

COMPILER CONSTRUCTION

Lex

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2017

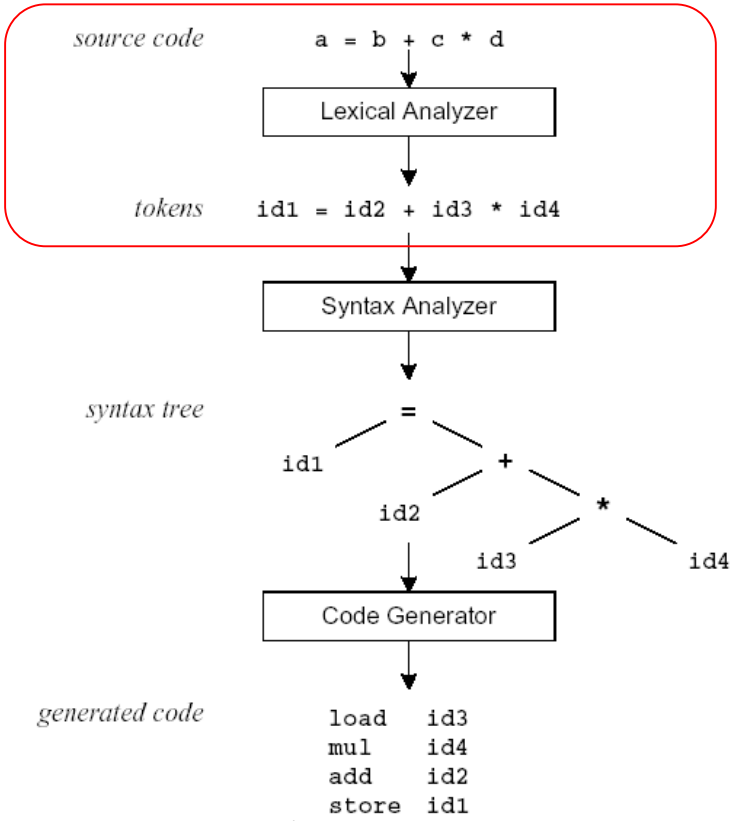
NCKU National Cheng Kung University

Courtesy of CSE4100, Lex and Yacc, CSE, University of Connecticut. Prof. Steven A. Demurjian; Lex Yacc tutorial, PLLAB, CS, NTHU. Dr. Kun-Yuan Hsieh.



Where are we?

- Lex and Yacc are able to do the following
- Here, our target is Lex





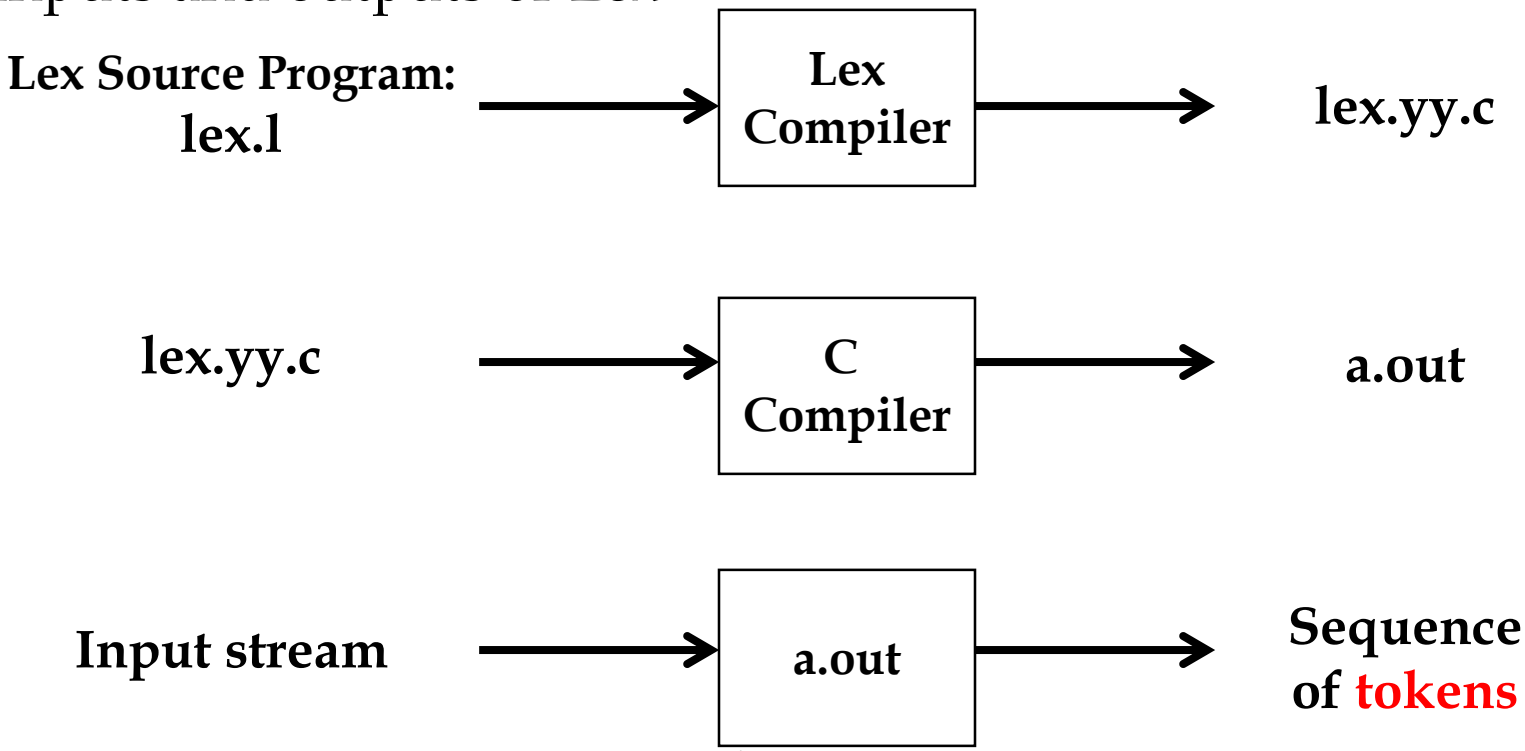
Lex and Yacc

- Two compiler writing tools:
  - Lexical Tokens and their Order of Processing (**Lex**)
  - Context Free Grammar for LALR(1) (**Yacc**)
- Both Lex and Yacc have Long History in Computing
  - Lex and Yacc – Earliest Days of Unix Minicomputers
  - Flex and Bison – From GNU
  - JFlex - Fast Scanner Generator for Java
  - BYacc/J – Berkeley
  - PCLEX and PCYACC from Abacus
  - **ANTLR**, CUP, PCYACC



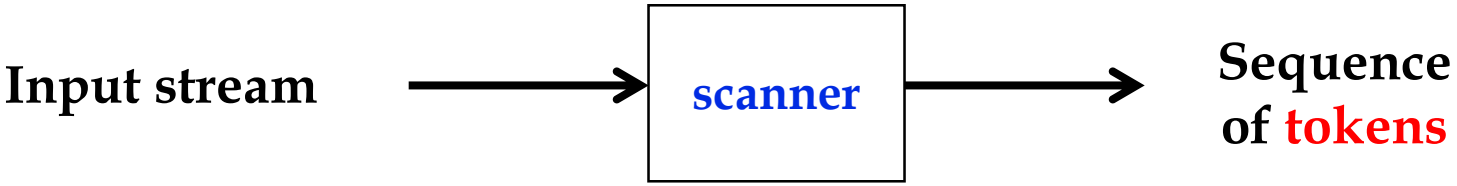
Lex – A Lexical Analyzer Generator

- A Unix Utility from early 1970s
- A Compiler that takes as source a specification for:
  - Tokens/Patterns of a Language
  - Generates a "C" Lexical Analyzer Program
- Inputs and outputs of Lex





Lex



- The main job of a *lexical analyzer (scanner)* is to break up an input stream into more usable elements (*tokens*)

```
a = b + c * d;
```
← **Input stream**

ID ASSIGN ID PLUS ID MULT ID SEMI ← **Tokens**

- Lex is an utility to help you rapidly generate your scanners



Lex (Cont'd)

- Lexical analyzers **tokenize** input streams
- Tokens are the **terminals** of a language
  - English
    - words, punctuation marks, …
  - Programming language
    - Identifiers, operators, keywords, …
- **Regular expressions define terminals/tokens**



Lex Source to C Program

- The input (.l) is translated to a C program (lex.yy.c) which
  - reads an input stream,
  - partitions the input into strings which match the given expressions, and
  - copies it to an output stream if necessary



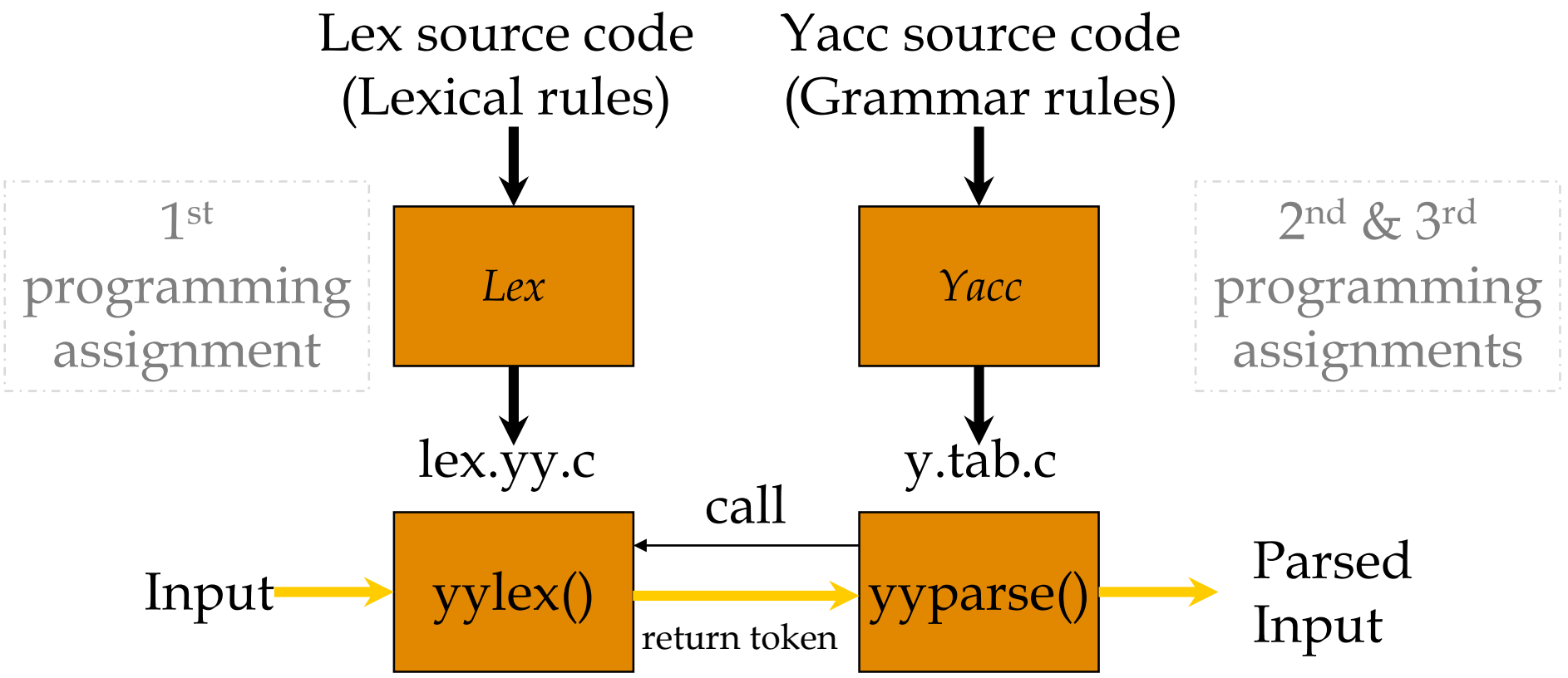
Lex vs. Yacc

- Lex
  - Generates C code for a lexical analyzer, or scanner
  - Uses **patterns** that match strings in the input and converts the strings to **tokens**

- Yacc
  - Generates C code for syntax analyzer, or parser
  - Uses **grammar rules** that allow it to analyze tokens from Lex and create a **syntax tree**

- **Together they form a compiler**

Lex & Yacc


Lex source code
(Lexical rules)
Yacc source code
(Grammar rules)
1st programming assignment
Lex
Yacc
2nd & 3rd programming assignments
lex.yy.c
y.tab.c
call
Input
yylex()
yyparse()
Parsed Input
return token



Format of a Lexical Specification

- lex.l is divided into 3 parts:

1. Declarations:
   - Defs, Constants, Types, #includes, etc. that can occur in **a C Program**
   - Regular Definitions (expressions)
2. Translation rules:
   - Pairs of (**Regular Expression**, **Action**)
   - Informs lexical analyzer of action when pattern is recognized
3. Auxiliary procedures:
   - Designer Defined **C Code**
   - E.g., symbol table codes

lex.l file format:

DECLARATIONS
%%
TRANSLATION RULES
%%
AUXILIARY PROCEDURES

lex.l Example



```
%{
#define  T_IDENTIFIER  300
#define  T_INTEGER     301
#define  T_REAL        302
#define  T_STRING      303
#define  T_ASSIGN      304
#define  T_ELSE        305
#define  T_IF          306
#define  T_THEN        307
#define  T_EQ          308
#define  T_LT          309
#define  T_NE          310
#define  T_GE          311
#define  T_GT          312
%}

letter          [a-zA-Z]
digit           [0-9]
ws              [ \t\n]+
id              [A-Za-z][A-Za-z0-9]*
comment         "(*"([^*]|\n|"*"+[^)])*"*"+")"
integer         [0-9]+/([^0-9]|"..")
real            [0-9]+"."[0-9]*([0-9]|"E"[+-]?[0-9]+)
string          \'([^']|\'\')*\'
%%

":="            {printf(" %s ", yytext);return(T_ASSIGN);}
"else"          {printf(" %s ", yytext);return(T_ELSE);}
```

User defined values to each token (else lex will assign)

Regular expression rules for later token definitions

Token definitions and action

lex.l Example



```
"then"          {#ifdef PRNTFLG
                 printf(" %s ", yytext);
                 #endif
                 return(T_THEN);
                }
"<="            {printf(" %s ", yytext);return(T_EQ);}
"<"             {printf(" %s ", yytext);return(T_LT);}
"<>"            {printf(" %s ", yytext);return(T_NE);}
">="            {printf(" %s ", yytext);return(T_GE);}
">"             {printf(" %s ", yytext);return(T_GT);}

{id}            {printf(" %s ", yytext);return(T_IDENTIFIER);}
{integer}       {printf(" %s ", yytext);return(T_INTEGER);}
{real}          {printf(" %s ", yytext);return(T_REAL);}
{string}        {printf(" %s ", yytext);return(T_STRING);}
{comment}       {/* T_COMMENT */}
{ws}            {/* spaces, tabs, newlines */}
%%

yywrap()        {return 0;}

main()
{
  int i;
  do {
    i = yylex();
  } while (i!=0);
}
```

Conditional compilation action

Token definitions and action

Discard

EOF for input

Three lex&yacc variables:

1. **yytext** = "currenttoken"
2. **yylen** = 12
3. **yylval** = 300



Internal Variables in Lex

- You may find the variables useful when writing translation rules
- char ***yytext**;
  - Pointer to current lexeme terminated by '\0'
- int **yylen**;
  - Number of chacters in yytex but not '\0'
- **yylval**:
  - Global variable through which the token value can be returned to Yacc
  - Parser (Yacc) can access yylval, yylen, and yytext
- How are these used?

  Consider **integer tokens**:

  yylval = ascii_to_integer (yytext);

  → Conversion from *string* to actual *integer value*



Internal Variables in Lex (Cont'd)

- FILE ***yyin**
  - The input of the lex, pointing to the current file position
  - Default is set to stdin
- FILE ***yyout**
  - The output of the lex program
  - Default is set to stdout
- **yylineno**
  - The current line number of yyin



Lex Library Routines

- **yylex()**
  - The default main() contains a call of yylex()
- **yymore()**
  - return the next token
- **yyless(n)**
  - retain the first n characters in yytext
- **yywarp()**
  - is called whenever Lex reaches an end-of-file
  - The default yywarp() always returns 1



Lex Regular Expressions (Extended Regular Expressions)

- A regular expression matches a set of strings
- Regular expression
  - Operators
  - Character classes
  - Arbitrary character
  - Optional expressions
  - Alternation and grouping
  - Context sensitivity
  - Repetitions and definitions



Operators

“ \ [ ] ^ - ? . * + | ( ) $ / { } % < >

- Considered as the meta-character of the regular expressions used in Lex
- If they are to be used as **text characters**, an escape should be used

  \$ = “$”

  \\ = “\”
- Every character is always a text character, except *blank*, *tab* (\t), *newline* (\n) and the list above



Character Classes [ ]

- `[abc]`
  - matches **a single character**, which may be `a`, `b`, **or** `c`
- Every operator meaning is ignored
  - except `\` `-` **and** `^`
- Examples:

`[ab]`	=> `a` or `b`
`[a-z]`	=> `a` or `b` or `c` or … or `z`
`[-+0-9]`	=> all the digits and the two signs
`[^a-zA-Z]`	=> any character which is not a letter



Arbitrary Character .

- The operator character .
  - is the class of all characters, except newline

An escape character example:

- [\40-\176]
  - matches all printable characters in the ASCII character set, from 40_{octal} (space) to 176_{octal} (tilde~)



Optional & Repeated Expressions

- a? => zero or one instance of a
- a* => zero or more instances of a
- a+ => one or more instances of a

- Examples:

  ab?c => ac or abc

  [a-z]+ => all strings of lower case letters

  [a-zA-Z][a-zA-Z0-9]*

  => all alphanumeric strings with a leading alphabetic character



Precedence of Operators

- Level of precedence
  1. Kleene closure (*), ?, +
  2. concatenation
  3. alternation (|)
- All operators are **left associative**
- Ex: `a*b|cd*` = `((a*)b)|(c(d*))`



Pattern Matching Primitives

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line / complement
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
[ab]	a or b
a{3}	3 instances of a
“a+b”	literal “a+b” (C escapes still work)



Regular Expression and its Action

```
// Input stream
a = b + c;
```

```
// Input stream
a = b + c;

// Output
a operator: ASSIGNMENT b + c;
```

```
…
%%
<regexp> <action>
<regexp> <action>
…
%%
```

```
%%
[abcd]          {printf("%s ", yytext);}
"="             {printf("operator: ASSIGNMENT");}
…
```



Transition Rules

- regexp <one or more blanks> action (C code);
- regexp <one or more blanks> { actions (C code) }

- A null statement ; will ignore the input (no actions)
  - Example:

    [\t\n] ;

    Causes the three spacing characters to be ignored

```
// Input stream
a = b + c;
d = b * c;
↓ ↓
// Corresponding output for the rule
a=b+c;d=b*c;
```



Transition Rules (cont'd)

- Four special options for actions:
  |, **ECHO;, BEGIN**, and **REJECT;**
- | indicates that the action for this rule is from the action for the next rule
  - [\t\n] ;
  - " " |
    "\t" |
    "\n" ; ← **the three applies the same rule**
- The **unmatched token** is using a default action: **ECHO** from the input to the output



Transition Rules (cont'd)

- **REJECT**
  - Go do the next alternative rule
  - It causes whatever rule was second choice after the current rule to be executed

- The matching rules depend on the tool you use:
  - Matching rules for Flex:
  1. Match the longest possible token
  2. Of the tokens with the same length, prefer the pattern **earlier** in the source file



Transition Rules (cont'd)

- **BEGIN**
  - For conditional rules

BEGIN name1;

- Executing the action statement enters a start condition, which changes the start condition to **name1**

BEGIN 0;

- Executing the action statement resets the initial condition of the Lex automaton interpreter

Example:

```
%START AA BB CC
%%
^a              {ECHO; BEGIN AA;}
^b              {ECHO; BEGIN BB;}
^c              {ECHO; BEGIN CC;}
\n              {ECHO; BEGIN 0;}
<AA>magic         printf("first");
<BB>magic         printf("second");
<CC>magic         printf("third");
```



Usage

- To run Lex on a source file, type
  lex scanner.l
- It produces a file named lex.yy.c which is a C program for the lexical analyzer
- To compile lex.yy.c, type
  cc lex.yy.c –ll
- To run the lexical analyzer program, type

  ./a.out < inputfile



Versions of Lex

- Versions:
  - AT&T Lex
    http://www.combo.org/lex_yacc_page/lex.html
  - GNU Flex
    https://github.com/westes/flex
  - Win32 version of Flex:
    http://gnuwin32.sourceforge.net/packages/flex.htm
  - Lex on Cygwin :
    https://www.cygwin.com/
- Each Lex implementation may has its own **character**
  - Please refer to the online manual for Lex on The Lex & Yacc Page



QUESTIONS?



A lex.l example for tokens in **Pascal**

A Pascal lex.l



```
%{
#include "y.tab.h"
%}

letter                  [a-zA-Z]
digit                   [0-9]

ws                      [ \t\n]+
id                      [A-Za-z][A-Za-z0-9]*
comment                 "(*"([^*]|\n|"*"+[^)])*"*"+")"
integer                 [0-9]+/([^0-9]|"..")
real                    [0-9]+"."[0-9]*([0-9]|"E"[+-]?[0-9]+)
string                  \'([^']|\'\')*\'

%%

":="                    {return(T_ASSIGN);}
":"                     {return(T_COLON);}
"array"                 {return(T_ARRAY);}
"begin"                 {return(T_BEGIN);}
"case"                  {return(T_CASE);}
"const"                 {return(T_CONST);}
"downto"  {return(T_DOWNTO);}
"do"                    {return(T_DO);}
"else"                  {return(T_ELSE);}
"end"                   {return(T_END);}
"file"                  {return(T_FILE);}
"for"                   {return(T_FOR);}
```

A Pascal lex.l (Cont'd)



```
"function" {return(T_FUNCTION);}
/*  "goto"          {return(T_GOTO);} */
"if"                {return(T_IF);}
"label"             {return(T_LABEL);}
"nil"               {return(T_NIL);}
"not"               {return(T_NOT);}
"of"                {return(T_OF);}
/* "packed"         {return(T_PACKED);} */
"procedure"         {return(T_PROCEDURE);}
"end"               {return(T_END);}
"program" {return(T_PROGRAM);}
"record"  {return(T_RECORD);}
"repeat"  {return(T_REPEAT);}
"set"               {return(T_SET);}
"then"              {return(T_THEN);}
"to"                {return(T_TO);}
"type"              {return(T_TYPE);}
"until"             {return(T_UNTIL);}
"var"               {return(T_VAR);}
"while"             {return(T_WHILE);}
/* "with"   {return(T_WITH);} */
"+"                 {return(T_PLUS);}
"-"                 {return(T_MINUS);}
"or"                {return(T_OR);}
"and"               {return(T_AND);}
"div"               {return(T_DIV);}
"mod"               {return(T_MOD);}
"/"                 {return(T_RDIV);}
```

A Pascal lex.l (Cont'd)



```
"*"                 {return(T_MULT);}
"("                 {return(T_LPAREN);}
")"                 {return(T_RPAREN);}
"="                 {return(T_EQ);}
","                 {return(T_COMMA);}
".."                {return(T_RANGE);}
"."                 {return(T_PERIOD);}
"["                 {return(T_LBRACK);}
"]"                 {return(T_RBRACK);}
"<="                {return(T_EQ);}
"<"                 {return(T_LT);}
"<>"                {return(T_NE);}
">="                {return(T_GE);}
">"                 {return(T_GT);}
"in"                {return(T_IN);}
"^"                 {return(T_UPARROW);}
";"                 {return(T_SEMI);}

{id}                {return(T_IDENTIFIER);}
{integer}   {return(T_INTEGER);}
{real}              {return(T_REAL);}
{string}    {return(T_STRING);}
{comment}           {/* T_COMMENT */}
{ws}                {/* spaces, tabs, newlines */}
```