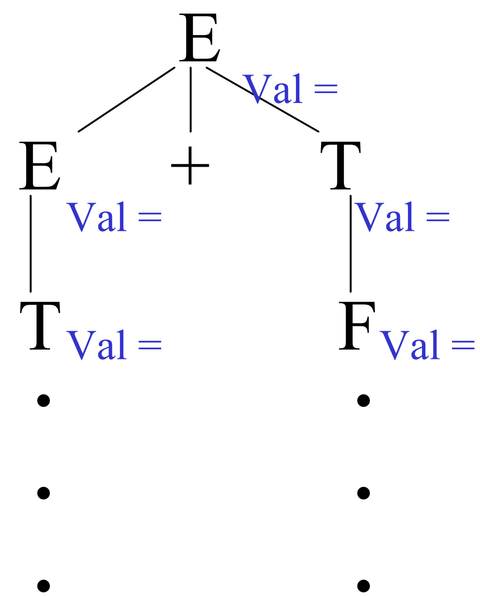


Example:

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

Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
E → 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



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

Example Problems using Synthesized Attributes

- Expression grammar – given a valid expression using constants (ex: 1 * 2 + 3), determine the associated value while parsing.
- Grid – Given a starting location of 0,0 and a sequence of north, south, east, west moves (ex: NESNNE), find the final position on a unit grid.

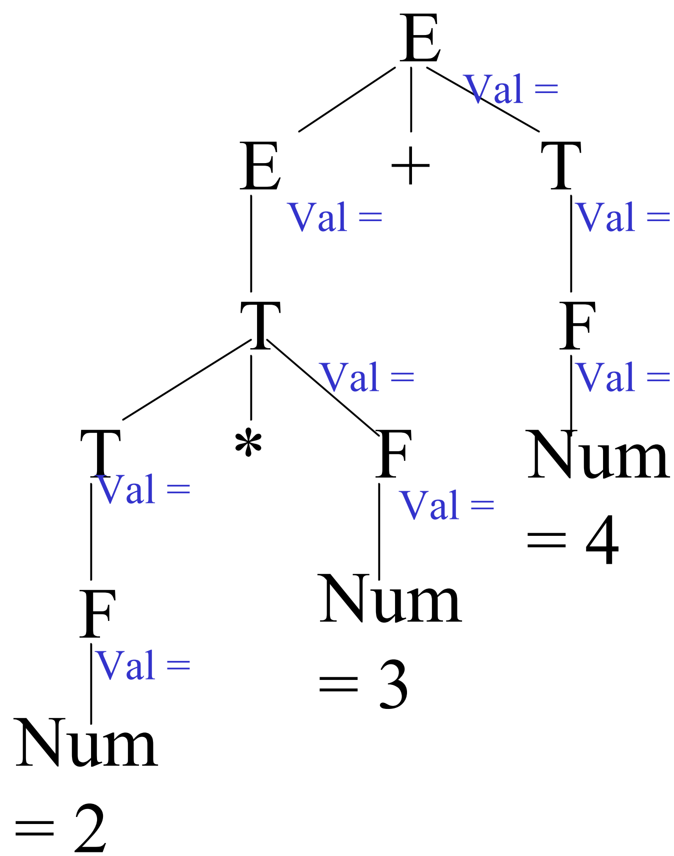
Synthesized Attributes – Expression Grammar

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

Synthesized Attributes –Annotating the parse tree

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

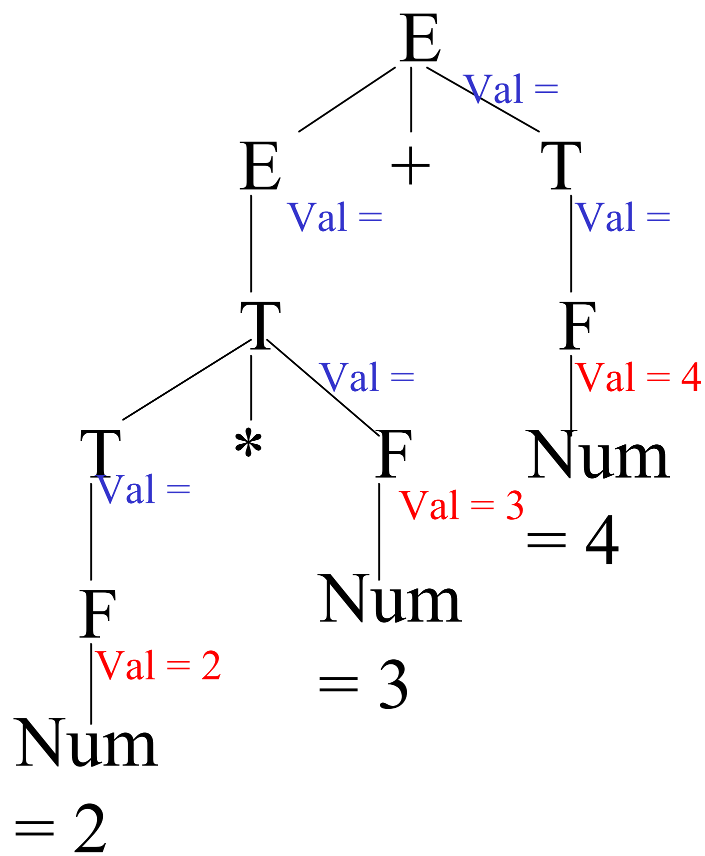
Input: 2 * 3 + 4



Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
E → 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

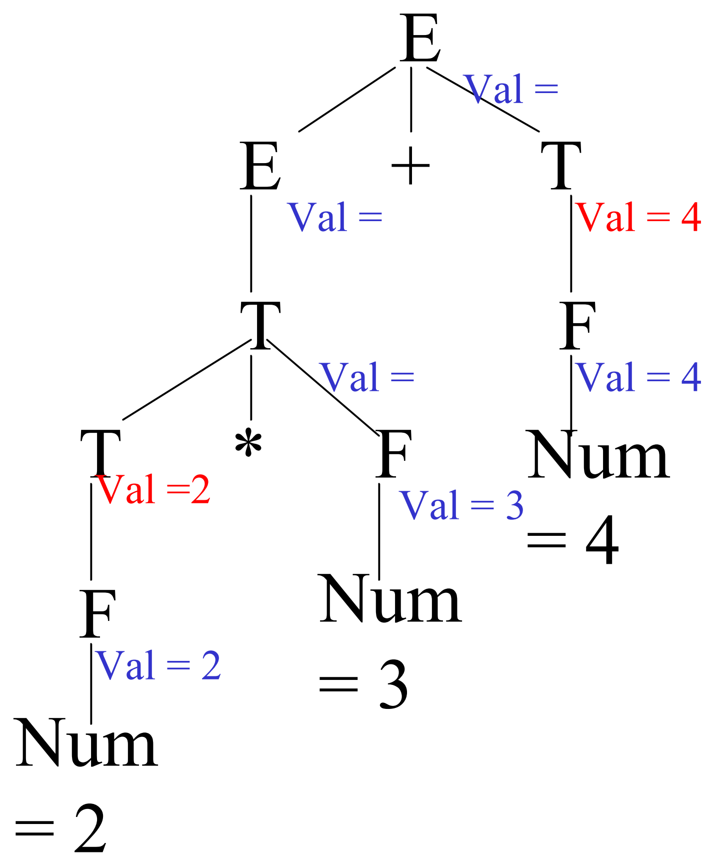
Input: 2 * 3 + 4



Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
E → 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

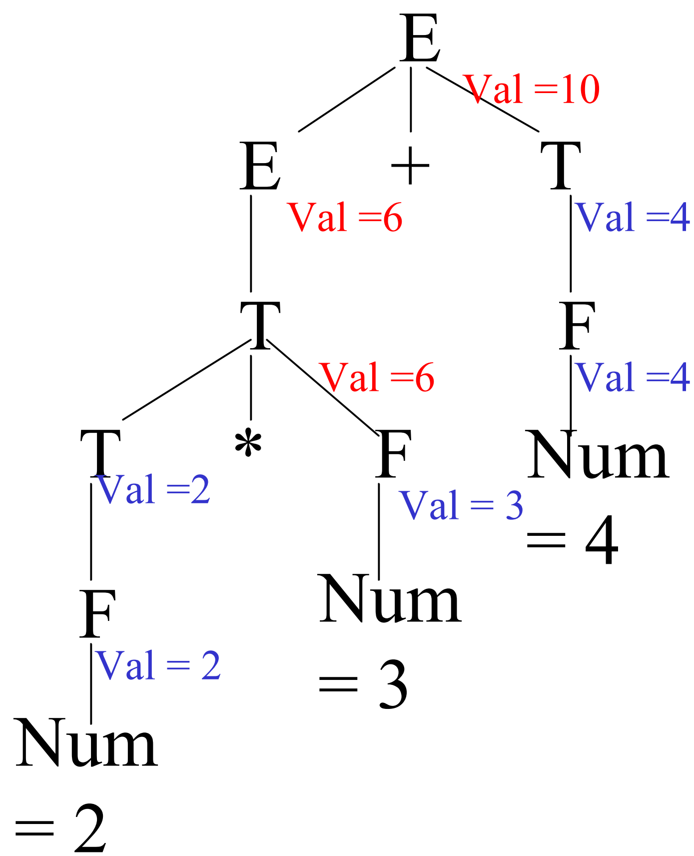
Input: 2 * 3 + 4



Synthesized Attributes –Annotating the parse tree

Production	**Semantic Actions**
E → 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

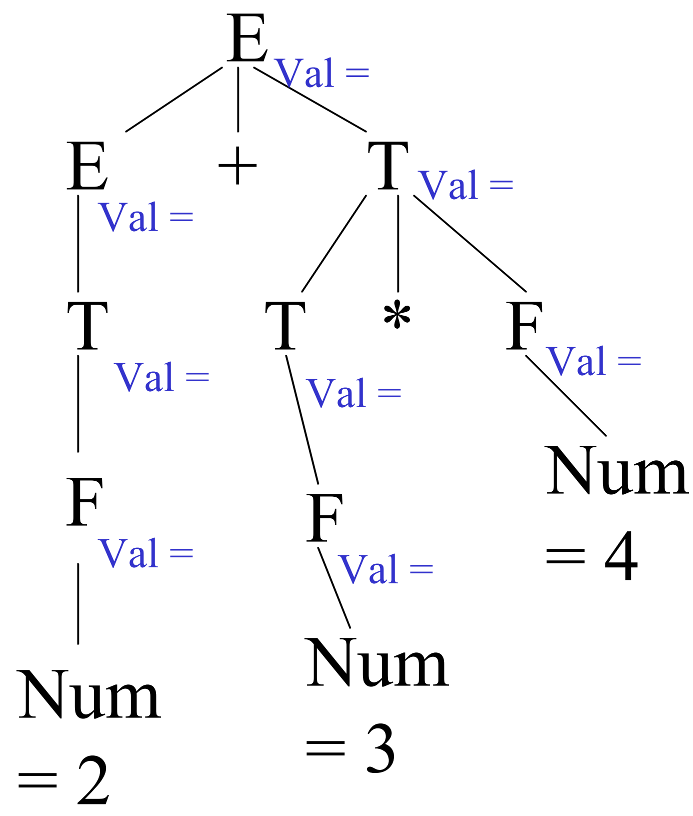
Input: 2 * 3 + 4



Synthesized Attributes –Annotating the parse tree

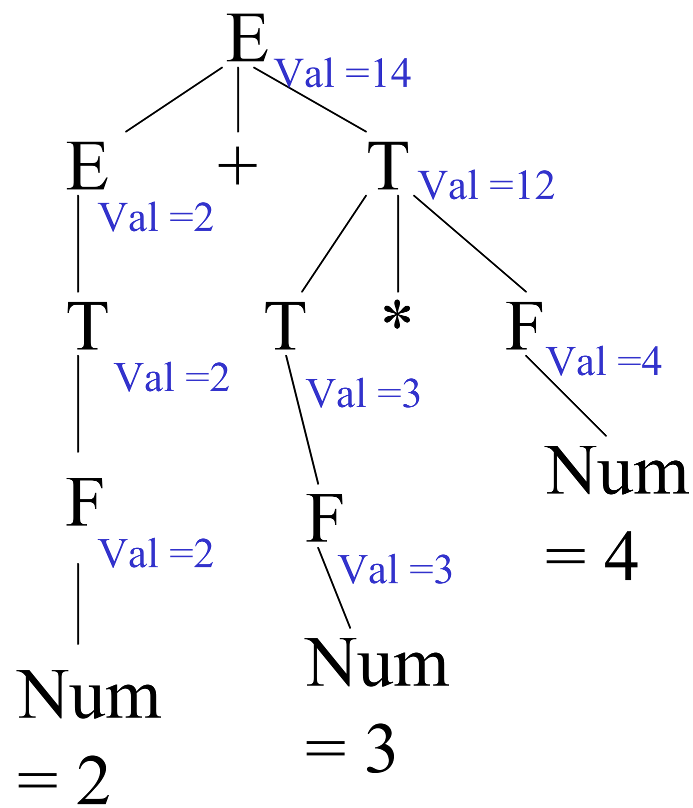
Production	Semantic Actions
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val

Input: 2 + 4 * 3



Synthesized Attributes –Annotating the parse tree

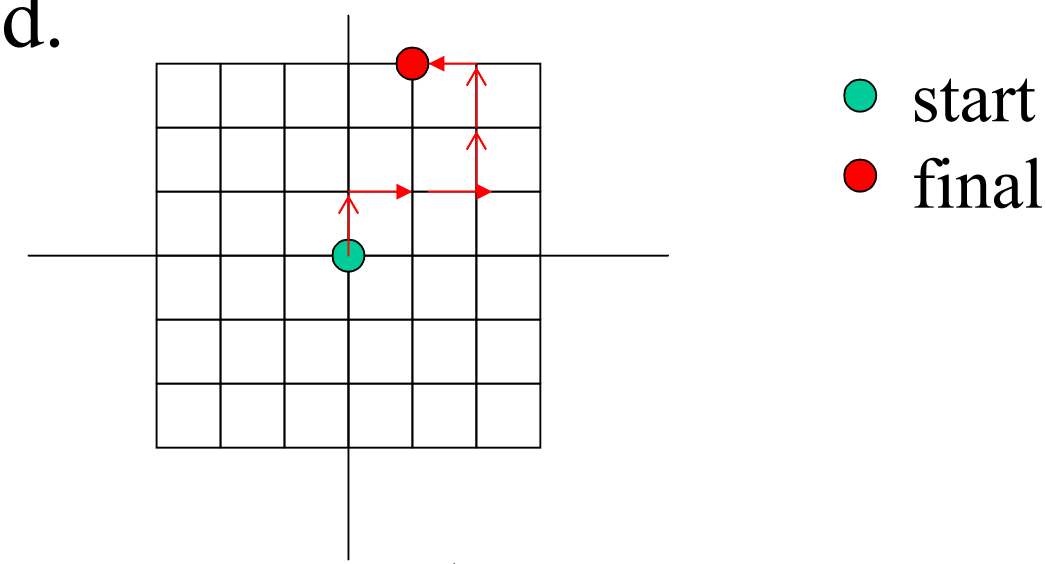
Production	Semantic Actions
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val



Input: 2 + 4 * 3

Grid Example

- Given a starting location of 0,0 and a sequence of north, south, east, west moves (ex: NEENNW), find the final position on a unit grid.



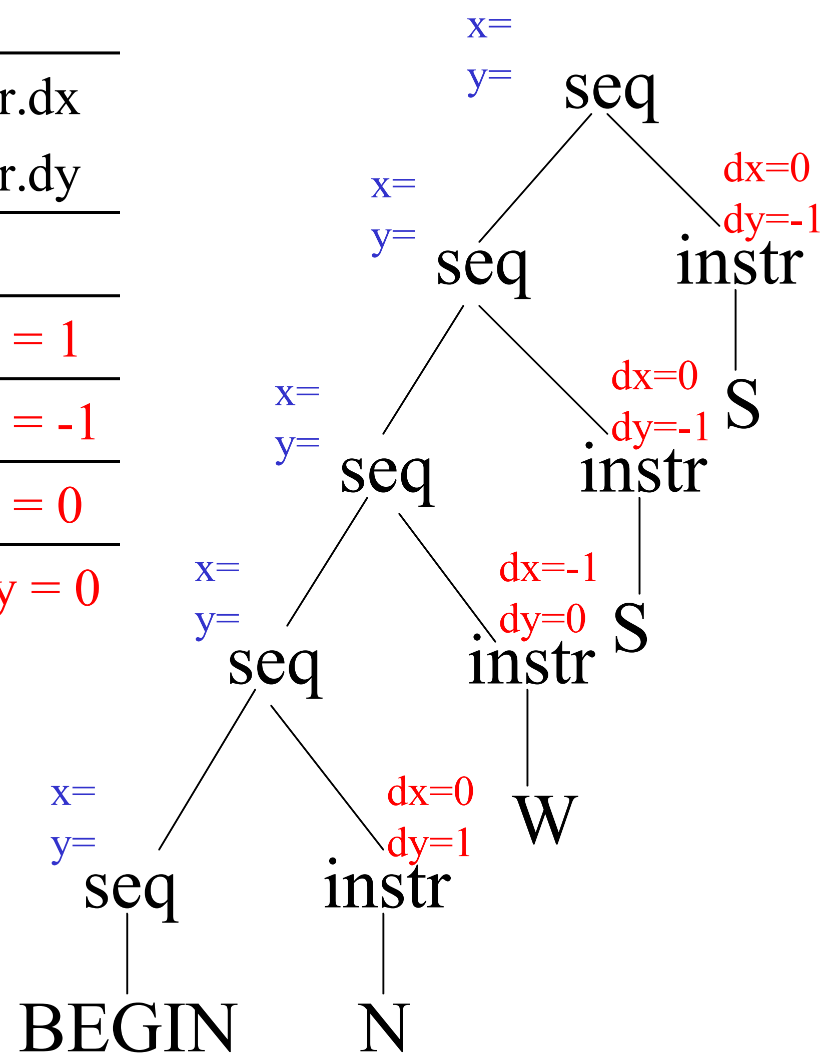
Synthesized Attributes – Grid Positions

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 → NORTH	instr.dx = 0, instr.dy = 1
instr → SOUTH	instr.dx = 0, instr.dy = -1
instr → EAST	instr.dx = 1, instr.dy = 0
instr → WEST	instr.dx = -1, instr.dy = 0

Synthesized Attributes –Annotating the parse tree

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 → NORTH	instr.dx = 0, instr.dy = 1
instr → SOUTH	instr.dx = 0, instr.dy = -1
instr → EAST	instr.dx = 1, instr.dy = 0
instr → WEST	instr.dx = -1, instr.dy = 0

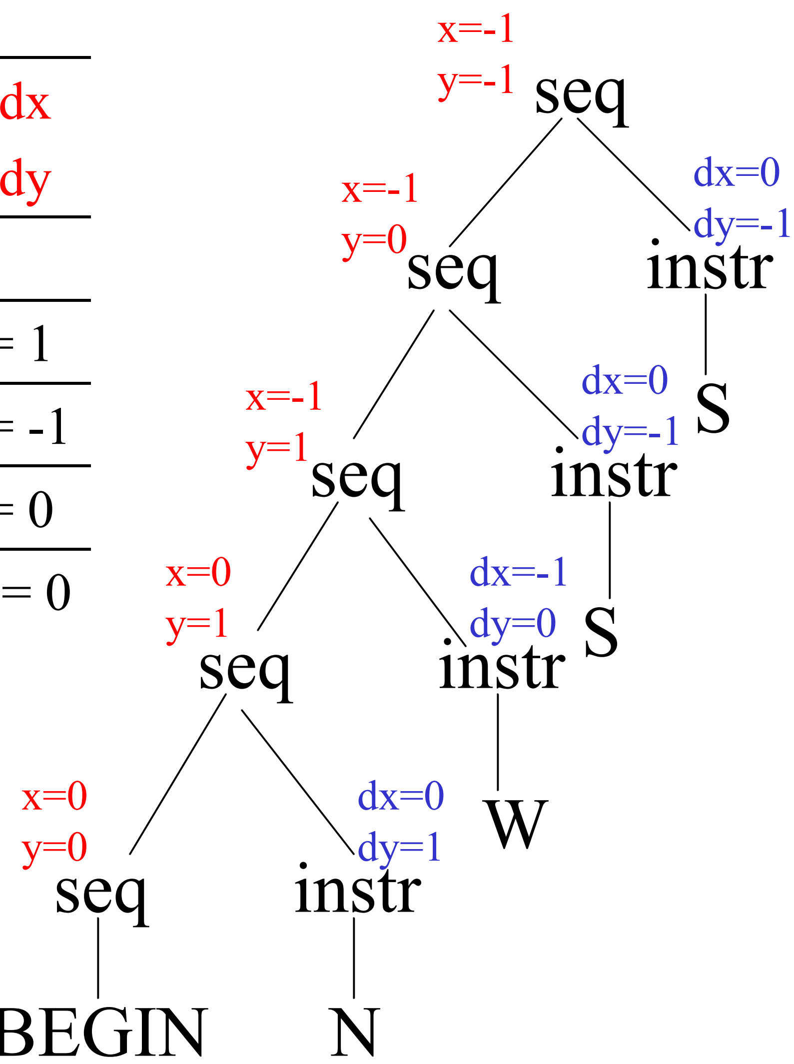
Input: BEGIN N W S S



Synthesized Attributes –Annotating the parse tree

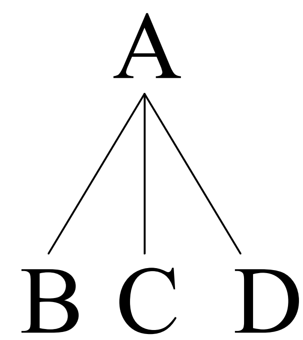
Production	**Semantic Actions**
seq → seq_1 instr	seq.x = seq_1.x + instr.dx seq.y = seq_1.y + instr.dy
seq → BEGIN	seq.x = 0, seq.y = 0
instr → NORTH	instr.dx = 0, instr.dy = 1
instr → SOUTH	instr.dx = 0, instr.dy = -1
instr → EAST	instr.dx = 1, instr.dy = 0
instr → WEST	instr.dx = -1, instr.dy = 0

Input: BEGIN N W S S



Inherited Attributes

Inherited attributes – if an attribute is not synthesized, it is inherited.



Example:

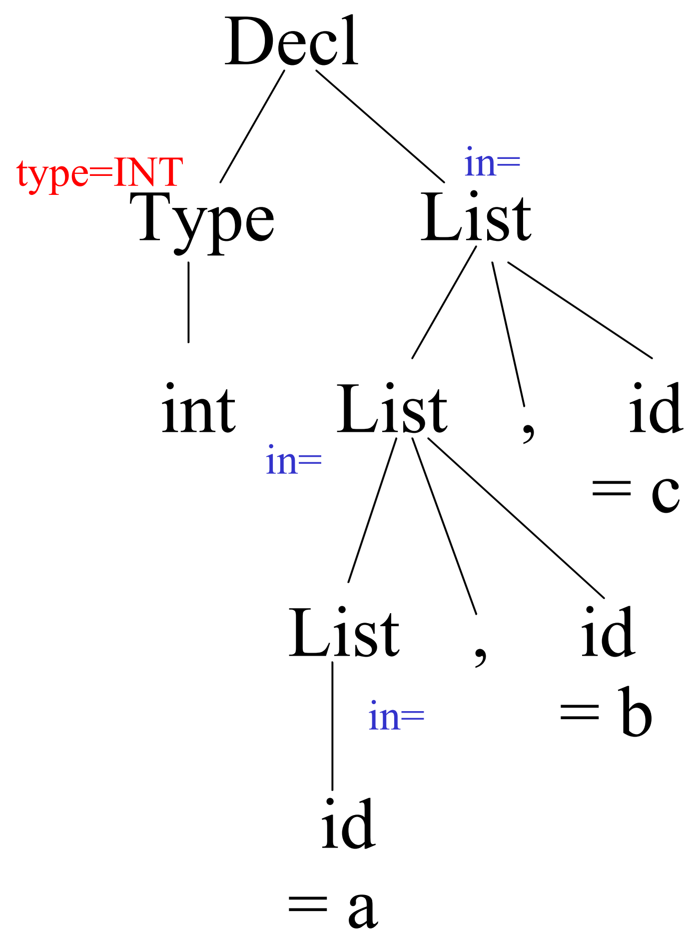
Production	**Semantic Rules**
A → B C D	B.b := A.a + C.b

Inherited Attributes – Determining types

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)

Inherited Attributes – Example

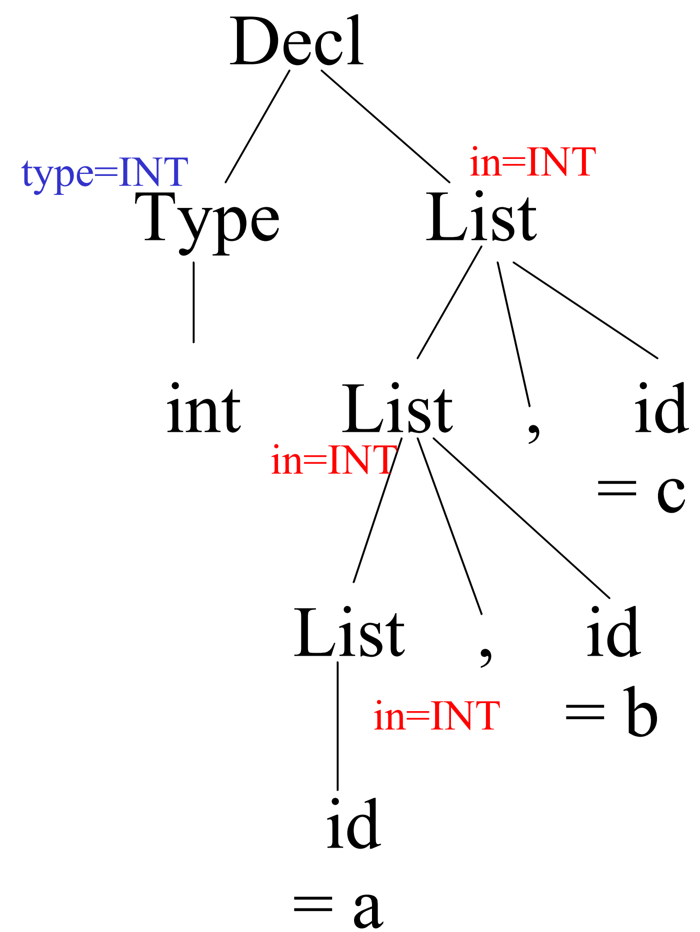
Productions	**Semantic Actions**
Decl → Type List	List.in = Type.type
Type → int	Type.type = INT
Type → real	T.type = REAL
List → $List_1$, id	$List_1$.in = List.in, addtype(id.entry.List.in)
List → id	addtype(id.entry,List.in)



Input: int a,b,c

Inherited Attributes – Example

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)



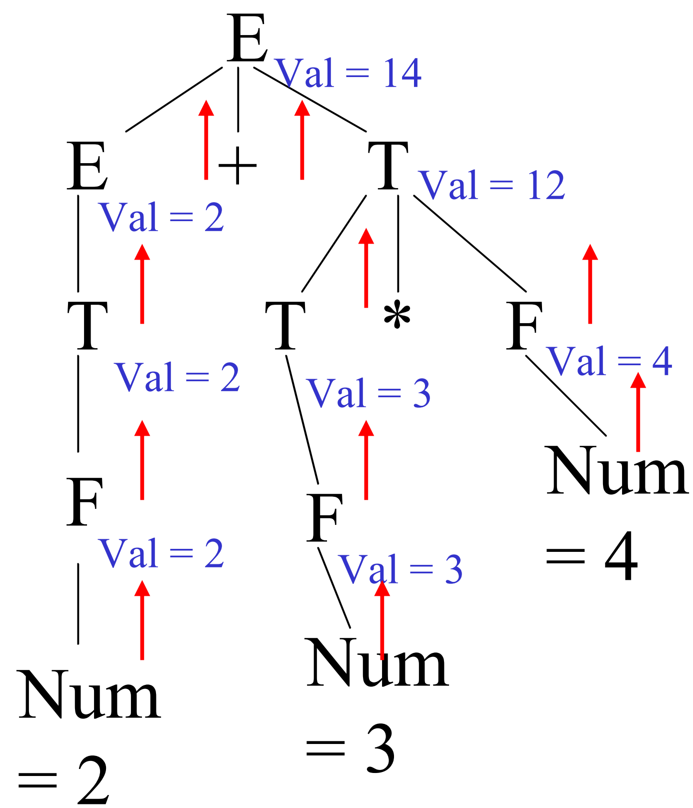
Input: int a,b,c

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.
- Dependencies matter when considering syntax directed translation in the context of a parsing technique.

Attribute Dependencies

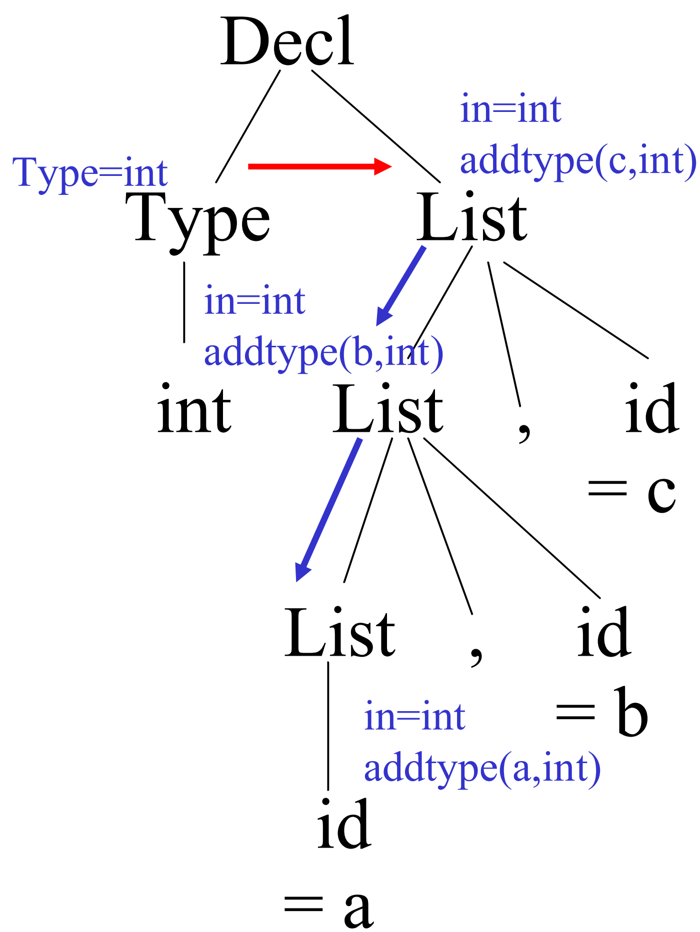
Production	Semantic Actions
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val



Synthesized attributes – dependencies always up the tree

Attribute Dependencies

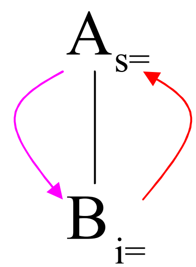
Productions	**Semantic Actions**
Decl → Type List	List.in = Type.type
Type → int	Type.type = INT
Type → real	T.type = REAL
List → $List_1$, id	$List_1$.in = List.in, addtype(id.entry.List.in)
List → id	addtype(id.entry,List.in)



Attribute Dependencies

Circular dependences are a problem

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



Synthesized Attributes and LR Parsing

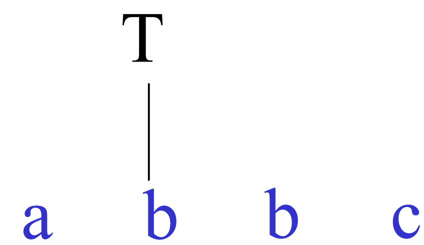
Synthesized attributes have natural fit with LR parsing

- Attribute values can be stored on stack with their associated symbol
- When reducing by production A → α, both α and the value of α's attributes will be on the top of the LR parse stack!

Synthesized Attributes and LR Parsing

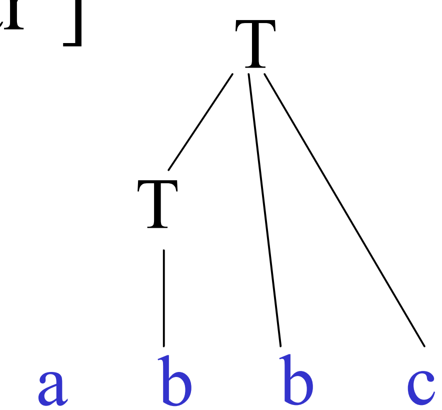
Example Stack:

$0[attr],a1[attr],T2[attr],b5[attr],c8[attr]



Stack after T → T b c:

$0[attr],a1[attr],T2[attr']



Other SDD types

L-Attributed definition – edges can go from left to right, but not right to left. Every attribute must be:

- Synthesized or
- Inherited (but limited to ensure the left to right property).

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.

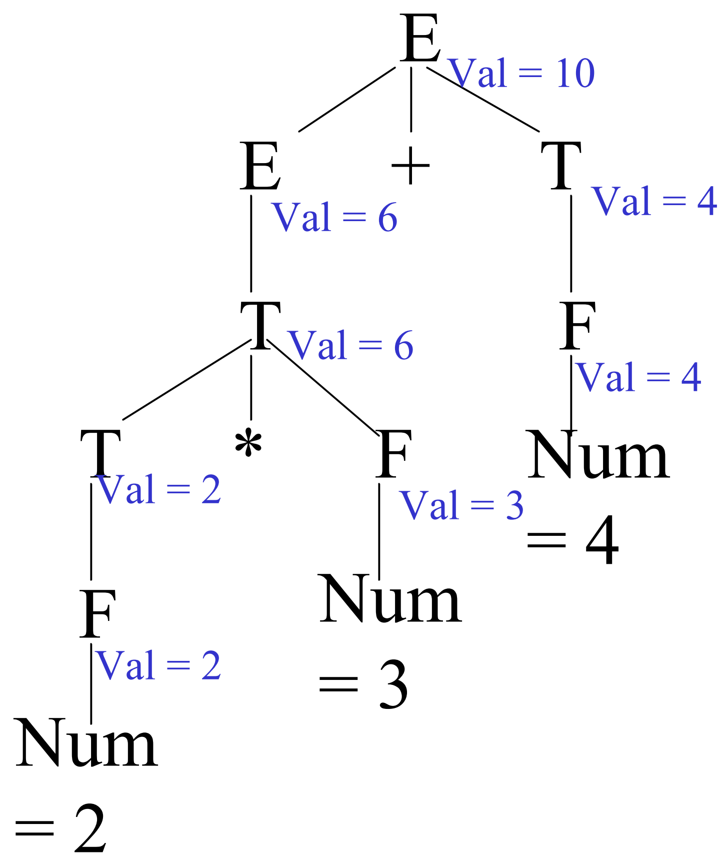
  For A : B C D,

  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
%%
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
         ;
%%
```

Expression Grammar in YACC

```
%token NUMBER CR
%%
lines     :  lines line
          |  line
          ;

line      :   expr   CR              {System.out.println($1.ival); }
          ;

expr      :   expr '+' term          {$$ = new ParserVal($1.ival + $3.ival); }
          |   term
          ;

term      :    term '*' factor        {$$ = new ParserVal($1.ival * $3.ival);
          |    factor
          ;
factor    :    '(' expr ')'          {$$ = new ParserVal($2.ival); }
          |    NUMBER
          ;
%%
```

Associated Lex Specification

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

In Java:

```
      yyparser.yylval =
          new ParserVal(Integer.parseInt(yytext()));
      return Parser.INT;
```

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

- Actions can be embedded in productions. This changes the numbering ($1,$2,…)
- Embedding actions in productions 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  {action 2}  ;
```

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

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;
   …
}
```

- 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 pt.

```
%union{  double f_value; }
%token <f_value> NUMBER
%type <f_value> expr term factor
%%
expr        :   expr '+' term          { $$ = $1 + $3; }
                |   term
                ;
term        :    term '*' factor       { $$ = $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);}
%%
```

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
           |    S              {$$.dx = 0; $$.dy = -1;} …  ;
```

missing action

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

Java's ParserVal class

```
public class ParserVal
  {
    public int ival;
    public double dval;
    public String sval;
    public Object obj;
    public ParserVal(int val)
         {  ival=val; }
    public ParserVal(double val)
         {  dval=val; }
    public ParserVal(String val)
         { sval=val; }
    public ParserVal(Object val)
         { obj=val; }
}
```

If ParserVal won't work…

Can define and use your own Semantic classes:

```
/home/u1/white/byacc -Jsemantic=Semantic gen.y
```

```
%%
grid : seq          {System.out.println("Done: "
                       + $1.ival1 + " " + $1.ival2);}
   ;
seq : seq instr    {$$.ival1 = $1.ival1 + $2.ival1;
                    $$.ival2 = $1.ival2 + $2.ival2;}
   |   BEGIN
   ;
instr : N | S | E | W ;
%%
public static final class Semantic {
   public int ival1 ;
   public int ival2 ;
   public Semantic(Semantic sem) {
       ival1 = sem.ival1; ival2 = sem.ival2; }
public Semantic(int i1,int i2) {
       ival1 = i1; ival2 = i2; }
public Semantic() { ival1=0;ival2=0;} }
```

Grid Example (Java)

/home/u1/white/byacc -Jsemantic=Semantic gen.y

Grid Example (Java)

```
%%
B       {yyparser.yylval = new Parser.Semantic(0,0);
         return Parser.BEGIN;}
N       {yyparser.yylval = new Parser.Semantic(0,1);
         return Parser.N;}
S       {yyparser.yylval = new Parser.Semantic(0,-1);
         return Parser.S;}
E       {yyparser.yylval = new Parser.Semantic(1,0);
         return Parser.E;}
W       {yyparser.yylval = new Parser.Semantic(-1,0);
         return Parser.W;}
[ \t\n] {;}
%%
```