SML/NJ Language Processing Tools
DRAFT

Aaron Turon
adrassi@cs.uchicago.edu

October 24, 2006

This document was written with support from NSF grant CNS-0454136, "CRI: Standard ML Software Infrastructure."

Contents

Notice

This is an **early draft** manual for `ml-ulex` and `ml-antlr`. The early release of these tools and this manual is intended for gathering feedback. The interfaces described herein will likely undergo substantial revision before the general release of these tools.

Chapter 1

Overview

In software, language recognition is ubiquitous: nearly every program deals at some level with structured input given in textual form. The simplest recognition problems can be solved directly, but as the complexity of the language grows, recognition and processing become more difficult.

Although sophisticated language processing is sometimes done by hand, the use of scanner and parser generators[1] is more common. The Unix tools `lex` and `yacc` are the archetypical examples of such generators. Tradition has it that when a new programming language is introduced, new scanner and parser generators are written in that language, and generate code for that language. Traditional *also* has it that the new tools are modeled after the old `lex` and `yacc` tools, both in terms of the algorithms used, and often the syntax as well. The language Standard ML is no exception: `ml-lex` and `ml-yacc` are the SML incarnations of the old Unix tools.

This manual describes two new tools, `ml-ulex` and `ml-antlr`, that follow tradition in separating scanning from parsing, but break from tradition in their implementation: `ml-ulex` is based on *regular expression derivatives* rather than subset-construction, and `ml-antlr` is based on $LL(k)$ parsing rather than $LALR(1)$ parsing.

1.1 Motivation

Most parser generators use some variation on *LR* parsing, a form of *bottom-up* parsing that tracks possible interpretations (reductions) of an input phrase until only a single reduction is possible. While this is a powerful technique, it has the following downsides:

- Compared to predictive parsing, it is more complicated and difficult to understand. This is particularly troublesome when debugging an *LR*-ambiguous grammar.

- Because reductions take place as late as possible, the choice of reduction cannot depend on any semantic information; such information would only become available *after* the choice was made.

[1] "Scanner generator" and "parser generator" will often be shortened to "scanner" and "parser" respectively. This is justified by viewing a parser generator as a parameterized parser.

- Similarly, information flow in the parser is strictly bottom-up. For (syntactic or semantic) context to influence a semantic action, higher-order programming is necessary.

The main alternative to *LR* parsing is the top-down, *LL* approach, which is commonly used for hand-coded parsers. An *LL* parser, when faced with a decision point in the grammar, utilizes lookahead to unambiguously predict the correct interpretation of the input. As a result, *LL* parsers do not suffer from the problems above. *LL* parsers have been considered impractical because the size of their prediction table is exponential in k — the number of tokens to look ahead — and many languages need $k > 1$. However, Parr showed that an approximate form of lookahead, using tables linear in k, is usually sufficient.

To date, the only mature *LL* parser based on Parr's technique is his own parser, `antlr`. While `antlr` is sophisticated and robust, it is designed for and best used within imperative languages. The primary motivation for the tools this manual describes is to bring practical *LL* parsing to a functional language. Our hope with `ml-ulex` and `ml-antlr` is to modernize and improve the Standard ML language processing infrastructure, while demonstrating the effectiveness of regular expression derivatives and $LL(k)$ parsing. The tools are more powerful than their predecessors, and they raise the level of discourse in language processing.

1.2 Outline

This manual is organized into three parts: usage, theory, and implementation. Each of these parts is further broken down into two chapters, one on `ml-ulex` and one on `ml-antlr`. The usage section is self-contained, and gives a fairly complete specification of the two tools. Full details on the algorithms used are given in the theory section. Data structures, system organization, and other code-related particulars are described in the implementation section.

Part I

Usage

Chapter 2

Usage: `ml-ulex`

2.1 Overview

ML-ULEX is used for generating "lexers," which discern the lexical structure of an input string. If the generated module is called `Lexer`, it will contain a type `strm` and a function

```
val lex : strm -> (token * strm) option
```

where `token` is a type determined by the user of `ml-ulex`. Thus, a lexer is a token reader, in the sense of the Basis library `StringCvt.reader` type.

The tool is invoked from the command-line as follows:

```
ml-ulex [options] file
```

where `file` is the name of the input `ml-ulex` specification, and where `options` may be any combination of:

`--dot`	generate DOT output (for graphviz; see `http://www.graphviz.org`). The produced file will be named `file.dot`, where `file` is the input file.
`--match`	enter interactive matching mode. This will allow interactive testing of the machine; presently, only the `INITIAL` start state is available for testing (see Section **??** for details on start states).
`--ml-lex-mode`	operate in `ml-lex` compatibility mode. See Section 2.7 for details.

2.2 Specification format

A `ml-ulex` specification is a list of semicolon-terminated *declarations*. Each declaration is either a *directive*

spec	::=	(*declaration* ;)*
declaration	::=	*directive*
	|	*rule*
directive	::=	`%charset` (`ASCII7` | `ASCII8` | `UTF8`)
	|	`%defs` *code*
	|	`%let` *ID* = *re*
	|	`%name` *ID*
	|	`%states` *ID*$^+$
code	::=	(...)
rule	::=	*re* => *code*

Figure 2.1: The top-level `ml-ulex` grammar

2.3 Directives

2.3.1 The `%defs` directive

2.3.2 The `%let` directive

2.3.3 The `%name` directive

2.3.4 The `%states` directive

2.4 Regular expressions

2.5 Using the generated code

2.6 An example

```
%name CalcLex;

%let digit = [0-9];
%let int = {digit}+;
%let alpha = [a-zA-Z];
%let id = {alpha}({alpha} | {digit})*;

%defs (
  open CalcParse.Tok
);

let      => ( KW_let );
in       => ( KW_in );
{id}     => ( ID (yytext()) );
{int}    => ( NUM (valOf (Int.fromString (yytext()))) );
"="      => ( EQ );
```

re	::=	any nonreserved, nonwhitespace character or escape code
	|	*re* `*`
	|	*re* `?`
	|	*re* `+`
	|	`^` *re*
	|	*re* `|` *re*
	|	*re* `&` *re*
	|	*re* `/` *re*
	|	*re* `$`
	|	a double-quoted string, as in SML
	|	{ *ID* }
	|	`[` $\hat{}^?$ (*char* `-` *char* | *char*)$^+$ `]`
	|	*re re*
	|	(*re*)
	|	`.`
	|	`_`

Figure 2.2: The `ml-ulex` grammar for regular expressions

```
"+"      => ( PLUS );
"-"      => ( MINUS );
"*"      => ( TIMES );
"("      => ( LP );
")"      => ( RP );
" " | \n | \t
         => ( continue() );
.        => ( (* handle error *) );
```

2.7 `ml-lex` compatibility

Chapter 3

Usage: `ml-antlr`

3.1 Overview

3.2 Specification format

$$
\begin{array}{rcl}
\mathit{spec} & ::= & (\ \mathit{declaration}\ \texttt{;}\ )^{*} \\
\mathit{declaration} & ::= & \mathit{directive} \\
 & | & \mathit{nonterminal} \\
\mathit{directive} & ::= & \texttt{\%defs}\ \mathit{code} \\
 & | & \texttt{\%import}\ \mathit{STRING} \\
 & | & \texttt{\%keywords}\ \mathit{symbol}^{+} \\
 & | & \texttt{\%name}\ \mathit{ID} \\
 & | & \texttt{\%start}\ \mathit{ID} \\
 & | & \texttt{\%tokens}\ \texttt{:}\ \mathit{tokdef}\ (\ \texttt{|}\ \mathit{tokdef}\ )^{*} \\
\mathit{code} & ::= & \texttt{(}\ \ldots\ \texttt{)} \\
\mathit{tokdef} & ::= & \mathit{datacon}\ (\ \texttt{(}\ \mathit{STRING}\ \texttt{)}\ )^{?} \\
\mathit{datacon} & ::= & \mathit{ID} \\
 & | & \mathit{ID}\ \texttt{of}\ \mathit{monotype} \\
\mathit{monotype} & ::= & \text{usual SML syntax} \\
\mathit{symbol} & ::= & \mathit{ID} \\
 & | & \mathit{STRING}
\end{array}
$$

nonterminal	::=	*ntdef*
	|	`%extend` *ntdef*
	|	`%replace` *ntdef*
	|	`%drop` ID^+
ntdef	::=	*ID* $formals^?$ `:` *prodlist*
formals	::=	`(` *ID* (`,` *ID* $)^*$ `)`
prodlist	::=	*production* (`|` *production* $)^*$
production	::=	$\texttt{\%try}^?$ $named\text{-}item^*$ (`%where` *code* $)^?$ (`=>` *code* $)^?$
named-item	::=	(*ID* `:` $)^?$ *item*
item	::=	*prim-item* `?`
	|	*prim-item* `+`
	|	*prim-item* `*`
prim-item	::=	*symbol* $args^?$
	|	`(` *prodlist* `)`
args	::=	`@` *code*

3.3 An example

`calc.grm`

```
%name CalcParse;
%tokens
  : KW_let  ("let")  | KW_in   ("in")
  | ID of string     | NUM of Int.int
  | EQ      ("=")    | PLUS    ("+")
  | TIMES   ("*")    | MINUS   ("-")
  | LP      ("(")    | RP      (")")
  ;

exp(env)
  : "let" ID "=" exp@(env)
    "in" exp@(AtomMap.insert(env, Atom.atom ID, exp1))
      => ( exp2 )
  | addExp@(env)
  ;

addExp(env)
  : multExp@(env) ("+" multExp@(env))*
      => ( List.foldl op+ multExp SR1 )
  ;

multExp(env)
  : prefixExp@(env) ("*" prefixExp@(env))*
      => ( List.foldl op* prefixExp SR1 )
```

```
  ;

prefixExp(env)
  : atomicExp@(env)
  | "-" prefixExp@(env)
      => ( ~prefixExp )
  ;

atomicExp(env)
  : ID
      => ( valOf(AtomMap.find (env, Atom.atom ID)) )
  | NUM
  | "(" exp@(env) ")"
  ;
```

Part II

Theory

Chapter 4

Theory: `ml-antlr`

Part III

Implementation

Chapter 5

Implementation: `ml-antlr`