Yacc and Lex

Yacc and Lex (1)

- need a compiler/interpreter for high level languages
- language has to be *easily* parsed:
  - it should have reserved words, regular grammar and one symbol lookahead
- the compiler:
  - symbol table (for variables, procedures, labels)
  - parser
  - syntax analysis - needs to be simple?
  - code generation - into machine code
  - optimisation
  - activation records for context switching
- Interpreter / Hardware.
- PL0
  - example of a complete environment: it has a syntax checker, a parser, a compiler and an interpreter
  - it uses a symbol table.
  - it is not optimised.
  - uses activation records.

Yacc and Lex (1)

- not the best way to generate compilers - difficult to extend the embedded language description.
- is there a better way?
- yes: use tools to aid compiler building such as **yacc (bison)** and **lex (flex)**.

LEX

- it is a problem oriented specification for character string matching
- it produces a language that recognises *only* regular expressions
- recognised strings that match regular expressions.
- each matched string invokes a user specified operation (code)
- it always chooses the longest match.
- one may need substantial look ahead (not one character).

Lex

Basic example:

input - ***abcdefh***

rules - ***ab***

abcdefg

finds **ab,** then starts matching again from c

- To compile the lex code first use lex/flex

lex my_prog.l

then use the gcc compiler to produce the binary

gcc -o myprog lex.yy.c –lfl (if using flex) OR

gcc -o myprog lex.yy.c –ll (if using lex)

To run the program use:

./myprog < input > output

Lex

- Note lex.yy.c contains the program yylex()
- default:

lex uses a default main()

read from standard input

write to standard output

- can use:

call yylex() from your own code.

- can incorporate into code generated by yacc automatically.
- other functions such as yywrap() deals with reading multiple files etc.
- many distributions of linux do not provide all the basic function definitions (such ass yywrap or yyerror) – definitions of these functions can be found on the web

Lex

- examples of **lex** code:

delete all blanks and tabs at the end of each line of text:

```
%%
[\t ]+$    ;
```

replace multiple blanks or tabs by single blank:

```
%%
[\t ]+$    ;
[\t ]  printf(" ");
```

- scans for both rules at the same time
- observes at the end of a string of tabs or blanks a newline character
- executes the desired action
- search for the word integer in the input:

```
%%
integer      printf("found keyword INT");
```

Lex

- if action contains more than one statement enclose in braces:
- **lex** generates a deterministic finite automaton from the regular expressions in the source.
- format for a **lex** file:

```
{definitions}        optional
%%
{rules}              optional
%%                   optional
{user subroutines}   optional
```

- minimal **lex** program:

%% *copies input to output*

- ***use of regular expressions:***
- *similar to those used in Unix.*

Lex

- operators:

" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

xyz"++" matches the string xyz++

xyz\+\+ same as above

[abc] matches either a, b or c

[a-z0-9<>@@] matches lowercase characters, digits, angle brackets or underline

[-+0-9] matches all digits and plus and minus signs

[^abc] matches all characters except a, b or c

[^a-zA-Z] matches any character that isn't a letter

.(period) matches any character except the newline

ab?c matches either ab or abc

a* matches zero or more occurrences of the character a

a+ matches one or more occurrences of the character a

[a-z]+ matches all strings of lower case characters

[A-Za-z][A-Za-z0-9]* matches all alphanumeric strings beginning with an alphabetic character

(ab|cd) matches either ab or cd

Lex

a(b|c)d matches abd or acd only

(**ab|cd+)?(ef)*** matches abefef, efefef, cdef or cddd and not abc, abcd or abcdef

^abc only matches abc if it occurs at the beginning of a line

abc$ matches abc if it occurs at the end of a line (same as abc/\n)

ab/cd matches ab if followed by cd

{digit} looks for a predefined string called "digit" and inserts it in the string

a{1,5} looks for 1 to 5 occurrences of a i.e. a, aa, aaa, aaaa, aaaaa

- more examples:

```
[ \t\n]    ;
```

blanks, tabs and newlines are ignored - can write as:

```
" "    |
"\t"      |
"\n"      ;
```

Note quotes around \t and \n not required

Lex

- when a pattern is matched, the string is placed in the variable yytext so can be printed out:

[a-z]+ printf("%s",yytext);

or

[a-z]+ ECHO;

- it is possible to output the length of the string matched using yyleng:

[a-zA-Z]+ {words++; chars += yyleng; }

- one can manipulate the matched words (treat as an array) and backtrack using yyless which puts characters back onto the input string.

Note by default the lex library imposes a limit of 100 characters to backup.

Lex

- ambiguous source rules:

1/ choose the longest match.

2/ if two rules match, choose the one defined first

e.g.

integer: *keyword action;*

[a-z]+: *identifier action;*

- input of integers is an identifier as it matches [a-z]+
- input of integer is a keyword as it matched both rules but the rule for integer comes first
- source definitions

lex file format:

{*definitions*}

%%

{rules}

%%

{user routines}

- lex turns rules into program code (usually in C).

Lex

- context sensitive analysis:

need to deal with preprocessor statements differently to normal code e.g. #defines in C

- methods:

use of flags - when only a few rules change from one context to another - user defined

use of start conditions - allows lex to define the code required

Lex

Example lex program to extract basic word statistics for a user specified file – based on example from “Unix programming tools: Lex and Yacc”, John Levine, 1992, O’Reilly, pages 32-35.

```
%{
unsigned charCount = 0, wordCount = 0, lineCount = 0;
%}

word [^ \t\n]+
eol \n

%%

{word} { wordCount++; charCount += yyleng; }
{eol} { charCount++; lineCount++; }
. charCount;

%%
```

Lex

```
main(argc,argv)
int argc;
char **argv;
{
 if (argc > 1) {
    FILE *file;
    file = fopen(argv[1], "r");
    if (!file) {
      fprintf(stderr,"Could not open file %s\n",argv[1]);
      exit(1);
    }
   yyin = file;
 }
 yylex();
 printf("The statistics are as follows: %u %u %u\n",charCount,
   wordCount, lineCount);
 return 0;
}
#ifndef yywrap
yywrap() { return 1; }
#endif
```

Yacc

- **yacc** is a Unix compiler
- **yacc** converts user defined grammar into a subroutine called yyparse() (written in c) that handles the input process
- this subroutine calls a user-supplied routine (usually yylex()) to return to the next basic input item
- specification of grammar is very general: LALR(1) with disambiguating rules

LALR(1): Look Ahead Left Recursive, (1) means one symbol lookahead

- **yacc** requires a specification of the input process
- rules describing the input structure
- code to be invoked when the rules are recognised
- low-level routine to do the the basic input
- **yacc** then generates a *parser* using **lex** to identify the basic tokens from the input stream
- **yacc** attempts to *recognise* combinations of these tokens that match the grammar

Yacc

- an action is invoked when a match is found
- example grammar rules:

d*ate: month_name day ',' year ;*

- this matches: February 14, 1966.
- structure recognised by **lex** called a 'terminal symbol' or token
- structure recognised by **yacc** called a 'non-terminal symbol'
- the advantage is that it is easy to add new rules e.g.

d*ate: month_name '/' day '/' year ;*

- this matches 2/14/1966
- input errors are detected as early as possible with a left to right scan
- this reduces the chance of reading and computing with bad data input
- this way bad data can be quickly found

Yacc

- constructions difficult to handle in **yacc** are typically difficult for human interpretation
- these cases reveal poor design of your language

yacc file format:

declarations   *optional*

%%

rules

%%   *optional*

programs   *optional*

- rule format:

A : BODY ;

where A is a *non-terminal symbol* and BODY is a sequence of zero or more *non-terminal symbols*, *terminal symbols* and *literals*

- a literal is a single character in quotes: 'a' etc

Yacc

- C escape sequences allowed as literals:

```
'\n' newline          '\r' return
'\'' single quote     '\e' backslash 'e'
'\t' tab              '\b' backspace
'\f' formfeed     '\xxx' octal 'xxx'
```

- cannot use **NULL** (\0 or 0)
- typical rules:

```
A  :  B C D ;
A  :  E F ;
A  :  G ;
    or:
A  :  B C D
   |  E F
   |  G
      ;
```

- one can have an empty rule:

```
empty  :  ;
```

Yacc

- tokens are declared in the *declaration* section:

%t*oken name1 name2...*

- start symbol declared in the *definition* section:

%s*tart symbol*

- actions:

```
A  :   '(' B ')'  { hello(1, "abc"); }
XXX:   YYY ZZZ
{  printf("a message\n");
flag = 25; }
```

Yacc

- a simple **yacc** calculator example (calc.y):

```
%{
#include <stdio.h>
%}
%union {
   long value;
}
%type <value> expression add_expr mul_expr
%token <value> NUMBER

%%
input:
     expression
   | input expression
expression:
     expression '\n' { printf("%ld\n", $1); }
   | expression '+' add_expr { $$ = $1 + $3; }
   | expression '-' add_expr { $$ = $1 - $3; }
   | add_expr { $$ = $1; }
add_expr:
     add_expr '*' mul_expr { $$ = $1 * $3; }
   | add_expr '/' mul_expr { $$ = $1 / $3; }
   | mul_expr { $$ = $1; }
mul_expr:
     NUMBER { $$ = $1; }
```

Yacc

- needs the following lex definition to work (calc.l)

```
%{
#include "y.tab.h"
%}
NUMBER  [0-9]+
OP [+-/*]
%%
{NUMBER}    { yylval.value = strtol(yytext, 0, 10); return NUMBER; }
({OP}|\n)   { return yytext[0]; }
.   { ; }
%%
#ifndef yywrap
   yywrap() { return 1; }
#endif
```

- to compile use:

% yacc -d calc.y

% lex calc.l

% gcc -o mycalc y.tab.c lex.yy.c –lfl (if using flex)

<u>Yacc</u>

- to run:

% ***mycalc***

12+45

57

2 + 3

5

100 + -50

syntax error

%

- <u>how does **YACC** parse the input?</u>
- it uses a stack and a finite state machine (FSM) with the four actions:

<u>Shift</u>

<u>Reduce</u>

<u>Accept</u>

<u>Error</u>

Yacc

- <u>Shift</u> used to push symbols onto the stack
- <u>Reduce</u> is used when the righthand side of the rule has been recognised (symbols on the stack) and these symbols can be replaced by the lefthand side of the rule
- <u>Accept</u> means the input has been parsed successfully
- <u>Error</u> means a syntax error has been found

- <u>recursive parsing example:</u>

statement: *Name '=' expression*

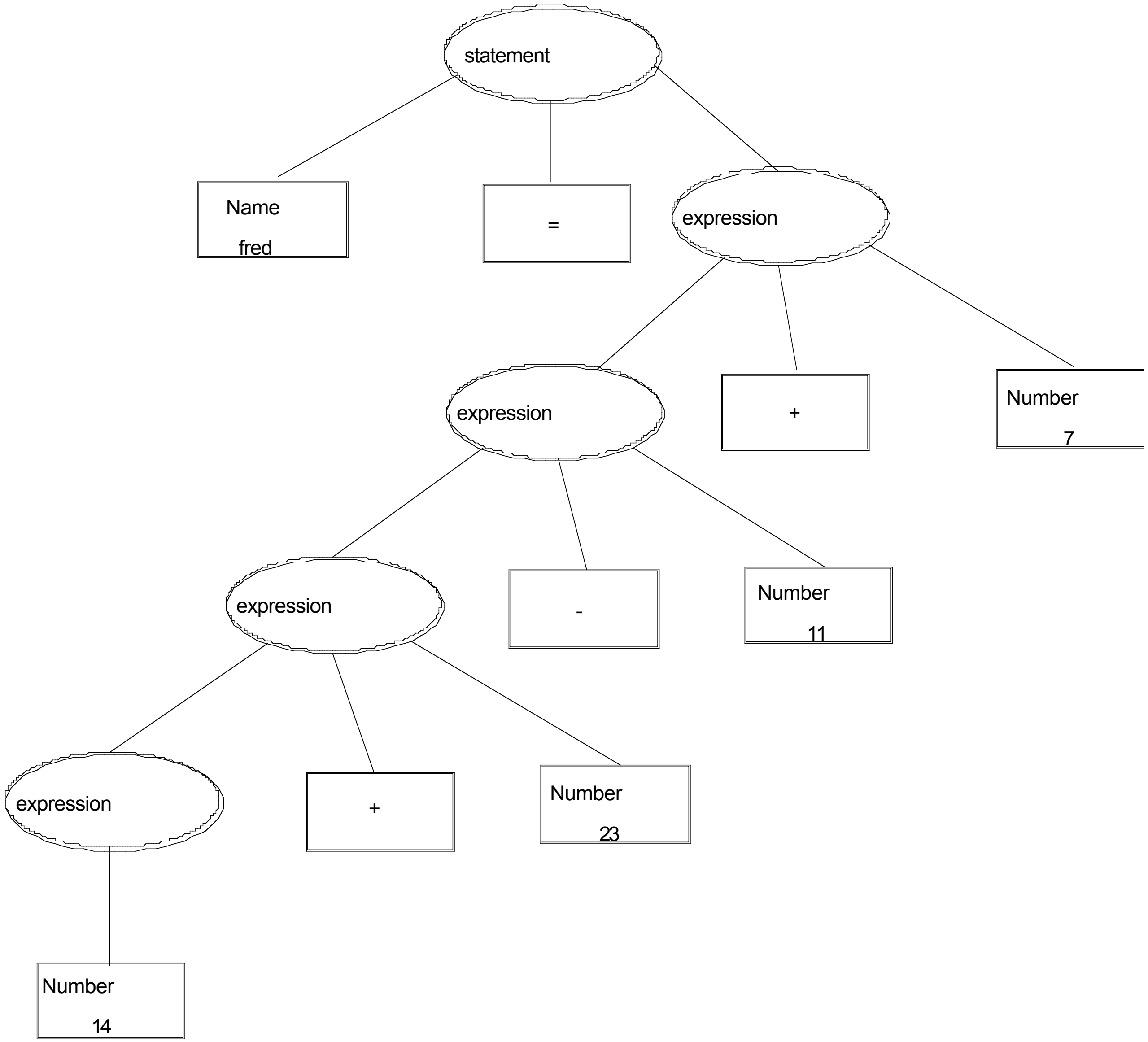
expression: *Number*

| *expression '+' Number*

| *expression '-' Number*

"mycalc = 14 + 23 -11 +7"

Yacc



statement
Name
fred
=
expression
expression
+
Number
7
expression
-
Number
11
expression
+
Number
23
Number
14

Yacc

- <u>problem grammars:</u>

```
phrase:    cart_animal AND CART
|   work_animal AND PLOW
;
cart_animal:   HORSE
|   GOAT
;
work_animal:   HORSE
|   OX
;
```

- unfortunately it needs two symbols lookahead
- for example input of "HORSE AND CART"- it cannot tell if HORSE is a cart_animal or a work_animal
- new first rule:

```
phrase:    cart_animal CART
|   work_animal PLOW
;
```

Yacc

- ambiguity

```
expression:    expression + expression { $$ = $1 + $2; }
|  expression '-  expresssion { $$ = $1 - $2; }
|  expression * expression { $$ = $1 * $2; }
|  expression '/ expression
{if($3 == 0)
yyerror("divide by zero");
else
$$ = $1 / $3;
}
|   '- expression { $$ = -$2; }
|   '+ expression { $$ = $2; }
|  NUMBER { $$ = $1; }
;
```

- parsing 2+3*4 gives (2+3)*4 or 2+(3*4) gives 16 shift/reduce conflicts
- it can be solved using *precedence* and *associativity*
- left associativity: group left to right e.g. 3+4+5 -> (3+4)+5

Yacc

- implicit precedence and associativity:

```
expression:    expression + mulexp
|  expression '-  mulexxp
| mulexp
;
mulexp:        mulexp * primary
|  mulexp '/ primary
|  NUMBER
;
primary:                 + expression
|  '- primary
| NUMBER
;
```

- use explicit precedence and associativity :

```
%left + -
%left * /'
%nonassoc UMINUS
```