

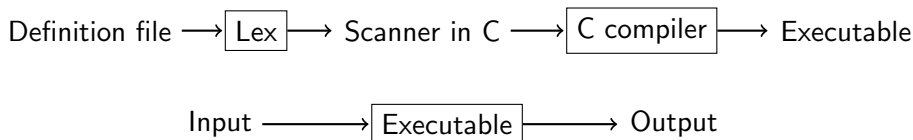
Lex

Jiří Techet Tomáš Masopust Alexander Meduna

Department of Information Systems
Faculty of Information Technology
Brno University of Technology
Božetěchova 2, Brno 61266, Czech Republic

Modern Formal Language Theory, 2007

- tool for generating scanners
- scanner described by rules in a **definition file**
- a **rule** is a pair
 - **lexical pattern** (described by regular expression)
 - **action** (written in C)
- Lex processes the definition file and outputs a **scanner** written in C
- this scanner can be compiled by a **C compiler** to produce an **executable**
- the executable processes its **input**, finds lexical patterns and executes associated actions to produce its **output**



Structure of Definition File – Example

Example

```
int num_lines = 0, num_chars = 0;
```

```
%%
```

```
\n    ++num_lines; ++num_chars;
```

```
.    ++num_chars;
```

```
%%
```

```
main()
```

```
{
```

```
    yylex();
```

```
    printf( "# of lines = %d, # of chars = %d\n",  
            num_lines, num_chars );
```

```
}
```

Structure of Definition File

- Lex definition file divided into 3 parts which are separated by **%%**:
 - 1 **definitions** – definitions of global user **variables**, **name definitions** and **start conditions**

```
int num_lines = 0, num_chars = 0;
```

- 2 **rules** – **patterns** at the beginning of a line and indented **actions** which are executed when the corresponding pattern is matched with the source. Any non-matched character is copied to the resulting file

```
\n      ++num_lines; ++num_chars;  
      .      ++num_chars;
```

- 3 **user code** – any auxiliary C function used in rules and main().
yylex() is used to start the lexical analysis

```
main()  
{  
  yylex();  
  printf( "# of lines = %d, # of chars = %d\n",  
          num_lines, num_chars );  
}
```

Name Definitions

- used to declare symbolic names for regular expressions
- are of the form **name definition**

Example

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

can be referenced as

```
{DIGIT}+"."{DIGIT}*
```

which is identical to

```
([0-9])+"."([0-9])*
```

Regular Expressions I

- patterns described by regular expressions

Regular Expressions

`x` match the character `x`

`.` any character except newline

`[xyz]` a “character class”; the pattern matches either `x`, `y`, or `z`

`[abj-oZ]` a “character class” with a range in it; matches `a`, `b`, any letter from `j` through `o`, or `Z`

`[^A-Z]` a “negated character class”, i.e., any character but those in the class. In this case, any character **except** an uppercase letter

`[^A-Z\n]` any character **except** an uppercase letter or a newline

`r*` zero or more `r`'s, where `r` is any regular expression

`r+` one or more `r`'s

Regular Expressions II

Regular Expressions

`r?` zero or one `r`'s

`r{2,5}` anywhere from 2 to 5 `r`'s

`r{2,}` 2 or more `r`'s

`r{4}` exactly 4 `r`'s

`{name}` the expansion of the `name` definition

`"[xyz]"` the literal string: `[xyz]"`

`\x` if `x` is `a`, `b`, `f`, `n`, `r`, `t`, or `v`, then the ANSI-C interpretation of `\x`. Otherwise, a literal `x` (used to escape operators such as `*`)

`\0` a NUL character (ASCII code 0)

`\123` the character with octal value 123

`\x2a` the character with hexadecimal value 2a

Regular Expressions III

Regular Expressions

(r) match an *r*; parentheses are used to override precedence

rs the regular expression *r* followed by the regular expression *s*

r|s either *r* or *s*

^r an *r*, but only at the beginning of a line

r\$ an *r*, but only at the end of a line

<s>r an *r*, but only in start condition *s*; **<s1,s2,s3>r** same, but in any of start conditions *s1*, *s2*, or *s3*

<*>r an *r* in any start condition, even an exclusive one

<<EOF>> an end-of-file

<s1,s2><<EOF>> an end-of-file when in start condition *s1* or *s2*

Regular Expressions IV

Operator precedence

- operators described above grouped by precedence – highest first
- e.g., `foo|bar*` is the same as `(foo)|(ba(r*))`

Character Class Expressions

- `[:alnum:]`, `[:alpha:]`, `[:blank:]`, `[:cntrl:]`, `[:digit:]`, `[:graph:]`, `[:lower:]`, `[:print:]`, `[:punct:]`, `[:space:]`, `[:upper:]`, `[:xdigit:]`
- set of characters equivalent to the corresponding standard C `isXXX` function (e.g., `isalnum()`)

Input Matching

Input Matching Rules

- if **no match found**, the next character from the input is copied to the output
- if **more than one match found**, the longest string is chosen
- if there are **more longest strings**, the pattern appearing first in the definition file is chosen
- associated action is executed and the remaining input is scanned for the next match

Global Variables

- can be used in actions

`yytext` matched string

`yylen` length of the matched string

Actions I

- an action is a C code which follows the associated pattern
- if no action is specified, the matched string is discarded

```
%%
```

```
/* replace sequence of tabs with space */
```

```
[\t]+          putchar( ' ' );
```

```
[\t]+$         /* ignore tabs at the EOL */
```

- if multi-line action is needed, it has to be enclosed within `{ }` or `%{ %}`
- the action `|` is used to specify the same action as the action for the next rule

Special Directives Used within Actions

ECHO copies `yytext` to the scanner's output

BEGIN used to place scanner to a start position

Special Directives Used within Actions

REJECT directs the scanner to proceed on to the second best rule which matched the input

Example

```
int word_count = 0;
%%
    /* call special() for 'frob' */
frob    special(); REJECT;
    /* count the number of words */
[^\t\n]+ ++word_count;
```

Actions III

Example

```
%%  
a      |  
ab     |  
abc    |  
abcd   ECHO; REJECT;
```

- outputs abcdabcaba for the input abcd

Special Directives Used within Actions

`yymore()` tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of `yytext` rather than replacing it

Actions IV

Example

```
%%  
mega-      ECHO; yymore();  
kludge     ECHO;
```

- outputs mega-mega-kludge for the input mega-kludge

Special Directives Used within Actions

`yylless(n)` returns all but the first `n` characters of the current token back to the input stream (will be rescanned)

`unput(c)` puts the character `c` back onto the input stream. It will be the next character scanned

`input()` reads the next character from the input stream

Start Conditions

- mechanism for conditionally activating and deactivating rules
- if a pattern is prefixed by `<sc>`, it will only be active when the scanner is in the start condition named `sc`

Example

```
<STRING>[~"]*           { /* eat up the string body ... */
```

is active only if the scanner is in the `STRING` start condition, and

```
<INITIAL,STRING,QUOTE>\. { /* handle an escape ... */
```

is active only if the scanner is either in `INITIAL`, `STRING`, or `QUOTE` condition

Start Condition Declaration

- start conditions declared in the first section
- activated by **BEGIN** action so rules with the given start condition will be active and rules with other start conditions will be inactive

Types of Start Conditions

inclusive (declared with **%s**) – also rules with no start condition are active

exclusive (declared with **%x**) – only rules with the given start condition are active (possible to define “mini-scanners” independent on the rest of the scanner)

- with the start condition **INITIAL**, only rules without start conditions are active
- **<*>** matches every start condition
- current start condition can be accessed by **YY_START**

Start Condition Example I

Example

```
%s example
```

```
%%
```

```
<example>foo    do_something();  
bar             something_else();
```

is equivalent to

```
%x example
```

```
%%
```

```
<example>foo    do_something();  
<INITIAL,example>bar    something_else();
```

Start Condition Example II

- several start conditions can be grouped

Example

- a (sub)scanner which discards C comments

```
%x comment
```

```
%%
```

```
int line_num = 1;
```

```
"/*" BEGIN(comment);
```

```
<comment>{
```

```
    [^*\n]* /* eat anything that's not a '*' */
```

```
    "*" + [^*/\n]* /* eat up '*'s not followed by '/'s */
```

```
    \n ++line_num;
```

```
    "*" + "/" BEGIN(INITIAL);
```

```
}
```

Generated Scanner

- output is written to `yyout` (`lex.yy.c` by default)
- it contains the routine `int yylex(void)` which runs the lexical analysis
- `int yylex(void)` can be changed by redefining `YY_DECL` macro

Example

```
#define YY_DECL float lexscan( float a, float b );
```

defines the scanning routine `lexscan` which takes two float parameters and returns float

- `yylex()` scans the global input file `yyin` (`stdin` by default)
- if it is not interrupted by a return statement (scanning can be resumed by calling `yylex()` again), it continues until it reaches EOF (returns 0)
- `yyrestart(FILE *)` can be used to continue scanning a new file

Command Line Options

```
flex [-bcdfhilnpstvwBFILTV78+? -C[aeFmr] -ooutput -Pprefix  
-Sskeleton] [--help --version] [file ...]
```

Selected Parameters

- o **outf** output file name
- P **pref** specifies prefix other than yy for Lex functions
- i case insensitive scanner

Options Within Definition File

- many options can be specified within the first section of the definition file
- ```
%option case-insensitive
```

# Example I

## Example

```
/* scanner for a toy Pascal-like language */

%{
/* need this for the call to atof() below */
#include <math.h>
%}

DIGIT [0-9]
ID [a-z] [a-z0-9]*

%%
```

## Example II

### Example

```
{DIGIT}+ {
 printf("An integer: %s (%d)\n", yytext,
 atoi(yytext));
}

{DIGIT}+"."{DIGIT}* {
 printf("A float: %s (%g)\n", yytext,
 atof(yytext));
}

if|then|begin|end|procedure|function {
 printf("A keyword: %s\n", yytext);
}
```

# Example III

## Example

```
{ID} printf("An identifier: %s\n", yytext);

"+"|"-"|"*"|"/"
 printf("An operator: %s\n", yytext);

"{ "[^}\n]*" }
 /* eat up one-line comments */

[\t\n]+
 /* eat up whitespace */


. printf("Unrecognized character: %s\n", yytext);

%%
```

## Example

```
main(int argc, char ** argv)
{
 ++argv, --argc; /* skip over program name */
 if (argc > 0)
 yyin = fopen(argv[0], "r");
 else
 yyin = stdin;

 yylex();
}
```

 [Flex documentation.](http://flex.sourceforge.net/manual/)  
`http://flex.sourceforge.net/manual/`