Lab 2

Lexical Analysis

Goals

In this lab you will learn:

1. What a **T**ranslator is
2. What are the main components of a **T**ranslator
3. How a Lexical Analyzer works
4. Fundamentals of LEX

Resources

Table 2.1: Lab Resources

Resource	Link
Crafting Interpreters - "A Map of the Territory"	https://craftinginterpreters.com/a-map-of-the-territory.html
Crafting Interpreters - "Scanning"	https://craftinginterpreters.com/scanning.html
The Dragon Book - Chapters 1-3	http://ce.sharif.edu/courses/94-95/1/ce414-2/resources/root/Text%20Books/Compiler%20Design/Alfred%20V.%20Aho,%20Monica%20S.%20Lam,%20Ravi%20Sethi,%20Jeffrey%20D.%20Ullman-Compilers%20-%20Principles,%20Techniques,%20and%20Tools-Pearson_Addison%20Wesley%20(2006).pdf

2.1 A brief overview on translators

Translator are programs that can be used to convert a program written in one language to another.

2.1.1 Types

There are various types of translators such as compilers, interpreters and assemblers.

1. A *compiler* converts the entire high-level language source code into machine code. If there are any syntax or semantic error, the program will not execute. As the compiler checks the whole program, the scanning time is high but the execution time is lower. Therefore, compiler-based languages such as C, C++ are considered as fast languages.

   Compiling is an implementation technique that involves translating a source language to some other—usually lower-level—form. When you generate bytecode or machine code, you are compiling. When we say a language implementation "is a compiler", we mean it translates source code to some other form but doesn't execute it. The user has to take the resulting output and run it themselves.

2. Conversely, when we say an implementation "is an *interpreter*", we mean it takes in source code and executes it immediately. It runs programs "from source".

2.1.2 Parts

1. **Scanning**

   The first step of any translator is scanning, also known as lexing, or lexical analysis. A scanner (or lexer) takes in the linear stream of characters and chunks them together into a series of something more akin to "words". In programming languages, each of these words is called a token. Some tokens are single characters, such as (and ,. Others may be several characters long, like numbers (123), string literals ("hi!"), and identifiers (min).

   The notion "Lexical" comes from the Greek root "lex", meaning "word".

   

   Note 2.1.1

   Some characters in a source file don't actually mean anything. Whitespace is often insignificant, and comments, by definition, are ignored by the language. The scanner usually discards these, leaving a clean sequence of meaningful token

2. **Parsing**

   The second step of a translator is parsing. This is where our syntax gets a grammar—the ability to compose larger expressions and statements out of smaller parts. Did you ever diagram sentences in a Romanian/English class? If so, you've done what a parser does, except that English has thousands and thousands of "keywords" and quite a lot of ambiguity. Programming languages are much simpler.

   A parser takes the flat sequence of tokens and builds a tree structure that mirrors the nested nature of the grammar. These trees have a couple of different names—parse tree or abstract syntax tree—depending on how close to the bare syntactic structure of the source language they are. In practice, these are usually called syntax trees.

Parsing has a long, rich history in computer science that is closely tied to the artificial intelligence community. Many of the techniques used today to parse programming languages were originally conceived to parse human languages by AI researchers who were trying to get computers into conversations.

It turns out human languages were too difficult for the rigid grammars those parsers could handle, but they were a perfect fit for the simpler artificial grammars of programming languages. Still, humans still manage to use simple grammars incorrectly, so the parser's job also includes letting us know when we do by reporting syntax errors.

2.2 Lexical Analysis

The first step in any compiler or interpreter is scanning. The scanner takes in raw source code as a series of characters and groups it into a series of chunks we call tokens. These are the meaningful "words" and "punctuation" that make up the language's grammar.

Lexical Analysis converts source code into a sequence of symbols that are relevant from a semantic perspective. These symbols, which are often the output of the lexical analizer define a middleware language.

Note 2.2.1

Any lexical analyzer has some general functions for counting lines, generating lists, trimming spaces or ignoring comments.

Let's look at a line of code:

Listing 2.2.1: C code

```
var language = "C";
```

Here, var is the keyword for declaring a variable. That three-character sequence "v-a-r" means something. But if we extract three letters out of the middle of language, like "g-u-a", those don't mean anything on their own.

That's what lexical analysis is about - scanning through the list of characters and grouping them together into the smallest sequences that represent something. Each of these groups of characters is called a lexeme. In the given line of code, the lexemes are: 'var', 'language', '=', 'C', ';'

The lexemes are only the raw substrings of the source code. However, in the process of grouping character sequences into lexemes, we also stumble upon some other useful information. When we take the lexeme and bundle it together with that other data, the result is a token. It includes useful information like:

1. **Keywords**

   Keywords are part of the shape of the language's grammar, so the parser often has code such as, "If the next token is while then do ..." That means the parser wants to know not just that it has a lexeme for some identifier, but that it has a reserved word, and which keyword it is.

2. **Lexeme Types** The parser could categorize tokens from the raw lexeme by comparing the strings, but that is not the prefered approach. Instead, at the point that we recognize a lexeme, we also remember which kind of lexeme it represents. We have a different type for each keyword, operator, bit of punctuation, and literal type.

3. **Literal values**

   There are lexemes for literal values—numbers and strings and the like. Since the scanner has to walk each character in the literal to correctly identify it, it can also convert that textual representation of a value to the living runtime object that will be used by the interpreter later.

Let's recap how the scanner and parser can work together.

The core of the scanner is a loop. Starting at the first character of the source code, the scanner figures out what lexeme the character belongs to, and consumes it and any following characters that are part of that lexeme. When it reaches the end of that lexeme, it emits a token. Then it loops back and does it again, starting from the very next character in the source code. It keeps repeating this process, "eating" characters and occasionally, outputing tokens, until it reaches the end of the input.

The part of the loop where we look at a handful of characters to decide which kind of lexeme it "matches" should sound familiar now that you worked a little with regular expressions. For instance, you might consider defining a regex for each kind of lexeme and using those to match characters. For example, if we would like to write down a rule for C language identifiers (variable names), we could use this regex:

Listing 2.2.2: regex sample — C code

```
[a-zA-Z_][a-zA-Z_0-9]*
```

The rules that determine how a particular language groups characters into lexemes are called its lexical grammar. Tools like Lex are designed expressly to let you define regexes, and they give you a complete scanner back.

Note 2.2.2

In C, as in most programming languages, the rules of their grammar are generally not simple enough for the language to be classified a regular language (the most restricted language form in Chomsky's hierarchy).

2.3 LEX

2.3.1 Introduction

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes regular expressions.

The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. At the boundaries between strings program sections provided by the user are executed. The Lex source file associates the regular expressions and

the program fragments. As each expression appears in the input to the program written by Lex, the corresponding fragment is executed.

Lex turns the user's expressions and actions (called source in this memo) into the host general-purpose language; the generated program is named *yylex*. The *yylex* program will recognize expressions in a stream (called input in this memo) and perform the specified actions for each expression as it is detected.

The *yylex()* function can be viewed as the implementation for a finite automaton which scans for all rules at once and executes the desired rule action. We generally use lex to associate terminal symbols with specific tokens that a parser could further use. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator, such as Yacc, assigns structure to the resulting pieces via the *yyparse()* function.

Input → yylex() → yyparse() → Syntactically analyzed input

The automaton is interpreted, rather than compiled, in order to save space. The result is still a fast analyzer. In particular, the time taken by a Lex program to recognize and partition an input stream is proportional to the length of the input. The number of Lex rules or the complexity of the rules is not important in determining speed, unless rules which include forward context require a significant amount of rescanning. What does increase with the number and complexity of rules is the size of the finite automaton, and therefore the size of the program generated by Lex.

In the program written by Lex, the user's fragments (representing the actions to be performed as each regular expression is found) are gathered as cases of a switch. The automaton interpreter directs the control flow. Opportunity is provided for the user to insert either declarations or additional statements in the routine containing the actions, or to add subroutines outside this action routine.

2.3.2 Lex Source

The general format of Lex source is:

Listing 2.3.1: sample.lex — C code

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The second %% is optional, but the first is required to mark the beginning of the rules. The absolute minimum Lex program is thus

Listing 2.3.2: sample.lex — C code

```
%%
```

(no definitions, no rules) which translates into a program which copies the input to the output unchanged.

In the outline of Lex programs shown above, the rules represent the user's control decisions; they are a table, in which the left column contains regular expressions and the right column

contains actions, program fragments to be executed when the expressions are recognized. An individual rule may look as:

Listing 2.3.3: sample.lex — C code

```
integer   printf("found keyword INT");
```

This rule would look for the string integer in the input stream and print the message *"found keyword INT"* whenever it appears. In this example the host procedural language is C and the C library function printf is used to print the string. The end of the expression is indicated by the first blank or tab character. If the action is merely a single C expression, it can just be given on the right side of the line; if it is compound, or takes more than a line, it should be enclosed in braces.

As a slightly more useful example, suppose it is desired to change a number of words from British to American spelling. Lex rules such as

Listing 2.3.4: sample.lex — C code

```
colour      printf("color");
mechanise   printf("mechanize");
petrol      printf("gas");
```

would be a start. These rules are not quite enough, since the word petroleum would become gaseum; a way of dealing with this will be described later.

Note 2.3.1

Remember that Lex is turning the rules into a program. Any source not intercepted by Lex is copied into the generated program.

When writing rules in lex, one can define names to be associated with specific translations. The name and translation must be separated by at least one blank or tab, and the name must begin with a letter. The translation can then be called out by the name syntax in a rule. Using D for the digits and E for an exponent field, for example, might abbreviate rules to recognize numbers:

Listing 2.3.5: num.lex — C code

```
D                   [0-9]
E                   [DEde][-+]?{D}+
%%
{D}+                printf("integer");
{D}+"."{D}*({E})?   printf("real");
```

The definitions section may also contain other commands, including the selection of a host language, a character set table, a list of start conditions, or adjustments to the default size of arrays within Lex itself for larger source programs, but you will learn more about these later on.

2.3.3 Lex Regular Expressions

In the previous laboratory you have learned about regular expressions using Python. Lex also supports regular expressions, most of which should fit to what you have already learned. Besides the known expressions, LEX also supports some context dependency expressions, such as:

- /, which is a more general version of the $ character, and indicates trailing context. The expression *"ab/cd"* matches the string *ab*, but only if followed by *cd*. Thus *"ab$"* is the same as *"ab/ \n"*
- <>, used for specifying left context. Left context is handled in Lex by start conditions. If a rule is only to be executed when the Lex automaton interpreter is in start condition x, the rule should be prefixed by *"< x >"* using the angle bracket operator characters. If we considered "being at the beginning of a line" to be equivallent to start condition ONE, then the ^operator would be equivalent to *"< ONE >"*. Start conditions will be explained in more detail in the next laboratory.

2.3.4 LEX Actions

When an expression written as above is matched, Lex executes the corresponding action. This section describes some features of Lex which aid in writing actions.

Note 2.3.2

Note that there is a default action, which consists of copying the input to the output. This is performed on all strings not otherwise matched. Thus the Lex user who wishes to absorb the entire input, without producing any output, must provide rules to match everything.

When Lex is being used with Yacc, the case above is the normal situation. One may consider that actions are what is done instead of copying the input to the output; thus, in general, a rule which merely copies can be omitted. Also, a character combination which is omitted from the rules and which appears as input is likely to be printed on the output, thus calling attention to the gap in the rules.

One of the simplest actions we can do is to ignore the input. This can be achieved by specifying a C null statement, ";" as an action. A frequent rule is

Listing 2.3.6: tab.lex — C code

```
[ \t\n]    ;
```

which causes the three spacing characters (blank, tab, and newline) to be ignored. Another easy way to avoid writing actions is the action character "|", which indicates that the action for this rule is the action for the next rule. The previous example could also have been written as:

Listing 2.3.7: tab.lex — C code

```
" "     |
"\t"    |
"\n"    ;
```

In more complex actions, the user will often want to know the actual text that matched some expression. Lex leaves this text in an external character array named *yytext*. Thus, to print the name found, a rule like

Listing 2.3.8: chars.lex — C code

```
[a-z]+   printf("\%s", yytext);
```

will print the string in *yytext*. This action is very common, so it may be written in a shorter manner as *"ECHO:"*.

Listing 2.3.9: chars.lex — C code

```
[a-z]+   ECHO;
```

Sometimes it's useful to know where the matched string ended. For this, LEX offers a counter called *yyleng* that stores the number of matched characters. To count the number of both characters and words found in the input string, we can write the rule:

Listing 2.3.10: words.lex — C code

```
[a-zA-Z]+ { words ++; chars += yyleng ;}
```

The rule above increments the `chars` variable with the character count given by *yyleng* and the `words` variable with each match. The last character of the matched string could then be accessed via `yytext[yyleng-1]`.

Occasionally, a LEX action may decide or enforce that a given rule did not correctly match a given string. For this situation, LEX offers two routines called *yyless* and *yymore*, which will be discussed in more detail in the next laboratory.

In addition to these routines, Lex also permits access to the I/O routines it uses. They are:

1. input() which returns the next input character
2. output(c) which writes the character c on the output
3. unput(c) which pushes the character c back onto the input stream to be read later by input()

Concept 2.3.1: Lookahead

Any lexical analyzer uses a lookahead. If the lexer only looks at the current unconsumed character, that means it has a one character of lookahead. The smaller this number is, generally, the faster the scanner runs. The rules of the lexical grammar dictate how much lookahead is needed. Fortunately, most languages in wide use only peek one or two characters ahead.

Lex does not look ahead at all if it does not have to, but every rule ending in + * ? or $ or containing / implies lookahead. Lookahead is also necessary to match an expression that is a prefix of another expression.

A Lex library routine that the user will sometimes want to redefine is *yywrap()* which is called whenever Lex reaches an end-of-file. If yywrap returns a 1, Lex continues with the normal wrap-up on end of input. Sometimes, however, it is convenient to arrange for more input to arrive from a new source. In this case, the user should provide a *yywrap* which arranges for new input and returns 0. This instructs Lex to continue processing. The default yywrap always returns 1. This routine is also a convenient place to print tables, summaries, etc. at the end of a program.

Note 2.3.3

Note that it is not possible to write a normal rule which recognizes end-of-file; the only access to this condition is through yywrap. In fact, unless a private version of input() is supplied a file containing nulls cannot be handled, since a value of 0 returned by input is taken to be end-of-file.

2.3.5 Ambiguous Rules

Lex can handle ambiguous specifications. When more than one expression can match the current input, Lex chooses as follows:

1. The longest match is preferred.
2. Among rules which matched the same number of characters, the rule given first is preferred.

Thus, suppose the rules

Listing 2.3.11: sample.lex — C code

```
integer   keyword action ...;
[a-z]+    identifier action ...;
```

to be given in that order. If the input is integers, it is taken as an identifier, because [a-z]+ matches 8 characters while integer matches only 7. If the input is integer, both rules match 7 characters, and the keyword rule is selected because it was given first. Anything shorter (e.g. int) will not match the expression integer and so the identifier interpretation is used.

The principle of preferring the longest match makes rules containing expressions like .* dangerous. For example, '.*' might seem a good way of recognizing a string in single quotes. But it is an invitation for the program to read far ahead, looking for a distant single quote. Presented with the input

'first' quoted string here, 'second' here

the above expression will match

'first' quoted string here, 'second'

which is probably not what was wanted. A better rule is of the form

Listing 2.3.12: sample.lex — C code

```
'[^'\n]*'
```

which, on the above input, will stop after 'first'. The consequences of errors like this are mitigated by the fact that the . operator will not match newline. Thus expressions like .* stop on the current line.

Note 2.3.4

Don't try to defeat this with expressions like `(.|\n)+` or equivalents; the Lex generated program will try to read the entire input file, causing internal buffer overflows.

Lex is normally partitioning the input stream, not searching for all possible matches of each expression. This means that each character is accounted for once and only once. For example, suppose it is desired to count occurrences of both she and he in an input text. Some Lex rules to do this might be

Listing 2.3.13: sample.lex C code

```
she    s++;
he     h++;
\n     |
.      ;
```

where the last two rules ignore everything besides *he* and *she*. Since *she* includes *he*, Lex will normally not recognize the instances of *he* included in *she*, since once it has passed a *she* those characters are gone.Sometimes the user would like to override this choice.

`REJECT` is useful whenever the purpose of Lex is not to partition the input stream but to detect all examples of some items in the input, and the instances of these items may overlap or include each other. The action `REJECT` means "go do the next alternative". It causes whatever rule was second choice after the current rule to be executed. The position of the input pointer is adjusted accordingly. Suppose the user really wants to count the included instances of he:

Listing 2.3.14: sample.lex C code

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

these rules are one way of changing the previous example to do that. After counting each expression, it is rejected; whenever appropriate, the other expression will then be counted. In this example, of course, the user could note that she includes he but not vice versa, and omit the REJECT action on he; in other cases, however, it would not be possible a priori to tell which input characters were in both classes.

Consider the two rules

Listing 2.3.15: sample.lex C code

```
a[bc]+   { ... ; REJECT;}
a[cd]+   { ... ; REJECT;}
```

If the input is *ab*, only the first rule matches, and on *ad* only the second matches. The input string *accb* matches the first rule for four characters and then the second rule for three characters. In contrast, the input *accd* agrees with the second rule for four characters and then the first rule for three.

2.4 Practice problems

To review what you have learned so far, try solving some exercises. Test the examples below to get familiar with the syntax and behavior.

Exercise 2.4.1 *

1. Removing the trailing spaces and tabs and the EOL of an input text and replacing the remaining spaces and tabs with a singular space.

   Listing 2.4.1: sample.lex — C code

   ```
   %%
   [ \t]+$     ;
   [ \t]+      printf("")
   %%
   ```

2. Identifying integer and real numbers (Fortran convention) in the input string

   Listing 2.4.2: sample.lex — C code

   ```
   D                           [0-9]
   E                           [DEde][ -+]?{D}+
   %%
   {D}+                        printf (" integer ");
   {D}+"."{D}*({E})?           |
   {D}*"."{D}+({E})?           |
   {D}+{E}                     printf (" real ");
   %%
   ```

   Let's look at some more examples.

3. Generating a histogram of word lengths from an imput text, where a word can consist of a stream of a-z characters.

   Listing 2.4.3: sample.lex — C code

   ```
   int   lung [99];
   %%
   [a-z ]+      lung[yyleng]++;
   .            |
   \n           ;

   %%

   yywrap ()
   {
     int i;
     printf("Length; Word Count \n");
     for (i = 0; i < 99; i++)
       if (lung[i] > 0)
         printf("\%6d\%7d\n", i, lung[i]);

     return(1);
   }
   ```

4. Get familiar with REJECT with the example below which determines the counts of "she" and "he" words.
   In: "she reads, he cooks, she is happy, he is sad"
   Out: "she: 2; he: 4"

Listing 2.4.4: sample.lex C code

```
D       he
int     s, h;
%%
s{D}    s++;
{D}     h++;
\n      ;
.       ;
%%
yywrap (){
  printf ("She: \% i; he:\% i", s, h);
  return 1;
}
```

Exercise 2.4.2 **

Implement a program which returns the character count in a file, ignoring newline characters.

Exercise 2.4.3 **

Implement a program that counts comments marked with // at the beginning of a line (may be preceded by any number of tab or space characters)