Lecture 4d: YACC and Syntax Directed Translation

COSC 4316

Last Revised 3/21/2017

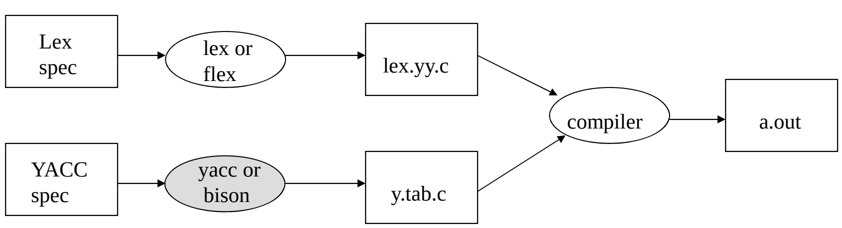
(grateful acknowledgement to Robert van Engelen and Elizabeth White for some of the material from which these slides have been adapted)

Part 1: Introduction to YACC

ANTLR, Yacc, and Bison

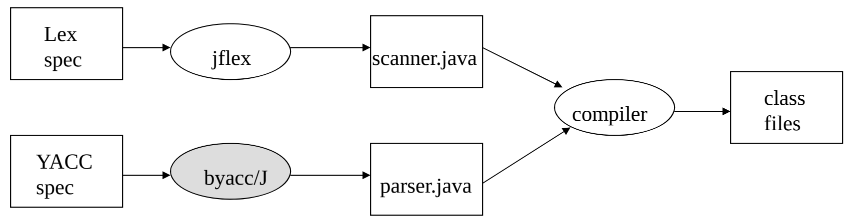
- *ANTLR* tool (ANother Tool For Language Recognition)
  - Generates LL(k) parsers
  - Developed by Terrance Parr at Univ. of San Francisco (1992)
- *Yacc* (Yet Another Compiler Compiler)
  - Generates LALR(1) parsers
  - Developed by Stephen Johnson at Bell Laboratories (1972)
- *Bison*
  - Improved version of **yacc**
  - Developed by Robert Corbett, of the GNU Project (1988)

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 Specification

- A *yacc specification* consists of three parts:

  yacc declarations, and C declarations within **%{ %}**
  %%
  translation rules
  %%
  user-defined auxiliary procedures
- The *translation rules* are productions with actions:

  $production_1$ { $semantic\ action_1$ }
  $production_2$ { $semantic\ action_2$ }
  …
  $production_n$ { $semantic\ action_n$ }

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

```
        |    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.

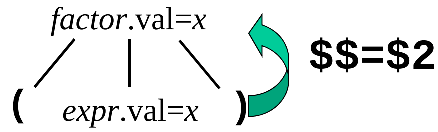
Synthesized Attributes

- Semantic actions may refer to values of the *synthesized attributes* of terminals and nonterminals in a production:

  $X : Y_1\ Y_2\ Y_3 \ldots Y_n \quad \{\ action\ \}$

  - **`$$`** refers to the value of the attribute of X
  - **`$`***i* refers to the value of the attribute of Y_i
- For example

  `factor : '(' expr ')' { $$=$2; }`



Writing a Grammar in Yacc

- Productions in Yacc are of the form

  Nonterminal **:** tokens/nonterminals { *action* }

  | tokens/nonterminals { *action* }

  …

  ;
- Tokens that are single characters can be used directly within productions, e.g. **`'+'`**
- Named tokens must be declared first in the declaration part using

  `%token` *TokenName*

Example 1

```
%{ #include <ctype.h> %}
%token DIGIT
%%
line    : expr '\n'           { printf("%d\n", $1); }
        ;
expr    : expr '+' term       { $$ = $1 + $3; }
        | term                { $$ = $1; }
        ;
term    : term '*' factor     { $$ = $1 * $3; }
        | factor              { $$ = $1; }
        ;
factor  : '(' expr ')'        { $$ = $2; }
        | DIGIT               { $$ = $1; }
        ;
%%
int yylex()
{ int c = getchar();
  if (isdigit(c))
  { yylval = c-'0';
    return DIGIT;
  }
  return c;
}
```

Also results in definition of
`#define DIGIT xxx`

Attribute of **`term`** (parent)

Attribute of **`factor`** (child)

Attribute of token (stored in **`yylval`**)

Example of a very crude lexical analyzer invoked by the parser

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`**

Note: **`yacc`** generates **`y.tab.c`** by default, **`bison`** generates ***filename*`.tab.c`** by default.

`Bison` works more like **`yacc`** when invoked with the **`–y`** option

- 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
  ```

Creates the **`y.tab.h`** *file*

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**)

COSC 4316 Timothy J. McGuire

Using yacc with an associated Lex Specification

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

COSC 4316 Timothy J. McGuire

Associated Flex specification

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


byacc/J 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`**.
- By defining operator precedence levels and left/right associativity of the operators, we can specify ambiguous grammars in **`yacc`**, such as $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid -E \mid$ <u>num</u>
- 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
…
%%
…
expr : expr PLUS expr
           | expr MINUS expr
…
```

PLUS and MINUS have lowest priority and are left associative
MULT and DIV have higher priority and are left associative
Unary MINUS has highest priority and is non-associative

Part 2: Syntax Directed Translation

Syntax-Directed Definitions

- A *syntax-directed definition* (or *attribute grammar*) binds a set of *semantic rules* to productions
- Terminals and nonterminals have *attributes* holding values set by the semantic rules
- A *depth-first traversal* algorithm traverses the parse tree thereby executing semantic rules to assign attribute values
- After the traversal is complete the attributes contain the translated form of the input

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

Attributes

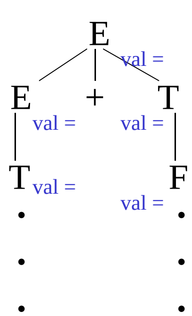
- Attribute values may represent
  - Numbers (literal constants)
  - Strings (literal constants)
  - Memory locations, such as a frame index of a local variable or function argument
  - A data type for type checking of expressions
  - Scoping information for local declarations
  - Intermediate program representations

Annotating a Parse Tree With Depth-First Traversals

procedure *visit*(*n* : *node*);
begin
  for each child *m* of *n*, from left to right **do**
    visit(*m*);
  evaluate semantic rules at node *n*
end

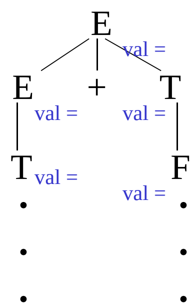
Example Attribute Grammar

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

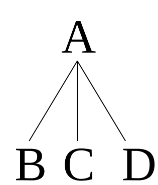


Production	Semantic Actions
E → E_1 + T	$\boxed{\text{E.val = } E_1\text{.val + T.val}}$
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val

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).

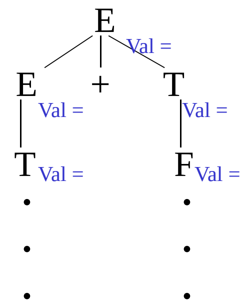


Example:

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

Synthesized Attributes –Annotating the parse tree

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



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

Yacc/Bison only support S-attributed definitions

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

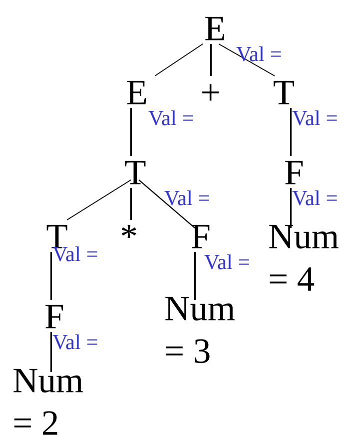
Production	**Semantic Actions**
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val

COSC 4316 Timothy J. McGuire

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

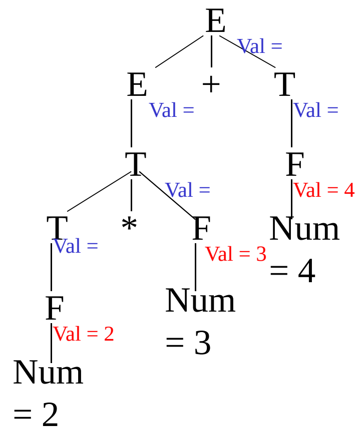


COSC 4316 Timothy J. McGuire

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

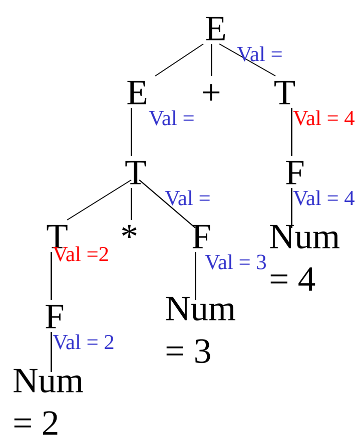


COSC 4316 Timothy J. McGuire

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

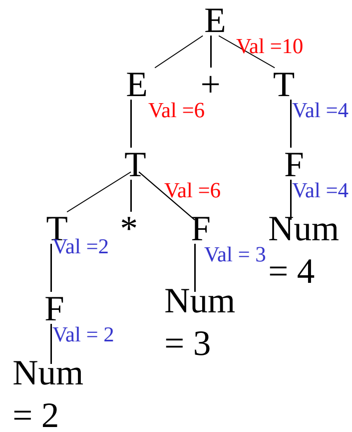
Input: 2 * 3 + 4



COSC 4316 Timothy J. McGuire

Synthesized Attributes –Annotating the parse tree

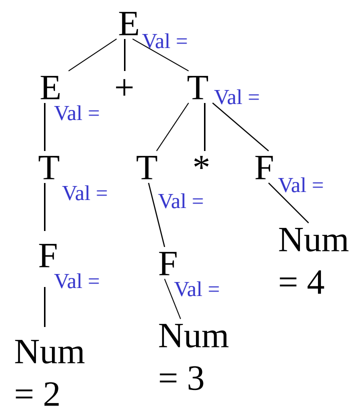
Production	Semantic Actions
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val



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

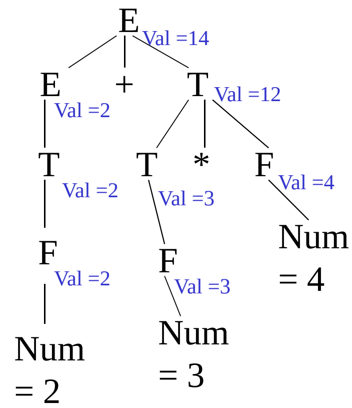


Input: 2 + 4 * 3

Synthesized Attributes –Annotating the parse tree

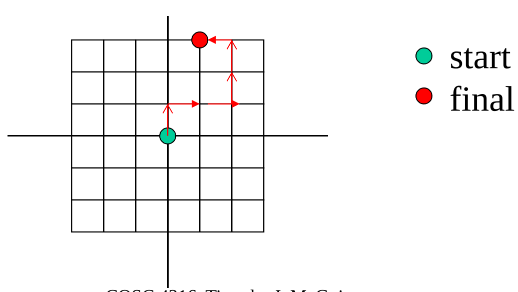
Production	**Semantic Actions**
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val

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.



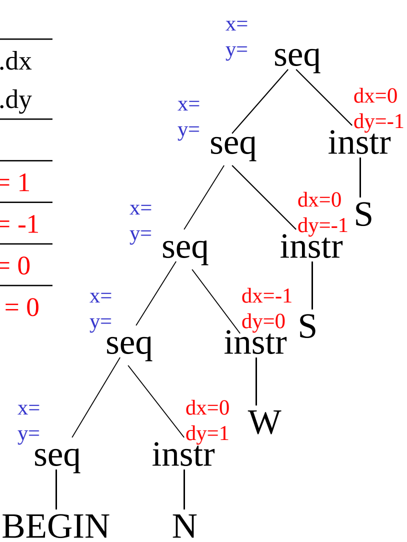
Synthesized Attributes – Grid Positions

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

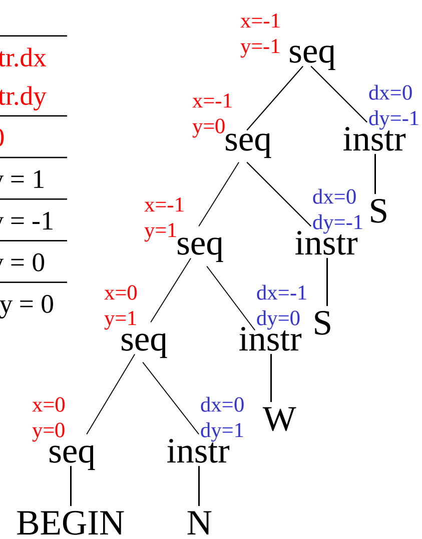
Input: BEGIN N W S S



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

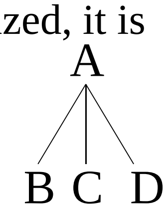
Input: BEGIN N W S S



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

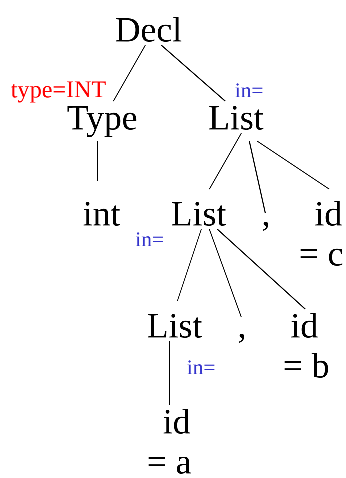
B's attribute is inherited from its parent (and sibling)

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

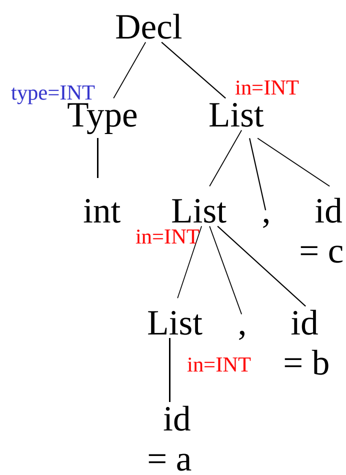
Productions	**Semantic Actions**
Decl → Type List	List.in = Type.type
Type → int	Type.type = INT
Type → real	T.type = REAL
List → $List_1$, id	$List_1$.in = List.in, addtype(id.entry.List.in)
List → id	addtype(id.entry,List.in)



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)



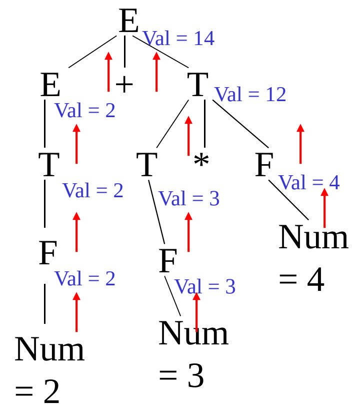
Input: int a,b,c

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

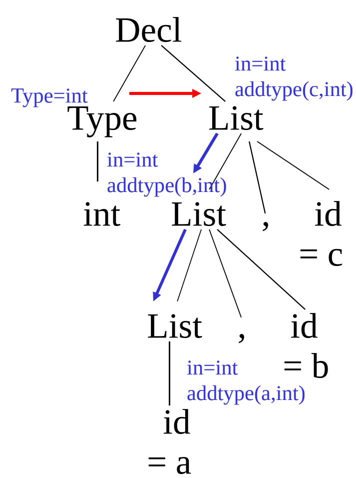
Production	Semantic Actions
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → num	F.val = value(num)
F → (E)	F.val = E.val



Synthesized attributes –
dependencies are always up the tree

Attribute Dependencies

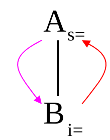
Productions	Semantic Actions
Decl → Type List	List.in = Type.type
Type → int	Type.type = INT
Type → real	T.type = REAL
List → $List_1$, id	$List_1$.in = List.in, addtype(id.entry.List.in)
List → id	addtype(id.entry,List.in)



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

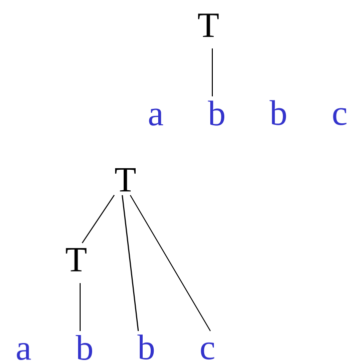
Synthesized attributes have natural fit with LR parsing

- Attribute values can be stored on stack with their associated symbol
- When reducing by production $A \rightarrow \alpha$, 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']



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;*

Example 1 Revisited

```
%{ #include <ctype.h> %}
%token DIGIT
%%
line    : expr '\n'           { printf("%d\n", $1); }
        ;
expr    : expr '+' term       { $$ = $1 + $3; }
        | term                { $$ = $1; }
        ;
term    : term '*' factor     { $$ = $1 * $3; }
        | factor              { $$ = $1; }
        ;
factor  : '(' expr ')'        { $$ = $2; }
        | DIGIT               { $$ = $1; }
        ;
%%
int yylex()
{ int c = getchar();
  if (isdigit(c))
  { yylval = c-'0';
    return DIGIT;
  }
  return c;
}
```

Also results in definition of **`#define DIGIT xxx`**

Attribute of **`factor`** (child)

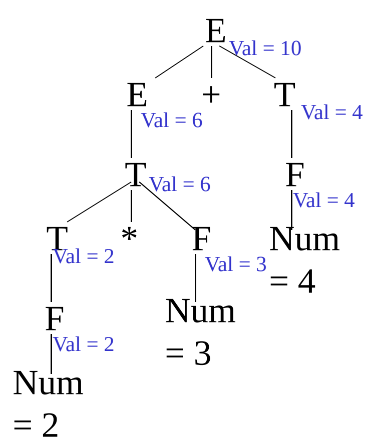
Attribute of **`term`** (parent)

Attribute of token (stored in **`yylval`**)

Example of a very crude lexical analyzer invoked by the parser

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
         ;
%%
```

COSC 4316 Timothy J. McGuire

Expression Grammar in BYACC/J

```
%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
         ;
%%
```

COSC 4316 Timothy J. McGuire

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.

Example 2

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines   : lines expr '\n'        { printf("%g\n", $2); }
        | lines '\n'
        | /* empty */
        ;
expr    : expr '+' expr          { $$ = $1 + $3; }
        | expr '-' expr          { $$ = $1 - $3; }
        | expr '*' expr          { $$ = $1 * $3; }
        | expr '/' expr          { $$ = $1 / $3; }
        | '(' expr ')'           { $$ = $2; }
        | '-' expr %prec UMINUS { $$ = -$2; }
        | NUMBER
        ;
%%
```

Double type for attributes and **yylval**

Example 2 (cont'd)

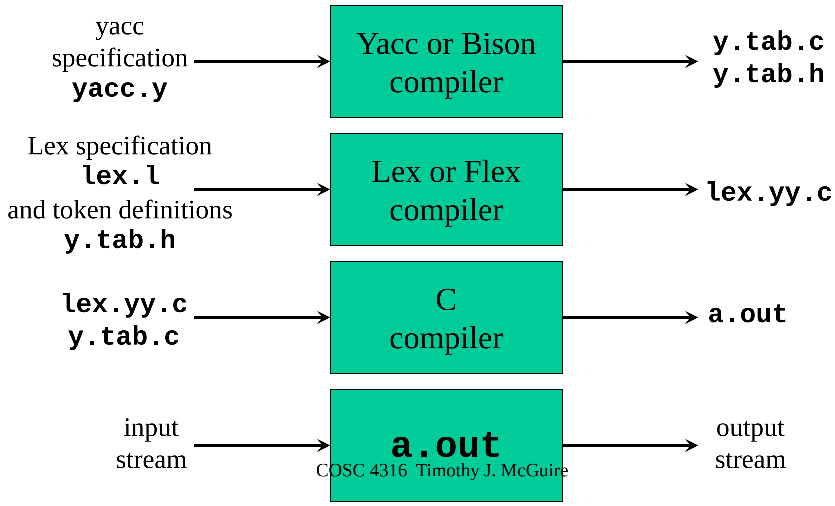
```
%%
int yylex()
{ int c;
  while ((c = getchar()) == ' ')
    ;
  if ((c == '.') || isdigit(c))
  { ungetc(c, stdin);
    scanf("%lf", &yylval);
    return NUMBER;
  }
  return c;
}
int main()
{ if (yyparse() != 0)
    fprintf(stderr, "Abnormal exit\n");
  return 0;
}
int yyerror(char *s)
{ fprintf(stderr, "Error: %s\n", s);
}
```

Crude lexical analyzer for fp doubles and arithmetic operators

Run the parser

Invoked by parser to report parse errors

Combining Lex/Flex with Yacc/Bison



Lex Specification for Example 2

```
%option noyywrap
%{
#include "y.tab.h"
extern double yylval;
%}
number  [0-9]+\.?|[0-9]*\.[0-9]+
%%
[ ]           { /* skip blanks */ }
{number}      { sscanf(yytext, "%lf", &yylval);
                return NUMBER;
              }
\n|.          { return yytext[0]; }
```

Generated by Yacc, contains **`#define NUMBER xxx`**

Defined in **`y.tab.c`**

```
yacc -d example2.y
lex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

```
bison -d -y example2.y
flex example2.l
gcc y.tab.c lex.yy.c
./a.out
```

Error Recovery in Yacc

```
%{
…
%}
…
%%
lines   : lines expr '\n'      { printf("%g\n", $2; }
        | lines '\n'
        | /* empty */
        | error '\n'           { yyerror("reenter last line: ");
                                 yyerrok;
                               }
        ;
…
```

Error production: set error mode and skip input until newline

Reset parser to normal mode

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;
   …
 }
```

This is more generally applicable than redefining YYSTYPE

- 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);}
%%
```


Another Example Attribute Grammar in Yacc

```
%token DIGIT
%%
L : E '\n'         { printf("%d\n", $1); }
  ;
E : E '+' T        { $$ = $1 + $3; }
  | T              { $$ = $1; }
  ;
T : T '*' F        { $$ = $1 * $3; }
  | F              { $$ = $1; }
  ;
F : '(' E ')'      { $$ = $2; }
  | DIGIT          { $$ = $1; }
  ;
%%
```

Synthesized attribute of parent node **F**

Bottom-up Evaluation of S-Attributed Definitions in Yacc

Stack	val	Input	Action	Semantic Rule
$	_	**3*5+4n$**	shift	
$ 3	*3*	***5+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *F*	*3*	***5+4n$**	reduce $T \rightarrow F$	$$ = $1
$ *T*	*3*	***5+4n$**	shift	
$ *T* *****	*3 _*	**5+4n$**	shift	
$ *T* *** 5**	*3 _ 5*	**+4n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *T* ***** *F*	*3 _ 5*	**+4n$**	reduce $T \rightarrow T$ ***** F	$$ = $1 * $3
$ *T*	*15*	**+4n$**	reduce $E \rightarrow T$	$$ = $1
$ *E*	*15*	**+4n$**	shift	
$ *E* **+**	*15 _*	**4n$**	shift	
$ *E* **+ 4**	*15 _ 4*	**n$**	reduce $F \rightarrow$ **digit**	$$ = $1
$ *E* **+** *F*	*15 _ 4*	**n$**	reduce $T \rightarrow F$	$$ = $1
$ *E* **+** *T*	*15 _ 4*	**n$**	reduce $E \rightarrow E + T$	$$ = $1 + $3
$ *E*	*19*	**n$**	shift	
$ *E* **n**	*19 _*	**$**	reduce $L \rightarrow E$ **n**	*print* $1
$ *L*	*19*	**$**	accept	

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

Using Translation Schemes for L-Attributed Definitions

Production	Semantic Rule
$D \rightarrow T\ L$	L.in := T.type
$T \rightarrow$ **int**	T.type := ‘integer’
$T \rightarrow$ **real**	T.type := ‘real’
$L \rightarrow L_1$ **, id**	L1.in := L.in; *addtype*(**id**.entry, L.in)
$L \rightarrow$ **id**	*addtype*(**id**.entry, L.in)

Translation Scheme

$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ **int** { T.type := ‘integer’ }
$T \rightarrow$ **real** { T.type := ‘real’ }
$L \rightarrow$ { L1.in := L.in } L1 **, id** { *addtype*(**id**.entry, L.in) }
$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

Implementing L-Attributed Definitions in Top-Down Parsers

Attributes in L-attributed definitions implemented in translation schemes are passed as arguments to procedures (synthesized) or returned (inherited)

$D \to T$ { L.in := T.type } L
$T \to$ **int** { T.type := 'integer' }
$T \to$ **real** { T.type := 'real' }

```
void D()
{ Type Ttype = T();
  Type Lin = Ttype;
  L(Lin);
}
Type T()
{ Type Ttype;
  if (lookahead == INT)
  { Ttype = TYPE_INT;
    match(INT);
  } else if (lookahead == REAL)
  { Ttype = TYPE_REAL;
    match(REAL);
  } else error();
  return Ttype;
}
void L(Type Lin)
{ … }
```

Output: synthesized attribute

Input: inherited attribute

Implementing L-Attributed Definitions in Bottom-Up Parsers

- More difficult and also requires rewriting L-attributed definitions into translation schemes
- Insert marker nonterminals to remove embedded actions from translation schemes, that is
  $A \to X$ { actions } Y
  is rewritten with marker nonterminal N into
  $A \to X\ N\ Y$
  $N \to \varepsilon$ { actions }
- Problem: inserting a marker nonterminal may introduce a conflict in the parse table

Emulating the Evaluation of L-Attributed Definitions in Yacc

$D \rightarrow T$ { L.in := T.type } L
$T \rightarrow$ **int** { T.type := ‘integer’ }
$T \rightarrow$ **real** { T.type := ‘real’ }
$L \rightarrow$ { L_1.in := L.in } L_1 **, id**
    { *addtype*(**id**.entry, L.in) }
$L \rightarrow$ **id** { *addtype*(**id**.entry, L.in) }

```
%{
Type Lin; /* global variable */
%}
%%
D   : Ts L
    ;
Ts  : T        { Lin = $1; }
    ;
T   : INT      { $$ = TYPE_INT; }
    | REAL     { $$ = TYPE_REAL; }
    ;
L   : L ‘,’ ID { addtype($3, Lin);}
    | ID       { addtype($1, Lin);}
    ;
%%
```


Rewriting a Grammar to Avoid Inherited Attributes

Production	Production	Semantic Rule
$D \rightarrow L$ **:** T	$D \rightarrow$ **id** L	*addtype*(**id**.entry, L.type)
$T \rightarrow$ **int**	$T \rightarrow$ **int**	T.type := ‘integer’
$T \rightarrow$ **real**	$T \rightarrow$ **real**	T.type := ‘real’
$L \rightarrow L_1$ **, id**	$L \rightarrow$ **, id** L_1	*addtype*(**id**.entry, L.type)
$L \rightarrow$ **id**	$L \rightarrow$ **:** T	L.type := T.type