CP 465 Database II

LEX-YACC

Ilias S. Kotsireas

ikotsire@wlu.ca

LEX, YACC

- developed at Bell Labs in 1970's
- tools for compiler development
- more generally: look for patterns in the input stream
- GNU, FSF distribute **flex** (fast Lex) and **bison** (replacement to YACC (Yet Another Compiler Compiler)
- Windows users: **ANTLR** `http://www.antlr.org/`
  (ANother Tool for Language Recognition)
- `ftp ftp.oreilly.com` anonymous ftp server
  directory: published/oreilly/nutshell/lexyacc
- **Main idea:** write programs that transform structured input
- huge range of applications: text search (grep), prog. lang. compiler

- two main tasks:
  (1) divide the input stream into meaningful units
  (2) discover the relationships among these units

(1) these units are called **tokens** (i.e. SQL keywords)
lexical analysis

Lex takes a set of descriptions of tokens (lex specification) and produces a C routine that can identify these tokens. **lexer**

The token descriptions are specified via regular expressions.

(2) after the input has been divided into tokens, a program needs to establish relationships among the tokens (i.e. SQL expressions, statements, variables)
syntactic analysis, parsing

List of rules that define the relationships between tokens: **grammar**

Yacc takes a description of a grammar (in its BNF notation) and produces a C routine that can parse the grammar. **parser**

A Yacc parser detects:

- when a sequence of input tokens matches a one of the rules of the grammar
- when a sequence of input tokens does not match any of the rules of the grammar **syntax error**

Yacc parsers are easy to modify and maintain

SUMMARY

divide input stream into units, establish relationships among them
Lex & Yacc

A simple Lex program

simple.l

```
%%
.|\n     ECHO;
%%
```

copies standard input to standard output (Unix cat command)

compilation instructions

lex simple.l (produces the lexer, C program lex.yy.c)

cc lex.yy.cc -o simple -ll (produces the executable simple)

./simple (CNTR D to terminate)

Word Recognizer (verbs)

Build a program that recognizes different types of English words.

Start with recognizing a set of verbs.

Run the program:

```
This will be easy
This: is not a verb
will: is a verb
be: is a verb
easy: is not a verb
```

Structure of a Lex program

Definition Section

Contains any C code or header files

The code is surrounded by the delimiters %{ and %}

The code is copied by Lex directly into the generated C file

The symbol %% marks the end of the definition section

Rules Section

Each rule is made up of two parts:

- pattern (regular expression)
- action (any valid C code)

Patterns and actions are separated by whitespace

The lexer executes the action, when it recognizes the corresp. pattern

The symbol %% marks the end of the rules section

The 4 rules of the Word Recognizer program:

- `[\t ]+  /* ignore whitespace */ ;`

  Pattern: a tab followed by a whitespace, once or more times
  Action: ; (do nothing)

- `is | am | ...`

- `[a-zA-Z]+   { printf("%s: is not a verb\n", yytext); }`

  Pattern: any ABtical string with at least one character
  Action: printf statement

- `.|\n   { ECHO; /* normal default anyway */ }`

  Pattern: any single character other than a newline, matched by

  `\n`

  Action: ECHO; prints the matched input on the output

Rules concerning matching

a verb will be matched by the verb rule **and** by the ABtical string rule

which of the corresp. two actions is to be executed?

consider the input tokens: "is", "island" **2 disambiguation rules**

- Lex patterns only match a given input string once.
- Lex executes the action for the longest possible match for the current input.

User Subroutines Section

contains any legal C code, which is copied into the generated C file, after the end of the generated code

In the main, we call `yylex();` i.e. the C routine produced by Lex, the lexer

Word Recognizer (verbs, adverbs, adjectives, ...)

Symbol Tables

Listing more words (extending to as many as we want, and as many parts of speech as we want) is not a practical way to write this program.

It would be more practical to build a table of words as the lexer is running, so that we can add new words without modifying/recompiling the Lex program.

Allow for dynamic declaration of parts of speech, as the lexer is running, reading the words from an input file with the syntax:

```
noun door window
verb eat drink
```

This table of words is a simple **symbol table** commonly used in lex/yacc

Adding a symbol table changes the Lex program substantially:

BEFORE: put separate patterns in the lexer for each word to match

AFTER: a single pattern matches any word and we consult the symbol table to decide which part of speech has been found

The names of the parts of speech (noun, verb, ...) become "reserved words", since they introduce a declaration line.

We still have a separate lex pattern/action for each reserved word.

We also need to add symbol table maintenance routines:

- `add_word()` puts a new word into the symbol table
- `lookup_word()` looks up a word already entered

The variable `state` keeps track of whether we are looking up words, or declaring them (in which case, it remembers the kind of words)

When we see a line starting with a name of a part of speech, we set the state variable to declare that kind of word.

If we see a

```
\n
```

then we switch back to the normal lookup state.

In the definition section, we define an `enum`, to record the types of individual words.

The caret symbol matches a pattern at the beginning of the line.

We reset state to lookup, at the beginning of each line.

In the user subroutines section, we create/search a linked list of words.

Yacc Grammars

We need to recognize specific sequences of tokens and perform appropriate actions.

A description of such a set of actions is called a **grammar**.

Example we need to recognize common sentences. Simple sentence types include: (noun verb) (noun verb noun)

Notation to describe grammars (BNF-like) $\rightarrow$
(a set of tokens can be replaced by a new symbol)

Example

subject $\rightarrow$ noun | pronoun object $\rightarrow$ noun

sentence $\rightarrow$ subject verb object

AIM: build a Yacc grammar, parser.

modify our lexical analyzer to return values useful to the parser.

Lexer-Parser Communication

We need to use a lex lexer and a yacc parser together

The parser is the higher-level routine, it calls the lexer `yylex()` whenever it needs a token from the input

The lexer scans the input recognizing tokens

As soon as it finds a token of interest to the parser, it returns the token's code as the value of `yylex()`

Not all tokens are of interest to the parser (e.g. comments, whitespace)

For these ignored tokens, the lexer doesn't return anything, it rather continues on to the next token

The lexer and the parser have to use the same token codes

Yacc defines the token codes (usually as small integers) with the # define preprocessor

```
# define NOUN 257
# define VERB 259
# define ADVERB 260
...
```

Token code 0 $\rightsquigarrow$ end of input

Yacc can optionally write a C header file containing all the token definitions `y.tab.h`

This file is included in the lexer and the preprocessor symbols are used in the lexer action code.

Differences with the previous lexer

- we changed the part of speech names used in the lexer, agree with the token names in the parser
- we added `return` statements to pass to the parser the token codes for the words that it reognizes
- we added a rule to mark the end of a sentence (period followed by a newline)
- we omitted the `main` routine, as it will now be provided within the parser

Yacc Parser

- similar structure to that of a lex lexer
- definition section: code block enclosed in %{ and %}
- definitions of all the tokens we expect to receive from the lexer
- we choose token names in a meaningful manner, uppercase
- the rules section is enclosed in %% and %%
- user subroutines section: `main()` calls repeatedly `yyparse()` until the lexer's input file finishes
- `yyparse()` is the C routine generated by Yacc, the parser

The Rules Section

In the Rules Section, we describe the grammar as a set of **production rules**

Rule structure: (lhs : rhs) lhs is a name, rhs is a list of symbols, action code

By default, the first rule is the highest-level rule (it contains the action executed when we parse the sentence)

User Subroutines Section

Two routines: `main()` and `yyerror()` (provided by the lexer)

Upon recognizing `subject subject`, `yyerror()` recognizes the special rule error

Lex/Yacc compilation instructions

```
lex example.l   (generates the file lex.yy.c)

yacc -d example.y   (generates the files y.tab.c y.tab.h)

cc -c lex.yy.c y.tab.c   (compiles the C files)

cc -o example lex.yy.o y.tab.o -ll
  (links the files and produces the executable example)


./example    (CNTR D to terminate)
```