`iflex`: A Lexical Analyzer Generator for Icon

Ray Pereda

Unicon Technical Report UTR-02

February 25, 2000

Abstract

`iflex` is software tool for building language processors. It is based on `flex`, a well-known tool for the C programming language. This paper describes the experienced gained in creating `iflex` and a brief description of how to use the tool.

The University of Nevada, Las Vegas
Department of Computer Science
Las Vegas, Nevada 89154, USA

1. Introduction

Building a language processor such as a compiler is a complex task. Parsing is the extraction of the grammatical structure of a sentence in some language. The first step in parsing is extracting the lexical items or "words" in a sentence. iflex is tool for doing just that, i.e. scanning, or also known more formally as lexical analysis.

If this report is your first lex, you may wish to also read "Lex & Yacc," by [Levine92]. Also, the book by [Jeffery00] has a chapter on iflex. iyacc, the companion program for iflex is documented in [Pereda00].

The name of the iflex tool stands for Icon Lexical Analyzer. It was created by modifying a program called flex to generate Icon code as an alternative to C. Flex documented in [Paxson92]. The flex program in turn is based on a classic UNIX program called lex that dates back to 1975. Lex is documented in [Lesk75]. The iflex tool takes a lexical specification and produces a lexical analyzer that corresponds to that specification. A lexical analyzer is a fancy name for a scanner, and the lexical analyzer generated by iflex is simply a procedure named yylex(). The specification of the lexical structure of many languages can be concisely and precisely stated using this notation. You only need to know the basics of regular expressions to get a useful understanding of them. The iflex tool specifications consist of a list of regular expressions.

2. Example #1, A Word Count Program

There is a UNIX program called wc, short for word count, that counts newlines, words, and characters in a file. In this section, you will see how to build such a program using iflex. A short, albeit simplistic, definition of a word is any sequence of non-white space characters. White space characters are blanks and tabs. Below is a complete iflex
program that operates like wc:

```
ws    [ \t]
nonws [^ \t\n]
%{
global cc, wc, lc
%}
%%
{nonws}+ cc +:= yyleng; wc +:= 1
```

```
{ws}+ cc +:= yyleng
\n lc +:= 1; cc +:= 1
%%
procedure main()
  cc := wc := lc := 0
  yylex()
  write(right(lc, 8), right(wc, 8), right(cc, 8))
end
```

All iflex programs, including this program, consist of three sections, separated by lines containing two percent signs. The three sections are the definitions section, the rulessection, and the procedures section. In the word count program, the definitions section has two definitions, one for white space (ws) and one for non-white space (nonws). These definitions are followed by code to declare three global variables, which will be used as counters. The variables cc, wc, and lc are used to count the characters, words, and lines, respectively. The rules section in this example contains three rules. White space, words, and newlines each have a rule that matches and counts their occurrences. The procedure section has one procedure, main(). It calls the lexical analyzer and then prints out the counts. There are many ways to write this word count program, with different performance characteristics. If speed is your primary consideration you can look in the documentation for flex to get five progressively more complex but faster versions of word count. The documentation is available on the Internet, just search for flex and Vern Paxson, the author.

3. Example #2, A Lexical Analyzer for a Desktop Calculator

The above example illustrates using iflex to write standalone programs, but the function yylex() produced by iflex is usually called by a parser algorithm. The yylex()function can be used to produce a sequence of words, and a parser such as that generated by the iyacc program combines those words into sentences. So it makes sense to study how iflex is used in this typical context. One obvious difference is that in the earlier example, yylex() was only called once to process an entire file; in contrast, when a parser uses yylex() it calls it repeatedly, and yylex() returns with each word that it finds. You will see this in the following example. A calculator program is simple enough to understand in one sitting and complex enough to get a sense of how to use iflex with its parser generator counterpart, iyacc. In general, in a desktop calculator program the user types in complex formulas and the calculator evaluates them and prints out the answer. First things first: what are the *words* of this little language? Numbers, math operators, and variable names. A number is one or more digits followed by an optional decimal point and one or more digits. In regular expressions, you can write this as

```
[0-9]+(\.[0-9]*)?
```

The 0 through 9 is specified with a range using a dash for characters within the square brackets. The plus sign means one or more occurrences, whereas the star means *zero* or more occurrences. The backslash period means literally match a period; without a backslash a period matches any single character. The parentheses are used for grouping and the question mark means zero or more times. The math operators are simple "words" composed of one character such as the plus sign, minus sign, and star for multiplication. Variable names need to be meaningful; so why not let them be any combination of letters, digits, and underscores. You do not want to confuse them with numbers, so refine the definition by making sure that variables do not begin with a number. This definition of variable names corresponds to the following regular expression:

```
[a-zA-Z_][a-zA-Z0-9_]*
```

Here is some more information about the three sections that make up an iflex program. The definitions section contains Icon code that is copied verbatim into the generated final program, before the generated scanner code. You can put $include statements there to define symbolic constants for the different kinds of words in your scanner. The rules section contains a series of rules, composed of two parts: a regular expression and a fragment of Icon code that executes whenever that regular expression matches part of the input. The procedures section contains arbitrary code that is copied verbatim after the generated scanner's code. A complete scanner specification for the desktop calculator looks like:

```
%{
# y_tab.icn contains the symbol definitions for integer values
# representing the terminal symbols NAME, NUMBER, and
# ASSIGNMENT. It is generated with iyacc ?d calc.y
$include y_tab.icn
%}
letter [a-zA-Z_]
digiletter [a-zA-Z0-9_]
%%
{letter}{digiletter}* { yylval := yytext; return NAME }
[0-9]+(\.[0-9]+)? { yylval := numeric(yytext); return NUMBER }
\n return 0 /* logical EOF */
``:='' return ASSIGNMENT
[ \t]+ ; /* ignore whitespace */
. return ord(yytext)
%%
```

There are a couple of details about iflex worth noting in this scanner. The iflex tool maintains a global variable named yytext that holds the characters that match a given regular expression. For example, a plus operator is returned by the scanner rule that says to return the character code corresponding to yytext, ord(yytext), on the regular expression that consists of a lone period (. matches any one character). Even if yytext is not part of yylex()'s return value for a token, there are situations in which the string is of interest to the parser, which reads lexical values from a global variable called yylval. When a variable name is encountered, it makes sense to copy yytext over into yylval. On the other hand, when a number is encountered, the numeric value corresponding to the characters in yytext is computed and stored in yylval. Since Icon allows a variable to hold any type of value, there is no need for a union or some other messy construct to handle the fact that different tokens have different kinds of lexical values.

Before you conclude your study of iflex, two subtle points are worth knowing. The matches allowed by a list of regular expressions are often ambiguous, and this is normal and healthy. For example, does count10 match a variable name and then an integer, or just one variable name? The iflex tool matches the longest substring of input that can match the regular expression. So it matches count10 as one word, which is a variable name in this case. There is one more sticky point: what if two rules match the exact same input characters, with no longest match to break the tie? In this case, iflex picks the first rule listed in the specification that matches, so the order of the rules can be important.

Appendix: A Summary of the Regular Expression Operators:

This list the most commonly used operators in iflex. Advanced users will want to consult the documentation for flex or its predecessor, lex, for a more complete list.

Commonly used iflex operators

Operator	**Description**
.	Matches any single character except the newline character.
*	Matches zero or more occurrences of the preceding expression.
[]	This is a character class that matches any character within the brackets. If the first character is a caret (^), then it changes the meaning to match any character except those within the brackets. A dash inside the brackets represents a character range, so [0-9] is equivalent to [0123456789].
^	Matches the beginning of a line. This interpretation is used only when the caret is the first character of a regular expression.
$	Matches the end of a line. This interpretation is used only when the dollar symbol is the last character of a regular expression.
\	Used to escape the special meaning of a character. For example, \$ matches the dollar sign, not the end of a line.
+	Matches one or more occurrences of the preceding expression. For example, [0-9]+ matches “``1234"” or “734,” but not the empty string.
?	Matches zero or more occurrences of the preceding expression. For example, -?[0-9]+ matches a number with an optional leading negative sign.
|	Matches either the preceding regular expression or the one following it. So Mar|Apr|May matches any one of these three months.
“…”	Matches everything in quotes literally.
( )	Groups a regular expression together, overriding the default operator precedence. This is useful for creating more complex expressions with *, +, and |.

References

[Jeffery00] Jeffery, C., Mohamed, S., Pereda, R. and Parlett, R., *Programming with Unicon: very high-level object-oriented application and system programming.* To be published.

[Lesk75] Lesk, M. E., and Schmidt, E. LEX – Lexical Analyzer Generator, *Computer Science Technical Report No. 39*, Bell Laboratories, Murray Hill, New Jersey (October 1975).

[Levine92] Levine, J. R., Mason, T., and Brown, D. *Lex & Yacc*, O'Reilly and Associates, Cambridge, Massachusetts, 1992.

[Paxson95] Paxson, Vern, flex – a fast lexical analyzer generator, many replication on the Internet. Also available on Redhat Linux via the man command.

[Pereda00] Pereda, Ray, *iyacc – a Parser Generator of Icon, Unicon Technical Report 00-01*, University of Nevada, Las Vegas, Las Vegas, Nevada, (February 2000).