Where Syntax Meets Semantics
Three “Equivalent” Grammars

G1: \[<\text{subexp}> ::= a \mid b \mid c \mid <\text{subexp}> - <\text{subexp}>\]

G2: \[<\text{subexp}> ::= <\text{var}> - <\text{subexp}> \mid <\text{var}>\]
\[<\text{var}> ::= a \mid b \mid c\]

G3: \[<\text{subexp}> ::= <\text{subexp}> - <\text{var}> \mid <\text{var}>\]
\[<\text{var}> ::= a \mid b \mid c\]

These grammars all define the same language: the language of strings that contain one or more a’s, b’s or c’s separated by minus signs.

But the parse trees are not equivalent.
Parse of:
\[ a - b - c \]

**G2 parse tree:**
```
G2:  <subexp> ::= <var> - <subexp> | <var>
     <var> ::= a | b | c
```

**G3 parse tree:**
```
G3:  <subexp> ::= <subexp> - <var> | <var>
     <var> ::= a | b | c
```

(a - b) - c
Why Parse Trees Matter

- We want the structure of the parse tree to correspond to the semantics of the string it generates
- This makes grammar design much harder: we’re interested in the structure of each parse tree, not just in the generated string
- Parse trees are where syntax meets semantics
Outline

- Operators
- Precedence
- Associativity
- Other ambiguities: dangling else
- Cluttered grammars
- Parse trees and EBNF
- Abstract syntax trees
- Reverse Polish Notation and Evaluation
Operators

• Special syntax for frequently-used simple operations like addition, subtraction, multiplication and division
• The word *operator* refers both to the token used to specify the operation (like + and *) and to the operation itself
• Usually predefined, but not always
• Usually a single token, but not always
Operator Terminology

- **Operands** are the inputs to an operator, like 1 and 2 in the expression **1+2**
- **Unary** operators take one operand: −1
- **Binary** operators take two: **1+2**
- **Ternary** operators take three: **a?b:c**
More Operator Terminology

- In most programming languages, binary operators use an *infix* notation: \( a + b \)
- Sometimes you see *prefix* notation: \(+ \ a \ b\)
  - What is \((\ast (\ast 3 4) 2))\?  
  - What is \(\ast + 3 4 2\) ?
- Sometimes *postfix* notation: \(a \ b +\)
  - What is \((2 (3 4 +) \ast))\?  
  - What is \(2 3 4 + \ast\) ?
- Unary operators, similarly:
  - (Can’t be infix, of course)
  - Can be prefix, as in \(-1\)
  - Can be postfix, as in \(a++\)
Outline

• Operators
• Precedence
• Associativity
• Other ambiguities: dangling else
• Cluttered grammars
• Parse trees and EBNF
• Abstract syntax trees
• Reverse Polish Notation and Evaluation
Working Grammar – G4

\[
\text{G4: } \langle \text{exp} \rangle \ ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{exp} \rangle \cdot \langle \text{exp} \rangle \mid (\langle \text{exp} \rangle) \mid a \mid b \mid c
\]

This generates a language of arithmetic expressions using parentheses, the operators + and *, and the variables a, b and c.
Issue #1: Precedence

Our G4 grammar can generate two trees for \(a+b*\text{c}\). In the left tree, the addition is performed before the multiplication, which is not the usual convention for operator precedence.

\[
G4: \quad <exp> ::= <exp> + <exp> | <exp> * <exp> | \langle <exp> \rangle | a | b | c
\]
Operator Precedence

• Applies when the order of evaluation is not completely decided by parentheses
• Each operator has a precedence level, and those with higher precedence are performed before those with lower precedence, as if parenthesized
• Most languages put * at a higher precedence level than +, so that:

\[ a + b \times c = a + (b \times c) \]
Precedence Examples

• C (15 levels of precedence—too many?)
  \[ a = b < c ? * p + b * c : 1 << d () \]

• Pascal (5 levels—not enough?)
  \[ a <= 0 \text{ or } 100 <= a \quad \text{Error!} \]

• Smalltalk (1 level for all binary operators)
  \[ a + b * c \]
Precedence In The Grammar

• Fix precedence: Modify grammar to put * below + in the parse tree in G5.
• <exp> defined in terms of <mulexp> therefore higher in parse tree and lower precedence.
Correct Precedence

G5:  \(<\text{exp}\> ::= <\text{exp}\> + <\text{exp}\> | <\text{mulexp}\>

<\text{mulexp}\> ::= <\text{mulexp}\> * <\text{mulexp}\>

| (\<\text{exp}\>)

| a | b | c

Our new grammar generates this tree for \texttt{a+b*c}. It generates
the same language as before, but no longer generates parse
trees with incorrect precedence.
Parse

G5:  
\[ <exp> ::= <exp> + <exp> \mid <mulexp> \]
\[ <mulexp> ::= <mulexp> \ast <mulexp> \mid ( <exp> ) \mid a \mid b \mid c \]

\[
\begin{align*}
\text{a} & + \text{b} \ast \text{c} \\
<exp> & = <exp> + <exp> \\
& = <exp> + <mulexp> \\
& = <mulexp> + <mulexp> \\
& = \text{a} + <mulexp> \\
& = \text{a} + <mulexp> \ast <mulexp> \\
& = \text{a} + \text{b} \ast \text{c}
\end{align*}
\]
For the language defined by the grammar above, give the parse tree and select the value of each:

1. $1-2*3$? 
   $-5\ 3\ -3\ 7$
2. $1-2-3$? 
   $-4\ 2\ -2\ 0$
3. $1-3-2$? 
   $-4\ 2\ -2\ 0$

Which are valid for the above grammar?

1. $1-1-1$
2. $2*3$
3. $2*2*3$
4. $2*3-1$
Outline

• Operators
• Precedence
• **Associativity**
  • Other ambiguities: dangling else
  • Cluttered grammars
  • Parse trees and EBNF
  • Abstract syntax trees
  • Reverse Polish Notation and Evaluation
Issue #2: Associativity

Our grammar G5 generates both these trees for \( a+b+c \). The first one is not the usual convention for operator associativity.

G5:

\[
<\text{exp}> ::= <\text{exp}> + <\text{exp}> \mid <\text{mulexp}>
\]

\[
<\text{mulexp}> ::= <\text{mulexp}> * <\text{mulexp}> \mid (<\text{exp}>)
\]

\[
\mid a \mid b \mid c
\]
Operator Associativity

• Applies when the order of evaluation is not decided by parentheses or by precedence

• *Left-associative* operators group left to right: \( a+b+c+d = ((a+b)+c)+d \)

• *Right-associative* operators group right to left: \( a+b+c+d = a+(b+(c+d)) \)

• Most operators in most languages are left-associative, but there are exceptions
Associativity Examples

- **C**
  - `a<<b<<c` — most operators are left-associative
  - `a=b=0` — right-associative (assignment)

- **ML**
  - `3-2-1` — most operators are left-associative
  - `1::2::nil` — right-associative (list builder)

- **Fortran**
  - `a/b*c` — most operators are left-associative
  - `a**b**c` — right-associative (exponentiation)
Associativity In The Grammar

G5: \[<\text{exp}\> ::= <\text{exp}\> + <\text{exp}\> | <\text{mulexp}\>\]
\[<\text{mulexp}\> ::= <\text{mulexp}\> * <\text{mulexp}\>\]
\[| (<\text{exp}\>)\]
\[| a | b | c\]

G6: \[<\text{exp}\> ::= <\text{exp}\> + <\text{mulexp}\> | <\text{mulexp}\>\]
\[<\text{mulexp}\> ::= <\text{mulexp}\> * <\text{rootexp}\> | <\text{rootexp}\>\]
\[<\text{rootexp}\> ::= (<\text{exp}\>) | a | b | c\]

- Fix associativity: Modify the grammar to make trees of +s grow down to the left (and likewise for *s) in G6.
- G5 ambiguous: \(<\text{exp}\>+<\text{exp}\>\) is right and left recursive.
- G6: Left recursive rule alone defines left associativity, \(<\text{exp}\> + <\text{mulexp}\>\) and \(<\text{mulexp}\> * <\text{rootexp}\>\)
Correct Associativity

Our new grammar generates this tree for \( a+b+c \). It generates the same language as before, but no longer generates trees with incorrect associativity.

\[
\text{G6: } \quad \langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{mulexp} \rangle \mid \langle \text{mulexp} \rangle \\
\langle \text{mulexp} \rangle ::= \langle \text{mulexp} \rangle \ast \langle \text{rootexp} \rangle \mid \langle \text{rootexp} \rangle \\
\langle \text{rootexp} \rangle ::= (\langle \text{exp} \rangle) \mid a \mid b \mid c
\]
Exercise 2

For the language defined by the grammar, give the parse tree and the value of the following:

a) $1 - 2 \cdot 3$?  -5  3 –3 7
b) $1 - 2 - 3$?  -4  2 –2 0
c) $1 - 3 - 2$?  -4  2 –2 0
Exercise 2 Continued

Starting with this grammar:

\[
\begin{align*}
\text{G6:} & \quad \langle \text{exp} \rangle : = \langle \text{exp} \rangle + \langle \text{mulexp} \rangle \mid \langle \text{mulexp} \rangle \\
\langle \text{mulexp} \rangle & : = \langle \text{mulexp} \rangle \ast \langle \text{rootexp} \rangle \mid \langle \text{rootexp} \rangle \\
\langle \text{rootexp} \rangle & : = (\langle \text{exp} \rangle) \\
& \quad \mid a \mid b \mid c
\end{align*}
\]

d) Add subtraction and division (- and /) with the customary precedence and associativity.
e) Add a left-associative operator % between the + and * in precedence.
f) Add a right-associative = operator, at lower precedence than any of the others.
g) Add parenthesis to reflect the parse under definition f?
   a) \( a = b + c \)
   b) \( a + b = b \ast c \)
Outline

• Operators
• Precedence
• Associativity
• Other ambiguities: dangling else
• Cluttered grammars
• Parse trees and EBNF
• Abstract syntax trees
• Reverse Polish Notation and Evaluation
Issue #3: Ambiguity

- **G4** was *ambiguous*: it generated more than one parse tree for the same string
- Fixing the associativity (in G5) and precedence (in G6) problems eliminated all the ambiguity
- This is usually a good thing: the parse tree corresponds to the meaning of the program, and we don’t want ambiguity about that
- Not all ambiguity stems from confusion about precedence and associativity...
Dangling Else In Grammars

\[
\begin{align*}
<\text{stmt}> & ::= <\text{if-stmt}> \mid s1 \mid s2 \\
<\text{if-stmt}> & ::= \text{if } <\text{expr}> \text{ then } <\text{stmt}> \text{ else } <\text{stmt}> \\
& \quad \mid \text{if } <\text{expr}> \text{ then } <\text{stmt}> \\
<\text{expr}> & ::= e1 \mid e2
\end{align*}
\]

This grammar has a classic “dangling-else ambiguity.” The statement we want to derive is

\[
\text{if } e1 \text{ then if } e2 \text{ then } s1 \text{ else } s2
\]

and the next slide shows two different parse trees for it...
Most languages that have this problem choose this parse tree: `else` goes with nearest unmatched `then`.

Ambiguous if-then-else parse trees

if `e1` then (if `e2` then `s1` else `s2`)
Eliminating The Ambiguity

• We want to insist that if this expands into an `if`, that `if` must already have its own `else`.
• First, we make a new non-terminal `<full-stmt>` that generates everything `<stmt>` generates.
• Except that it can not generate `if` statements with no `else`:

```latex
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt>
             | if <expr> then <stmt>
<expr> ::= e1 | e2
```
G7:

<full-stmt> ::= <full-if> | s1 | s2
<full-if> ::= if <expr> then <full-stmt> else <full-stmt>

<i-stmt> ::= if <expr> then <full-stmt> else <i-stmt> | if <expr> then <i-stmt>
expr ::= e1 | e2

- Use the new <full-stmt> non-terminal here.
- The effect is the new grammar can match an else part with an if part only if all the nearer if parts are already matched.
G7:

<full-stmt> ::= <full-if> | s1 | s2
<full-if> ::= if <expr> then <full-stmt> else <full-stmt>
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <full-stmt> else <stmt> |
          if <expr> then <stmt>
<expr> ::= e1 | e2

\[
\text{if e1 then if e2 then s1 else s2}
\]
if e1 then if e2 then s1 else s2

Correct Parse Tree
\[
<\text{full-stmt}> ::= \text{if } <\text{expr}> \text{ then } <\text{full-stmt}> \text{ else } <\text{full-stmt}> | \text{s1} | \text{s2} | \text{s3}
\]

\[
<\text{stmt}> ::= <\text{if-stmt}> | \text{s1} | \text{s2} | \text{s3}
\]

\[
<\text{if-stmt}> ::= \text{if } <\text{expr}> \text{ then } <\text{full-stmt}> \text{ else } <\text{stmt}> | \text{if } <\text{expr}> \text{ then } <\text{stmt}>
\]

\[
<\text{expr}> ::= \text{e1} | \text{e2}
\]

\[
\text{if e1 then if e2 then s1 else s2 else s3}
\]
Dangling Else

• We fixed the grammar, but…
• The grammar trouble reflects a problem with the language, which we did not change
• A chain of if-then-else constructs can be very hard for people to read
• Especially true if some but not all of the else parts are present
Exercise 3

Give the parse tree for each \(<stmt>\).

\(<full>\) ::= \(\text{if } <expr> \text{ then } <full> \text{ else } <full> \) | 
\(s1 \mid s2 \mid s3\)

\(<stmt>\) ::= \(\text{if } <expr> \text{ then } <full> \text{ else } <stmt> \) | 
\(\text{if } <expr> \text{ then } <stmt> \) | 
\(s1 \mid s2 \mid s3\)

\(<expr>\) ::= \(e1 \mid e2\)

1. if \(e1\) then \(s1\)
2. if \(e1\) then if \(e2\) then \(s1\)
3. if \(e1\) then if \(e2\) then \(s1\) else \(s2\)
4. if \(e1\) then if \(e2\) then \(s1\) else \(s2\) else \(s3\)
5. if \(e1\) then \(s1\) else if \(e2\) then \(s1\) else \(s2\)
Clearer Styles

```c
int a=0;
if (0==0)
  if (0==1) a=1;
  else a=2;

int a=0;
if (0==0) {
  if (0==1) a=1;
  else a=2;
}
```

Better: correct indentation

Even better: use of a block reinforces the structure
Languages That Don’t Dangle

• Some languages define if-then-else in a way that forces the programmer to be more clear
• Algol does not allow the `then` part to be another `if` statement – though it can be a block containing an `if` statement
• Ada requires each `if` statement to be terminated with an `end if`
Outline

- Operators
- Precedence
- Associativity
- Other ambiguities: dangling else
- **Cluttered grammars**
- Parse trees and EBNF
- Abstract syntax trees
- Reverse Polish Notation and Evaluation
Clutter

- The new if-then-else grammar is harder for people to read than the old one
- It has a lot of clutter: more productions and more non-terminals
- Same with G4, G5 and G6: we eliminated the ambiguity but made the grammar harder for people to read
- This is not always the right trade-off
Reminder: Multiple Audiences

• In Chapter 2 we saw that grammars have multiple audiences:
  ▪ Novices want to find out what legal programs look like
  ▪ Experts—advanced users and language system implementers—want an exact, detailed definition
  ▪ Tools—parser and scanner generators—want an exact, detailed definition in a particular, machine-readable form

• Tools often need ambiguity eliminated, while people often prefer a more readable grammar
Options

• Rewrite grammar to eliminate ambiguity
• Leave ambiguity but explain in accompanying text how things like associativity, precedence, and the dangling else should be parsed
• Do both in separate grammars
Outline

• Operators
• Precedence
• Associativity
• Other ambiguities: dangling else
• Cluttered grammars
• **Parse trees and EBNF**
• Abstract syntax trees
• Reverse Polish Notation and Evaluation
EBNF and Parse Trees

- You know that \{x\} or x* means "zero or more repetitions of x" in EBNF
- So \(<exp> ::= <mulexp> \{+ <mulexp>\}\) should mean a \(<mulexp>\) followed by zero or more repetitions of "+ \(<mulexp>\)"
- But what then is the associativity of that + operator? What kind of parse tree would be generated for a+a+a?
Two Camps

• Some people use EBNF loosely:
  ▪ Use {} anywhere it helps
  ▪ Add a paragraph of text dealing with ambiguities, associativity of operators, etc.

• Other people use EBNF strictly:
  ▪ Use \(<exp> ::= <mulexp> \{+ <mulexp>\}\) only for left-associative operators
  ▪ Use recursive rules for right-associative operators: \(<expa> ::= <expb> [ = <expa> ]\)
About Syntax Diagrams

• Similar problem: what parse tree is generated?
• As in loose EBNF applications, add a paragraph of text dealing with ambiguities, associativity, precedence, and so on
Outline

• Operators
• Precedence
• Associativity
• Other ambiguities: dangling else
• Cluttered grammars
• Parse trees and EBNF
• Abstract syntax trees
• Reverse Polish Notation and Evaluation
Full-Size Grammars

• In any realistically large language, there are many non-terminals
• Especially true when in the cluttered but unambiguous form needed by parsing tools
• Extra non-terminals guide construction of unique parse tree
• Once parse tree is found, such non-terminals are no longer of interest
Abstract Syntax Tree

- Language systems usually store an abbreviated version of the parse tree called the *Abstract Syntax Tree*
- Details are implementation-dependent
- Usually, there is a node for every operation, with a subtree for every operand
parse tree

Abstract Syntax Tree

Example
Outline

• Operators
• Precedence
• Associativity
• Other ambiguities: dangling else
• Cluttered grammars
• Parse trees and EBNF
• Abstract syntax trees
• Reverse Polish Notation and Evaluation
AST to Reverse Polish Notation and Evaluation

- Abstract Syntax Tree holds the infix expression operations and operands.
- Traversing the AST in *post-order* produces the RPN.
- RPN operator precedence is unambiguous.
- Operands are arranged in proper order for post-order evaluation.
- Easily evaluated on hardware using a stack.
Parse Tree: 1-2*3+4

\[
\begin{align*}
<exp> & ::= <exp> - <mulexp> \mid <exp> + <mulexp> \mid <mulexp> \\
<mulexp> & ::= <mulexp> \ast <rootexp> \mid <rootexp> \\
<rootexp> & ::= 1 \mid 2 \mid 3 \mid 4
\end{align*}
\]

```
1 - 2 * 3 + 4
```

\[
\text{Parse Tree: 1-2*3+4}
\]
Abstract Syntax Tree: 1 - 2 * 3 + 4
Post-order traversal and RPN

abstract syntax tree

Postorder traversal

RPN

void postOrder(tree t) {
    if (t != null) {
        postOrder(t.left);
        postOrder(t.right);
        cout << t.data;
    }
}

\[ 1 \ 2 \ 3 \ast - 4 + \]
RPN Evaluation using a Stack

1. Scan RPN input from left to right.

   1 | 2 | 3 | 4 input
   push input onto stack.

   + | - | * input
   1. pop operand2
      pop operand1
   2. push operand1 * operand2
      or
      push operand1 – operand2
      or
      push operand1 + operand2

2. After RPN is scanned, the expression value is stack top.

<table>
<thead>
<tr>
<th>Input</th>
<th>Stack</th>
<th>Push</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 * - 4 +</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2 3 * - 4 +</td>
<td>1 2</td>
<td>2</td>
</tr>
<tr>
<td>3 * - 4 +</td>
<td>1 2 3</td>
<td>3</td>
</tr>
<tr>
<td>* - 4 +</td>
<td>1 6</td>
<td>2 * 3</td>
</tr>
<tr>
<td>- 4 +</td>
<td>-5</td>
<td>1 - 6</td>
</tr>
<tr>
<td>4 +</td>
<td>-5 4</td>
<td>4</td>
</tr>
<tr>
<td>±</td>
<td>-1</td>
<td>-5 + 4</td>
</tr>
</tbody>
</table>

Java Expression Parser and Evaluator

Try the Java applet, which is given to enter expressions in the form specified in the expression grammar, parse, generate RPN, and evaluate the RPN. Source code for the applet is available.
Exercise 4

parse tree

1. Give the AST of the parse tree.
2. The corresponding RPN.
3. The value from the RPN evaluation.
Exercise 5

Given the above BNF rules, generate the AST for expressions:

- $6 + 5 - 8$
- $2 + 9 / 3 - 1$

Traverse the AST nodes in post order to produce the Reverse Polish Notation.

Evaluate and compare with the results of the parser applet.
Parsing, Revisited

• When a language system parses a program, it goes through all the steps necessary to find the parse tree
• But it usually does not construct an explicit representation of the parse tree in memory
• Most systems construct an AST instead
• We will see ASTs again in Chapter 23
Parser Generators

• There are many tools that automate scanning and parsing stages of programming languages

• In Java: javaCC, sableCC

• Basic Unix tool
  ▪ YACC : converts a context-free grammar into a runnable parser (*.y → y.tab.c)
YACC and Semantic Actions

- YACC can generate a parser based on a grammar which specifies
  - a set of production rules
  - semantic actions
- Format for a pair of rule and semantic action:
  
  ```
  name : non-terminals and tokens    { action; } 
  | alternatives    { action; } 
  ;
  ```
Grammar rules for YACC

Definitions:
\%
\%start line means the whole input should match line
\%union lists all possible types for values associated with parts of the grammar and gives each a field-name
\%type gives an individual type for the values associated with each part of the grammar, using the field-names from the \%union declaration
Grammar rules for YACC

Actions, C declarations and code:

$$ \quad \text{resulting value for any part of the grammar} \\
$1, \ $2, \ \text{etc. values from sub-parts of the grammar} \\
yyparse \quad \text{routine created by YACC from (expected input, action) lists. (It actually returns a value indicating if it failed to recognise the input.)} \\
yylex \quad \text{routine called by yyparse for all its input. We are using getchar, which just reads characters from the input.} \\
yyerror \quad \text{routine called by yyparse whenever it detects an error in its input.}
YACC program for an infix calculator

/*** File: calc.y ***/

%{
  /* C declarations used in actions */
  include <stdio.h>
%}

/* yacc definitions */

%union {int a_number;}
%start line
%type <a_number> exp term factor number digit
YACC program for an infix calculator

```plaintext
%%
/* production rules */
line    : exp ';' '
'     { printf ("result is %d\n", $1); }
  ;
exp    : term     { $$ = $1; }
  | exp '+' term    { $$ = $1 + $3; }
  | exp '-' term    { $$ = $1 - $3; }
  ;
term   : factor     { $$ = $1; }
  | term '*' factor    { $$ = $1 * $3; }
  | term '/' factor    { $$ = $1 / $3; }
  ;
factor : number     { $$ = $1; }
  | '(' exp ')'     { $$ = $2; }
  ;
/* semantic actions (in C)*/
```
YACC program for an infix calculator

number : digit { $$ = $1; }
| number digit { $$ = $1*10 + $2; }
;
digit : '0' { $$ = 0; }
| '1' { $$ = 1; }
| '2' { $$ = 2; }
| '3' { $$ = 3; }
| '4' { $$ = 4; }
| '5' { $$ = 5; }
| '6' { $$ = 6; }
| '7' { $$ = 7; }
| '8' { $$ = 8; }
| '9' { $$ = 9; }
;
YACC program for an infix calculator

%%% /* C code */

int main (void) {return yyparse ( );}

int yylex (void) {return getchar ( );}

void yyerror (char *s) {fprintf (stderr, "%s\n", s);}

%%% /* C code */
How YACC is used?

$>\text{yacc calc.y} \quad \rightarrow \quad \text{y.tab.c (rename it as calc.c)}$

$>\text{gcc -o calc calc.c} \quad \rightarrow \quad \text{calc}$

$>\text{calc}$

2+3*5–7;
result is 10
Using FLEX and YACC together

- YACC cannot represent numbers as [0-9]+ nor easily obtain the corresponding value, nor can it easily be used to ignore white space and comments.
- We need to use both FLEX and YACC together; FLEX for the simple parts (e.g. numbers, white space, comments) and YACC for more complex parts (e.g. expressions).
YACC code for infix calculator

/*** File: calcy.y ***/
{%
#include <stdio.h>
%
%

%union {int a_number;}
%start line
%token <a_number> number
%type <a_number> exp term factor

%%

line   : exp ';'            {printf ("result is %d\n", $1);}
    ;
YACC code for infix calculator

exp  : term             { $$ = $1; } 
    | exp '+' term          { $$ = $1 + $3; } 
    | exp '-' term          { $$ = $1 - $3; } 
    ;

term : factor           { $$ = $1; } 
    | term '*' factor       { $$ = $1 * $3; } 
    | term '/' factor       { $$ = $1 / $3; } 
    ;

factor : number         { $$ = $1; } 
    | '(' exp ')'           { $$ = $2; } 
    ;

%%

int main (void) {return yyparse ( );}
void yyerror (char *s) {fprintf (stderr, "\%s\n", s);}
FLEX code for infix calculator

/*** File: calcl.l ***/
%
#include "y.tab.h"
%

%%

[0-9]+ {yylval.a_number = atoi(yytext); return number;}
[ \t\n] ;
[-+*/();] {return yytext[0];}
. {ECHO; yyerror("unexpected character");}

%%

int yywrap (void) {return 1;}

 Chapter Three  Modern Programming Languages  72
Declarations

YACC declarations

\%token declare each grammar rule used by YACC that is recognised by FLEX and give type of value

FLEX declarations and actions

y.tab.h gives FLEX the names and type declarations etc. from YACC

yyval name used for values set in FLEX e.g.

yyval.a_number = atoi (yytext);

yyval.a_name = findname (yytext);
How FLEX and YACC is used?

```
$> yacc –d calcy.y  →  y.tab.c  y.tab.h
       (rename y.tab.c as calcy.c)

$> gcc –c calcy.c   →  calcy.o

$> flex calcl.l    →  lex.yy.c  (rename it as calcl.c)

$> gcc –c calcl.c   →  calcl.o

$> gcc –o calc calcl.o calcy.o  →  calc

$> calc
   2+3*5–7;
   result is 10
```
Conclusion

• Grammars define syntax, and more
• They define not just a set of legal programs, but a parse tree for each program
• The structure of a parse tree corresponds to the order in which different parts of the program are to be executed
• Thus, grammars contribute (a little) to the definition of semantics