Compiler Design
Lexical Analysis
The Lexical-Analyzer Generator Lex

conf. dr. ing. Ciprian-Bogdan Chirila

chirila@cs.upt.ro

http://www.cs.upt.ro/~chirila

Outline

- Use of Lex
- Structure of Lex Programs
- Conflict Resolution in Lex
- The Lookahead Operator

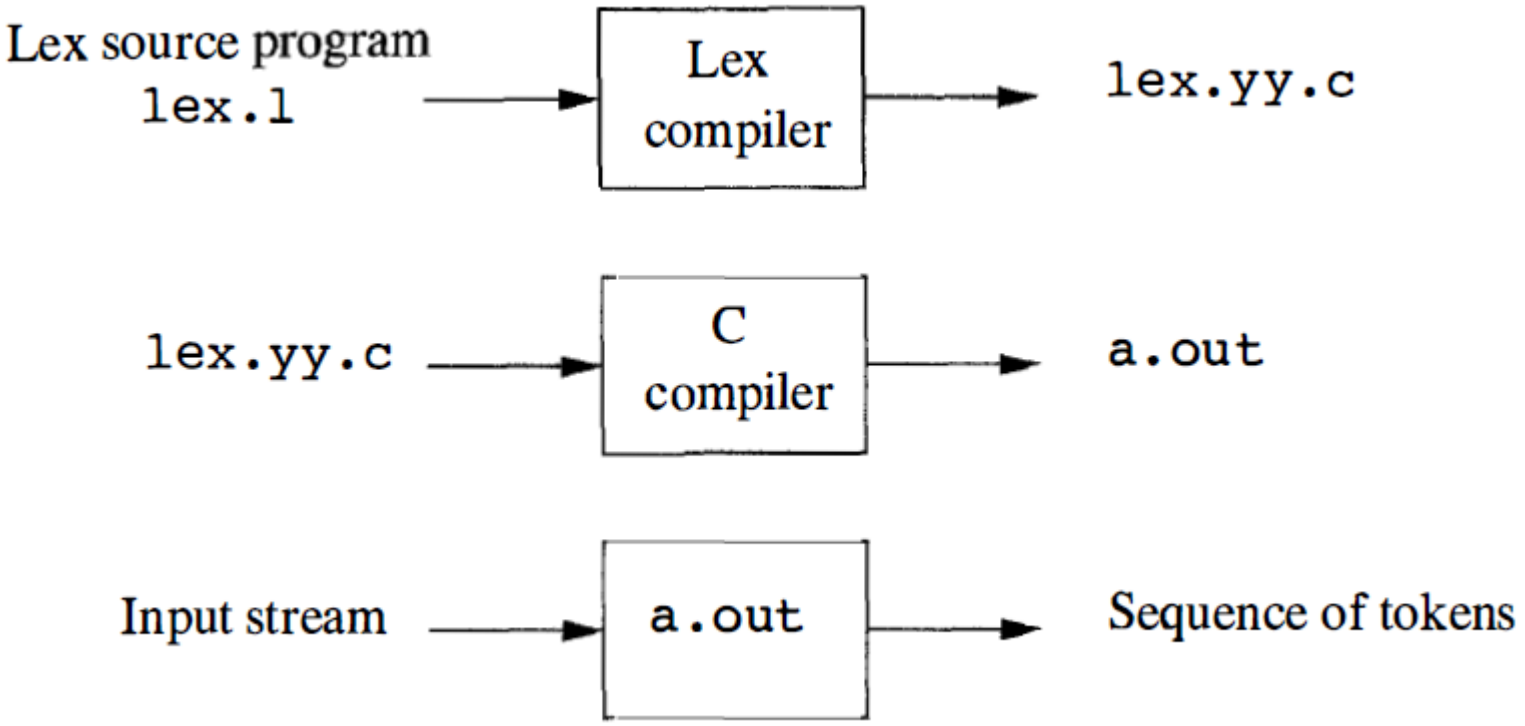
Lex

- is a lexical analyzer Generator
- Flex is a more recent implementation
- allows to specify a lexical analyzer
- by specifying regular expressions to describe patterns for tokens

Lex

- Lex language
  - the input notation for the Lex Compiler
- Lex compiler
  - transforms the input patterns into a transition diagram and generates code
  - in a file called lex.yy.c
  - simulates transition diagrams
  - transitions from regular expressions to transition diagrams is subject of other sections

Use of Lex


Lex source program
lex.l
Lex
compiler
lex.yy.c
lex.yy.c
C
compiler
a.out
Input stream
a.out
Sequence of tokens

Use of Lex

- lex.l
  - input file written in the Lex language
  - describes the lexical analyzer to be generated
- The Lex compiler
  - transforms lex.l to a C program
  - in a file that is always called lex.yy.c
- lex.yy.c
  - is compiled by the C compiler into a file called a.out
  - a working lexical analyzer that can take a stream of input characters and produce a stream of tokens

Use of Lex

- a.out
  - is a subroutine of the parser
  - is a C function that returns an integer
    - which is a code for one of the possible token names
- the attribute value
  - numeric code
  - a pointer to the symbolic table
  - or nothing
- is placed in a global variable *yylval*
- which is shared between lexical analyzer and parser
- yy refers to the Yacc parser-generator
- commonly used in conjunction with Lex

Structure of Lex Programs

```
declarations
%%
translation rules
%%
auxiliary functions
```

Structure of Lex Programs

- The declarations section includes
  - declarations of variables
  - manifest constants
    - identifiers declared to stand for a constant
    - e.g. the name of a token
  - regular definitions

Structure of Lex Programs

- The translation rules each have the form
  - **`Pattern {Action}`**
  - Each pattern is a regular expression
    - may use regular definitions from the declaration section
  - The actions are fragments of code, typically written in C
  - multiple variants of Lex were created generating code for other languages

Structure of Lex Programs

- The third section holds whatever additional functions are used in the actions
- can be compiled separately and loaded with the lexical analyzer

Structure of Lex Programs

- the lexical analyzer created by Lex behaves in concert with the parser as follows
- when called by the parser
- the lexical analyzer begins reading its remaining input
- one character at a time
- until it finds the longest prefix of the input that matches one of the patterns *Pi*

Structure of Lex Programs

- it then executes the associated action Ai
- typically, Ai will return to the parser
- but if it does not
  - e.g. because Pi describes whitespace or comments
- then the lexical analyzer proceeds to find additional lexemes
- until one of the corresponding actions causes a return to the parser

Structure of Lex Programs

- the lexical analyzer returns a single value, the token name, to the parser

- uses the shared, integer variable yylval to pass additional information about the lexeme found, if needed

Lexical rules example

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Example of Lex program Declarations

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%
```

Example of Lex program Declarations

- anything between %{ and }% will be copied directly to the file lex.yy.c
  - not treated as regular definition
- used to place manifest constants definitions
- to use C #define statements
- to associate unique integer codes with each of the manifest constants LT, IF etc.

Example of Lex program Declarations

- regular definitions use extended notation for regular expressions
- regular definitions used in later definitions or in patterns are surrounded by curly braces
  - e.g. delim is defined to be the shorthand for the character class including
    - blank
    - tab \t
    - new line \n
  - ws is defined to be one or more delimiters {delim}+

Example of Lex program Declarations

- parentheses
  - are used for grouping meta-symbols
  - do not stand for themselves
  - e.g. id and number
- E in the definition of number
  - stands for himself

Example of Lex program Declarations

- to use Lex meta-symbols like +,*,?
- to stand for themselves we must precede them with a backslash
  - e.g. we use \. in the definition of number

Example of Lex program Translation rules

```
%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%
```

Example of Lex program Translation rules

- ws has an associated empty action
- when finding a white space
  - we do not return to the parser
  - we look for another lexeme
- if – simple regular expression pattern
  - to see two letters i and f and not followed by any letter or digit
  - otherwise we see an identifier
- then, else
  - are treated similarly

Example of Lex program Translation rules

- the pattern of id is matched by keywords like if
- when the longest matching prefix matches multiple patterns
- Lex chooses whichever pattern is listed first

Example Lex program Auxiliary functions

```
%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Example Lex program Auxiliary functions

- two functions
  - installID()
  - innstallNum()
- the lines that appear between %{ and }% are copied directly to the file lex.yy.c
- may be used in the actions

Actions taken when id is matched

- to call the auxiliary function `installID()` to place the lexeme found in the symbol table
- to return a pointer to the symbol table placed in the global variable yylval
- to be used by the parser or by a later component of the compiler

Example Lex program Auxiliary functions

- the `installID()` function has available to it two variables
  - yytext if a pointer to the begin of the lexeme
    - similar to lexemeBegin
  - yylength is the length of the found lexeme
- the token name ID is returned to the parser
- the action for the number pattern is similar
  - uses the `installNum()` auxiliary function

Conflict resolution in Lex

- Rules that Lex uses to decide on the proper lexeme to select
- when several prefixes of the input match one or more patterns:
  - Always prefer a longer prefix to a shorter prefix
  - If the longest possible prefix matches two or more patterns
    - prefer the pattern listed first in the Lex program

The Lookahead Operator

- Lex automatically reads one character
  - ahead of the last character
  - that forms the selected lexeme
- then retracts the input so only the lexeme itself is consumed from the input

The Lookahead Operator

- Sometimes we want a certain pattern
  - to be matched to the input
  - only when it is followed by a certain other characters
- If so, we may use the slash / in a pattern to indicate the end of the part of the pattern that matches the lexeme

The Lookahead Operator

- what follows / is an additional pattern
- that must be matched before we can decide that the token in question was seen
- but what matches this second pattern is not part of the lexeme

Lookahead Operator Example

- in Fortran and some other languages, keywords are not reserved
- that situation creates problems, such as a statement
- `IF(I,J) = 3` where
  - **`IF`** is the name of an array
  - not a keyword
- this statement contrasts with statements of the form

```
IF ( condition ) THEN ...
```

  - where **`IF`** is a keyword.

Lookahead Operator Example

- fortunately, we can be sure that the keyword **`IF`** is always followed by a left parenthesis
  - some text - the condition - that may contain parentheses
  - a right parenthesis and
  - a letter
- thus, we could write a Lex rule for the keyword **`IF`** like

```
IF / \( .* \) {letter}
```

Lookahead Operator Example

- IF matches the two letters
- the slash announces that
  - additional pattern follows
  - will not match the lexeme
- in this pattern
  - left parenthesis
    - which is a meta-symbol
    - must be escaped with backslash
  - dot
    - any character except newline
  - dot star
    - any string without new line
  - right parenthesis
  - letter
    - regular definition representing the character class of all letters

Lookahead Operator Example

- to preprocess the input to delete the whitespaces

```
IF  (A<(B+C)*D)  THEN
```

- the first two characters match if
- the next character matches \(
- the next 9 characters match .*
- the next two match \) and letter

Bibliography

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman – Compilers, Principles, Tehcniques and Tools, Second Edition, 2007