## Contents

1 Overview 1

2 ML-ULex 3
   2.1 Specification format ........................................ 4
   2.2 Directives ................................................... 6
      2.2.1 The %arg directive ...................................... 6
      2.2.2 The %defs directive ................................... 6
      2.2.3 The %let directive ..................................... 6
      2.2.4 The %name directive .................................... 6
      2.2.5 The %header directive ................................ 7
      2.2.6 The %states directive .................................. 7
   2.3 Rules ....................................................... 7
      2.3.1 Regular expression syntax ............................... 8
      2.3.2 EOF rules ................................................. 8
      2.3.3 Actions .................................................. 9
   2.4 Using the generated code .................................. 9
   2.5 ml-lex compatibility ...................................... 10

3 ML-Antlr 11
   3.1 Background definitions .................................... 12
   3.2 Specification format ..................................... 12
   3.3 Directives ................................................ 14
      3.3.1 The %defs directive ................................... 14
      3.3.2 The %entry directive ................................ 14
      3.3.3 The %import directive ................................. 15
      3.3.4 The %keywords directive ............................ 15
      3.3.5 The %name directive .................................. 15
      3.3.6 The %header directive ................................ 16
      3.3.7 The %refcell directive ............................... 16
      3.3.8 The %start directive ................................. 16
      3.3.9 The %tokens directive ............................... 17
   3.4 Nonterminal definitions .................................. 17
      3.4.1 Extended BNF constructions .......................... 17
      3.4.2 Inherited attributes ................................. 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.3</td>
<td>Selective backtracking</td>
<td>18</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Semantic predicates</td>
<td>19</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Actions</td>
<td>19</td>
</tr>
<tr>
<td>3.5</td>
<td>The $LL(k)$ restriction</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Position tracking</td>
<td>22</td>
</tr>
<tr>
<td>3.7</td>
<td>Using the generated code</td>
<td>22</td>
</tr>
<tr>
<td>4</td>
<td>The ml-lpt-lib library</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>The <code>AntlrStreamPos</code> structure</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>The <code>AntlrRepair</code> structure</td>
<td>26</td>
</tr>
<tr>
<td>5</td>
<td>A complete example</td>
<td>27</td>
</tr>
</tbody>
</table>
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 \texttt{lex} and \texttt{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 \texttt{lex} and \texttt{yacc} tools, both in terms of the algorithms used, and often the syntax as well. The language Standard ML is no exception: \texttt{ml-lex} and \texttt{ml-yacc} are the SML incarnations of the old Unix tools.

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

Motivation

Most parser generators use some variation on \textit{LR} parsing, a form of \textit{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 \textit{LR}-ambiguous grammar.

- Because reductions take place as late as possible, the choice of reduction cannot

\(^1\)“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.
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 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.
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

\[
\text{val lex} : \text{AntlrStreamPos.sourcemap} \rightarrow \text{strm} \\
\quad \rightarrow \text{lex\_result} \times \text{AntlrStreamPos.span} \times \text{strm}
\]

where `lex\_result` is a type that must be defined by the user of `ml-ullex`. 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-ullex` offers the following improvements:

- Unicode is supported under the UTF8 encoding.
- Regular expressions can include intersection and negation operators.
- The position span of a token is automatically computed and returned to the lexer’s caller (as can be seen by the specification of the `lex` function above).
- 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-ullex [options] file
```

where `file` is the name of the input `ml-ullex` specification, and where `options` may be any combination of:
CHAPTER 2. ML-ULEX

```
--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 2.2.6 for details on start states).

--ml-lex-mode operate in ml-lex compatibility mode. See Section 2.5 for details.

--table-based generate a table-based lexer.

--fn-based generate a lexer that represents states as functions and transitions as tail calls.

--minimize generate a minimal machine. Note that this is slow, and is almost never necessary.

--strict-sml generate strict SML (i.e., do not use SML/NJ extensions). This flag is useful if you want to use the output with a different SML system.

The output file will be called file.sml.
```

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 grammar is given in Figure 2.1.

There are a few lexical details of the specification format worth mentioning. First, SML-style comments (\(\ast \ldots \ast\)) 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.
2.1. SPECIFICATION FORMAT

Figure 2.1: The ml-ulex grammar
CHAPTER 2.  ML-ULEX

2.2 Directives

2.2.1 The %arg directive

Specifies an additional curried parameter, appearing after the sourcemap parameter, that will be passed into the lex function and made available to all lexer actions.

2.2.2 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 = CalcParseToks.token
  fun eof() = CalcParseTokens.EOF
  fun helperFn x = (* ... *)
}
```

The definitions must at least fulfill the following signature:

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

unless EOF rules are specified, in which case only the lex_result type is needed (see Section 2.3.2). 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).

2.2.3 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,

```ml
%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,

```ml
%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.

2.2.4 The %name directive

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

2.2.5 The %header directive

The %header directive allows one to specify the lexer’s header. This directive is useful when you want to use a functor for the lexer. For example,

```
%header (functor ExampleLexFn (Extras : EXTRA_SIG));
```

Note that it is an error to include both the %header and %name directive in the same lexer specification.

2.2.6 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 2.3.3 for details on rule actions.

2.3 Rules

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. 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 determined 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 FOO is defined, and the following rules appear, with no other rules belonging to FOO:

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

If the current start state is not FOO, none of the rules will be considered. Otherwise, on input “aabbbc” 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.
\[ [. ] = \Sigma \]
\[ [re_1 \cdot re_2] = [re_1] \cdot [re_2] \]
\[ [re^*] = \bigcup_{i=0}^{\infty} [re]^i \]
\[ [\sim re] = \left( \bigcup_{i=0}^{\infty} \Sigma^i \right) \setminus [re] \]
\[ [re_1 \cup re_2] = [re_1] \cup [re_2] \]
\[ [re ?] = [re] \cup \{\epsilon\} \]

\[ [re_1 \& re_2] = [re_1] \cap [re_2] \]
\[ [re_1 \mid re_2] = [re_1] \cup [re_2] \]
\[ [re \{n\}] = [re]^n \]
\[ [re \{n, m\}] = \bigcup_{i=n}^{m} [re]^i \]

Figure 2.2: Semantics for regular expressions

2.3.1 Regular expression syntax

The syntax of regular expressions is given in Figure 2.1; constructs are listed in precedence order, from most tightly-binding to least. Escape codes are the same as in SML, but also include \uxxxxx and \Uxxxxxxxx, where xxxx 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.

The semantics for ml-ulex regular expressions are shown in Figure 2.2; they are standard. Some examples:

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

2.3.2 EOF rules

It is sometimes useful for the behavior of a lexer when it reaches the end-of-file to change depending on the current start state. Normally, there is a single user-defined eof function that defines EOF behavior, but EOF rules can be used to be more selective, as in the following example:

\[
\begin{align*}
\text{<INITIAL>} & \text{ <EOF>> } => ( \text{ Tok.EOF } ); \\
\text{<COMMENT>} & \text{ <EOF>> } => ( \text{ print "Error: unclosed comment"; } \\
& \text{ Tok.EOF } );
\end{align*}
\]

Other than the special <EOF>> symbol, EOF rules work exactly like normal rules.
2.4. USING THE GENERATED CODE

2.3.3 Actions

Actions are arbitrary SML code enclosed in parentheses. The following names are in scope:

- **YYBEGIN** a function taking a start state and returning unit; changes to that start state.
- **yysetStrm** a function taking a ULexBuffer.stream and returning unit; changes the current input source. The functions yystreamify, yystreamifyInstream and yystreamifyReader can be used to construct the stream; they work identically to the corresponding functions described in Section 2.4
- **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 Word.word values
- **continue** a unit to lex_result function that recursively calls the lexer on the input following the matched prefix, and returns its result. The span for the resulting token begins at the left border of the match that calls continue,
- **skip** a function identical to continue, except that it moves forward the left marker for the span of the returned token. Thus skip should be used for skipping whitespace.
- **yysm** the sourcemap for the lexer, to be used with the functions in the AntlrSourcePos module.
- **yypos** the position of the left border of the matched RE, starting from 0.
- **yylineno** the current line number, starting from 1. Note that lines are counted using the Unix line-ending convention.
- **yycolno** the current column number, starting from 1.
- **?** any name bound in the %defs section.

2.4 Using the generated code

The generated lexer module has a signature that includes the following specifications:

- `type prestrm
  type strm = prestrm * start_state
val streamify : (unit -> string) -> strm
val streamifyReader : (char, 'a) StringCvt.reader -> 'a -> strm
val streamifyInstream : TextIO.instream -> strm`


```
val lex : AntlrStreamPos.sourcemap -> strm -> lex_result * AntlrStreamPos.span * strm
```

where `lex_result` is the result type of the lexer actions, and `start_state` is an algebraic datatype with nullary constructors for each defined start state. Note that `lex_result` must be defined as a type using the `#defs` directive. In this interface, lexer start states are conceptually part of the input stream; thus, from an external viewpoint, start states can be ignored. It is, however, 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 `AntlrStreamPos` module is part of the `ml-lpt-lib` library described in Chapter 4. An `AntlrStreamPos.sourcemap` value, combined with an `AntlrStreamPos.pos` value, compactly represents position information (line number, column number, and so on). An `AntlrStreamPos.span` is a pair of `pos` values that specify the character position of the leftmost character of the scanned token and the position of the character following the token (respectively). By default, the span will cover the entire sequence of characters consumed by a call to the lexer, but one may use the `skip` function (see Section 2.3.3) to ignore leading whitespace, etc. when computing the span. Code to compute the span is generated automatically by `ml-ulex`.

### 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 ANTLR_LEXER signature:

```ml
signature ANTLR_LEXER = sig
  type strm
  val getPos : strm -> AntlrStreamPos.pos
end
```

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

```ml
val parse : (Lex.strm -> ParserToks.token * AntlrStreamPos.span * Lex.strm) -> Lex.strm -> result_ty option * strm * ParserToks.token AntlrRepair.repair list
```

where `result_ty` is determined by the semantic actions for the parser. The ParserTokens module is generated by ml-antlr (see Section 3.7) and the AntlrRepair module is available in the ml-lpt library (see Chapter 4).

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 [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.

- **--latex**
  generate a simple LaTeX version of the grammar, named file.tex.

- **--unit-actions**
  ignore the action code in the grammar, and instead return () for every production.

The output file will be called file.sml.

### 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 $\alpha$; we write this relation as $A \rightarrow \alpha$. Suppose $\alpha A \beta$ is a symbol string, and $A$ is a nonterminal symbol. We write $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if $A \rightarrow \gamma$ 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 (or just 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 supporting infrastructure for the grammar. The nonterminal definitions specify the grammar itself. The grammar for ml-antlr is given in Figure 3.1.
3.2. SPECIFICATION FORMAT

\[
\begin{align*}
\text{spec} & ::= ( \text{declaration} ; )^* \\
\text{declaration} & ::= \text{directive} \\
& \quad | \text{nonterminal} \\
\text{directive} & ::= \text{%defs code} \\
& \quad | \text{%entry ID (, ID)*} \\
& \quad | \text{%import STRING ( %dropping symbol+)?)} \\
& \quad | \text{%keywords symbol (, symbol)*} \\
& \quad | \text{%name ID} \\
& \quad | \text{%header code} \\
& \quad | \text{%refcell ID : monotype = code} \\
& \quad | \text{%start ID} \\
& \quad | \text{%tokens : tokdef ( | tokdef)*} \\
& \quad | \text{%nonterms : datacon ( | datacon)*} \\
\text{code} & ::= (\ldots) \\
\text{tokdef} & ::= \text{datacon ( ( STRING ) )}\? \\
\text{datacon} & ::= \text{ID} \\
& \quad | \text{ID of monotype} \\
\text{monotype} & ::= \text{standard SML syntax for monomorphic types} \\
\text{nonterminal} & ::= \text{ID formals}\?: \text{prodlst} \\
\text{formals} & ::= (\text{ID (, ID)*}) \\
\text{prodlst} & ::= \text{production ( | production)*} \\
\text{production} & ::= \text{%try\? named-item\? ( %where code )\? ( => code )?} \\
\text{named-item} & ::= (\text{ID : })\? \text{item} \\
\text{item} & ::= \text{prim-item ?} \\
& \quad | \text{prim-item +} \\
& \quad | \text{prim-item *} \\
\text{prim-item} & ::= \text{symbol ( @ code )}\? \\
& \quad | (\text{prodlst}) \\
\text{symbol} & ::= \text{ID} \\
& \quad | \text{STRING} \\
\text{ID} & ::= \text{standard SML identifier} \\
\text{STRING} & ::= \text{standard SML double-quoted string} \\
\end{align*}
\]

Figure 3.1: The \texttt{ml-antlr grammar}
SML-style comments (\(\text{(* ... *)}\)) are treated as ignored whitespace anywhere they occur in the specification, except in segments of code. The \textit{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.

Most \texttt{ml-antlr} declarations are \textit{cumulative}: they may appear multiple times in a grammar specification, with each new declaration adding to the effect of the previous ones. Thus, for instance, the specification fragment

\begin{verbatim}
%tokens : foo ;
%tokens : bar of string ;
\end{verbatim}

is equivalent to the single directive

\begin{verbatim}
%tokens : foo | bar of string ;
\end{verbatim}

and similarly for nonterminal definitions and so on. All declarations are cumulative except for the \%start and \%name directives. The reason for treating specifications in this way is to give the \%import directive very simple semantics, as described below.

### 3.3 Directives

#### 3.3.1 The \%defs directive

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

\begin{verbatim}
%defs (   
  fun helperFn x = (* ... *)
);
\end{verbatim}

All definitions given will be in scope for the semantic actions (see Section 3.4.5).

#### 3.3.2 The \%entry directive

It is often useful to parse input based on some fragment of a grammar. When a non-terminal is declared to be an \textit{entry point} for the grammar via \%entry, \texttt{ml-antlr} will generate a separate \texttt{parse} function that expects the input to be a string derived from that nonterminal. Given a grammar with a nonterminal \texttt{exp} and the declaration

\begin{verbatim}
%entry exp;
\end{verbatim}

the generated parser will include a function

\begin{verbatim}
val parseexp : (Lex.strm -> ParserToks.token * AntlrStreamPos.span * Lex.strm) 
  -> Lex.strm 
  -> exp_ty option * strm * ParserToks.token AntlrRepair.repair list
\end{verbatim}

where \texttt{exp_ty} is the type of the actions for the \texttt{exp} nonterminal. Note that if \texttt{exp} has inherited attributes (Section 3.4.2) they will appear as a tuple argument, curried after the lexer argument:
3.3. DIRECTIVES

\[
\text{val } \text{parseexp} : (\text{Lex.strm } \to \text{ParserToks.token } \times \text{AntlrStreamPos.span } \times \text{Lex.strm}) \\
\quad \to \text{attributes} \\
\quad \to \text{Lex.strm} \\
\quad \to \text{exp\_ty option } \times \text{strm } \times \text{ParserToks.token AntlrRepair.repair list}
\]

Finally, the \textit{start} symbol (Section 3.3.8) is always an entry point to the grammar, but the generated function is simply called \text{parse}.

3.3.3 The \texttt{%import} directive

The \texttt{%import} directive is used to include one grammar inside another. The string given in the directive should hold the path to a grammar file, and \texttt{\backslash} characters must be escaped. By default, all declarations appearing in the specified file are included in the resulting grammar, except for \texttt{%start}, \texttt{%entry}, and \texttt{%name} declarations. However, individual tokens or nonterminals can be dropped by listing them in the \texttt{%dropping} clause of an \texttt{%import} declaration. Since nonterminal definitions are cumulative (Section 3.4), the imported nonterminals can be extended with new productions simply by listing them. The final grammar must, of course, ensure that all used tokens and nonterminals are defined.

3.3.4 The \texttt{%keywords} directive

When a syntax error is discovered, \texttt{ml-antlr} attempts to find a single-token repair to the input that will allow the parse to continue. Changes to the input involving keywords can drastically alter the meaning of the input, so it is usually desirable to favor non-keyword repairs. The \texttt{%keywords} directive is used to tell \texttt{ml-antlr} which tokens should be considered keywords.

3.3.5 The \texttt{%name} directive

The prefix to use for the name of the generated parser functor is specified using \texttt{%name}. In addition to the functor, \texttt{ml-antlr} will generate a module to define the \texttt{token} datatype. If the declaration

\[
\text{\texttt{%name} Example;}
\]

appears in the specification, then the parser functor will be named \texttt{ExampleParseFn} and the tokens module will be called \texttt{ExampleTokens}.

3.3.6 The \texttt{%header} directive

The \texttt{%header} directive allows one to specify the parser’s functor. This directive is useful when you want to add additional parameters to the functor, but the declaration must include the \texttt{Lex} structure with signature \texttt{ANTLR_LEXER}. For example,

\[
\text{\texttt{%header} (}
\quad \text{functor ExampleParseFn (}
\quad \quad \text{structure } \text{Lex } : \texttt{ANTLR_LEXER}
\quad \text{)}
\text{)}
\]
3.3.7 The %refcell directive

Because semantic actions must be pure (for backtracking and error repair), they cannot make use of standard reference cells to communicate information. Nonterminals may inherit attributes (Section 3.4.2), which allows information to flow downward, but in some cases flowing information this way can become extremely tedious. For example, a data structure may only need to be updated at a single point in the grammar, but in order to properly thread this state through the grammar, an inherited attribute would have to be added and propagated through every nonterminal.

The %refcell directive is used to declare a backtracking-safe reference cell and make it available to all semantic actions. Reference cells are declared by giving the name, type, and initial value for the cell. Each cell is bound in the semantic actions as a standard SML ref value. Thus, for example, we might have the following specification fragment:

```ml
%refcell symbols : StringSet.set = ( StringSet.empty );

exp
  : INT
  | (exp)
  | ID => ( symbols := StringSet.add(!symbols, ID); ID )

;  
```

The result of this fragment is that all symbol uses are tracked, in any use of the exp nonterminal, but without having to manually thread the data structure state through the grammar.

The final contents of a reference cell is returned as an extra result from the parse following the repair list.

3.3.8 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.

3.3.9 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:
%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 nonterminal definitions is given in Figure 3.1. As an illustration of the grammar, consider the following example, which defines a nonterminal with three productions, taking a formal parameter `env`:

```plaintext
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.

As with most directives, the non-terminal definitions are cumulative. For example, the definition of `atomicExp` above could also be written as three separate rules.

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

### 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:

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

All of the extended BNF constructions have implications for the actions of a production; see Section 3.4.5 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) : (* ... *) ;
```

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+ "")"

; %try B* ";"
```

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+ ")"
)

; %try B* ";"
```

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

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:

```sml
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 Actions

Actions for productions are just SML code enclosed in parentheses. Because of potential backtracking and error repair, action code should be pure (except that they may update ml-antlr refcells; see the %refcell directive).

In scope for an 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 overriden using the syntax name=symbol. Overriding a default name does not change the automatic number for other default names. Consider:
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:

\[
\text{plusList} : ((\text{exp } "+" \text{ exp } \Rightarrow (\text{ exp1 + exp2 })) \text{ ; } \Rightarrow (\text{ SR }))^* \Rightarrow (\text{ SR })
\]

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

\[
\text{plusList} : ((\text{exp } "+" \text{ exp } \Rightarrow (\text{ exp1 + exp2 })) \text{ ; } \text{ })^*
\]

since "+" and " ; " represent nullary token values.

### 3.5 The LL(k) restriction

When working with any parser, one must be aware of the restrictions its algorithm places on grammars. When 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:

\[
\begin{align*}
\text{prod1} & \Rightarrow AA \\
\text{prod1} & \Rightarrow AB \\
\text{prod1} & \Rightarrow BC \\
\text{prod2} & \Rightarrow AC \\
\text{prod2} & \Rightarrow C
\end{align*}
\]

At \( k = 1 \), the productions can generate the following sets:

\[
\begin{align*}
\text{prod1} & \{A, B\} \\
\text{prod2} & \{A, C\}
\end{align*}
\]
and $k = 2$,

\[
\begin{align*}
\text{prod1} & \{A, B, C\} \\
\text{prod2} & \{C, \texttt{<EOF>}\}
\end{align*}
\]

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

\[
\begin{align*}
\text{if} & \, \text{LA}(0) = A \, \text{then} \\
& \, \text{if} \, \text{LA}(1) = A \, \text{or} \, \text{LA}(1) = B \, \text{then} \\
& \quad \text{predict prod1} \\
& \, \text{else if} \, \text{LA}(1) = C \, \text{then} \\
& \quad \text{predict prod2} \\
& \, \text{else if} \, \text{LA}(0) = B \, \text{then} \\
& \quad \text{predict prod1} \\
& \, \text{else if} \, \text{LA}(1) = C \, \text{then} \\
& \quad \text{predict prod2}
\end{align*}
\]

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

\[
\text{foo} : A \ A \ A \ B \\
\mid A \ A \ A \ A \\
\]

and will generate the following error message:

\[
\text{Error: lookahead computation failed for 'foo',} \\
\text{with a conflict for the following productions:} \\
\text{foo ::= A A A A EOF} \\
\text{foo ::= A A A B EOF} \\
\text{The conflicting token sets are:} \\
k = 1: \{A\} \\
k = 2: \{A\} \\
k = 3: \{A\}
\]

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 foo productions can only be distinguished by a token at $k = 4$ depth. This situation can usually be resolved using left-factoring, which lifts the common prefix of multiple productions into a single production, and then distinguishes the old productions through a subrule:

\[
\text{foo} : A \ A \ A \ (B \mid A)
\]

Recall that subrule alternatives can have their own actions:
foo : A A A ( B => ( "got a B" )
  | A => ( "got an A" )
  )
;

making left-factoring a fairly versatile technique.

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

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

foo : B (A A)*
;

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

### 3.6 Position tracking

\texttt{ml-antlr} includes built-in support for propagating position information. Because the lexer module is required to provide a \texttt{getPos} function, the tokens themselves do not need to carry explicit position information. A position span is a pair to two lexer positions (the type \texttt{AntlrStreamPos.span} is an abbreviation for \texttt{AntlrStreamPos.pos} \( \ast \) \texttt{AntlrStreamPos.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 \( \ast \) or \( + \) operators is the span of the entire matched list:

foo : A\ast \Rightarrow ( \ast A\_SPAN starts at the first A and ends at the last \ast )

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{XyzParseFn} and \texttt{XyzTokens} 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:
3.7. USING THE GENERATED CODE

val parse : (Lex.strm -> ParserToks.token * AntlrStreamPos.span * Lex.strm) -> Lex.strm -> result_ty option * strm * ParserToks.token AntlrRepair.repair list

where result_ty is the type of the semantic action for the grammar’s start symbol. The parse function 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 unrepairable error occurred, NONE is returned for the yield of the parse.

Note that if the start symbol for the grammar includes an inherited attribute (or a tuple of attributes), it will appear as an additional, curried parameter to the parser following the lexer parameter. Suppose, for example, that a grammar has a start symbol with an inherited Int.int AtomMap.map, and that the grammar yields Int.int values. The type of its parse function is as follows:

val parse :
  (strm -> ParserToks.token * strm) ->
  Int.int AtomMap.map ->
  strm ->
  Int.int option * strm * ParserToks.token AntlrRepair.repair list

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

val repairToString :
  (‘token -> string) -> AntlrStreamPos.sourcemap -> ‘token repair -> string

which can be used to produce error messages from the parser’s repair actions. There is a more refined version of this function that allows one to specialize the string representation of tokens based on whether the repair adds or deletes them.

datatype add_or_delete = ADD | DEL

val repairToString :
  (add_or_delete -> ‘token -> string) -> AntlrStreamPos.sourcemap ->’token repair -> string

Likewise, the tokens module (ParserTokens in this example) includes a function:

val toString : token -> string

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

let
  val sm = AntlrStreamPos.mkSourcemap()
  val (result, strm', errs) = Parser.parse (Lexer.lex sm) strm
  val errStrings = map (AntlrRepair.repairToString ParserTokens.toString sm) errs
  in
    print (String.concatWith "\n" errStrings)
end
The `toString` function will convert each token to its symbol as given in a `%tokens` directive, using abbreviations when they are available. By substituting a different function for `toString`, this behavior can be altered.
Chapter 4

The \texttt{ml-lpt-lib} library

4.1 The \texttt{AntlrStreamPos} structure

\begin{verbatim}
structure AntlrStreamPos : sig

    type pos = Position.int
    type span = pos * pos
    type sourceloc = { fileName : string option, lineNo : int, colNo : int }
    type sourcemap

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

    val mkSourcemap : unit -> sourcemap
    val mkSourcemap' : string -> sourcemap

    val same : sourcemap * sourcemap -> bool

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

        (* resynchronize to a full source location *)
    val resynch : sourcemap -> pos * sourceloc -> unit

    val sourceLoc : sourcemap -> pos -> sourceloc
    val fileName : sourcemap -> pos -> string option
    val lineNo : sourcemap -> pos -> int
    val colNo : sourcemap -> pos -> int
    val toString : sourcemap -> pos -> string
    val spanToString : sourcemap -> span -> string

end
\end{verbatim}

4.2 The \texttt{AntlrRepair} structure

structure AntlrRepair : sig
datatype 'tok repair_action
  = Insert of 'tok list
  | Delete of 'tok list
  | Subst of {'old : 'tok list,
              'new : 'tok list}
  | FailureAt of 'tok

val actionToString : ('tok -> string) -> 'tok repair_action -> string
val repairToString : ('tok -> string) -> AntlrStreamPos.sourcemap -> 'tok repair -> string

datatype add_or_delete = ADD | DEL

(* return a string representation of the repair action. This version
* uses the add_or_delete information to allow different token names
* for deletion (more specific) and addition (more general). *)
val actionToString' : (add_or_delete -> 'tok -> string) -> 'tok repair_action -> string
val repairToString' : (add_or_delete -> 'tok -> string) -> AntlrStreamPos.sourcemap -> 'tok repair -> string

end
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.

Library

```
structure CalcLexer
functor CalcParseFn
structure CalcTest

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

calc.grm : ml-antlr
calc.lex : ml-ulex
calc-test.sml
```

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

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

%defs ( 
  structure T = CalcTokens 
  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 Calc;

%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@2(env)
  "in" exp@3(AtomMap.insert(env, Atom.atom ID, exp1)) => ( exp2 )
  | addExp@2(env)
;

addExp(env)
  : multExp@2(env) ("+") multExp@2(env) => ( List.foldr op 0 (multExp::SR) )
;

multExp(env)
  : prefixExp@2(env) ("*") prefixExp@2(env) => ( List.foldr op* 1 (prefixExp::SR) )
;

prefixExp(env)
  : atomicExp@2(env)
  | "-" prefixExp@2(env)
   => ( `prefixExp )
;

atomicExp(env)
  : ID
   => ( valOf(AtomMap.find (env, Atom.atom ID)) )
  | NUM
  | "(" exp@2(env) ")"
;

Figure 5.3: The ml-antlr specification: calc.grm
structure CalcTest =
structure CP = CalcParseFn(CalcLexer)

fun tok2s (ID s) = s
| tok2s (NUM n) = Int.toString n
| tok2s tok = CalcTokens.toString tok

(* val calc : TextIO.instream -> Int.int *)
fun calc instrm = let
val sm = AntlrStreamPos.mkSourcemap()
val lex = CalcLexer.lex sm
val strm = CalcLexer.streamifyInstream instrm
val (r, strm', errs) = CP.parse lex AtomMap.empty strm
in
  print (String.concatWith "\n" (List.map (AntlrRepair.repairToString tok2s sm)
     errs));
  r
end

end

Figure 5.4: The driver: calc-test.sml
Chapter 6

Change history

Here is a history of changes to the SML/NJ Language Processing Tools. More details can be found in the SML/NJ NOTES and README files.

SML/NJ 110.77

Fixed an inconsistency in the way that ml-antlr and ml-ulex handled the contents of a %defs declaration. ml-ulex made these definitions visible in the UserDeclarations substructure, whereas ml-antlr hid them. We have changed the behavior of ml-ulex to match that of ml-antlr (i.e., hide the user definitions). We chose to hide the user definitions in ml-ulex because they are usually not useful outside the lexer, hiding them reduces the size of the generated code, and definitions that are needed outside the lexer can be defined in an external module. Note that the UserDeclarations substructure remains visible when ml-ulex is run in ml-lex compatibility mode.

Added the actionToString’ and repairToString’ functions to the AntlrRepair structure. These functions allow one to specialize the printing of tokens based on whether they are being added or deleted.

Removed the toksToString function from the tokens structure that ml-antlr generates. It was originally for use by the AntlrRepair structure, but that structure does not use it.

SML/NJ 110.72

Added --strict-sml flag to ml-ulex for MLton compatibility.

Added %header directive to the ml-antlr parser generator.