

Introduction to Lex

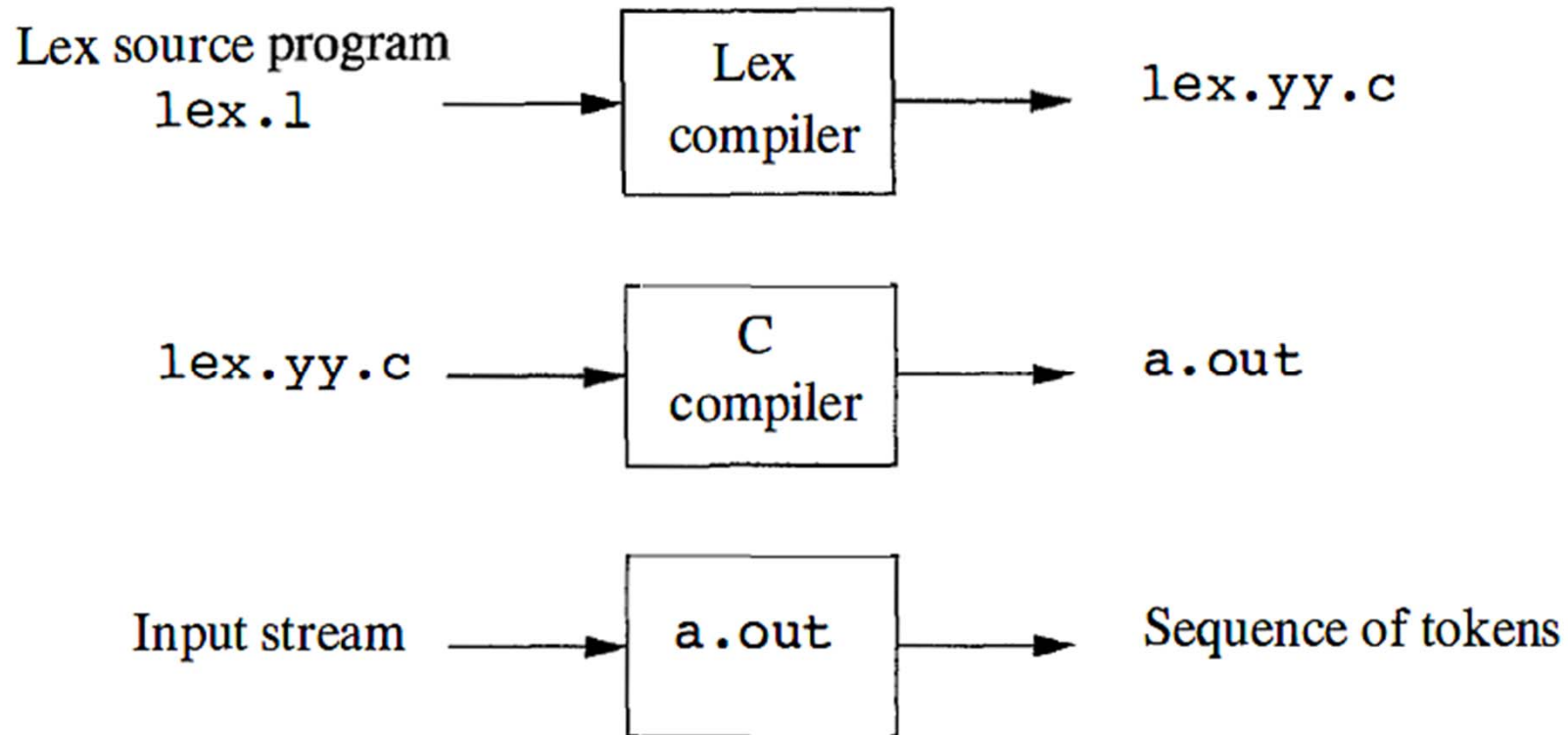
Fan Wu

Department of Computer Science and Engineering
Shanghai Jiao Tong University

Overview

- Lex is a tool for creating lexical analyzers.
- Lex source is a table of regular expressions and corresponding program fragments.
- The recognition of the expressions is performed by a deterministic finite automaton.
- As each expression appears in the input, the corresponding fragment is executed.

Usage Paradigm of Lex



Structure of Lex Programs

%{

C declarations and includes

%}

declarations

%%

translation rules

%%

user subroutines

Structure of Lex Programs Cont'd

- **%{...%}**: Anything within these brackets is copied directly to the file lex.yy.c
- **Declarations**: variables, identifiers declared to stand for a constant, and regular definitions
- **Translation rules**: pattern descriptions and actions

Pattern { Action }

- **User subroutines**: user-written codes
- Three parts are separated by **%%**

Policy for None-translated Source

- Any source not intercepted by Lex is **copied** into the generated program.
 - Any line which is not part of a Lex rule or action which begins with a **blank** or **tab**
 - Anything included between lines containing only **%{** and **%}**
 - Anything after the **second %%** delimiter

Position of Copied Source

- source input prior to the first `%%`
 - external to any function in the generated code
- after the first `%%` and prior to the second `%%`
 - appropriate place for declarations in the function generated by Lex which contains the actions
- after the second `%%`
 - after the Lex generated output

Default Rules and Actions

- The first and second part must exist, but may be empty, the third part and the second %% are optional.
- If the third part dose not contain a main(), a default main() will be linked.
- Unmatched patterns will perform a default action, which copys the input to the output

Simple Lex Source Examples

- A minimum Lex program:

%%

It only copies the input to the output unchanged.

- Deleting three spacing characters:

%%

[\t\n];

- Deleting all blanks and tabs at the ends of lines :

%%

[\t]+\$;

at the end of the line, same as /\n

A Lex Source Example

```
%{  
/*  
 * Example lex source file  
 * This first section contains necessary  
 * C declarations and includes  
 * to use throughout the lex specifications.  
 */  
#include <stdio.h>  
%}  
  
bin_digit [01]  
%%
```

A Lex Source Example Cont'd

```
{bin_digit}* {  
/* match all strings of 0's and 1's */  
/* Print out message with matching text */  
printf("BINARY: %s\n", yytext);  
}  
([ab]*aa[ab]*bb[ab]*)|([ab]*bb[ab]*aa[ab]*) {  
/* match all strings over (a,b) containing aa and bb */  
printf("AABB\n");  
}  
\n ; /* ignore newlines */
```

A Lex Source Example Cont'd

```
% %  
/*  
 * Now this is where you want your main program  
 */  
int main(int argc, char *argv[]) {  
/*  
 * call yylex to use the generated lexer  
 */  
yylex();  
/*  
 * make sure everything was printed  
 */  
fflush(yyout);  
exit(0);  
}
```

Lex Regular Expressions

- Elementary Operations
 - single characters
 - except “ \ . \$ ^ [] - ? * + | () / { } % < > ”
 - concatenation (putting characters together)
 - alternation (a|b|c)
 - [ab] == a|b
 - [a-k] == a|b|c|...|i|j|k
 - [a-z0-9] == any letter or digit
 - [^a] == any character but a
 - Kleene Closure (*)
 - Positive Closure (+)

Lex Regular Expressions Cont'd

- Special Operations
 - `.` matches any single character (except newline)
 - `"` and `\` quote the part as text
 - `\t` tab
 - `\n` newline
 - `\b` backspace
 - `\"` double quote
 - `\\` `\`
 - `?` the preceding is optional
 - `ab? == a|ab`
 - `(ab)? == ab|ε`

Lex Regular Expressions Cont'd

- Special Operations Cont'd
 - ^ at the beginning of the line
 - \$ at the end of the line, same as $\backslash n$
 - [^] anything except
 - $\backslash "[^\\"]^*\backslash "$ is a double quoted string
 - {n,m} m through n occurrences
 - $a\{1,3\}$ is a or aa or aaa
 - {definition} translation from definition
 - / matches only if followed by right part of /
 - 0/1 the 0 of 01 but not 02 or 03 or ...
 - () grouping

Regular Definitions

- NAME REGUGLAR_EXPRSSION
 - digs [0-9] +
 - integer {digs}
 - plainreal {digs} "." {digs}
 - expreal {digs} "." {digs} [Ee] [+ -] ? {digs}
 - real {plainreal} | {expreal}
- NAME must be a valid C identifier
- {NAME} is replaced by prior regular expression

Regular Definitions Cont'd

- The definitions can also contain variables and other declarations used by the Code generated by Lex.
 - These usually go at the start of this section, marked by `%{` at the beginning and `%}` at the end or the line which begins with a blank or tab .
 - Includes usually go here.
 - It is usually convenient to maintain a line counter so that error messages can be keyed to the lines in which the errors are found.
 - `%{`
 - `int linecount = 1;`
 - `%}`

Translation Rules

- Pattern `<white spaces> { program statements }`
- A null statement will ignore the input
- Four special options:
 - The unmatched token is using a default action that ECHO from the input to the output
 - | indicates that the action for this rule is from the action for the next rule
 - REJECT means going to the next alternative
 - BEGIN means entering a start condition

Transition Rule Example

```
{real}      {return FLOAT;}  
{newline} {linecounter++;}  
{integer} {  
    printf("I found an integer\n");  
    return INTEGER;  
}
```

Ambiguous Source Rules

- When more than one expression can match the current input
 - The longest match is preferred.
 - Among rules which matched the same number of characters, the rule given first is preferred.
- To override the choice, use action REJECT

```
she    {s++; REJECT;}  
he     {h++; REJECT;}  
.|\\n  ;
```

Multiple States

- Lex allows the user to explicitly declare multiple states (in Definitions section)

`%s COMMENT`

- Default states is INITIAL or 0
- Transition rules can be classified into different states, which will be matched depending on the state in
- BEGIN is used to change state

<code><INITIAL>.</code>	<code>{ECHO;}</code>
<code><INITIAL>"/**"</code>	<code>{BEGIN COMMENT;}</code>
<code><COMMENT>.</code>	<code>{:}</code>
<code><COMMENT>"/**"</code>	<code>{BEGIN INITIAL;}</code>

Lex Special Variables

- `yytext` -- a string containing the lexeme
- `yylen` -- the length of the lexeme
- `yyin` -- the input stream pointer

- Example:

```
{integer} {  
    printf("I found an integer\n");  
    sscanf(yytext,"%d", &x);  
    return INTEGER;  
}
```

将yytext以整数的形式存入x

Lex Library Function Calls

- `yylex()`
 - default `main()` contains a `return yylex();`
- `yywarp()`
 - called by lexical analyzer if end of the input file
 - default `yywarp()` always return 1
- `yyless(n)`
 - `n` characters in `yytext` are retained
- `yyomore()`
 - the next input expression recognized is to be appended to the end of this input

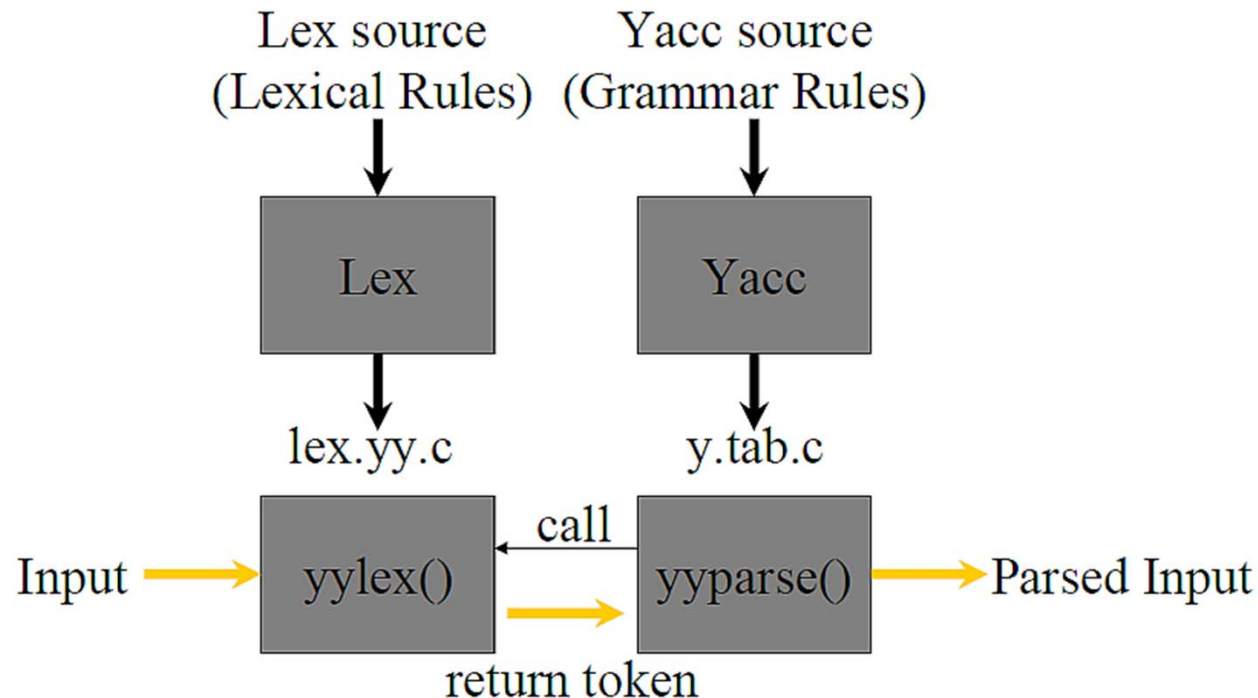
User Subroutines

- The actions associated with any given token are normally specified using statements in C. But occasionally the actions are complicated enough that it is better to describe them with a function call, and define the function elsewhere.
- Definitions of this sort go in the last section of the Lex input.

Example

```
%{ int lengs[100]; }%
%%
[a-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);
}
```

Using Yacc Together with Lex



- Yacc will call `yylex()` to get the token from the input so that each Lex rule should end with:
 `return(token);`
where the appropriate token value is returned.

Resources

- FLex Manual:
<http://flex.sourceforge.net/manual/>
- Doug Brown, John Levine, and Tony Mason, “lex & yacc”, second edition, O'Reilly.
- Thomas Niemann, “A Compact Guide to Lex & Yacc”.
- Lex/Yacc Win32 port:
<http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html>
- Parser Generator:
www.bumblebeesoftware.com