SML/NJ Language Processing Tools: User Guide

Aaron Turon
adrassi@cs.uchicago.edu

January 24, 2007
Copyright ©2006. All rights reserved.

This document was written with support from NSF grant CNS-0454136, “CRI: Standard ML Software Infrastructure.”
5 A complete example 23
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 1.0 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.

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.

- 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

\(^1\)“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.
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-ulox 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.
Chapter 2

ML-ULex

Lexers analyze the lexical structure of an input string, and are usually specified using regular expressions. ML-ULEX is a lexer generator for Standard ML. The module it generates will contain a type `strm` and a function

```ml
val lex : strm -> lex_result * strm
```

where `lex_result` is a type that must be defined by the user of `ml-ulex`. Note that the lexer always returns a token: we assume that end-of-file will be explicitly represented by a token. Compared to ML-Lex, `ml-ulex` offers the following improvements:

- Unicode is fully supported.
- Intersection and negation of REs are supported.
- The specification format is somewhat cleaner.
- The code base is much cleaner, and supports multiple back-ends, including DFA graph visualization and interactive testing of rules.

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 2.5 for details.

The output file will be called `file.sml`. 
CHAPTER 2. ML-ULEX

2.1 Specification format

A ml-ulex specification is a list of semicolon-terminated declarations. Each declaration is either a directive or a rule. Directives are used to alter global specification properties (such as the name of the module that will be generated) or to define named regular expressions. Rules specify the actual regular expressions to be matched. The top-level grammar is given in Figure 2.1.

There are a few lexical details of the specification format worth mentioning. First, SML-style comments (* ... *) are treated as ignored whitespace anywhere they occur in the specification, except in segments of code. The ID symbol used in the grammar stands for alpha-numeric-underscore identifiers, starting with an alpha character. The code symbol represents a segment of SML code, enclosed in parentheses. Extra parentheses occurring within strings or comments in code need not be balanced.

A complete example specification appears in Chapter 5.

2.2 Directives

2.2.1 The %defs directive

The %defs directive is used to include a segment of code in the generated lexer module, as in the following example:

```ml
%defs (  
    type lex_result = CalcParserToks.token  
    fun eof () = CalcParserToks.EOF  
    fun helperFn x = (* ... *)  
  )
```

The definitions must at least fulfill the following signature:

```ml
type lex_result  
val eof : unit -> lex_result
```

All semantic actions must yield values of type lex_result. The eof function is called by ml-ulex when the end of file is reached – it acts as the semantic action for the empty input string. All definitions given will be in scope for the rule actions (see Section 2.3).

2.2.2 The %let directive

Use %let to define named abbreviations for regular expressions; once bound, an abbreviation can be used in further %let-bindings or in rules. For example,
2.3. RULES

\%let digit = [0-9];

introduces an abbreviation for a regular expression matching a single digit. To use abbreviations, enclose their name in curly braces. For example, an additional \%let definition can be given in terms of digit,

\%let int = \{digit\}+;

which matches arbitrary-length integers. Note that scoping of let-bindings follows standard SML rules, so that the definition of int must appear after the definition of digit.

2.2.3 The \%name directive

The name to use for the generated lexer module is specified using \%name.

2.2.4 The \%states directive

It is often helpful for a lexer to have multiple start states, which influence the regular expressions that the lexer will match. For instance, after seeing a double-quote, the lexer might switch into a STRING start state, which contains only the rules necessary for matching strings, and which returns to the standard start state after the closing quote.

Start states are introduced via \%states, and are named using standard identifiers. There is always an implicit, default start state called INITIAL. Within a rule action, the function YYBEGIN can be applied to the name of a start state to switch the lexer into that state; see 2.3.2 for details on rule actions.

2.3 Rules

Recall that the lex function of the generated lexer module is a “token” reader. In general, when lex is applied to an input stream, it will attempt to match a prefix of the input with a regular expression given in one of the rules. When a rule is matched, its action (associated code) is evaluated and the result is returned. Hence, all actions must belong to the same type, but no restrictions are placed on what that type is.

Rules are specified by an optional list of start states, a regular expression, and the action code. The rule is said to “belong” to the start states it lists. If no start states are specified, the rule belongs to all defined start states.

Rule matching is influenced by three factors: start state, match length, and rule order. A rule is only considered for matching if it belongs to the lexer’s current start state. If multiple rules match an input prefix, the rule matching the longest prefix is selected. In the case of a tie, the rule appearing first in the specification is selected.

For example, suppose the start state F00 is defined, and the following rules appear, with no other rules belonging to F00:

```plaintext
<F00> a+  => ( Tokens.as );
<F00> a+b+ => ( Tokens.asbs );
<F00> a+bb* => ( Tokens.asbs );
```

If the current start state is not F00, none of the rules will be considered. Otherwise, on input “aabbcc” all three rules are possible matches. The first rule is discarded, since the others match a longer prefix. The second rule is then selected, because it matches the same prefix as the third rule, but appears earlier in the specification.
CHAPTER 2. ML-ULEX

Figure 2.2: The ml-ulex grammar for regular expressions

2.3.1 Regular expression syntax

The syntax of regular expressions is given in Figure 2.2; constructs are listed in precedence order, from most tightly-binding to least. Escape codes are the same as in SML, but also include \uxxxx and \uxxxxxxxxx, where xxx represents a hexadecimal number which in turn represents a Unicode symbol. The specification format itself freely accepts Unicode characters, and they may be used within a quoted string, or by themselves.

Some examples:

\begin{verbatim}
0|1|2|3 denotes {0, 1, 2, 3}
[0123] denotes {0, 1, 2, 3}
0123 denotes {0123}
0* denotes \{\epsilon, 0, 00, \ldots\}
00* denotes \{0, 00, \ldots\}
0+ denotes \{0, 00, \ldots\}
[0 - 9]{3} denotes \{000, 001, 002, \ldots, 999\}
0 * &(...)* denotes \{\epsilon, 00, 0000, \ldots\}
\^ (abc) denotes \Sigma^* \setminus \{abc\}
[\^ abc] denotes \Sigma \setminus \{a, b, c\}
\end{verbatim}

2.3.2 Actions

Actions are arbitrary SML code enclosed in parentheses. The following names are in scope:
2.4. USING THE GENERATED CODE

YYBEGIN a function taking a start state and returning unit; changes to that start state.

yytext the matched text as a string.

yysubstr the matched text as a substring (avoids copying).

yyunicode the matched Unicode text as a list of Word32.words

continue a unit to “token” function which recursively calls the lexer on the input following the matched prefix, and returns its result. This can be used, for example, to skip whitespace.

yylineno the current line number, starting from 0.

yypos the current character, starting from 0.

? any name bound in the %defs section.

2.4 Using the generated code

The generated lexer module has a signature including the following:

```
type prestrm
type strm = prestrm * start_state
val streamify : (int -> string) -> strm
val lex : StreamPos.sourcemap -> strm -> token * strm
val getPos : strm -> StreamPos.pos
```

where token is the result type of the lexer actions, and start_state is an algebraic datatype with nullary constructors for each defined start state. In this interface, lexer start states are conceptually part of the input stream; thus, from an external viewpoint start states can be ignored. However, it is sometimes helpful to control the lexer start state externally, allowing contextual information to influence the lexer. This is why the strm type includes a concrete start_state component.

Note that the StreamPos module is part of the ml-lpt-lib library described in Chapter 4. A StreamPos.sourcemap value, combined with a StreamPos.pos value, compactly represents position information (line number, column number, and so on).

2.5 ml-lex compatibility

Running ml-ulex with the --ml-lex-mode option will cause it to process its input file using the ML-Lex format, and interpret the actions in a ML-Lex-compatible way. The compatibility extends to the bugs in ML-Lex, so in particular yylineno starts at 2 in --ml-lex-mode.
Chapter 3

ML-Antlr

Parsers analyze the syntactic structure of an input string, and are usually specified with some variant of context-free grammars. ml-antlr is a parser generator for Standard ML based on Terence Parr’s variant of $LL(k)$ parsing. The details of the parsing algorithm are given in the companion implementation notes; the practical restrictions on grammars are discussed in Section 3.5. A parser generated by ml-antlr is a functor; it requires a module with the LEXER signature:

```
signature LEXER = sig
  type strm
  type pos = StreamPos.pos
  val getPos : strm -> pos
end
```

Applying the parser functor will yield a module containing a parse function:

```
val parse : (strm -> ParserToks.token * strm) -> strm ->
  result_ty option * strm * ParserToks.token Repair.repair list
```

where `result_ty` is determined by the semantic actions for the parser. The ParserToks module is generated by ml-antlr (see Section 3.7) and the Repair module is available in the ml-lpt library (see chapter ??).

Notable features of ml-antlr include:

- Extended BNF format, including Kleene-closure (*), positive closure (+), and optional (?) operators.
- Robust, automatic error repair.
- Selective backtracking.
- “Inherited attributes”: information can flow downward as well as upward during a parse.
- Semantic predicates: a syntactic match can be qualified by a semantic condition.
- Grammar inheritance.
- Convenient default actions, especially for EBNF constructions.
- Convenient abbreviations for token names (e.g., "(" rather than LP)

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

```
ml-antlr file
```

where `file` is the name of the input ml-ulex specification. The output file will be called `file.sml`.
CHAPTER 3. ML-ANTLR

spec ::= ( declaration ; )∗
declaration ::= directive
    | nonterminal
directive ::= %defs code
    | %import STRING
    | %keywords symbol+
    | %name ID
    | %refcell ID : monotype ::= code
    | %start ID
    | %tokens : tokdef ( | tokdef )∗
code ::= ( . . )
tokdef ::= datacon ( ( STRING ) )?
datacon ::= ID
    | ID of monotype
monotype ::= standard SML syntax for monomorphic types
symbol ::= ID
    | STRING

Figure 3.1: The top-level ml-antlr grammar

3.1 Background definitions

Before describing ml-antlr, we need some terminology. A context-free grammar (CFG) is a set of
token (or terminal) symbols, a set of nonterminal symbols, a set of productions, and a start symbol S,
which must be a nonterminal. The general term symbol refers to both tokens and nonterminals.
A production relates a nonterminal A to a string of symbols a; we write this relation as A → a.
Suppose αAβ is a symbol string, and A is a nonterminal symbol. We write αAβ ⇒ αγβ if A → γ
is a production; this is called a one-step derivation. In general, a CFG generates a language,
which is a set of token strings. The strings included in this language are exactly those token
string derived in one or more steps from the start symbol S.

A parser recognizes whether an input string is in the language generated by a given CFG,
usually computing some value (such as a parse tree) while doing so. The computations
performed during a parse are called semantic actions.

3.2 Specification format

A ml-antlr specification is a list of semicolon-terminated declarations. Each declaration is either
a directive or a nonterminal definition. Directives are used to alter global specification properties
(such as the name of the functor that will be generated) or to define the tokens for the grammar.
The nonterminal definitions specify the grammar itself. The top-level grammar for ml-antlr is
given in Figure 3.1.

There are a few lexical details of the specification format worth mentioning. First, SML-style
comments ((* . . . *)) are treated as ignored whitespace anywhere they occur in the specification,
except in segments of code. The ID symbol used in the grammar stands for alpha-numeric-
derescore identifiers, starting with an alpha character. The code symbol represents a segment
of SML code, enclosed in parentheses. Extra parentheses occurring within strings or comments
in code need not be balanced. The STRING symbol represents a double-quoted string. Escape
codes may be used, so \ is written as \\.

A complete example specification appears in Chapter 5.
3.3 Directives

3.3.1 The %defs directive

The %defs directive is used to include a segment of code in the generated parser:

```ml
%defs
  fun helperFn x = (* ... *)

);  
```

All definitions given will be in scope for the semantic actions (see Section 3.4.6).

3.3.2 The %import directive

An %import directive can appear only once in a specification. The string given in the directive should hold the path to a grammar file (recall that `\` characters must be escaped). By default, all of the nonterminal definitions appearing in the specified file are included in the grammar. The token definitions of the imported file are not used. See Section 3.4.5 for details on changing or removing inherited nonterminals.

3.3.3 The %name directive

The name to use for the generated parser functor is specified using %name. In addition to the functor, ml-antlr will generate a module to define the token datatype; if the parser is named Example, then this module will be called ExampleToks.

3.3.4 The %start directive

A particular nonterminal must be designated as the start symbol for the grammar. The start symbol can be specified using %start; otherwise, the first nonterminal defined is assumed to be the start symbol.

3.3.5 The %tokens directive

The alphabet of the parser is defined using %tokens. The syntax for this directive resembles a datatype declaration in SML, except that optional abbreviations for tokens may be defined. For example:

```ml
%tokens
  : KW_let   ("let") | KW_in    ("in")
    | ID of string | NUM of Int.int
    | EQ        ("=") | PLUS     ("+")
    | LP        ("(") | RP       ")")

;  
```

Within nonterminal definitions, tokens may be referenced either by their name or abbreviation; the latter must always be double-quoted.

3.4 Nonterminal definitions

The syntax of nontermal definitions is given in Figure 3.2. As an illustration of the grammar, consider the following example, which defines a nonterminal with three productions, taking a formal parameter env:
nonterminal ::= ntdef
                    | %extend ntdef
                    | %replace ntdef
                    | %drop ID+
ntdef ::= ID forms? : proplist
formals ::= ( ID ( , ID )* )
proplist ::= production ( | production )* 
production ::= %try? named-item* ( %where code )? ( => code )? 
named-item ::= ( ID : )? item 
item ::= prim-item ?
         | prim-item +
         | prim-item *
prim-item ::= symbol ( @ code )?
         | ( proplist )
symbol ::= ID
         | STRING

Figure 3.2: The ml-antlr grammar for nonterminal definitions

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

Note that actions are only allowed at the end of a production, and that they are optional.

3.4.1 Extended BNF constructions

In standard BNF syntax, the right side of a production is a simple string of symbols. Extended BNF allows regular expression-like operators to be used: *, +, and ? can follow a symbol, denoting 0 or more, 1 or more, or 0 or 1 occurrences respectively. In addition, parentheses can be used within a production to enclose a subrule, which may list several |-separated alternatives, each of which may have its own action. In the following example, the nonterminal item_list matches a semicolon-terminated list of identifiers and integers:

    item_list : (( ID | INT ) ";")* ;

All of the extended BNF constructions have implications for the actions of a production; see Section 3.4.6 for details.

3.4.2 Inherited attributes

In most parsers, information can flow upward during the parse through actions, but not downward. In attribute grammar terminology, the former refers to synthesized attributes, while the latter refers to inherited attributes. Since ml-antlr is a predictive parser, it allows both kinds of attributes. Inherited attributes are treated as parameters to nonterminals, which can be used in their actions or semantic predicates. Formal parameters are introduced by enclosing them in parentheses after the name of a nonterminal and before its production list; the list of parameters will become a tuple. In the following, the nonterminal expr takes a single parameter called env:

    expr(env) : (* ... * ) ;
3.4. NONTERMINAL DEFINITIONS

If a nonterminal has a formal parameter, any use of that nonterminal is required to apply it to an actual parameter. Actual parameters are introduced in a production by giving the name of a nonterminal, followed by the @ sign, followed by the code to compute the parameter. For example:

```
assignment : ID "::=" expr@Env.emptyEnv ;
```

### 3.4.3 Selective backtracking

Sometimes it is inconvenient or impossible to construct a nonterminal definition which can be unambiguously resolved with finite lookahead. The %try keyword can be used to mark ambiguous productions for selective backtracking. For backtracking to take place, each involved production must be so marked. Consider the following:

```
A : %try B* ";"
  | %try B* "(" C+ ")"
;
```

As written, the two productions cannot be distinguished with finite lookahead, since they share an arbitrary long prefix of B nonterminal symbols. Adding the %try markers tells ml-antlr to attempt to parse the first alternative, and if that fails to try the second. Another way to resolve the ambiguity is the use of subrules, which do not incur a performance penalty:

```
A : B* ( ";"
  | "(" C+ ")"
 )
;
```

This is essentially left-factoring. See Section 3.5 for more guidance on working with the LL(k) restriction.

### 3.4.4 Semantic predicates

A production can be qualified by a semantic predicate by introducing a %where clause. Even if the production is syntactically matched by the input, it will not be used unless its semantic predicate evaluates to true. A %where clause can thus introduce context-sensitivity into a grammar. The following example uses an inherited env attribute, containing a variable-value environment:

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

In this example, if a variable is mentioned that has not been defined, the error is detected and reported during the parse as a syntax error.

Semantic predicates are most powerful when combined with selective backtracking. The combination allows two syntactically identical phrases to be distinguished by contextual, semantic information.
3.4.5 Modifying inherited nonterminals

Nonterminal definitions imported using the `%import` directive can be altered in several ways. In the simplest case, a nonterminal can be dropped entirely using the `%drop` keyword. Alternatively, the productions of a definition can be replaced using `%replace`, or new productions can be added using `%extend`. Note that these alterations may appear anywhere in the grammar; the order is irrelevant. For each imported nonterminal, only one of `%drop`, `%replace`, or `%extend` may be used. The resulting grammar must, of course, ensure that all used nonterminals are defined.

3.4.6 Actions

Actions for productions are just SML code enclosed in parentheses. In scope for the action are all the user definitions from the `%defs` directive. In addition, the formal parameters of the production are in scope, as are the semantic yield of all symbols to the left of the action (the yield of a token is the data associated with that token's constructor). In the following example, the first action has `env` and `exp` in scope, while the second action has `env` and `NUM` in scope:

```sml
atomicExp(env)
  : "(" exp@{env} ")" => ( exp )
  | NUM => ( NUM )
  ;
```

Notice also that the actual parameter to `exp` in the first production is `env`, which is in scope at the point the parameter is given; `exp` itself would not be in scope at that point.

An important aspect of actions is naming: in the above example, `exp` and `NUM` were the default names given to the symbols in the production. In general, the default name of a symbol is just the symbol's name. If the same name appears multiple times in a production, a number is appended to the name of each yield, start from 1, going from left to right. A subrule (any items enclosed in parentheses) is by default called `SR`. Any default name may be overridden using the syntax `name=symbol`. Overriding a default name does not change the automatic number for other default names. Consider:

```sml
foo : A bar=A A ("," A)* A*
```

In this production, the names in scope from left to right are: `A1`, `bar`, `A3`, `SR`, `A4`.

The EBNF operators `*`, `+` and `?` have a special effect on the semantic yield of the symbols to which they are applied. Both `*` and `+` yield a list of the type of their symbol, while `?` yields an option. For example, if `ID*` appeared in a production, its default name would be `ID`, and if the type of value of `ID` was `string`, it would yield a `string` list; likewise `ID?` would yield a `string` option.

Subrules can have embedded actions that determine their yield:

```sml
plusList : ((exp "+" exp => ( exp1 + exp2 )) ";" => ( SR ))* => ( SR )
```

The `plusList` nonterminal matches a list of semicolon-terminated additions. The innermost subrule, containing the addition, yields the value of the addition; that subrule is contained in a larger subrule terminated by a semicolon, which yield the value of the inner subrule. Finally, the semicolon-terminated subrule is itself within a subrule, which is repeated zero or more times. Note that the numbering scheme for names is restarted within each subrule.

Actions are `optional`: if an action is not specified, the default behavior is to return all nonterminals and non-nullary tokens in scope. Thus, the last example can be written as:

```sml
plusList : ((exp "+" exp => ( exp1 + exp2 )) ";")*
```

since "+" and ";" represent nullary token values.
3.5 The \textit{LL}(k) restriction

When working with any parser, one must be aware of the restrictions is algorithm places on grammars. When \texttt{ml-antlr} analyzes a grammar, it attempts to create a prediction-decision tree for each nonterminal. In the usual case, this decision is made using lookahead token sets. The tool will start with $k = 1$ lookahead and increment up to a set maximum until it can uniquely predict each production. Subtrees of the decision tree remember the tokens chosen by their parents, and take this into account when computing lookahead. For example, suppose we have two productions at the top level that generate the following sentences:

\begin{verbatim}
prod1 ==> AA
prod1 ==> AB
prod2 ==> BC
prod2 ==> AC
prod2 ==> C
\end{verbatim}

At $k = 1$, the productions can generate the following sets:

\begin{verbatim}
prod1 \{A, B\}
prod2 \{A, C\}
\end{verbatim}

and $k = 2$,

\begin{verbatim}
prod1 \{A, B, C\}
prod2 \{C, \textless E\text{EOF}\textgreater\}\end{verbatim}

Examining the lookahead sets alone, this grammar fragment looks ambiguous even for $k = 2$. However, \texttt{ml-antlr} will generate the following decision tree:

\begin{verbatim}
if LA(0) = A then
  if LA(1) = A or LA(1) = B then
    predict prod1
  else if LA(1) = C then
    predict prod2
else if LA(0) = B then
  predict prod1
else if LA(1) = C then
  predict prod2
\end{verbatim}

In \texttt{ml-antlr}, only a small amount of lookahead is used by default ($k = 3$). Thus, the following grammar is ambiguous for \texttt{ml-antlr}:

\begin{verbatim}
foo : A A A B
  | A A A A
  ;
\end{verbatim}

and will generate the following error message:

Error: lookahead computation failed for 'foo',
with a conflict for the following productions:

\begin{verbatim}
foo ::= A A A A E\text{EOF}
foo ::= A A A B E\text{EOF}
\end{verbatim}

The conflicting token sets are:

\begin{verbatim}
  k = 1: \{A\}
  k = 2: \{A\}
  k = 3: \{A\}
\end{verbatim}
Whenever a lookahead ambiguity is detected, an error message of this form is given. The listed 
productions are the point of conflict. The \( k = \ldots \) sets together give examples that can cause the 
ambiguity, in this case an input of \( AAA \).

The problem with this example is that the two \texttt{foo} productions can only be distinguished by 
a token at \( k = 4 \) depth. This situation can usually be resolved using \textit{left-factoring}, which lifts the 
common prefix of multiple productions into a single production, and then distinguishes the old 
productions through a subrule:

\[
\texttt{foo : A A A (B | A)}
\]

Recall that subrule alternatives can have their own actions:

\[
\texttt{foo : A A A ( B \Rightarrow ( \text{"got a B"} ) \ |
\texttt{A \Rightarrow ( \"got an A\"") })}
\]

making left-factoring a fairly versatile technique.

Another limitation of predictive parsing is \textit{left-recursion}, where a nonterminal recurs without 
any intermediate symbols:

\[
\texttt{foo : foo A A | B}
\]

Left-recursion breaks predictive parsing, because it is impossible to make a prediction for a left-
recursive production without already having a prediction in hand. Usually, this is quite easily 
resolved using EBNF operators, since left-recursion is most often used for specifying lists. Thus, 
the previous example can be rewritten as

\[
\texttt{foo : B (A A)*}
\]

which is both more readable and more amenable to \textit{LL}(k) parsing.

### 3.6 Position tracking

\texttt{ml-antlr} includes built-in support for propagating position information. Because the lexer mod-
ule is required to provide a \texttt{getPos} function, the tokens themselves do not need to carry explicit 
position information. A position \texttt{span} is a pair to two lexer positions (the type \texttt{StreamPos.span} 
is an abbreviation for \texttt{StreamPos.pos * StreamPos.pos}). Within action code, the position span 
of any symbol (token, nonterminal, subrule) is available as a value; if the yield of the symbol is 
named \texttt{Sym}, its span is called \texttt{Sym\_SPAN}. Note that the span of a symbol after applying the \* or + 
operators is the span of the entire matched list:

\[
\texttt{foo : A\* \Rightarrow (\* A\_SPAN starts at the first A and ends at the last \*)}
\]

In addition, the span of the entire current production is available as \texttt{FULL\_SPAN}. 


3.7 Using the generated code

When \texttt{ml-antlr} is run, it generates a tokens module and a parser functor. If the parser is given the name \texttt{Xyz} via the \texttt{%name} directive, these structures will be called \texttt{Xyz} and \texttt{XyzToks} respectively. The tokens module will contain a single datatype, called \texttt{token}. The data constructors for the \texttt{token} type have the same name and type as those given in the \texttt{%tokens} directive; in addition, a nullary constructor called \texttt{EOF} will be available.

The generated parser functor includes the following:

\begin{verbatim}
val parse : (strm -> ParserToks.token * strm) -> strm ->
  result_ty option * strm * ParserToks.token Repair.repair list
\end{verbatim}

where \texttt{result_ty} is the type of the semantic action for the grammar’s start symbol. The parser is given a lexer function and a stream. The result of a parse is the semantic yield of the parse, the value of the stream at the end of the parse, and a list of error repairs. If an unreparable error occurred, \texttt{NONE} is returned for the yield of the parse.

The \texttt{Repair} module is part of the \texttt{ml-lpt-lib} library; it is fully described in Chapter 4. It includes a function \texttt{repairToString}:

\begin{verbatim}
val repairToString : StreamPos.sourcemap -> ('a list -> string) ->
  'a repair -> string
\end{verbatim}

Likewise, the tokens module (\texttt{ParserToks} in this example) includes a function \texttt{toksToString}:

\begin{verbatim}
val toksToString : token list -> string
\end{verbatim}

Thus, although error reporting is customizable, a reasonable default is provided, as illustrated below:

\begin{verbatim}
let
  val sm = StreamPos.mkSourcemap()
  val (result, strm', errs) = Parser.parse (Lexer.lex sm) strm
  val errStrings = map (Repair.repairToString sm ParserToks.toksToString) errs
in
  print (concatWith "\n" errStrings)
end
\end{verbatim}
Chapter 4

The ml-lpt-lib library

4.1 The StreamPos structure

structure StreamPos : sig

    type pos = Position.int
    type span = pos * pos
    type sourcemap

    (* the result of moving forward an integer number of characters *)
    val forward : pos * int -> pos

    val mkSourcemap : unit -> sourcemap
    val same : sourcemap * sourcemap -> bool

    (* log a new line occurrence *)
    val markNewLine : sourcemap -> pos -> unit

    val lineNo : sourcemap -> pos -> int
    val colNo : sourcemap -> pos -> int
    val toString : sourcemap -> pos -> string

end

4.2 The Repair structure

structure Repair : sig

    datatype 'a repair_action
        = Insert of 'a list
        | Delete of 'a list
        | Subst of {
            old : 'a list,
            new : 'a list
        }
        | FailureAt of 'a
type 'a repair = StreamPos.pos * 'a repair_action

val repairToString : StreamPos.sourcemap -> ('a list -> string) -> 'a repair -> string

d
Chapter 5

A complete example

This chapter gives a complete example of a simple calculator language implemented using both ml-ulex and ml-antlr. Figure 5.1 gives the CM file for the project. Note that we are assuming that the ml-ulex and ml-antlr tools have been run by hand.

Library

structure CalcLex
functor CalcParse
structure CalcTest

is
$/basis.cm
$/smlnj-lib.cm
$/ml-lpt-lib.cm

calc.grm.sml
calc.lex.sml
calc-test.sml

Figure 5.1: The CM file: sources.cm
%name CalcLex;

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

%defs (
    structure T = CalcParse.Tok
    type lex_result = T.token
    fun eof() = T.EOF
);

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

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

Figure 5.2: The ml-ulex specification: calc.lex
\%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.foldr op+ 0 multExp::SR )
    ;

multExp(env)
  : prefixExp(env) ("*" prefixExp(env)) *
    => ( List.foldr op* 1 prefixExp::SR )
    ;

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

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

Figure 5.3: The ml-antlr specification: calc.grm
structure CalcTest =
  struct
    structure CP = CalcParse(CalcLex)

    fun calc str = let
      val sref = ref str
      fun input _ = (!sref before sref := "")
      val sm = StreamPos.mkSourcemap()
      val lex = CalcLex.lex sm
      val strm = CalcLex.streamify input
      val (r, strm', errs) = CP.parse lex AtomMap.empty strm
      in
        print (String.concatWith "\n" (map (Repair.repairToString sm Tok.toksToString) errs));
        r
      end
  end

Figure 5.4: The driver: calc-test.sml