Lexical and Parser Tools

CSE 413, Autumn 2002

Programming Languages

http://www.cs.washington.edu/education/courses/413/02au/

References

» Modern Compiler Implementation in Java, Appel

http://www.cs.princeton.edu/~appel/modern/java/

» lex & yacc, Levine, Mason, Brown

http://www.oreilly.com/catalog/lex/index.html

» The Lex & Yacc Page (C)

http://dinosaur.compilertools.net/

» GNU flex and bison (C)

http://www.gnu.org/manual/flex-2.5.4/flex.html

http://www.gnu.org/manual/bison-1.35/bison.html

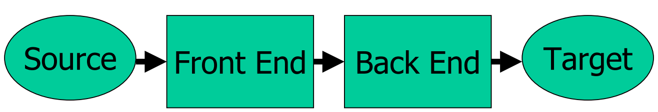
» JLex and CUP (Java)

http://www.cs.princeton.edu/~appel/modern/java/JLex/

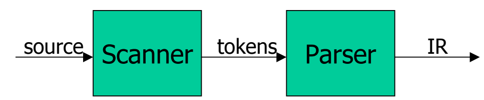
http://www.cs.princeton.edu/~appel/modern/java/CUP/

Structure of a Compiler

- First approximation
  - » Front end: analysis
    - Read source program and understand its structure and meaning
  - » Back end: synthesis
    - Generate equivalent target language program

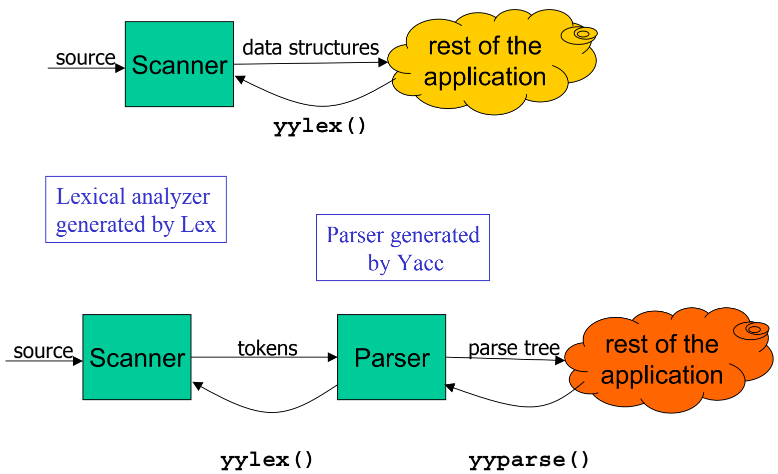


Front End



- Split into two parts
  - » **Scanner**: Responsible for converting character stream to token stream
    - Also strips out white space, comments
  - » **Parser**: Reads token stream; generates IR
- Both of these can be generated automatically or by hand
  - » Source language specified by a formal grammar
  - » Tools read the grammar and generate scanner & parser (either table-driven or hard coded)

Lex and Yacc



Lex

- Lex is a lexical analyzer generator
- Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream
  - » editor-script type transformations
  - » segmenting input in preparation for a parsing routine, ie, tokenizing
- You define the scanner by providing the patterns to recognize and the actions to take

Lex Input

```
%{
Declarations
%}
Definitions
%%
Rules
%%
User Subroutines
```

- Declarations - *optional* user supplied header code
- Definitions - *optional* definitions to simplify the Rules section
- Rules - token definition patterns and associated actions in C
- User subroutines - *optional* helper functions

Lex Output

- Generates a scanner function written in C
  - » the file is lex.yy.c
  - » the function is yylex()
  - » yylex() implements the DFA that corresponds to the regular expressions you supplied using:
    - transition table for the DFA
    - action code invoked at the accept states

Lex Rules

- Lines in the rules section have the form
  `expression action`
- `expression` is the pattern to recognize
  - » regular expressions defined as we have in class with extensions for convenience
- `action` is the action to take when the pattern is recognized
  - » arbitrary C code that becomes part of the DFA code

pattern definition match operators

x	the character "x"
[xy]	any character selected from the list given (in this case, x or y)
[x-z]	any character from the range given (in this case, x, y or z)
[^x]	any character but x, ie the complement of the character(s) specified.
.	any single character but newline.
x?	an optional x, ie, 0 or 1 occurrences of x
x*	0 or more instances of x.
x+	1 or more instances of x, equivalent to xx*
x{m,n}	m to n occurrences of x
^x	an x at the beginning of a line.
x$	an x at the end of a line.
\x	an "x", even if x otherwise has some special meaning
"xy"	string "xy", even if xy would otherwise have some special meaning
x|y	an x or a y.
x/y	an x but only if followed by y.
{xx}	the translation of xx from the definitions section.

actions

- When an `expression` is matched, Lex executes the corresponding `action`.
  - » the action is defined as one or more C statements
  - » if a section of the input is not matched by any pattern, then the default action is taken which consists of copying the input to the output

context for the action code

- There are several global variables defined at the point when the action code is called
  - » yytext - null terminated string containing the lexeme
  - » yyleng - the length of yytext string
  - » yylval - structured variable holding attributes of the recognized token
  - » yylloc - structured variable holding location information about the recognized token

count.l - count chars, words, lines

```
%{
   int numchar = 0, numword = 0, numline = 0;
%}

%%

\n              {numline++; numchar++;}
[^ \t\n]+       {numword++; numchar+= yyleng;}
.               {numchar++;}

%%

main()
{
   yylex();
   printf("%d\t%d\t%d\n", numchar, numword, numline);
}
```

declarations

rules

user code

create and run count

create the scanner source file `lex.yy.c`

```
[finson@walnut cse413]$ flex count.l
```

build the executable program `count`

```
[finson@walnut cse413]$ gcc -o count lex.yy.c -ll
```

run the program on the definition file `count.l`

```
[finson@walnut cse413]$ ./count < count.l
220     32      17
```

run the program on the generated source file `lex.yy.c`

```
[finson@walnut cse413]$ ./count < lex.yy.c
35824   5090    1509
```


histo.l - histogram of word lengths

```
   int lengs[100];
%%
[a-zA-Z]+       lengs[yyleng]++;
.|\n            ;
%%
yywrap() {
   int i;
   printf("Length  No. words\n");
   for(i=0; i<100; i++)
        if (lengs[i] > 0)
                printf("%5d%10d\n",i,lengs[i]);
   return(1);
}
```

declarations

rules

user code

Create and run histo

create the scanner source file `lex.yy.c`

```
[finson@walnut cse413]$ flex histo.l
```

build the executable program `histo`

```
[finson@walnut cse413]$ gcc -o histo lex.yy.c -ll
```

run the program on the definition file `histo.l`

```
[finson@walnut cse413]$ ./histo < histo.l
Length  No. words
    1        14
    2         3
    3         3
    5         5
    6         6
```

Yacc: Yet-Another Compiler Compiler

- Yacc provides a general tool for describing the input to a computer program.
  - » ie, Yacc helps write programs whose actions are directed by a language generated by some grammar
- The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized.
  - » Yacc turns the specification into a subroutine that handles the input process

Yacc Input

```
%{
Declarations
%}
Definitions
%%
Productions
%%
User Subroutines
```

- Declarations - *optional* user supplied header code
- Definitions - *optional* definitions to simplify the Rules section
- Productions - the grammar to parse and the associated actions
- User subroutines - *optional* helper functions

Yacc Output

- Generates a parser function written in C
  - » the file is y.tab.c
  - » the function is yyparse()
  - » yyparse() implements a bottom-up, LALR(1) parser for the grammar you supplied

Yacc Productions

- Lines in the productions section have the form
  `production action`
- `production` is the grammar expression
  - » almost exactly the same as the productions we have defined in class
- `action` is the action to take when the pattern is recognized
  - » arbitrary C code that becomes part of the DFA code

example productions

```
parameters : parameter
           | parameters ',' parameter
           ;

parameter :  T_KW_INT T_ID
               {printf("PARSED PARAMETER %s\n",$2);}
           ;

declarations  : declaration
              | declarations declaration
              ;

declaration   : T_KW_INT T_ID ';'
               {printf ("PARSED LOCAL VARIABLE %s\n",$2);}
              ;
```


Production format

- A grammar production has the form
  - » A : BODY ;
  - » A is the non-terminal
  - » BODY is a sequence of zero or more non-terminals and terminals
  - » ':' takes the place of '::=' or '→' in our grammars
  - » ';' means the end of the production or set of productions

actions

- When a `production` is matched, Yacc executes the corresponding `action`.
  - » the action is defined as one or more C statements
  - » the statements can do anything
  - » if no action code is specified, then the default action is to return the value of the first item in right hand side of the production for use in higher level accumulations

context for the action code

- The value of the items in the production is available when the action code is called
- You can set the value of this non-terminal by setting $$
- You can access the values of the parsed items with $1, $2, etc

```
parameter :  T_KW_INT T_ID
                    {printf("PARSED PARAMETER %s\n",$2);}
             ;
```

dp.y: declarations and definitions

```
%{
#define YYERROR_VERBOSE 1
void yyerror (char *s);
%}
%union {
  int intValue;
  char *stringValue;
}
%token <stringValue>T_ID
%token <intValue>T_INT
%token T_KW_INT
%token T_KW_RETURN
%token T_KW_IF
%token T_KW_ELSE
%token T_KW_WHILE
%token T_OP_EQ
%token T_OP_ASSIGN
```

miscellaneous C declarations

token attributes

tokens and keywords

dp.y: productions and actions

```
%%

program  : functionDefinition
         | program functionDefinition
         ;

functionDefinition : T_KW_INT T_ID '(' ')' '{' statements '}'
                    {printf("PARSED FUNCTION %s\n",$2);}
         | T_KW_INT T_ID '(' parameters ')' '{' statements '}'
                    {printf("PARSED FUNCTION %s\n",$2);}
         | T_KW_INT T_ID  '(' ')' '{' declarations statements '}'
                    {printf("PARSED FUNCTION %s\n",$2);}
         | T_KW_INT T_ID  '('parameters ')' '{' declarations statements '}'
                    {printf("PARSED FUNCTION %s\n",$2);}
         ;

etc, etc
```

look familiar?

dp.y: user subroutines

```
%%

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

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

simple error reporting function

simple main program

dp.l: lexical (part 1)

```
%{
   #include "dp.tab.h"
%}
alpha          [a-zA-Z]
numeric        [0-9]
comment        "//".*$
whitespace     [ \t\n\r\v\f]+
%%
[!>+\-*(){},;]         {return yytext[0]; }
"int"                  { return T_KW_INT; }
"return"               { return T_KW_RETURN; }
"if"                   { return T_KW_IF; }
"else"                 { return T_KW_ELSE; }
"while"                { return T_KW_WHILE; }
"=="                   { return T_OP_EQ; }
"="                    { return T_OP_ASSIGN; }
```

dp.l: lexical (part 2)

```
{whitespace}           { }
{alpha}({alpha}|{numeric}|_)* {
                           yylval.stringValue = strdup(yytext);
                           return T_ID;
                         }
{numeric}+             {
                           yylval.intValue = atoi(yytext);
                           return T_INT;
                         }
{comment}              { }
```


sample run of the dp parser

```
[finson@walnut cse413]$ ./parse < scanx.d
PARSED PARAMETER x
PARSED PARAMETER y
PARSED FUNCTION someFunction
PARSED PARAMETER x
PARSED LOCAL VARIABLE aSimpleInt
PARSED LOCAL VARIABLE a_S_I_2
PARSED LOCAL VARIABLE k
PARSED LOCAL VARIABLE z
PARSED FUNCTION main
[finson@walnut cse413]$
```


Summary

- The actions can be *any* code you need
- If you ever need to build a program that parses character strings into data structures
  - » think Lex/Yacc, flex/bison, JLex/CUP
- You can quickly build a solid parser with well defined tokens and grammar
  - » tokens are regular expressions
  - » grammar is standard Backus-Naur Form