Scanning and Parsing Tools Using Lex and Yacc

CS 4447 / CS 9545 -- Stephen M. Watt
University of Western Ontario

Using Lex on Unix

- lex myscanner.l
- mv lex.yy.c myscanner.c
- cc myscanner.c –o myscanner –ll
- myscanner < infile >outfile

Format of a Lex File

```
        /* Definitions */
A       [a-z]
%%

        /* Rules */
0|1|2   { printf("(%s)", yytext); return 1; }
x{A}+   { printf("<%s>", yytext); return 2; }
.       { printf("[%s]", yytext); return 3; }
%%

       /* Code */
int main() { while (yylex())  ; }
```

yylex

- The scanning function constructed by lex is called “yylex”
- **`int yylex();`**
- Default action is to echo.

yylex

- Will do whatever you tell it.
- Can return an integer value
- Typically 0/1 for success failure

- Can return an integer *code* saying what *kind* of token was matched
- Can pass other values back through global variables.

- Use codes > 255, so that we can pass back chars codes, e.g.

```
return '+';
```

(Remember that char is an integer type.)

Example – Part 1

```
%{
#define ID       300
#define NUM      301
#define ASSIGNOP 302

union {
    int   numval;
    char *idval;
} tok;

char *stralloc(char *);
%}

%%
```

Example – Part 2

```
[a-z]+ { tok.idval = stralloc(yytext); return ID; }
[0-9]+ { tok.numval = atoi(yytext); return NUM; }
":="   { return ASSIGNOP; }
";"    { return ';' ; }
.      { /* Don't echo */ }
%%
```

Example – Part 3

```
#include <string.h>
#include <stdlib.h>

char *stralloc(char *s)
{ return strcpy((char*)malloc(strlen(s)+1), s); }

int main()
{
    int code;
   while (code = yylex(), code) {
        switch (code) {
        case NUM: printf(”%d”, tok.numval); break;
        case ID:  printf(”%s”, tok.idval);
                  free(tok.idval); break;
        ...
       }
   }
}
```

Start conditions

- Sometimes have a few different contexts, e.g.
  - In a string “…”
  - In a comment /*…*/
  - Anything else
- In this situation, rules can be simplified using “Start conditions”

Start conditions in Flex

- Somewhat easier and more powerful than in standard “lex”. (negative conditions, …)

- Declare condition using **`%x`** ***`name`***
- Enter mode with action **`BEGIN(`*`name`*`)`**
- Limit rule with **`<`*`name`*`>`**

Example of Start Conditions

```
%x InComment
%%

"/*"                 { BEGIN(InComment); }
<InComment>"*/"      { BEGIN(INITIAL); }
<InComment>.         { /* Yum, yum, ... */ }
```

Other rules …

Using Yacc on Unix

- yacc –d mygrammar.y
- mv y.tab.c mygrammar.c
  mv y.tab.h mygrammar.h

Example – Part 1

Write Yacc grammar

```
%token NUM ID ASSIGNOP
%start MyProgram
%%

MyProgram: MyStatement | MyProgram MyStatement ;

MyStatement: ID ASSIGNOP MyExpression ';' ;

MyExpression: ID | NUM ;
```

File “mygram.h”

```
# define NUM 257
# define ID 258
# define ASSIGNOP 259
```

Example – Part 2

Extract scanner declarations into separate file

```
/* File eg2funs.h */
typedef union {
    int   numval;
    char *idval;
} TOKTYPE;

extern TOKTYPE tok;
extern char    *stralloc(char *);
```

Example - Part 3

Put your C code in its own file

```c
/* File eg2funs.c */
#include <string.h>
#include <stdlib.h>
#include "eg2funs.h"

TOKTYPE tok;

char *stralloc(char *s)
{
        return strcpy((char*)malloc(strlen(s)+1), s);
}

main() {
        yyparse();
}
```

Example – Part 4

Modify Scanner to include headers

```
%{
#include "eg2gram.h"
#include "eg2funs.h"
%}

%%
[a-z]+ { tok.idval = stralloc(yytext); return ID; }
[0-9]+ { tok.numval = atoi(yytext); return NUM; }
":="   { return ASSIGNOP; }
";"    { return ';' ; }
.      { /* Don't echo */ }
```

Example – Part 5

Build it

```
% lex eg2scan.l
% mv  lex.yy.c eg2scan.c

% yacc -d eg2gram.y
% mv y.tab.h eg2gram.h
% mv y.tab.c eg2gram.c

% cc -c eg2scan.c
% cc -c eg2gram.c
% cc -c eg2funs.c

% cc eg2funs.o eg2scan.o eg2gram.o -ll -ly -o eg2
```

Parser Actions

- Each grammar rule may have an associated “action”

```
Expr: Term
          { do something }
    | Term ADDOP Expr
          { do something else }
    ;
```

- In these actions, one may calculate a value for the LHS: **`$$ = ...`**
- The values of the parts of the RHS are available as **`$1, $2,`** etc.
- By default these are integers.

Example

```
Expr   : Term
                  { $$ = $1 ; }
       | Term '+' Expr
                  { $$ = $1 + $3; }
       ;

Term   : Factor
                  { $$ = $1; }
       | Factor '*' Expr
                  { $$ = $1 * $3; }
       ;

Factor : NUM      { $$ = tok.numval; }
       ;
```

Values of Tokens - Problem

- The strategy of using a global variable to communicate token information (e.g. tok, in our example) has problems.

- This is OK for rules like:

```
Factor : NUM { $$ = tok.numval; } ;
```

  but does not work for rules like:

```
MyStatement : ID ASSIGNOP MyExpression ';'
                 { $$ = ... }
            ;
```

- By the time the action is evaluated, **`tok`** has a value related to **`';'`** and the other values have been over-written.

Values of Tokens - Solution

- Yacc solves this, by defining the global variable **`yylval`** to use (e.g. instead of **`tok`**).
- Yacc knows about this variable, and will keep the needed values on a stack for $1,..,$n as needed.
- Including the header (y.tab.h   from yacc –d) in the lex program, provides the necessary declaration for yylval there.

Different Types

- In our previous example, we wanted the communication variable **`tok`** to be able to take on different types.
  For this we used a C union type.

- Now we want **`yylval`** to be able to have all the same types, *plus* we want to have the grammar rules give values of their own types as well.

- For this we use the **`%union`** directive in Yacc, e.g:

```
%union {
    /* For tokens */
          int          numval;
    char       *idval;

    /* For other rules. */
    struct node *node;

}
```

Different Types (cont'd)

- Declare tokens and non-terminals as follows:

```
%token <numval> NUM
%token <idval>  ID
%token          ASSIGNOP
%type  <node>   MyProgram MyStatement MyExpression
```

- Then $$ and $1,…$n may be used as variables of the associated type:

```
 MyStatement : ID ASSIGNOP MyExpression ';'
           { $$ = mkNode("assign", mkId($1), $3); }
           ;
```

- $1 has type **`char *`** ,
  while $3 and $$ have type **`struct node *`**

- This can be made explicit by saying $<node>$, $<idval>1, $<node>3

Example 3 – eg3funs.h

```
extern char            *stralloc(char *);

struct node {
        char           *tag;
        struct node *l;
        struct node *r;
};

extern struct node  *mkNode(char *, struct node *,
                                     struct node *);
extern struct node  *mkId (char *);
extern struct node  *mkNum(int);
```

Examle 3 – eg3funs.c

```
#include <string.h>
#include <stdlib.h>
#include "eg3funs.h"

/* A useless, fake node type */
struct node *mkNode(char *t, struct node *a, struct node *b)
{ struct node *r = (struct node *) malloc(sizeof(struct node));
  r->tag = t; r->l = a; r->r = b;
  return r; }

struct node *mkId(char *a)
{ return mkNode(a, 0, 0); }

struct node *mkNum(int n)
{ char  buf[20];
  sprintf(buf, "%d", n);
  return mkNode(stralloc(buf), 0, 0); }

char *stralloc(char *s)
{ return strcpy((char*)malloc(strlen(s)+1), s); }

main() { yyparse(); }
```

Example 3 – eg3gram.y

```
%union {
        struct node     *node;
        int             numval;
        char            *idval;
}
%{
#include "eg3funs.h"
%}
%token <numval>  NUM
%token <idval>   ID
%token           ASSIGNOP
%type  <node>    MyProgram MyStatement MyExpression

%start MyProgram
%%
MyProgram       : MyStatement
                | MyProgram MyStatement
                        { $$ = mkNode("stat", $1, $2); }
                ;
MyStatement     : ID ASSIGNOP MyExpression ';'
                        { $$ = mkNode("assign", mkId($1), $3); }
                ;
MyExpression    : ID  { $$ = mkId($1); }
                | NUM { $$ = mkNum($1); }
                ;
```

Example 3 – eg3scan.l

```
%{
#include "eg3gram.h"
#include "eg3funs.h"
%}

%%
[a-z]+ { yylval.idval =stralloc(yytext); return ID; }
[0-9]+ { yylval.numval=atoi(yytext);     return NUM; }
":="   { return ASSIGNOP; }
";"    { return ';' ; }
.      { /* Don't echo */ }
```

Example 3 - Makefile

```
eg3:    eg3funs.o eg3scan.o eg3gram.o
        cc eg3funs.o eg3scan.o eg3gram.o -ll -ly -o eg3

%.c %.h: %.y
        yacc -d $<
        mv y.tab.h $*.h
        mv y.tab.c $*.c

.l.c:
        lex $<
        mv lex.yy.c $*.c

eg3funs.o: eg3funs.h
eg3gram.o eg3scan.o: eg3gram.h
```

Words with changing token type

- Some tokens have different type depending on the context, e.g.

```
int f() { int N;  N = 8; return N; }
```
N is an identifier

```
typedef struct node *N;
N n1, n2;
```
N is a typename, like "int"

- To handle this, we need to communicate the context between the parser and the scanner.

C typedefs

- An example of this is C typdefs.
- This can be handled by an action on an appropriate grammar rule, e.g.
  `declaration : declaration_specifiers init_declarator_list ';'`
- The action can check whether the **`delcaration_specifiers`** contain **`typedef.`**
- If they do, then the identifiers declared in **`init_declarator_list`** need to be recorded in a table as being type names.
- Then when the scanner sees a word matching the identifier pattern, it must check in the table to see whether to yield an ID or TYPENAME.

C typedefs

- Note, the typedef table must be pushed/popped when entering/leaving blocks, because they can have local typedefs.
- You don't need to push/pop the typedef table for the assignment, unless you want to.
- In that case you would modify the grammar rule for **`compound_statement.`**