Specifying lexers

Several tools have been built for constructing lexical analysers from special-purpose notations based on regular expressions.

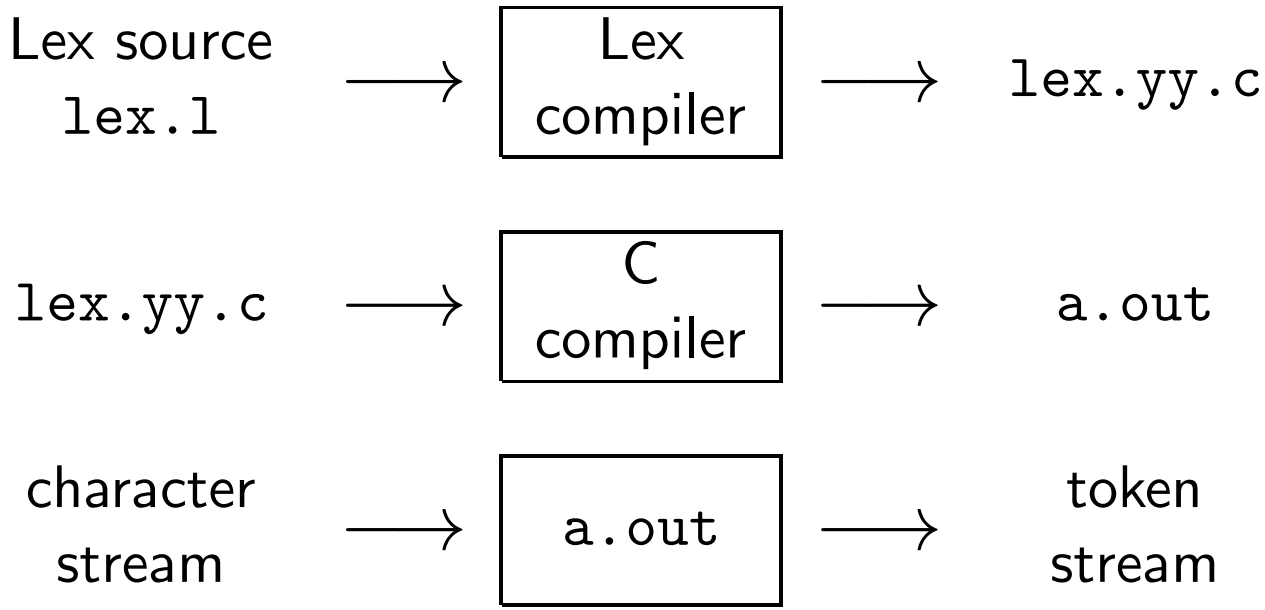
We shall now describe such a tool, named Lex, which is widely used in software projects developed in C.

Using this tool shows how the specification of patterns using regular expressions can be combined with actions, e.g., making entries into a symbol table, that a lexer may be required to perform.

We refer to the tool as the **Lex compiler** and to its input specification as the **Lex language**.

Specifying lexers (cont)

Lex is generally used in the following manner:



Specifying lexers/Lex specifications

A Lex specification (or source or program) consists of three parts:

```
declarations
%%
translation rules
%%
user code
```

The **declarations section** includes declarations of C variables, constants and regular definitions. The latter are used in the translation rules.

Specifying lexers/Lex specifications (cont)

The **translation rules** of a Lex program are statements of the form

$$\begin{array}{ll} p_1 & \{action_1\} \\ p_2 & \{action_2\} \\ \ldots & \quad \ldots \\ p_n & \{action_n\} \end{array}$$

where each p_i is a regular expression and each $action_i$ is a C program fragment describing what action the lexer should take when pattern p_i matches a lexeme.

The third section holds whatever **user code** (auxiliary procedures) are needed by the actions.

Specifying lexers/Lex specifications (cont)

A lexer created by Lex interacts with a parser in the following way:

1. the parser calls the lexer;
2. the lexer starts reading its current input characters;
3. when the longest prefix of the input matches a regular expression p_i, the corresponding $action_i$ is executed;
4. finally, two cases occur whether $action_i$ returns control to the parser or not:
   - 4.1 if so, the lexer returns the recognised token and lexeme;
   - 4.2 if not, the lexer forgets about the recognised word and go to step 2.

Specifying lexers/Lex declarations section

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

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

%%
```

Specifying lexers/Lex declarations section (cont)

First, we see a place for the declaration of the tokens. Depending on the parser, if any, used with Lex, these token may be declared by the parser. In this case, they are not declared here.

These declarations are surrounded by `%{` and `%}`. Anything between these brackets is copied verbatim in `lex.yy.c`.

Second, we see a series of regular definitions. Each definition consists of a name and a regular expression denoted by that name.

For instance, `delim` stands for the **character class** `[ \t\n]`, that is, any of the three characters: blank, tabulation (`\t`) or newline (`\n`).

Specifying lexers/Lex declarations section (cont)

Character classes. If we want to denote a set of letters or digits, it is often long to enumerate all the elements, like the **digit** regular expression.

So, instead of `4` | `1` | `2` we would shortly write [142].

If the characters are consequently ordered, we can use **intervals**, called in Lex *character classes*.

For instance we write [a-c] instead of `a` | `b` | `c`.

Or [0-9] instead of `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`.

We can now describe identifiers in a very compact way:

$$[\texttt{A-Za-z}][\texttt{A-Za-z0-9}]^\star$$

Specifying lexers/Lex declarations section (cont)

It is possible to have `]` and `-` in a character range: the character `]` must be first and `-` must be first or last.

The second definition is of white space, denote by the name `ws`. Note that we must write `{delim}` for **delim**, with braces inside regular expressions in order to distinguish it from the pattern made of the five letters `delim`.

The definitions of `letter` and `digit` illustrate the use of character classes (interval of (ordered) characters).

The definition of `id` shows the use of some Lex special symbols (or **metasymbols**): parentheses and vertical bar.

Specifying lexers/Lex declarations section (cont)

The definition of `num` introduces a few more features.

There is another metasymbol "?" with the obvious meaning.

We notice the use of a backslash to make a character mean itself instead of being interpreted as a metasymbol: `\.` means "the dot character", while `.` (metasymbol) means "any character." This works with any metasymbol.

Note finally that we wrote `[+\-]` because, in this context, the character "-" has the meaning of "range", as in `[0-9]`, so we must add a backslash. This action is called **to escape** (a character).

Another way of escaping a character is to use double-quotes around it, like `"."`

Specifying lexers/Lex translation rules

```
%%
{ws}      { /* no action and no return */ }
if        { return IF; }
then      { return THEN; }
else      { return ELSE; }
{id}      { yylval = lexeme(); return ID; }
{number}  { yylval = lexeme(); return NUM; }
"<"       { return LT; }
"<="      { return LE; }
"="       { return EQ; }
"<>"      { return NE; }
">"       { return GT; }
">="      { return GE; }
```

Specifying lexers/Lex translation rules (cont)

The translation rules follow the first %%.

The first rule says that if the regular expression denoted by the name `ws` maximally matches the input, we take no action. In particular, we do not return to the parser. Therefore, by default, this implies that the lexer will start again to recognise a token after skipping white spaces.

The second rule says that if the letters `if` are seen, return the token `IF`.

In the rule for `{id}`, we see two statements in the action. First, the Lex predefined variable `yylval` is set to the lexeme and the token `ID` is returned to the parser. The variable `yylval` is shared with the parser (it is defined in `lex.yy.c`) and is used to pass attributes about the token.

Specifying lexers/User code

Contrary to our previous presentation, the procedure `lexeme` takes here no argument. This is because the input buffer is directly and globally accessed in Lex through the pointer `yytext`, which corresponds to the first character in the buffer when the analysis started for the last time.

The length of the lexeme is given via the variable `yyleng`.

We do not show the details of the auxiliary procedures but the trailing section should look like

```
%%
char* lexeme () {
  /* returns a copy of the matched string
     between yytext[0] and yytext[yyleng-1] */
}
```

Specifying lexers/Lex longest-prefix match

If several regular expressions match the input, Lex chooses the rule which matches the most text. This is why the input `if123` is matched (recognised) as an identifier and not as the two tokens keyword (`if`) and number (123).

If Lex finds two or more matches of the same length, the rule listed *first* in the Lex input file is chosen.

That is why we listed the patterns `if`, `then` and `else` before `{id}`. For example, the input `if` is matched by `if` and `{id}`, so the first rule is chosen, and since we want the token keyword **if**, its regular expression is written *before* `{id}`.

Specifying lexers/Example

It is possible to use Lex alone. For instance, let `count.l` be the Lex specification

```
%{
int char_count=1, line_count=1;
%}
%%
.  {char_count++;}
\n {line_count++; char_count++;}
%%
int main () {
  yylex();  /* Calls the lexer */
  printf("There were %d characters in %d lines.\n",
          char_count,line_count);
  return 0;
}
```

Specifying lexers/Example (cont)

We have to compile the Lex specification into C code, then compile this C code and link the object code against a special library named `l`:

```
lex -t count.l > count.c
gcc -c -o count.o count.c
gcc -o counter count.o -ll
```

We can also use the C compiler `cc` with the same options instead of `gcc`. The result is a binary `counter` that we can apply on `count.l` itself:

```
cat count.l | counter
There were 210 characters in 12 lines.
```

Specifying lexers/Example (cont)

We can extend the previous specification to count words as well. For this, we need to define a regular expression for letters and bind it to a name, at the end of the declarations.

```
%{
int char_count=1, line_count=1, word_count=0;
%}
letter [A-Za-z]
%%
{letter}+ { word_count++; char_count += yyleng;
            printf ("[%s]\n",yytext); }
.         { char_count++; }
\n        { line_count++; char_count++; }
%%
...
```

Specifying lexers/Example (cont)

We can also use more regular expressions with names.

```
letter [A-Za-z]
digit  [0-9]
alpha  ({letter}|{digit})      /* No space inside! */
id     {letter}([_]*{alpha})*  /* No space inside! */
%%
{id} { word_count++; char_count += yyleng;
       printf ("word=[%s]\n",yytext); }
.    { char_count++; }
\n   { line_count++; char_count++; }
```

Specifying lexers/Example (cont)

By default, if there is no parser and no explicit `main` procedure, Lex will add one in the produced C code as if it were given in the user code section (at the end of the specification) as

```
int main ()
{
 yylex();
 return 0;
}
```