
CMCS 312: Programming Languages
ml-lex and ml-yacc

Umut A. Acar

October 6, 2006

Contents

1	Introduction	1
2	Lexing	1
2.1	Regular Expressions, ml-lex style	2
3	Parsing	2
4	Miscellaneous	3

1 Introduction

Compilers and interpreters generally start out by lexing (tokenizing) the program source text and then parsing the resulting tokens into a parse tree. These two subjects have significant bodies of theory and therefore deserve a course of their own: CMSC22610, Compilers-I.

2 Lexing

Program source text can be messy. People use whitespace and parenthesization differently. The compiler should not be treating each character in the source text individually. Instead, it should treat a group of characters as a token. Tokens are syntactic elements. Lexing generates a list of tokens from the source text. For example, LET, FN, +, and 1234 are all tokens in ML. The ml-lex program is a tool for generating lexers from a **lexical specification** of the **object language**.

```
ML code goes here ...
%%
Lexer definitions...
%%
regexp => (ML code to produce Token); ...
```

Figure 1: The form of lexical specs

The lexical specification consists of some boilerplate ML declarations, some character class/lex definitions, and the token specifications. These three sec-

tions are separated from each other by `%%`. Lex definitions have the form `classname=regex`. For example, `digit=[0-9]` defines a character class `digit` that includes all the digits from 0 to 9.

A token specification consists of a regular expression for matching a token and an action that communicates the token type plus some additional information to the parser. Actions are SML expressions. Token specifications have the form `regex => action`.

2.1 Regular Expressions, ml-lex style

Symbols	a ab ab1 “/” “(”
Empty String	“”
Alternation (OR)	true — false
Concatenation	sticktogether
Groupings	(10) — (34)
Repetition (Zero+)	(10)*
Repetition (Once+)	(10)+
Repetition (none or once)	(10)?
Character sets (Ranges)	[a-z], [a-zA-Z], [0-9]
Character classes (lex def'n)	{digits}
Wildcard (matches all characters)	.

Escaped characters newline (`\n`), space (`\`), tab (`\t`) ...

The parser provides a `Tokens` structure that contains functions for each of the token types. Each function takes at least two tupled arguments that represent the starting and ending points of the token in the program text. For the purpose of HW1, you may just use line numbers for the starting and ending points. Consequently, the starting and ending points should be the same (the line of the token). This position information can be used in error reporting. Some token functions take additional arguments called “semantic values”. The semantic value usually precedes the position information in a tupled argument (3-tuple). Be sure to check the generated `parser.grm.sig` to check how the token functions are supposed to be called.

If we get to a token that we want to ignore, we use the `lex` function to move on to the next token.

3 Parsing

Parsing turns the sequence of tokens into a parse tree. We generate parsers from specifications by means of the `ml-yacc` tool. The specification is partitioned into three sections (separated by `%%`) similar to the lex specifications: user declarations, parser declarations, and grammar rules. **Grammar rules** look similar to BNF except each line is followed by a “semantic action” to process the given grammatical element. BNF grammars are composed of **terminals** and **non-terminals**. Both must be declared up front. Nonterminals are metavariables standing in for some other grammar element each of which must be defined as

a BNF grammar. The ml-yacc specification also has a default case for every nonterminal (indicated by omitting a grammar element and only indicating a semantic action).

```
ML code ...
%%
%term TERMINALS
%nonterm nonterminals
Parser definitions ...
%%
nonterm : Terminal/Nonterminal sequence      (ML code for semantic action)
        | another terminal/nonterminal seq   (ML code for semantic action)
```

Figure 2: Parser spec (grm files)

Parsers find the true structure of expressions. As such, associativity and precedences must be declared explicitly in the parser specification.

The pitfall in specifying grammars is that you can specify ambiguous grammars. Ambiguous grammars can generate a single expression in the language in multiple ways. We can get rid of ambiguity by restructuring the grammar or defining some operator precedences in the parser declaration part of the specification. Precedence directives have the form `%assoc symbol symbol symbol` where `assoc` is `nonassoc` (i.e., give an error if ambiguous), `left`, or `right`. Symbols are either terminals or nonterminals and the space delimited sequence of can be of any length ≥ 1 . Directives that come earlier bind more weakly.

Modern parsers can also do some error recovery by trying different combinations of replacements or insertions to fix erroneous code. The `%prefer` directive allows us to specify a sequence (space delimited) of terminals we would prefer to insert for error recovery.

The `%eop` or `%noshift` directive can specify a space delimited list of terminals that signal that we would like to stop parsing.

4 Miscellaneous

```
print : string -> unit
Int.toString: int -> string
Int.fromString: string -> int option
```

```
Reference cells
val r = ref 0
!r
r := 1
```