About Lex

Alberto Ercolani

University of Trento

alberto.ercolani@unitn.it

November 02, 2017

Lex

- Lex is nowadays substituted by Flex: Fast Lex.
- It generates lexers whose language is described through regular expressions.
- Each regular expression can be annotated with a list of valid C statements called "semantic action".
- When the lexer matches an input text the corresponding semantic action is executed.
  - This makes the lexer more powerful than a regular automaton!
  - E.G.: It can accept $L = \{a^n b^n | n \geq 0\}$.

Lex matching characters

- Lex provides four different patterns to describe the language:
  - Strings of characters
  - Single characters
  - Classes of characters: [a-z], namely all the characters from a to z
  - A single meta character "•" matching any character but '\n'
- E.G.:
  - "int", "float", "boolean", "string"
  - 'i', 'f', 's', 'b'
  - [a-z], [a-zA-Z], [a-zA-Z_], [0-9A-F]
  - More about "•" later

Lex's regular expressions

- Let α and β be regular expressions formed by the previous patterns:
  - $\alpha \cdot \beta$ is the concatenation.
  - $\alpha|\beta$ is the alternation.
  - α^+ matches one or more repetitions.
  - α^* matches zero or more repetitions.
- These are the regular expressions you alredy do know.
- Apply recursively the definition to express any reg ex.
- E.G.:
  - "public" "static" "void"
  - "float" | "int"
  - [0-9a-z_]*
  - •"at" matches words: "cat", "rat", etc.

Lex's regular expressions, bis

- Let α and β be regular expressions formed by the previous patterns:
  - α? matches zero or one repetition.
  - $\alpha\{$n,m$\}$ | n$\leqslant$m, matches α from n to m times.
  - α\$, matches α if it appears at the end of the line.
  - ^α, matches α if it appears at the beginning of the line.
  - α/β matches α only if β follows it.
- These regular expressions are seldom used but available.

Lex's regular expressions, tris

- Let $\mathcal{C}$ be a character then:
  - [ˆ$\mathcal{C}$] is its complement.
  - [ˆCB]"at" is matched by "bat", "cat", "hat" and "fat" but not by "Cat" and "Bat".
- String complement is hard to achieve in Lex.

Lex's regular expressions, quater

- A special regular expression allows the matching of the end of file (EOF):
  - <<EOF>>
- This capability is useful when many files should be processed: in case of EOF match you can istruct the lexer to proceed on the next file.

Lex's file structure

```
%{
/* This code is copied verbatim
into the lexer's source code. */
%}
/* "Named" regular expressions here.*/
%%
/* "Anonymous" Regular expressions here.*/
%%
/* This section is copied verbatim
into the lexer's source. */
```

Lex's file structure

```
%{/* Copied verbatim in lexer's source. */%}

Type     ("int"|"float")
%%
{Type}              {/* A Semantic action.*/}
[a-z]+              {/* Another one.*/}
%%
int main(int iArgC, char**lpszArgV) {
        yylex(); // Starts lexing.
}
```

- Complying to this structure is enough for lexer generation, not compilation.

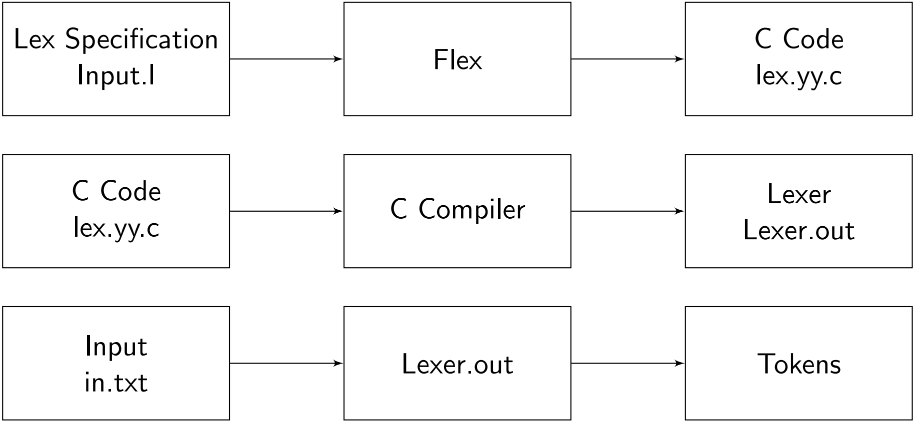
Lex's file structure

```
%{/* Copied verbatim in lexer's source. */%}

Type    ("int"|"float")
%%
{Type}          {/* A Semantic action.*/}
[a-z]+          {/* Another one.*/}
%%
int yywrap() {
        return 1;
}
int main(int iArgC, char**lpszArgV) {
        yylex(); // Starts lexing.
}
```

- "yywrap" declaration is needed. This function answers the question: When i find EOF should i stop lexing?

Generating/Compiling/Running a Lexer



- flex Input.l
- CC lex.yy.c -o Lexer.out -std=c99
- Lexer.out < In.txt

Lex's internal mechanics

- To work properly and to be of use Lex defines internally several functions, variables. Some of them are:

```
FILE* yyin;        /* Default value is stdin. */
FILE* yyout;       /* Default value is stdout. */
int yyleng;        /* Number of characters read. */
char* yytext;      /* The buffer on which
                   characters are copied
                   during pattern matching.
                   */
```

- This list will be incrementally refined, since very many of these entities are available.

Hands on!

- In the course lab section you will find:
  - simple_echo.zip
  - simple_word_recognizer.zip
  - word_counter.zip
  - simple_identifiers.zip
  - regular_expressions.zip
- These are very simple Lex specifications to get an idea.

Hands on!

- Your time:
  - Devise a lexer accepting a Windows file path.
  - Devise a lexer accepting a Linux file path.
  - Devise a lexer accepting a non regular language.

Lexer's context sensitivity

- Sometimes it is desirable to instruct the lexer to behave differently in case some lexem has been read.

Example

- “string content”
- /* comment content */
- \r
- int(x)

Lexer's context sensitivity (CS)

- Context sensitivity: having read a specific token we want the lexer to behave differently.
- How?
  - Making some regular expressions inactive.
- CS is achieved through multiple *start states*, also called *start conditions*.
- The Lexer can have many start states, exactly one is active.
- In the beginning the lexer is in a start state named "INITIAL".

Types of start states

- Start states are *exclusive* or *inclusive*.
- From an exclusive start state, only regular expressions related to it are reachable.
- From an inclusive start state, all regular expressions related to it and those unrelated to any other start state are reachable.

```
%{/**/%}

%x String        // Exclusive start states.
%s Cond          // Inclusive start states.

%%
...
%%
```

Associating rules to start states

```
%{/**/%}

%x Cond

%%
"\""              {/* Read " char. */ BEGIN Cond;}
<Cond>(^["])+     {/* Consume string content. */}
<Cond>"\""        {/* Read " char. */ BEGIN INITIAL;}
%%
```

- Associating a start state to a regular expression is simple:
    - prefix the reg ex with $<$ condition_name $>$
- To make the lexer enter a start state use BEGIN command.

Exclusive start states

```
%{/**/%}

%x String Unreacheable

%%
"\""              {/* Read " char. */ BEGIN String;}
<String>(^[ " ])+{/* Consume string content. */}
<String>"\""      {/* Read " char. */ BEGIN INITIAL;}
<Unreacheable>"end" {/* Will never be matched.*/}
%%
```

Inclusive start states

```
%{/**/%}

%s String

%%
"\""                {/* Read " char. */ BEGIN String;}
<String>(^["])+{/* Consume string content. */}
<String>"\""        {/* Read " char. */ BEGIN INITIAL;}
"end"               {/* Will be matched.*/}
%%
```

Start states stack

Start states are managed through a stack, currently three functions to manipulate it are available:

- void yy_push_state(int NewState)
  - Places the current start state on the top of the start state's stack and switches to NewState.
  - *Equivalent to BEGIN NewState;*
- void yy_pop_state()
  - Pops the top of the stack and switches to it.
  - *Equivalent to BEGIN;*
- int yy_top_state()
  - Returns the top of the stack.
  - *No meta command to do the same.*

Ambiguous specifications

- Ambiguous specifications are possible.
- Let α, β be reg exes s.t. $\mathcal{L}(\alpha) \subset \mathcal{L}(\beta)$

Example

- $\alpha =$ "int"
- $\beta =$ [a-zA-Z]$+$

- When "int" is matched, is it in α or β?

Ambiguous specifications: solution

- Lex attributes a precedence to every regular expressions: the closer to the beginning of the file, the higher the precedence.
- When a final node ϕ should be associated with a set of regular expression A := { $\alpha_{1...n}$ } the α_i s.t. $1 \leq i \leq n$ with maximum precedence is chosen and attributed.

In short

Regular expressions generating constant finite languages must be placed first, or they will be obscured.

Input consumption

- Whenever the lexer accepts a string, it fires the semantic action associated to it.
- The string is then consumed (can't be read again, no other action can be triggered by it).
- Unless, after the semantic action executed you force it to reject!

Input consumption

```
%{ int iCounter = 0; %}
%%
"abcde"          {iCounter+=1; REJECT;}
"abcd"           {iCounter+=1; REJECT;}
"abc"            {iCounter+=1; REJECT;}
"ab"             {iCounter+=1; REJECT;}
"a"              {iCounter+=1; }
.                {/*Consumes the rest.*/}
%%
int main(){
        yylex();
        /* iCounter = 5 when input is "abcde".*/
}
```

- REJECT command forces the lexer to reject and test the string on the next regular expression.

Bibliography

J. Grosch and H. Emmelmann
A tool box for compiler construction
In International Workshop on Compiler Construction (pp. 106-116). Springer, Berlin, Heidelberg

M. E. Lesk
LEX - A Lexical Analyzer Generator
Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975