Chapter 2: Syntax Directed Translation and YACC

長庚大學資訊工程學系　陳仁暉　助理教授
Tel: (03) 211-8800 Ext: 5990
Email: jhchen@mail.cgu.edu.tw
URL: http://www.csie.cgu.edu.tw/~jhchen

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

Attribute Grammars

- Associate ***attributes*** with tree nodes (internal and leaf).
- Rules 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)

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

Each node has single integer attribute ‘val’

Synthesized Attributes

Synthesized attributes – the value of a synthesized attribute for a node is computed using only information associated with 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

Example Problems for Synthesized

- Expression grammar – given a valid expression (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.

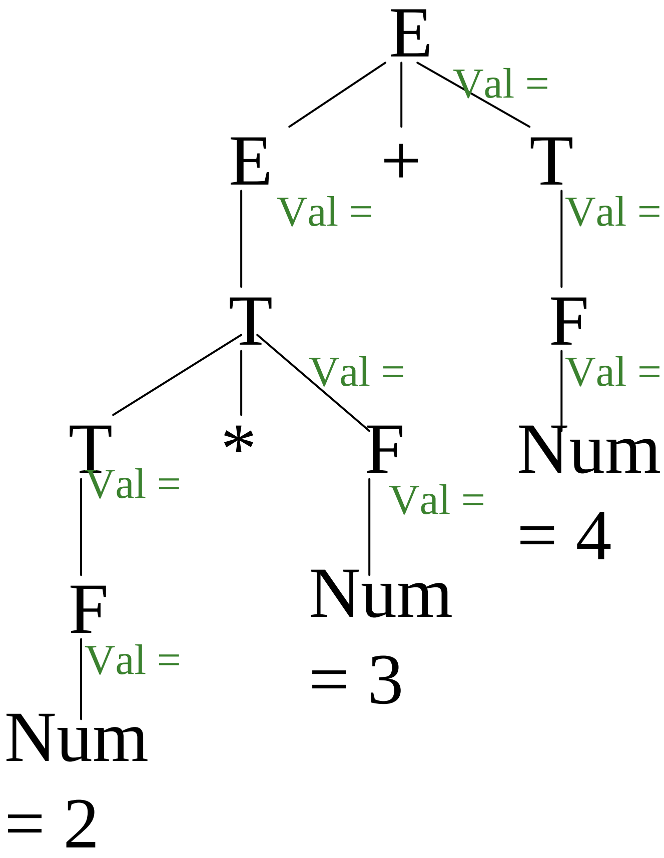
Synthesized Attributes – Expression Grammar

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

Synthesized Attributes –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

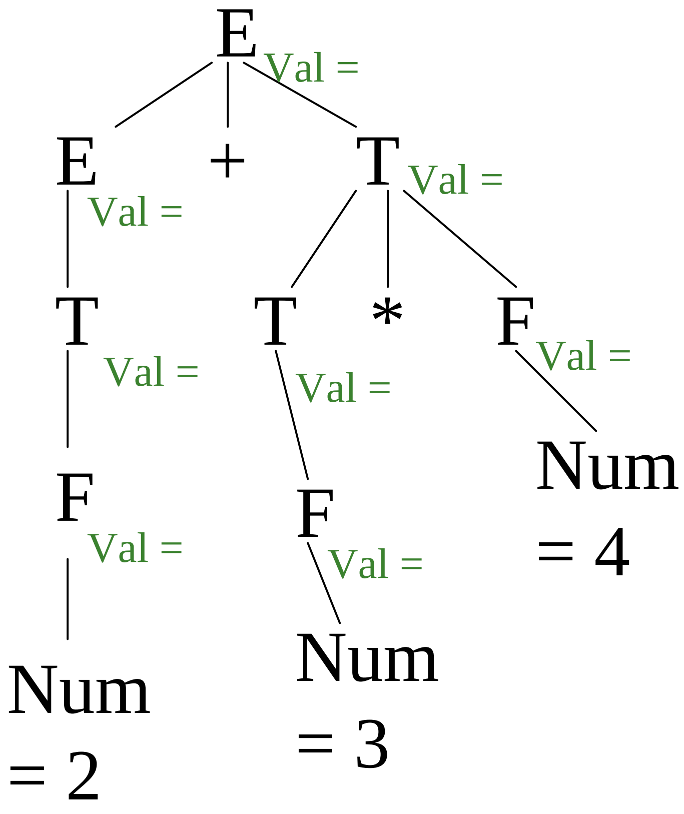
Input: 2 * 3 + 4



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



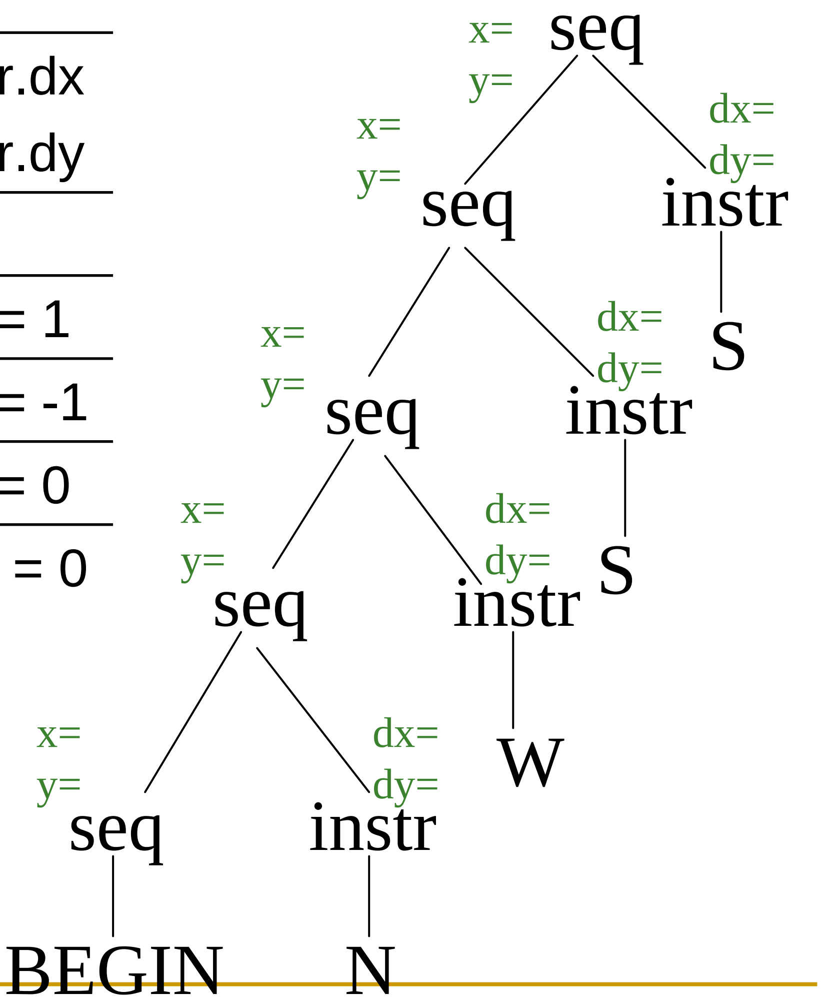
Synthesized Attributes – Grid Positions

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



Inherited Attributes

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

Example:

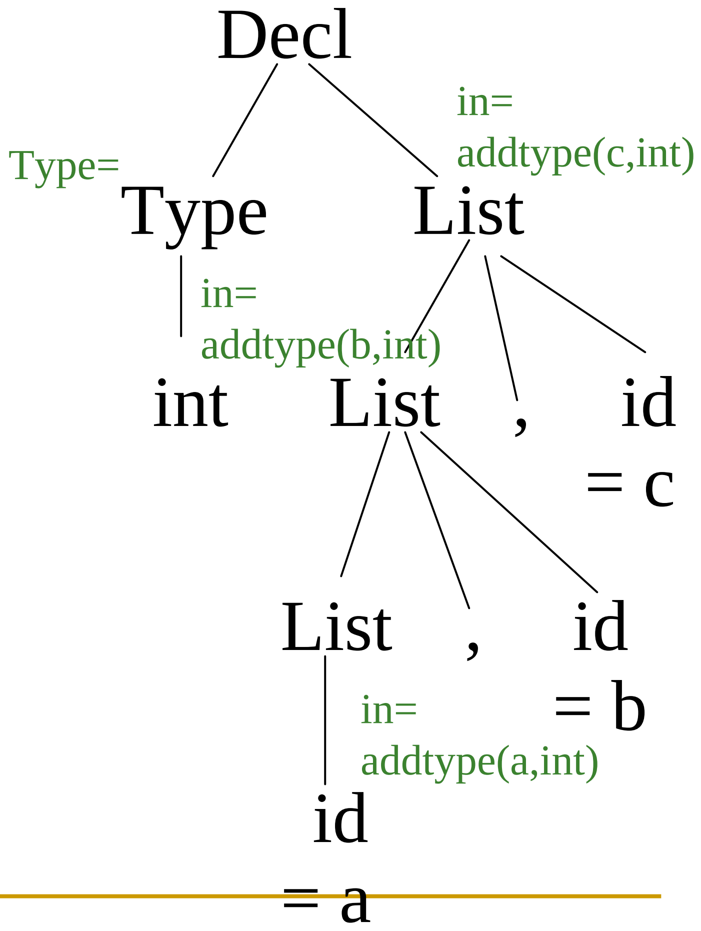
Production	**Semantic Rules**
A → B C D	B.b := A.a + C.b

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)



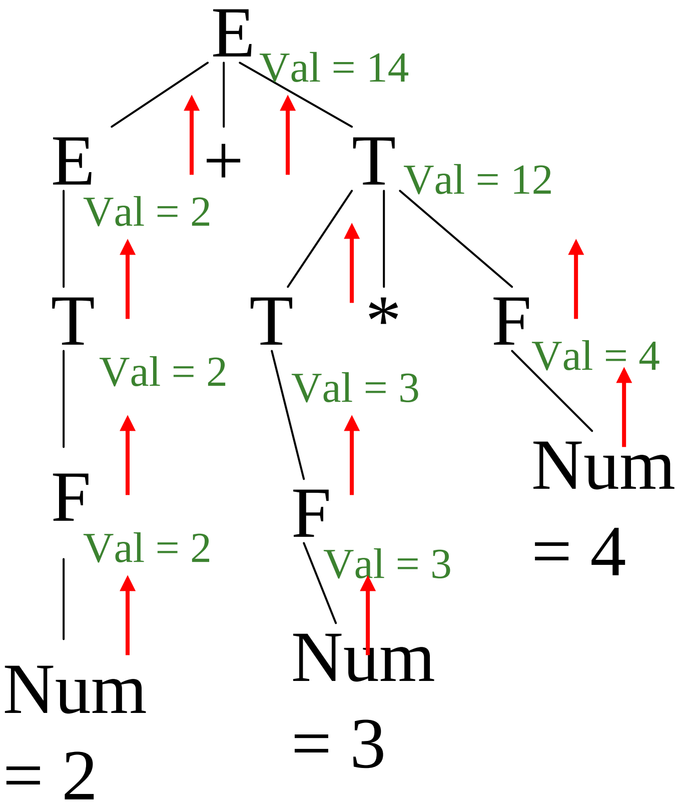
Input: int a,b,c

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

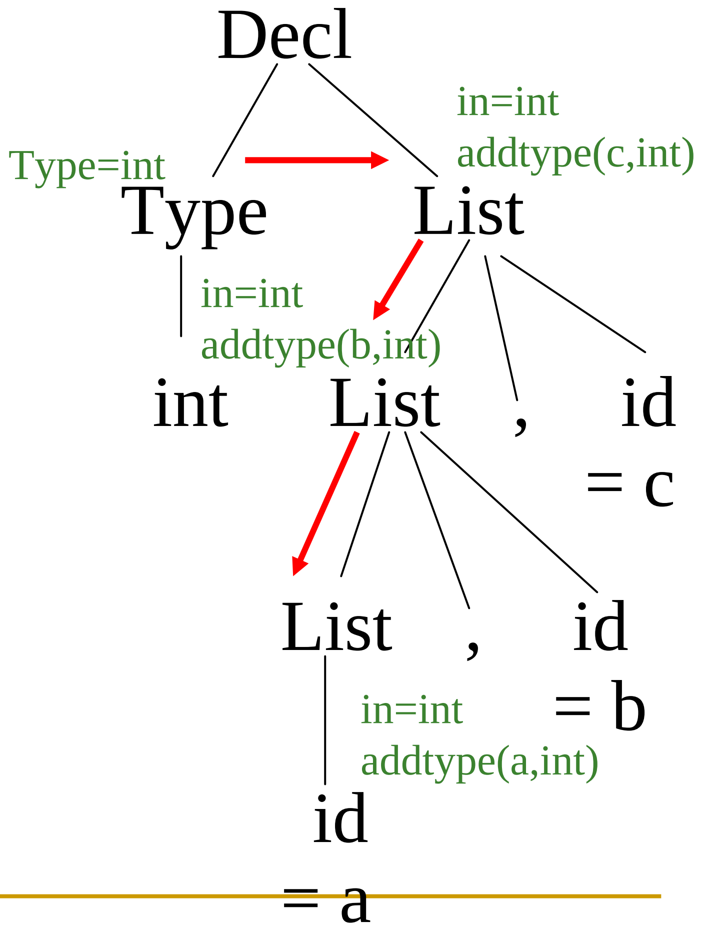
Production	**Semantic Actions**
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow$ num	$F.val =$ value(num)
$F \rightarrow (E)$	$F.val = E.val$



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)



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!

YACC

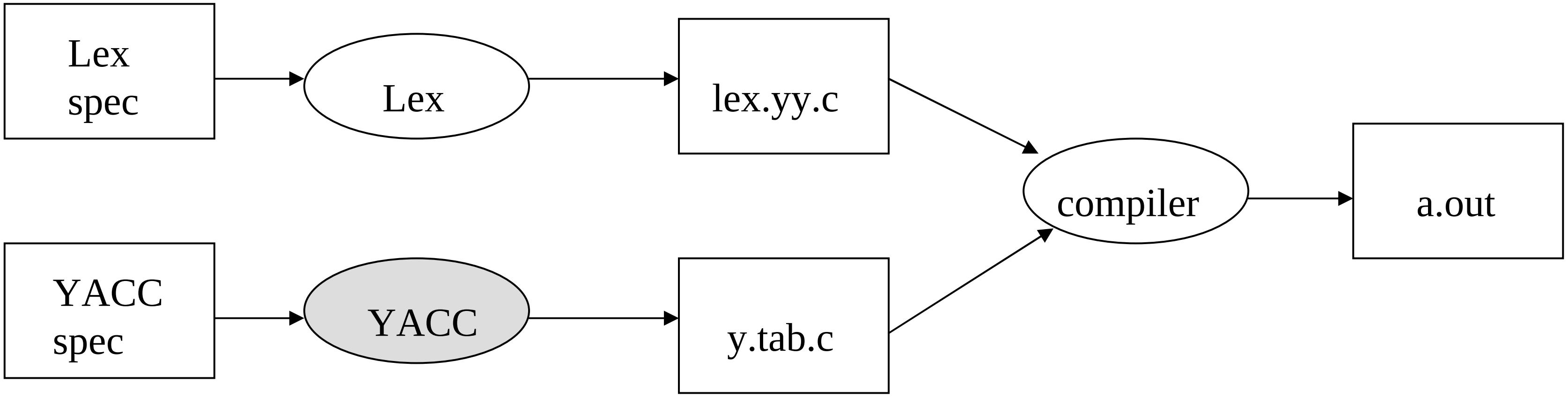
Yet Another Compiler-Compilers

S. C. Johnson, Bell Lab, 1975.

YACC

- Yacc takes a description of a grammar as its input and generates the table and code (in C) for an LALR parser. (“y.tab.c”)
- The input also includes semantic actions, and Yacc also generates code for carrying out these actions.
- The input specification for Yacc resembles that for Lex; it also consists of three parts.

YACC



Lex spec

Lex

lex.yy.c

YACC spec

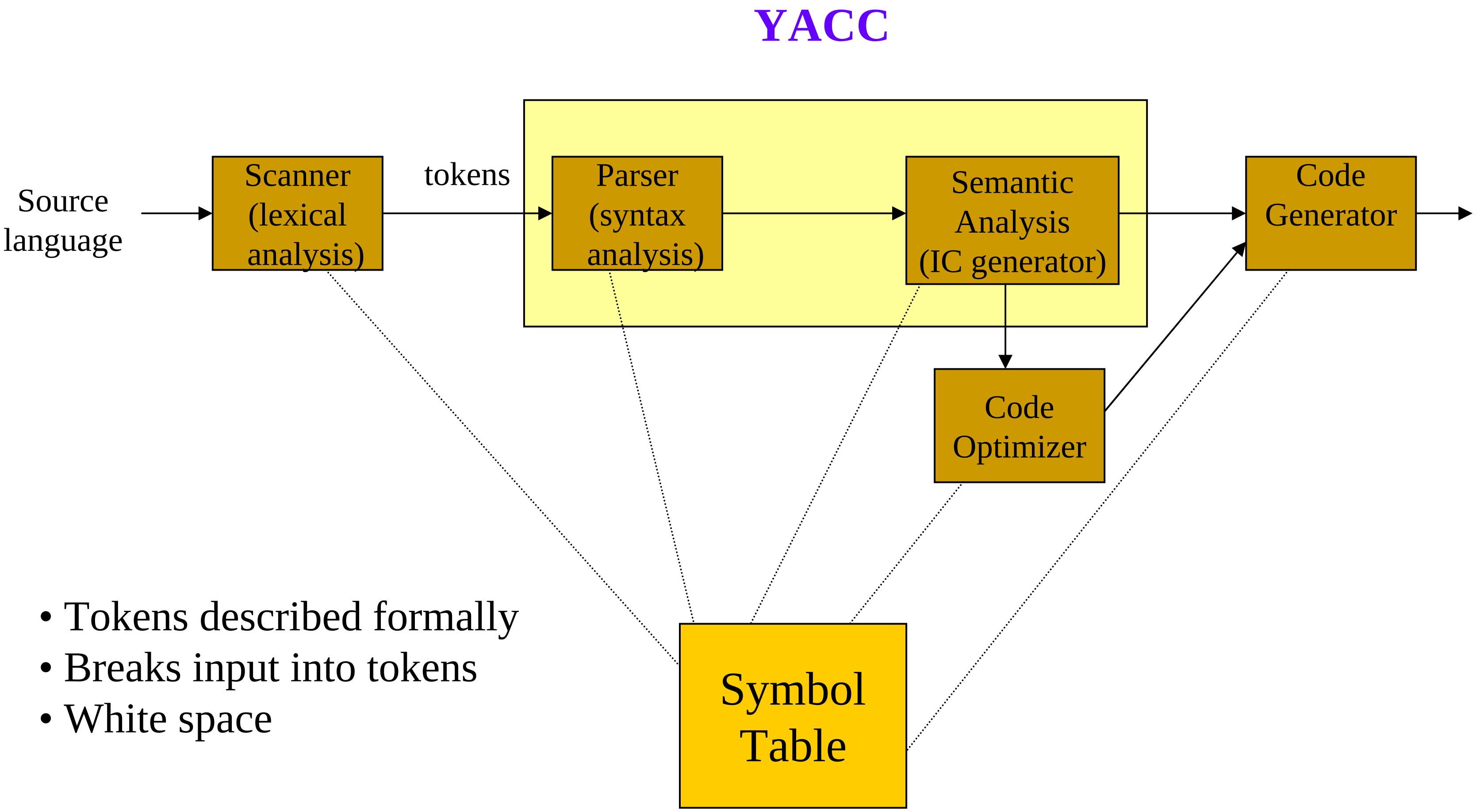
YACC

y.tab.c

compiler

a.out

Syntax & Semantic Analysis - Parsing



- Tokens described formally
- Breaks input into tokens
- White space

YACC Specifications

```
%{
    Declaration Section
    includes, comments, declarations (C code)
%}
    token definitions
%%
    Semantic actions with grammars
%%
    Supporting C/C++ code
```

Similar to Lex

YACC Declarations Section

- Includes:
  - Optional C/C++ code (%{ … %}) – copied directly into y.tab.c
  - 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++ 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 Lex

- *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

If using

#include “lex.yy.c”

- lex input.l

yacc input.y

cc y.tab.c –ly -ll

If compiling separately:

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

yacc –d input.y

cc y.tab.c lex.yy.c –ly -ll

Basic Lex/YACC example

```
%%
[a-zA-Z]+  {return(NAME);}
[0-9]{3}”-”[0-9]{4}
            {return(NUMBER); }
[ \n\t]              ;
%%
```

Lex

```
%token NAME NUMBER
%%
file    :     file  line
        |      line
        ;
line    :      NAME  NUMBER
        ;
%%
#include “lex.yy.c”
```

YACC

Expression Grammar Example

```
%{
#include “lex.yy.c”
extern int yylval;
%}
%token NUMBER MULTIPLY
%%
line         :   expr
             ;
expr         :   expr MULTIPLY term
             |   term
             ;
term         :   term ‘*’ factor
             |   factor
             ;
factor       :   ‘(‘ expr ‘)’
             |   NUMBER
             ;
%%
```

Associated Lex Specification

```
%%
\*          {return(MULTIPLY); }
\+          {return(‘+’); }
\(          {return(‘(’); }
\)          {return(‘)’); }
[0-9]+      {return(NUMBER);}
.           ;
%%
```

Grid Example

```
%{
#include “lex.yy.c”
%}
%token NORTH SOUTH EAST WEST
%token BEGIN
%%
seq          :   seq instr
            |   BEGIN
             ;
instr        :   NORTH
             |   SOUTH
             |   EAST
             |   WEST
             ;
%%
```

Associated Lex Specification

```
%%
N           {return(NORTH); }
S           {return(SOUTH); }
E           {return(EAST); }
W           {return(WEST); }
BEGIN       {return(BEGIN);}
.           ;
%%
```

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-hand Side (LHS) is $$. Right-hand Side (RHS) elements are numbered sequentially starting at $1.

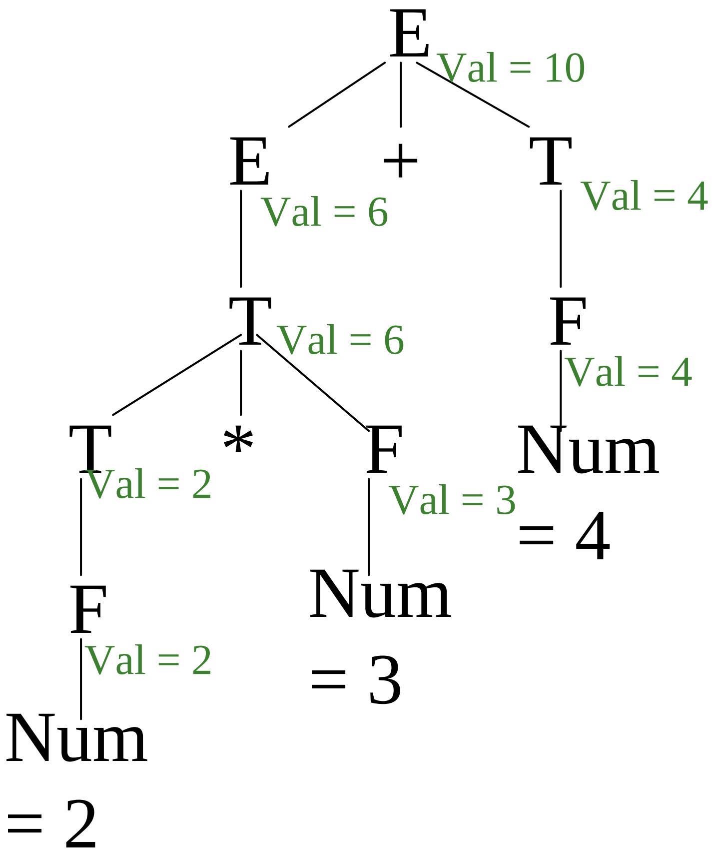
  For A : B C D,

  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
%%
line      :  expr                {printf(“Value = %d”,$1); }
          ;
expr      :  expr ‘+’ term       { $$ = $1 + $3; }
          |  term                { $$ = $1;  /* default – can be omitted */}
          ;
term      :  term ‘*’ factor     { $$ = $1 * $3; }
          |  factor
          ;
factor    :  ‘(‘ expr ‘)’        { $$ = $2; }
          |  NUMBER              { $$ = $1; }
          ;
%%
#include “lex.yy.c”
```

Associated Lex Specification

```
%%
\*            {return(‘*’); }
\+            {return(‘+’); }
\(            {return(‘(‘); }
\)            {return(‘)’); }
[0-9]*        {yylval = atoi(yytext);
                  return(NUMBER);}
%%
```

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	$\text{seq.x} = \text{seq}_1\text{.x} + \text{instr.dx}$ $\text{seq.y} = \text{seq}_1\text{.y} + \text{instr.dy}$
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+$3.dy; }
        |     BEGIN        {$$.x=0;  $$.y = 0; };
instr   :     N             {$$.dx = 0; $$.dy = 1;}
        |      S             {$$.dx = 0; $$.dy = -1;} …  ;
```

Precedence/Associativity 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