Ulex: A Lexical Analyzer Generator for Unicon

Katrina Ray, Ray Pereda, and Clinton Jeffery

Unicon Technical Report UTR – 02a

May 21, 2003

Abstract

Ulex is a software tool for building language processors. It implements a compatible subset of the well-known UNIX C tool called `lex(1)` for programs written in Unicon and Icon. This paper provides a brief description of Ulex and how to use the tool.

Department of Computer Science
New Mexico State University
Las Cruces, NM 88003

School of Computer Science
University of Nevada Las Vegas
Las Vegas, NV 89154

1. Introduction

Building a language processor such as a compiler is a complex task. A language processor must be able to extract the grammatical structure of a sentence in the language. This extraction is known as parsing. The first step in parsing is scanning to determine the lexical items or "words" in a sentence.

Ulex is a tool for building scanners that perform lexical analysis. Ulex stands for Unicon Lexical Analyzer. It was designed to function like the classic UNIX program called lex, except that it generates Unicon code rather than C. Lex dates back to 1975 and is documented in [Lesk75].

Ulex uses regular expression notation to specify lexical analysis; the lexical structure of many languages can be concisely and precisely stated using this notation.. The regular expressions supported in Ulex are given in Table 1.

Operator	Description
a	ordinary non-operator symbols match themselves
.	a period matches any single character except newline
|	alternation matches either the preceding or following expression
bc	concatenation is an implicit binary operator with low precedence
*	matches zero or more occurrences of the preceding expression
[ ]	matches any one character within the brackets
+	matches one or more occurrences of the preceding expression
?	matches zero or one occurrences of the preceding expression
"…"	matches characters in quotes literally
(e)	groups regular expressions, overriding operator precedence

Table 1: Regular Expressions in Ulex.

This document assumes you are somewhat familiar with regular expressions; if not, you may also wish to read Lex and Yacc, by [Levine92]. Ulex is usually used with Iyacc, a parser generator tool that is a companion program for Ulex. Iyacc is documented in [Pereda00]. Ulex and Iyacc are additionally described in [Jeffery03].

2. Ulex Program Structure

The Ulex tool takes a lexical specification and produces a lexical analyzer that corresponds to that specification. The specification consists of a list of regular expressions, with auxiliary code fragments, variables, and helper functions. The resulting generated analyzer is in the form of a procedure named yylex() .

All Ulex programs consist of a file with an extension of `.l` that contains three sections, separated by lines consisting of two percent signs. The three sections are the definitions section, the rules section, and the procedures section. The definitions

section has two kinds of components. *Macros* define shorthand for regular expressions that will be used in the next section. *Code fragments* enclosed by %{ and %} are copied verbatim to the generated lexical analyzer. The rules section contains the actual regular expressions that specify the lexical analysis that is to be performed. Each regular expression may be followed by an optional semantic action enclosed in curly brackets, which is a segment of Unicon code that will be executed whenever that regular expression is matched. The procedures section is also copied out verbatim into the generated lexical analyzer.

The yylex() function and its return value constitute the primary interface between the lexical analyzer and the rest of the program. yylex() returns a -1 if it consumes the entire input; returning different integer values from within semantic actions in the rules section allows yylex() to break the input up into multiple chunks of 1+ characters (called *tokens*), and to identify different kinds of tokens using different integer codes. In addition to the return value, the generated lexical analyzer also makes use of several global variables. The names and meanings of these are summarized in Table 2.

Variable Name	Description
yyin	File from which characters will be read; default: &input
yytext	String of characters matched by a regular expression
yyleng	Length of yytext (*yytext)
yychar	Integer category of the most recent token
yylval	Lexical value(s) (often a record) of the most recent token

Table 2: Ulex global variables.

3. Example 1: A Word Count Program

There is a UNIX program called wc, short for word count, that counts the number of lines, words, and characters in a file. This example demonstrates how to build such a program using Ulex. A short, albeit simplistic, definition of a word is any sequence of non-white space characters, where white space characters are blanks and tabs. See Listing 1 for a Ulex 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
        yyulex()
        write(right(lc, 8), right(wc, 8), right(cc, 8))
end
```

Listing 1. wc using ulex.

In the word count program, the definitions section consists of two definitions, one for white space characters (ws) and one for non-white space characters (nonws). These definitions are followed by code to declare three global variables: cc, wc, and lc. These are the counters for 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 counted values.

4. Example 2: A Lexical Analyzer for a Desktop Calculator

The previous example demonstrates using Ulex to create standalone programs. However, yylex() is typically called from a parser. The yylex() function can be used to produce a sequence of words so that a parser such as that generated by the iyacc program can combine those words into sentences. Thus it makes sense to study how ulex is used in this context. One obvious difference is that in the earlier example, yylex() was only called once to process the entire file. In contrast, when a parser uses yylex() , it calls the analyzer repeatedly, and yylex() returns with each word that it finds. This will be demonstrated in the example that follows.

A calculator program is simple enough to understand in one sitting and complex enough to get a sense of how to use Ulex with its parser generator counterpart: Iyacc. In a general desktop calculator program, the user types in complex formulas, which the calculator evaluates and then prints the result. The generated lexical analyzer must recognize the words of this language, which will be handled by the parser. In this case the words are 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, this may be written as:

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

The math operators are simple words composed of one character. Variable names can be any combination of letters, digits, and underscores. So as not to confuse them with numbers, refine the definition by making sure that the 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_]*
```

Recall that there are three sections to every Ulex program: a definitions section, a rules section, and a procedures section. The Ulex program for matching the words of a calculator is given in Listing 2.

```
%{
# y_tab.icn contains the integer codes for representing the
# terminal symbols NAME, NUMBER, and ASSIGNMENT.
$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 end-of-file
                         }
":="                     { return ASSIGNMENT }
[ \t]+                   {
                           # ignore white space
                         }
.                        { return ord(yytext) }
%%
```

Listing 2. Ulex program for recognizing the lexical elements of a calculator.

The definitions section has both a component that is copied directly to the generated lexical analyzer as well as a set of macros. The first rule matches variable names; the second rule matches numbers. The third rule returns 0 to indicate to the parser that it should evaluate the expression. The fourth rule lets the parser know that there was an assignment operator, and the fifth is used to ignore white space. The last rule matches everything else including the other mathematical operators. The character's numeric code (e.g. ASCII) is returned directly to the parser.

yylval is used to store the result whenever we match either a variable name or a number. This way when the lexical analyzer returns the integer code for name or number, the parser knows to look in yylval for the actual name or number that was matched. Since Unicon allows variables to hold any type of value there is no need for a complicated construct to handle the fact that different tokens have different types of lexical values.

Notice that the matches that are allowed by this set of regular expressions are somewhat ambiguous. For example, count10 may match a variable name and then an

integer, or one variable name. The Ulex tool is designed to match the longest substring of input that can match the regular expression. So count10 would be matched as one word, which is a variable in this case. In the case where two different expression match the same number of characters, the first rule listed in the specification will be used.

5. Conclusion

The examples in this paper introduce how Ulex may be useful in language processing either as a standalone application or in conjunction with a parser. Ulex is still in a preliminary state, but is now ready for testing; when mature it should be useful for experimental compiler prototyping.

Acknowledgements

The Ulex tool is primarily the work of Katrina Ray; she was supported by an NMSU fellowship and by the National Library of Medicine during its development.

Ray Pereda wrote the first version of this document to describe his iflex tool.

References

[Jeffery03] C. Jeffery, S. Mohamed, R. Pereda, and R. Parlett. *Programming with Unicon*. http;//unicon.sf.net/book/ub.pdf, 2003.

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

[Levine92] J.R. Levine, T. Mason, and D. Brown. *Lex & Yacc* , O"Reilly & Associates, Cambridge, Massachusetts, 1992.

[Pereda00] Ray Pereda. *Iyacc – a Parser Generator for Icon* , Unicon Technical Report 03, http://unicon.sf.net/utr/utr3.pdf, February 2000.

Appendix: Differences Between Ulex and UNIX lex(1)

This appendix summarizes the known differences between Ulex and the UNIX lex(1) tool. Ulex is a large subset of lex, but has several important limitations to note.

No lookahead – the lookahead operator / is not yet supported
Naïve semantic actions – semantic actions must be enclosed in curly brackets
No repetition – the repetition operators (e.g. a{2:5}) are not yet supported.