ML-GLR User's Manual

Version 1.0

Hans Leiß
Centrum für Informations- und Sprachverarbeitung
Universität München
Oettingenstr. 67
D-80538 München

April 23, 2009

Abstract

ML-GLR is a parser generator for acyclic context-free grammars building a tabular generalized LR-parser. It is derived from ML-Yacc[2] by replacing ML-Yacc's deterministic LALR(1)-parser by a chart parser with left-to-right filtering of chart entries, following Nederhof/Satta[8]. ML-GLR is intended for generating parsers and evaluators for ambiguous languages, in particular, fragments of natural languages. This document explains the syntax and usage of ML-GLR and points out the differences to ML-Yacc.

This document is based in large parts on the documentation of ML-Lex and ML-Yacc which carries the following copyright and disclaimer.

> Lexical analyzer generator for Standard ML. Version 1.6, October 1994
>
> Copyright (c) 1989-92 by Andrew W. Appel, James S. Mattson, David R. Tarditi
>
> This software comes with ABSOLUTELY NO WARRANTY. This software is subject only to the PRINCETON STANDARD ML SOFTWARE LIBRARY COPYRIGHT NOTICE, LICENSE AND DISCLAIMER, (in the file "COPYRIGHT", distributed with this software). You may copy and distribute this software; see the COPYRIGHT NOTICE for details and restrictions.

> ML-YACC COPYRIGHT NOTICE, LICENSE AND DISCLAIMER.
>
> Copyright 1989, 1990 by David R. Tarditi Jr. and Andrew W. Appel
>
> Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the names of David R. Tarditi Jr. and Andrew W. Appel not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.
>
> David R. Tarditi Jr. and Andrew W. Appel disclaim all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall David R. Tarditi Jr. and Andrew W. Appel be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

The same copyright notice and disclaimer apply to ML-GLR for Hans Leiß and Georg Klein.

Please send questions and bug reports to: `leiss@cis.uni-muenchen.de`

Contents

1 Introduction

ML-GLR is a parser generator in Standard ML that shares its input format and some of the internal machinery with ML-Yacc[2]. Unlike ML-Yacc, it is not restricted to LALR(1) grammars, but generates parsers for acyclic context-free grammars, including ambiguous ones. ML-GLR tests for acyclicity of the grammar and reports an error if there are cycles. If the grammar is cyclic, an input may have infinitely many analyses and the generated parser will loop.

ML-GLR is derived from David R. Tarditi and Andrew W. Appel's source code of ML-Yacc for SML/NJ, version 110.67[1]. It uses the grammar specification format of ML-Yacc, which in turn is similar to the format of Unix Yacc[6], but sacrifices some features of ML-Yacc, in particular:

- ML-GLR does not have an error correction facility,
- ML-GLR does not have the tracing of ML-Yacc's `lib/parser2.sml`,
- ML-GLR is not bootstrapping.

Error correction in ML-Yacc is based on the LALR(1) parsing strategy and cannot be extended to the chart parsing used in ML-GLR. Instead of the tracing of ML-Yacc's `parser2.sml`, ML-GLR has flags that cause a trace of its activities and a description of its internal parser data be printed and lets the user inspect the parse chart. To parse the grammar of ML-GLR's grammar specifications, we use ML-Yacc and hence don't need a bootstrapping parser.

Basic difference between ML-Yacc and ML-GLR

ML-Yacc computes from a grammar specification the LR(0)-automaton of all phrase forms, derives from it the LALR(1)-automaton by taking lookahead information into account, and delivers the deterministic pushdown automaton using the LALR(1)-automaton as parser.

In contrast, ML-GLR implements the tabular GLR-parser of Nederhof/Satta[8]. Basically, it proceeds as follows. From the specified grammar G compute the compacted LR(0)-automaton and the non-deterministic push-down automaton based on it; turn the push-down automaton into a binary one which pops at most two symbols off the stack at a time; this binary push-down automaton corresponds to a binary context-free grammar, the so-called *2LR-cover grammar* of G; use a modification of Cocke-Yonger-Kasami's tabular recognizer to parse inputs with respect to the cover grammar; finally, from the parse chart, extract the parse trees, transform them into parse trees with respect to G and evaluate them with the semantic annotations of rules in G. The compacted LR(0)-automaton and the binary push-down automaton increase the amount of possible sharing between computations of the non-deterministic push-down automaton. The chart serves to tabulate subcomputations and avoid the graph-structured stack of Tomita's GLR-parser[10].

Ambiguities

The intended use of ML-GLR is to generate parsers for fragments of natural language that can be specified by acyclic context-free grammars. Since the input grammar need not be deterministic, the generated parser may find several parse trees for the same input string. Hence, in general, it returns a list of values, one for each parse. In addition, semantic ambiguities arise since ML-GLR accepts grammars where two rules differ in their semantic annotations only. This is useful, for example, to provide several readings of the same sentence according to the relative scope of quantifiers in their noun phrases. Since rule numbers are included in the parse trees, the "same" syntactic structure can yield different meanings.

ML-GLR works with ambiguous lexers. This is useful because many words (resp. word forms) are overloaded. Although a lexer for natural language should be based on some standard lexicon, it is

possible to use ML-Lex for specifying a lexer that does not deliver tokens, but token lists instead, where a list represents a set of alternative tokens. Tokens made of the same word occurrence may differ in the grammar terminal and in the semantic value they carry.

In parsing of programming languages, operator precedence and associativity declarations are a means to exclude certain syntactic ambiguities by selecting one of several possible syntactic structures. ML-GLR admits terminal precedence and associativity declarations in the grammar specification. As with ML-Yacc, the precedence of a grammar rule is the one of its rightmost terminal, or the one of the explicit precedence annotation in the rule. ML-GLR remembers these precedences in the corresponding rules of the cover grammar and on finishing a parse, removes those trees that do not conform to the precedence declarations. Although more general methods to filter away unwanted parses are needed in the context of natural language parsing, operator precedence and associativity declarations can be useful, for example, to reduce ambiguities arising with nested occurrences of connectives like *and* and *or*.

Parsing and evaluation

The syntax rules in the source grammar may be annotated with Standard ML programs for semantic evaluation. In particular, higher-order functions can be used as values. Parsing and evaluation are implemented as separated phases. To fill the parse chart for input w, ML-GLR inserts into the field corresponding to a subword v of w the nonterminals of the cover grammar with the cover grammar's rule and the identifiers for the fields for v_1 and v_2 with $v = v_1v_2$ that caused the insertion; moreover, the source grammar's corresponding rule is remembered, if there is one. For atomic subwords v, the terminal and the number of values of tokens with that terminal are added to the field. When the parse chart for an input string is completed, trees are extracted and transformed to trees with respect to the source grammar. By bottom-up evaluation, the value of a tree is determined using the rule and value numbers attached to the tree, and an environment of token lists. Thus, in the current implementation the values play no role in filling the chart. An option to perform evaluation steps while filling the chart may be added in a future version.

Since the grammar is acyclic, the number of trees that can be extracted from the chart is finite, but may be exponential in the length of the input. Although the chart parser may find parse trees with different root nonterminals for the same input, ML-GLR expects the grammar to have a single start symbol. The reason for this restriction is that different start symbols are likely to carry values of different type, so that the union of these types has to be defined as the parser result type anyway. It is then natural to have a distinguished start symbol with values of this union type.

2 Installing ML-GLR

To install the stand-alone version of ML-GLR, unpack the archive in the SML/NJ root directory (whose subdirectory `bin` contains `ml-build`). Go to the main directory of ML-GLR, adjust the `build` script if you do not want to use `ml-glr` as the program name, and run

```
> make ml-glr
```

in a shell.

When the grammar specification has a `%graphview` declartation or when the parser-flags `trees` or `covertrees` are used, the graph visualizer `dot` [5] and the previewer `gv` [**?**] are needed. When the debug-flag is used, an HTML-version of the parse chart is be produced and presented with the browser `firefox`. Edit the values of `gv`, `dot` and `browser` in `lib/chartparser` and `src/glr.sml` and `src/export-glr.sml` if these programs are not installed.

3 Specification of grammar and lexer

Just like ML-Yacc, the parser generator ML-GLR generates a parser from a grammar specification file. The generated parser needs a lexer providing tokens of the kind expected by the parser's terminals, and such lexers can be produced with ML-Lex from a lexer specification file.

3.1 Grammar specification

ML-GLR grammar specification files have to be in the same format as those for ML-Yacc, consisting of three sections separated by **%%**, namely

ML-Yacc user declarations (Standard ML code)
%%
ML-Yacc declarations (terminals, nonterminals, precedences, ...)
%%
ML-Yacc rules (grammar rules)

Each section may contain comments in the format of Standard ML, i.e. enclosed like `(* ... *)`. For details, see ML-Yacc User's Manual[2]. Differences between ML-GLR and ML-Yacc are with respect to the ML-Yacc declarations only.

ML-Yacc user declarations: This section may contain declarations of values, types and structures to be used in the ML-Yacc declarations and rules sections. Often, such values and datatypes will be defined in an auxiliary structure, and then this structure is openend in the ML-Yacc user declarations, so that short names can be used in the ML-Yacc declarations and ML-Yacc rules sections.[1]

ML-Yacc declarations or rather **ML-GLR declarations**: ML-GLR supports the required ML-Yacc declarations of parser name, grammar terminals and nonterminals, and position type. These have to be given in the following format, where a symbol is an ML-identifier and a type is a monomorphic ML-type:

```
%name <symbol>
%term <symbol> | ...  <symbol> of <type> | ...
%nonterm <symbol> | ...  <symbol> of <type> | ...
%pos <type>
```

The lexer will assign to each token a terminal and a "payload" consisting of a value of the type of the terminal, if it has one (such as the integer value of an identifier of type `int`), and two values of the position type specified as `%pos`. The type specified as `%pos` will often be `int`, and the payload may hold the line and column numbers of the token in an input file. If the payload is supposed to hold the name of the input file as well, the position type could be `string * int`.

ML-GLR also supports the following of ML-Yacc's optional declarations, where code and pattern mean source code and value patterns of Standard ML, respectively:

```
%eop <symbol> ...
%start <symbol>
%verbose
%pure
%header (<code>)
%arg (<pattern>) :<type>
%token_sig_info (<code>)
```

[1] However, datatypes of the *terminal* symbols will occur in the signature `<name>_TOKENS` of the structure holding the token constructors derived from the grammar. In this case, use the long type names when declaring the terminals or use the `%token_sig_info`-declaration to add the types to the signature `<name>_TOKENS`, c.f. page 16.

and the two additional declarations

```
%graphview
%lexheader (<code>)
```

The %eop-declaration specifies the terminals that may follow the start symbol, also called end-of-parse symbols. The generated parsers read from their input stream up to an end-of-parse symbol, and only then start building the parse chart. Hence, for interactive usage it is necessary to declare end-of-parse symbols to enforce evaluation before an end of file is reached.

The %start-declaration specifies the start symbol of the grammar. (See the ML-Yacc documentation and remark 8.1 for how to simulate multiple start symbols.)

The %verbose flag tells ML-GLR to produce a human-readable description file <spec>.desc containing the compacted LR(0)-automaton, the 2LR covergrammar compiled from the source grammar, and the left-to-right filter between nonterminals of the cover grammar. The %graphview flag causes ML-GLR to produce a graphical representation of the compacted LR(0)-automaton and send it to the previewer.

The %pure flag tells ML-GLR that semantic actions are guaranteed to terminate and are essentially free of side effects, and that a simpler form of action-functions is to be generated.

By the %header declaration, the user can provide a header for the functor <name>ValsFun that computes an intermediate structure containing the rules of the cover grammar and the semantic actions. <name> is the parser name declared under %name. The declaration is of the form

```
%header (functor <name>ValsFun(structure Token:TOKEN
                               structure Grammar:GRAMMAR
                                 sharing type Token.term = Grammar.term
                               ...) : <name>_VALS)
```

where ... may contain additional argument types, structures and sharing constraints. The argument structure Grammar and the sharing constraint for term are specific to ML-GLR. Also specific to ML-GLR is the declaration

```
%lexheader (functor <name>LexFun(structure Tokens:<name>_TOKENS
                                 ...) : LEXER)
```

which has to match the %header-declaration of a lexer specification file for ML-Lex. These declarations need only be given if some additional arguments ... are involved or if one wants to use a lexer with an additional argument (see remark 5.1), in which case the %lexheader-declaration is

```
%lexheader (functor <name>LexFun(structure Tokens:<name>_TOKENS
                                   ...) : ARG_LEXER)
```

ML-GLR uses the functors <name>ValsFun and <name>LexFun to define structures <name>Vals, <name>Lex, and <name>Parser and write them to a file <spec>.grm.lnk.sml (cf. section 5).

The %arg declaration allows the specification of an additional argument for the generated parser.

The %token_sig_info declaration is used to provide ML-GLR with ML-declarations that are to be inserted in the <name>_TOKENS signature which ML-GLR constructs from the grammar's terminals. For example, when the type of a terminal depends on types provided by an argument structure of <name>ValsFun, these types can be included in <name>_TOKENS using the %token_sig_info declaration (cf. figure 10).

ML-GLR also supports operator precedence and associativity declarations for terminals:

```
%left <symbol> ...
%right <symbol> ...
%nonassoc <symbol> ...
```

All terminals mentioned in the same line have the given associativity and are of the same precedence; those in subsequent such lines are of higher precedence. Likewise, ML-GLR supports rule precedence declarations using annotations of the form

```
%prec <symbol>
```

on the right hand sides of grammar rules, immediately preceeding the semantic annotation. A rule inherits precedence and associativity from the terminal mentioned in the `%prec <term>` annotation, if there is one, or from the rightmost terminal on its right hand side, otherwise.

Since ML-GLR does not have ML-Yacc's error correction facility that is based on the LALR(1)-automaton of the grammar, it does *not* support ML-Yacc's declarations that are related to error recovery, i.e. those of the form

```
%noshift <symbol> ...
%keyword <symbol> ...
%prefer <symbol> ...
%subst <symbol> for <symbol> ...
%change <tokens> -> <tokens> | ...
%value <symbol>(<code>)
```

are ignored by ML-GLR. A warning is issued when they are used. Finally, ML-GLR does *not* support ML-Yacc's declaration

```
%nodefault
```

which is used by ML-Yacc to suppress the generation of a default action in certain states of the LR table in the parsers generated by ML-Yacc.

ML-Yacc rules: This section contains the context-free grammar rules with associated semantic actions and rule precedence. A grammar rule has the form

```
<nonterm> :  <term or nonterm> ...  %prec <term> (<action>)
```

where `<nonterm>` and `<term>` are symbols from the corresponding ML-Yacc declarations. The precedence declaration following the list of symbols on the right hand side is optional; if it is missing, the precedence of the last terminal on the right hand side is taken as the precedence of the rule. The `<action>` is a piece of ML code defining the value of the left hand side nonterminal from the values of the symbols on the right hand side. The code refers to a symbol's value by using the symbol extended by the number of its occurrence on the right hand side of the rule; if the symbol occurs only once, the number may be omitted. The code may refer to a symbol's payload positions by `<symbol>left` and `<symbol>right`, respectively. The value of the code is used for the value of the left hand side nonterminal, so its type must be the type declared for that nonterminal in the ML-Yacc declarations section. Several grammar rules with the same left hand side may be given by a list of right hand sides separated with `|`.

Example 3.1 Figure 1 shows the specification of an ambiguous grammar named `Amb`.

3.2 Lexer specification

Parsers generated by ML-GLR need suitable lexers, which can be produced by ML-Lex. From a lexer specification file `<spec>.lex`, ML-Lex produces an output file `<spec>.lex.sml` containing the lexer as an ML-structure of ML-functor. We shortly describe the format of lexer specifications used by ML-Lex and which instances will give a lexer that fits to a parser generated by ML-GLR. For details on ML-Lex, see the documentation [3].

An ML-Lex specification consists of three sections separated by `%%` and has the format

```
   (* ML-GLR user declarations *)
datatype dt = L of int | C of int
fun modify (L n) = C (3*n) | modify (C n) = C n

%% (* ML-GLR declarations *)
%name     Amb
%pos      int
%lexheader (functor AmbLexFun(structure Tokens: Amb_TOKENS
                              structure Interface: INTERFACE) : LEXER)
%term     b of int | c | EOF
%nonterm  S of dt | A of dt | E
%eop      EOF

%% (* ML-GLR rules, with different meanings for alike syntactic structure *)
S : A E (A)            (* constituent values = lexical values *)
  | A E (modify A)     (* constituent values = modified lexical ones *)
A : b (L b)            (* terminal with two values for token "a" *)
  | c (L 0)            (* another terminal for token "a" (see amb.lex) *)
E :   ()               (* deletion rule *)
```

Figure 1: Specification `examples/amb.grm` of a grammar with ambiguities

ML-Lex user declarations
%%
ML-Lex definitions
%%
ML-Lex rules

ML-Lex user declarations: This section consists of Standard ML declarations; they will be copied to a substructure `UserDeclarations` in the lexer generated. It has to contain two declarations

```
type lexresult = ...
fun eof () : lexresult = ...
```

If one wants to specify a lexer whose functions take an additional argument (which has to be declared as an ML-Lex definition `%arg (<pattern>:<type>);`), the `eof`-function must have the form

```
fun eof (<pattern>:<type>) : lexresult = ...
```

instead. The type `lexresult` is the type of values returned by actions of the lexer. ML-Yacc and ML-Lex are used to parse deterministic languages, where each lexical item has a unique reading. In this case, `lexresult` is just a type of tokens build from a terminal (possibly with a value of its type) and an occurrence position in the input. If lexical items may have several readings, which is the intended case for applications of ML-GLR, `lexresult` will be the type of token *lists*, each list element representing a different reading. For printing leaves of parse trees, we add a string component to this list.

If the lexer is to be used with a parser generated by ML-GLR, the user declarations need to contain additional declarations:

```
type pos = ...        (* as in %pos ... in the grammar specification *)
type svalue = Tokens.svalue
type ('a,'b) token = ('a,'b) Tokens.token
type lexresult = (svalue,pos) token list * string
type arg = ...        (* if %arg ... is in the grammar specification *)
```

The user declarations may fix the type of an additional argument to the lexer, abbreviations for structures, introduce auxiliary structures, etc.:

```
type lexarg = ...    (* if %arg (pat:...); is among the ML-Lex definitions *)
structure T = Tokens
structure Interface = struct ... end
...
```

ML-Lex definitions: This section may contain flags, provide names for the lexer structure or functor, and auxilary abbreviations to be used in the rules section. If the flag

```
%full
```

is present, ML-Lex will create a lexer for an 8-bit character set such as ISO Latin 9, permitting input characters with codes in the range 0-255; otherwise, the lexer accepts 7-bit ASCII characters with codes 0-127. See Roger Price's "User's Guide to ML-Lex and ML-Yacc"[9] on how to obtain lexers for other sets of Unicode characters, and for an explanation of the three additional flags

```
%reject
%count
%posarg
```

of minor importance.

If the lexer is to be combined with a parser generated by ML-Yacc or ML-GLR, it has to be parameterized by a structure `Tokens` which the parser generator produces from the terminals and position type of the grammar specification. In this case[2], an ML-Lex definition

```
%header (functor <name>LexFun(structure Tokens:<name>_TOKENS ...));
```

has to be provided, where `<name>` is given in the `%name`-declaration of the grammar specification, and `...` may contain additional argument types or structures etc.

If the lexer is to use an additional argument (see p.14 and remark 5.1), an ML-Lex definition

```
%arg (<pattern>:<type>);
```

is needed, where the ML-pattern typically is a variable and the type has to be monomorphic. Finally, there may be ML-Lex definitions that ease the formulation of the rules in the ML-Lex rules section. The definition

```
%s <identifier> ... ;
```

defines a list of identifiers for *states* of the lexer; a rule of the lexer may be restricted to some of these states. The definition

```
<identifier> = <regular expression>;
```

gives a name to a regular expression, which allows for a more compact formulation of the rules.

ML-Lex rules: This section consists of a sequence of unrestricted rules of the form

[2] In the general case, ML-Lex delivers the lexer as a structure `MLex` in the output file `<spec>.lex.sml`. If the ML-Lex definitions section contains a declaration

```
%structure <identifier>
```

the given identifier rather than `MLex` will be used to name this structure.

```
<regular expression> => (<code>);
```

or rules restricted to a list (formed by angle brackets `<...>`) of state identifiers

```
<<state> ...> <regular expression> => (<code>);
```

There is an implicit state `INITIAL` in which the lexer begins to operate. In a given state, the lexer uses the unrestricted rules and the rules that are restricted to this state. The lexer matches some piece of input using the regular expression of a rule and acts as the code of the rule says, thereby typically constructing a token from the recognized piece of input. If several rules are applicable, the rule with the longest match, or the first rule among several that match the same piece of input, is chosen. If no match is possible, the lexer raises an exception `LexError`.

The matched piece of input is stored in a string variable `yytext`, its initial position in the input file in `yypos`[3], and if the flag `%count` is present, the current line number in `yylineno`[4]. The rule's code can be built from ML-values defined in the ML-Lex user declarations or the values

```
value := yytext | yypos | yylineno
```

The code has to be a sequence of ML-expressions involving these values, extended by the following expressions:

```
expr := lex() | continue() | YYBEGIN <state> | REJECT()
```

Expression `lex()` recursively calls the lexer; in case `%arg` was given, one has to use `continue()` instead. Expression `YYBEGIN <state>` instructs the lexer to switch to the given state, and expression `REJECT()` makes the lexer behave as if the current rule had not matched (this can be used only when the flag `%reject` was given).

Omitting some details on matching escape sequences `\n`, `\t` etc. and reserved characters, the regular expressions admitted in ML-Lex rules and the strings matching them are as follows:

(i) any character c except the reserved characters in `?*+|()^$ /;.=<>[]{}"\` matches the regular expression c, and a reserved character c matches $\backslash c$,

(ii) any character in the string w except `\`, `-`, `^` matches the regular expression `[`w`]`,

(iii) any character whose code is in the range of those of c and d matches `[`c`-`d`]`,

(iv) any character except `\`, `-`, `^` that is not in the word w matches the regular expression `[^`w`]`,

(v) any character except newline matches the regular expression `.`,

(vi) any string w (possibly including reserved characters) matches the regular expression `"`w`"`,

(vii) any string w matches the regular expression `(`e`)`, if it matches e,

(viii) any string w matches the regular expression `{`i`}`, if w matches e and i is a name given to e,

(ix) any string matching one of e_1, e_2 matches the regular expression e_1`|`e_2,

(x) if string w_i matches e_i for $i = 1, 2$, then w_1w_2 matches the regular expression e_1e_2,

(xi) any concatenation of a (nonempty) sequence of strings matching e matches the regular expression e`*` (resp. e`+`),

[3] Due to a bug in ML-Lex, the first position of the file is given as 2; see remark 8.2 on ways to deal with this.

[4] According to the documentation of ML-Lex [3], this is less efficient than using one's own line counter defined among the ML-Lex user declarations.

(xii) the empty string as well as any string matching e matches the regular expression e`?`,

(xiii) any concatenation of k strings matching e matches the regular expressions e{k} and e{n`,`m} when $n \leq k \leq m$.

Example 3.2 Figure 2 shows a specification `amb.lex` for a lexer fitting to the parser generated from `amb.grm` of figure 1. Token constructors `T.b` and `T.c` correspond to the terminals `b` and `c`.

```
structure T = Tokens
structure I = Interface

type ('a,'b) token = ('a,'b) T.token
type svalue = T.svalue
type pos = I.pos
type lexresult = (svalue,pos) token list * string

open I              (* providing lin, col, init, line, eol, coln, error *)
fun eof() : lexresult = (T.EOF(!lin,!col)::nil,"")
%%
%header (functor AmbLexFun(structure Tokens: Amb_TOKENS
                           structure Interface: INTERFACE) : LEXER );
%full
%s WORK;
alpha=[a-zA-ZÄÖÜäöüß];
comment= %(.*)\n;
whitespace=[\t\ ]*;
%%
<INITIAL>{whitespace} => (init(); YYBEGIN WORK; lex());
<WORK>{whitespace}    => (eol(yypos,yytext); lex());
{comment}             => (eol(yypos,yytext); line(); YYBEGIN WORK; lex());
\n                    => (eol(yypos,yytext); line(); lex());
"a"                   => (coln(yypos); (T.b(1,!lin,!col)::T.b(2,!lin,!col)::
                                        T.c(!lin,!col)::nil,yytext));
"."                   => (let val (l,c)=(!lin,!col) in
                              init(); YYBEGIN INITIAL; eof() end);
.                     => (coln(yypos);
                              error ("ignoring char "^yytext,!lin,!col); lex());
```

Figure 2: Lexer specification `amb.lex`

The functor `AmbLexFun` in the `%header`-declaration takes an argument structure `Interface`, which needs to be generated with the functor from `interface.sml` shown in figure 3. In the user declarations section (which is copied to the body of `AmbLexFun`) the structure `Interface` is opend to have its values `pos`, `lin`, `col`, `init`, `eol`, `coln` and `error` available in the rules section.

See the documentation of ML-Lex[3] on how to build a stand-alone lexer from a specification `<spec>.lex` and auxiliary files like `interface.sml`. Below we will only explain how to use ML-GLR and ML-Lex to generate a parser from a grammar specified in `<spec>.grm` that uses a lexer generated from `<spec>.lex`.

```
signature INTERFACE =
  sig
      type pos
      val lin : pos ref
      val col : pos ref
      val init : unit -> unit
      val line : unit -> unit
      val eol : int * string -> unit
      val coln : int -> unit
      val error : string * pos * pos -> unit
      type arg
      val default : arg
  end

functor Interface (type arg val default:arg) : INTERFACE =
  struct
    type pos = int
    val lin = ref 1
    val col = ref 0
    val eolpos = ref 0
    fun init () = (lin := 1; col := 0; eolpos := 0)
    fun line () = (lin := !lin + 1)
    fun eol (yypos,yytext) = (eolpos := yypos+(size yytext))
    fun coln (yypos) =       (col    := yypos-(!eolpos))
    fun error (errmsg,lin:pos,col:pos) =
        TextIO.output(TextIO.stdOut,
                      "Error in line " ^ (Int.toString lin)
                                       ^ ", column " ^ (if lin = 1
                                                        then (Int.toString (col-2))
                                                        else (Int.toString col))
                                       ^ ": " ^ errmsg ^ "\n")
    type arg = arg
    val default = default
  end
```

Figure 3: An interface `interface.sml` to the lexer and parser

4 Operation of ML-Lex and ML-GLR

This section gives an overview of what ML-Lex and ML-GLR will do. It can be skipped on first reading.

4.1 Operation of ML-Lex

From a specification file `<spec>.lex` containing a `%header`- but no `%arg`-field, ML-Lex generates an output file `<spec>.lex.sml` which contains a functor

```
functor <name>LexFun(structure Tokens: <name>_TOKENS ...) : LEXER =
   struct
     structure UserDeclarations =
        struct <ML-Lex user declarations> end
     ... <defined using ML-Lex definitions and rules>...
     fun makeLexer (reader: int -> string) = ...
   end
```

The functor arguments are taken from the `%header`-field of the specification, its result signature is

```
signature LEXER =
   sig
       structure UserDeclarations :
          sig
              type ('a,'b) token
              type pos
              type svalue
              type lexresult = (svalue,pos) token list * string
          end
       val makeLexer : (int -> string) -> unit -> UserDeclarations.lexresult
   end
```

The type `lexresult`, which is not present in `ML-Yacc`'s `LEXER` signature, is fixed to be token *lists*, since ML-GLR is intended to be used with ambigous lexers. Type `token` is abstract in `LEXER`, so that lexers produced by `makeLexer` cannot examine the structure of tokens. The first argument of `makeLexer` is a function `reader:int -> string`, whose integer argument is the number of characters to be read in one go from the input file.

When the lexer specification has an `%arg`-field, the result signature of the functor is

```
signature ARG_LEXER =
   sig
       structure UserDeclarations :
          sig
              type ('a,'b) token
              type pos
              type svalue
              type lexresult = (svalue,pos) token list * string
              type arg
          end
       val makeLexer : (int -> string) -> UserDeclarations.arg ->
                                          unit -> UserDeclarations.lexresult
   end
```

In this case, the lexer produced takes an argument (such as a file name) before yielding a function from `unit` to `lexresult`. Lexers used by parsers generated by ML-GLR must match the signature `LEXER` or, when the lexing functions are supposed to take an additional argument, `ARG_LEXER`.

To build a lexer, one first needs to build a `Lexer:LEXER` structure using `<name>LexFun`. A suitable argument structure `Tokens:<name>_TOKENS` is produced by ML-GLR from the grammar specification `<spec>.grm`, and the signature `<name>_TOKENS` is written to file `<spec>.grm.sig`. All additional argument structures have to be provided by the user (cf. section 5).

The user is responsible for declaring types `token`, `pos`, and `svalue` among the ML-Lex user declarations in `<spec>.lex`, using corresponding types from the argument structure `Tokens`, as mentioned above (see page 9). It is necessary to declare `type lexresult = (svalue,pos) token list * string` and the rules have to deliver token lists rather than tokens, paired with strings.

4.2 Operation of ML-GLR

From the grammar specification file `<spec>.grm`, ML-GLR produces a functor

```
<name>ValsFun(structure Token: TOKEN
              structure Grammar: GRAMMAR
                sharing type Token.term = Grammar.term
              ...) : <name>_VALS =
   struct
```

```
    structure ParserData: PARSER_DATA = ...
    structure Tokens: <name>_TOKENS = ..
  end
```

and writes it out to file `<spec>.grm.sml`. The functor takes argument structures defining the datatypes of token and grammar etc., and additional argument structures `...` as specified in the `%header`-declaration of the specification.

The structure `ParserData:PARSER_DATA` contains all the values except for the lexer that are needed to call the polymorphic chart parser on the 2LR-cover grammar of an acyclic context-free grammar. The structure `Tokens:<name>_TOKENS` contains the token constructors determined from the terminals of the grammar and will be needed by the lexer.

The signatures of these two structures as well as the result signature of the functor are written to the file `<spec>.grm.sig`. They are as follows:

```
signature PARSER_DATA =
  sig
    type pos       (* the type of token positions *)
    type svalue    (* the type of semantic values of parse tree nodes *)
    type arg       (* the type of the user-supplied argument to the parser *)
    type result    (* the type of the result of the parse, produced by
                      applying Actions.extract to the root's semantic value. *)

    structure Grammar : GRAMMAR
    structure Token : TOKEN
    sharing type Token.term = Grammar.term

    val spec : string             (* the user's grammar specification file *)
    val grammar : Grammar.grammar (* 2LR-cover grammar of the user's grammar *)
    val nontermToTerm : Grammar.nonterm -> Grammar.term option
    val emptyNt : Grammar.nonterm
    val initNt : Grammar.nonterm
    val lrfilter : Grammar.nonterm * Grammar.nonterm -> bool

    (* a default semantic value, the evaluation functions associated with
       the source grammar rules, and a function to extract the final result
       from the values of the root of a tree: *)

    structure Actions :
      sig
        val void : svalue
        val actions : int * pos * (svalue * pos * pos) list * arg ->
            Grammar.nonterm * (svalue * pos * pos)
        val extract : svalue list -> result
      end
  end
```

This signature differs from the corresponding one of ML-Yacc in that it has a different `Actions` structure, various additional values, and no substructure `EC` for error correction.

The signature `<name>_TOKENS` specifies the token constructor functions for the terminals.

```
signature <name>_TOKENS =
  sig
    <token_sig_info>
    type ('a,'b) token
    type svalue
```

```
    val <terminal>: 'a * 'a -> (svalue,'a) token
    ...
    val <terminal>: <type> * 'a * 'a -> (svalue,'a) token
    ...
  end
```

This signature depends on the types of terminals. If these are not build from pervasive types only, they either have to be defined in the environment when the functors `<name>ValsFun` and `<name>LexFun` are defined, or must be an argument of or be defined in an argument structure of the functors, and then have to be included in `<name>_TOKENS` using the `%token_sig_info` declaration of the grammar specification. For an example, see section 8.

The result signature `<name>_VALS` of the functor `<name>ValsFun` instantiates the abstract types of tokens and values by the ones derived from the grammar specification:

```
signature <name>_VALS =
  sig
    structure Tokens : <name>_TOKENS
    structure ParserData : PARSER_DATA
      sharing type ParserData.Token.token = Tokens.token
      sharing type ParserData.svalue = Tokens.svalue
  end
```

Moreover, ML-GLR creates a file `<spec>.grm.lnk.sml` that applies the `<name>ValsFun` functor to build a structure `<name>Parser` containing the parser for the specified grammar:

```
structure <name>Vals = <name>ValsFun(structure Token = ChartParser.Token
                                     structure Grammar = ChartParser.Grammar  ...)
structure <name>Lex  = <name>LexFun(structure Tokens = <name>Vals.Tokens ...)
structure <name>Parser = ...
```

This file is used to join the lexer and parser together, see section 5.

4.3 The chart parser for binary grammars with LR-filter

Internally, the user parser produced by ML-GLR is based on a chart parser for binary grammars with a left-to-right filtering relation. For readers familiar with ML-Yacc, we present a few signatures of internal structures of ML-GLR and point out some differences to those of the LALR(1)-parser produced by ML-Yacc.

`TOKEN` is a signature for tokens revealing the internal structure of tokens. In contrast to ML-Yacc's signature `TOKEN` it does not have a substructure `LrTable:LR_TABLE`, but defines the type of terminals:

```
signature TOKEN =
  sig
      datatype term = T of int
      datatype ('a,'b) token = TOKEN of term * ('a * 'b * 'b)
      val sameToken : ('a,'b) token * ('a,'b) token -> bool
  end
```

`CHART_PARSER` is a signature for a polymorphic chart parser. It contains the substructures `Grammar` and `Chart` (`recognize` and `Chart` could be omitted) instead of `LrTable` in ML-Yacc's `LR_PARSER`.

```
signature CHART_PARSER =
```

```
sig
    structure Stream : STREAM
    structure Grammar : GRAMMAR
    structure Token : TOKEN
      sharing type Token.term = Grammar.term
    structure Chart : CHART where type term = Token.term

    exception ParseError

    val recognize : {grammar : Grammar.grammar,
                     lrfilter : Grammar.nonterm * Grammar.nonterm -> bool,
                     emptyNt : Grammar.nonterm,
                     initNt : Grammar.nonterm}
                    -> (Grammar.term * int) list list (* (terminal,|vals|) per token *)
                    -> bool ref                       (* progress report flag *)
                    -> Chart.chart

    val parse : {arg : 'arg,
                 spec: string,
                 grammar : Grammar.grammar,
                 emptyNt : Grammar.nonterm,
                 initNt : Grammar.nonterm,
                 lrfilter : Grammar.nonterm * Grammar.nonterm -> bool,
                 CGnontermToSGterm: Grammar.nonterm -> Grammar.term option,
                 CGnontermToSGstring: Grammar.nonterm -> string option,
                 lexer : (('b,'c) Token.token list * string) Stream.stream,
                 saction : int * 'c * ('b * 'c * 'c) list * 'arg
                           -> Grammar.nonterm * ('b * 'c * 'c),
                 void : 'b,
                 error : string * 'c * 'c -> unit,
                 debug : bool ref,
                 trees : bool ref,
                 covertrees : bool ref,
                 progress : bool ref
                } -> 'b list * ((('b,'c) Token.token list * string) Stream.stream)
end
```

Both `recognize` and `parse` need a 2LR-cover-grammar `grammar`, a left end marker `initNt` and a left-to-right filter relation `lrfilter`, all of which are produced by ML-GLR from the user's source grammar. The function `recognize` returns a parse chart whose fields contain grammar rules; from these, parse trees with respect to the cover grammar are reconstructed and translated to parse trees with respect to the source grammar. The function `parse` needs some additional arguments, in particular an ambigous lexer `lexer`, a default semantic value `void`, and the semantic `action`s from the source grammar, and delivers the list of semantic values of the recognized prefix of the input stream, and the remainder of the stream.

5 Joining lexer and parser

To obtain a user parser from the grammar and lexer specifications `<spec>.grm` and `<spec>.lex`, the functor `<name>ValsFun` produced by ML-GLR has to be applied to get a structure

```
structure <name>Vals =
   <name>ValsFun(structure Token = ChartParser.Token
                 structure Grammar = ChartParser.Grammar ...)
```

containing two substructures <name>Vals.ParserData and <name>Vals.Tokens. The functor <name>LexFun generated by ML-Lex has to be applied to the latter structure and further arguments as mentioned in the %header-declaration of <spec>.lex to produce a suitable lexer:

```
structure <name>Lex =
   <name>LexFun(structure Tokens = <name>Vals.Tokens ...)
```

Finally, the lexer has to be combined with the 2LR-cover grammar in ParserData and the polymorphic chart parser to produce the user's parser:

```
structure <name>Parser =
   Join(structure ParserData = <name>Vals.ParserData
        structure Lex = <name>Lex
        structure ChartParser = ChartParser)
```

This code to build the structure <name>Parser is generated from the grammar specification and written to <spec>.grm.lnk.sml. ML-GLR uses the %header declaration to know about argument structures of the functor <name>ValsFun besides Token and Grammar; similarly, the %lexheader declaration informs ML-GLR about argument structures of <name>LexFun besides structure Tokens = <name>Vals.Tokens.[5]

The heading functors may use additional argument structures structure X:SIGX besides Token, Grammar and Tokens; the functor call in the generated file will use arguments structure X = X for these. The user has to define corresponding argument structures X in a separate file mentioned in the description file <spec>.cm, to avoid cyclic dependencies between files. See examples/NL/nl.grm for how to use the %lexheader declaration with an additional argument structure Interface:INTERFACE, or ger.grm in section 8 with an additional argument Absyn:ABSYN and some type sharing constraints.

If the lexer specification file <spec>.lex does not contain a %header-declaration, but a %structure-declaration or none of these, then the file <spec>.lex.sml produced by ML-Lex contains a lexer *structure*, not a functor. The lexer is then not suited to work with the Tokens provided by the generated parser, and the <spec>.grm.lnk.sml-file will not be usable as well.

5.1 User parser signature

The structures <name>Lex:LEXER, <name>Vals.ParserData, and ChartParser can only be joined to the structure <name>Parser if they share some datatypes, as demanded by the functor definition

```
functor Join(structure Lex : LEXER
             structure ParserData: PARSER_DATA
             structure ChartParser : CHART_PARSER
               sharing ParserData.Token = ChartParser.Token
               sharing ParserData.Grammar = ChartParser.Grammar
               sharing type Lex.UserDeclarations.svalue = ParserData.svalue
               sharing type Lex.UserDeclarations.pos = ParserData.pos
               sharing type Lex.UserDeclarations.token = ParserData.Token.token
            ) : PARSER = ...
```

Parsers created by applying the functor Join of ML-GLR will match the signature PARSER:

```
signature PARSER =
```

[5] Recall that the %lexheader declaration in <spec>.grm has to match the %header declaration of the *lexer* specification file (without its trailing ;). It is easy to modify tool/tools.sml to suppress the generation of .lnk.sml-files, if one wants to write the corresponding definitions in the file <spec>.parse.sml containing the front end.

```
sig
    structure Token : TOKEN
    structure Stream : STREAM
    exception ParseError

    type pos     (* the type of token positions *)
    type result  (* the type of the result extracted from a parse tree *)
    type arg     (* the type of the user-supplied argument to the parser *)
    type svalue  (* the type of semantic values of parse tree nodes *)

    (* val makeLexer turns a reader into a stream of token alternatives *)

    type lexresult = (svalue,pos) Token.token list * string

    val makeLexer : (int -> string) -> lexresult Stream.stream

    (* val parse takes a stream of token alternatives and a function to print
       errors and returns a list of values of type result (for a prefix of
       the stream) and the unused remainder of the stream.  *)

    val parse : (lexresult Stream.stream) * (string * pos * pos -> unit) * arg
                -> result list * (lexresult Stream.stream)

    val sameToken : (svalue,pos) Token.token * (svalue,pos) Token.token -> bool

    val debug : bool ref
    val trees : bool ref
    val covertrees : bool ref
    val progress : bool ref
 end
```

This signature differs from the `PARSER` signature of ML-Yacc's functor `Join` in the type `lexresult`, the omission of a lookahead argument in the parse function, and the flags at the end. The first three flags control the printing of debugging information and graphic representation of parse trees; the last one can be used to get informed about the progress made while filling the chart.

If the lexer specification has an `%arg` declaration, the structure `<name>Parser` is built using `JoinWithArg` rather than `Join` and will match the folllowing signature:

```
signature ARG_PARSER =
    sig
        structure Token : TOKEN
        structure Stream : STREAM
        exception ParseError

        type pos
        type result
        type arg
        type svalue
        type lexresult = (svalue,pos) Token.token list * string
        type lexarg

        val makeLexer : (int -> string) -> lexarg -> lexresult Stream.stream
        val parse : (lexresult Stream.stream) * (string * pos * pos -> unit) * arg
                    -> result list * (lexresult Stream.stream)

        val sameToken : (svalue,pos) Token.token * (svalue,pos) Token.token -> bool
```

```
    val debug : bool ref
    val trees : bool ref
    val covertrees : bool ref
    val progress : bool ref
end
```

It extends PARSER by the type lexarg, which enters the type of makeLexer.

Remark 5.1 A parser <name>Parser:ARG_PARSER uses a lexer Lex:ARG_LEXER. A specification file <spec>.lex for a lexer of type ARG_LEXER needs to define

```
type arg = ty
```

for some type ty, in the user declarations section and give a declaration

```
%arg = (pat);
```

in the ML-Lex definitions section. The functor building the lexer must have ARG_LEXER as its result signature,

```
%header (functor <name>LexFun(structure Tokens:<name>_TOKENS ...):ARG_LEXER);
```

and all uses of lex and eof in the declaration and rules sections have to fit to the type arg -> unit -> lexresult. The front end structure for the parser needs to define

```
type lexarg = ty
```

In order to properly use JoinWithArg rather than Join to build <name>Parser in the generated file <spec>.grm.lnk.sml, we demand that <spec>.grm has a declaration

```
%lexheader (functor <name>LexFun(structure Tokens:<name>_TOKENS ...):ARG_LEXER);
```

even if no additional argument structures ... are used.

5.2 Front end to the user parser

The functions makeLexer, parse, and sameToken in the structure <name>Parser can be used to define various parsing functions, for example one to parse from a file, another to parse from a string, or one to parse from the interactive top-level.

To define such a front end to the parser, the user has to provide a reader int -> string to create a lexresult stream, and an error-printer string * pos * pos -> unit and a stop-token to create a parsing function. Figure 4 shows a front end structure with a function for parsing from the top-level. It assumes that <spec>.grm declares DOT as an end-of-parse terminal and that the lexer <spec>.lex delivers tokens built from DOT and EOF when, say, "." and "stop" are matched.

Section 8 contains an example with a front end structure extending this top-level parser for single sentences by two others: one with a read-eval-loop that shows the parse trees for each sentence, but delays the parse results until EOF is reached, and another one for reading and evaluating a sequence of sentences from a file. A front-end structure providing a read-eval-print-loop that is exited on typing stop can be found in examples/NL/nl.parse.sml.

```
local
    structure Parser = <name>Parser
    structure Tokens = <name>Vals.Tokens
    structure Interface = Interface
in
  structure <name> : sig
                       val parse : unit -> Parser.result list
                       val debug : bool ref
                       val trees : bool ref
                     end
= struct
    val debug = Parser.debug
    val trees = Parser.trees
    val DOT = Tokens.DOT(!Interface.lin,!Interface.col)

    fun invoke lexer =
        Parser.parse(lexer,Interface.error,Interface.default)

    fun loop stopToken lexer =
        let
            val (result,lexer) = invoke lexer
            val ((nextTokens,nextWord),lexer) = Parser.Stream.get lexer
        in
            if Parser.sameToken(List.hd nextTokens,stopToken) then result
            else loop stopToken lexer
        end
    fun parse () =
        let val _ = Interface.init()
            fun reader (_:int) = case (TextIO.inputLine TextIO.stdIn)
                                   of SOME s => s | NONE => ""
        in loop DOT (Parser.makeLexer reader) end
  end
end
```

Figure 4: Front end structure `<name>` using `<name>Parser` to parse from top-level

5.3 Signatures and structures exported by the ML-GLR libraries

The library of ML-GLR, used by all generated user parsers and by the parser generator itself and defined in `lib/ml-glr-lib.cm`, exports the following signatures, functors and structures:

```
signature STREAM
signature TOKEN
signature PARSER_DATA
signature CHART_PARSER
signature LEXER
signature PARSER
signature ARG_LEXER
signature ARG_PARSER
functor Join
functor JoinWithArg
structure Stream
structure ChartParser
```

All of these but `ChartParser:CHART_PARSER` have analogs in the library of ML-Yacc, but except for `Stream:STREAM`, the exported names have different meanings in ML-GLR and ML-Yacc.

The ML-GLR parser generator is defined in `src/ml-glr.cm` and exports the structures

```
structure ParseGen :
  sig
    val parseGen : string -> bool -> unit
  end

structure ExportParseGen :
  sig
    val parseGen : (string * string list) -> OS.Process.status
  end
```

The first is used to apply the parser generator interactively on a file and a debug flag without leaving SML/NJ, while the second wraps an interrupt handler around the first and exits SML/NJ.

ML-GLR uses the LR-parser `yacc.grm.sml` created by ML-Yacc to parse the user's grammar specification `<spec>.grm` into a value of type `Grammar.grammar`, computes its 2LR cover grammar and outputs a chart parser for that to `<spec>.grm.sml`. To avoid name conflicts between the libraries of ML-Yacc and ML-GLR, there is an auxiliary library extracted from ML-Yacc, defined in `src/parseGenParser-util.cm`, which just exports the structure

```
structure ParseGenParser :
  sig
    structure Header : HEADER
    val parse : string -> bool -> Header.parseResult * Header.inputSource
  end;
```

but none of the above signatures or functors. Its `parse` function essentially parses a file `<spec>.grm` into a result of type `Grammar.grammar`. ML-GLR uses this structure `ParseGenParser` as an argument to its own functor `ParseGenFun` that builds the parser generator (see `src/link.sml` and `lib/glr.sml`).

6 Using the parser generator

ML-GLR can be used in different ways to generate a parser: as a tool built into SML/NJ's compilation manager, or interactively from the SML top level, or as a separate program called from a Unix shell.

6.1 Using ML-GLR as a tool of the compilation manager CM

The easiest way to use ML-GLR is by first letting the compilation manager `CM` of SML/NJ[4] know about a new "tool" that automatically calls `ml-glr` from a project's `.cm`-file (SML/NJ's analog to a Unix Makefile). To do so, edit the file `$HOME/.smlnj-pathconfig` by adding lines telling `CM` that there is a tool `MLGLR` and where the makefile for it and its library are to be found in the file system. These lines look like

```
mlglr-tool.cm    <path to install directory of ml-glr>/tool
ml-glr-lib.cm    <path to install directory of ml-glr>/lib
```

Subsequently, a project's `.cm`-file may specify that the tool be applied to a grammar specification file. Figure 5 shows the project description for the grammar in `amb.grm` and lexer `amb.lex`:

The parser with front end structure `Amb` is then generated by typing

```
- CM.make "examples/amb/amb.cm";
```

```
Library
  structure Amb         (* front end structure *)
  structure AmbParser
is
  $/basis.cm
  $/ml-glr-lib.cm

  amb.lex               (* lexer specification *)
  amb.grm:MLGLR         (* grammar specification *)
  ../grm/interface.sml  (* interface functor *)
  amb.interface.sml     (* interface structure *)
  amb.parse.sml         (* front end *)
```

Figure 5: Project description `examples/amb/amb.cm`

The suffix `.lex` causes the ML-LEX tool to apply `ml-lex` to `amb.lex` and produce `amb.lex.sml`. By the definitions in `tool/*`, the line `amb.grm:MLGLR` tells `CM` to call `ml-glr` on the file `amb.grm` and produce `amb.grm.sig`, `amb.grm.sml` and `amb.grm.lnk.sml`. If `:MLGLR` were omitted, `ml-yacc` would be used to produce the first two files, which would fail since it expects a different `LEXER` signature. If `$HOME/.sml-pathconfig` has an additional line

```
glr-ext.cm       <path to the install directory of ml-glr>/tool/
```

one can use `amb.glr` instead of `amb.grm:MLGLR` in the project's `.cm` file to tell `CM` to apply `ml-glr` to the grammar specification in `amb.glr`.

6.2 Using ML-GLR with CM interactively

An alternative way to use `ml-glr` interactively in SML/NJ with CM is to

(i) "make" the parser generator in SML/NJ using

```
- CM.make "scr/ml-glr.cm";
```

(ii) apply the parser generator to the grammar specification by

```
- ParseGen.parseGen "<spec>.grm" false;
```

to create the files `<spec>.grm.sig`, `<spec>.grm.sml` and `<spec>.grm.lnk.sml`,

(iii) "make" the parser using CM and a description file mentioning the files exported in step (ii),

```
- CM.make "<spec>.cm";
```

Note that the files generated in step (ii) will just be compiled, not recomputed.

In the case of `examples/grm/amb/` the modified description file would be as in figure 6:

6.3 Using ML-GLR from the shell

On a Unix shell, apply ML-GLR to the grammar specification by

```
> ml-glr <spec>.grm
```

```
Library
  structure Amb
is
  $/basis.cm
  $/ml-glr-lib.cm
  amb.lex
  amb.grm.sig           (* signatures Amb_TOKENS, Amb_VALS *)
  amb.grm.sml           (* functor AmbValsFun *)
  amb.grm.lnk.sml       (* structures AmbVals, AmbLex, AmbParser *)
  ../grm/interface.sml
  amb.interface.sml
  amb.parse.sml         (* front end structure Amb *)
```

Figure 6: Desription file `amb2.cm` using previously generated files

This generates the files `<spec>.sig`, `<spec>.sml`, and `<spec>.lnk.sml`. Then call SML/NJ on the description file which mentions these files by

```
> sml <spec>.cm <spec>.run.sml
```

where `<spec>.run.sml` contains a call to the parse-function for the `<name>Parser` of the specification. Likewise, one can generate `<spec>.lex.sml` by

```
> ml-lex <spec>.lex
```

and then mention `<spec>.lex.sml` in the description file, to avoid its recompilation.

6.4 Debug mode and inspection of the compact LR(0) automaton

The parser generator can be called with a debug flag and then emits information about its actions and prints the compacted LR(0)-automaton, internal numbering of terminals and nonterminals, the 2LR-cover grammar and the LR-filter on its nonterminals. Most of the information is also contained in the file `<spec>.grm.desc` obtained with the `%verbose` declaration.

To call the parser generator with the debug flag, either do it interactively with

```
- CM.make "src/ml-glr.cm";
- ParseGen.parseGen "<spec>.grm" true;
```

or by using the shell command

```
> ml-glr <spec>.grm -debug
```

Information on the computation of the compacted LR(0)-automaton can be obtained by setting `val DEBUG = true` in `src/mkcover.sml` and making a new version of `ml-glr`.

If `<spec>.grm` contains a `%graphview` declaration (and `<spec>.grm` is touched), a graphical representation of the compact LR(0)-automaton of the grammar is written to `<spec>.grm.graph.dot`, translated to `<spec>.grm.graph.ps` and shown at the screen using the `gv` previewer.[6]

[6] A graphical representation of the ordinary LR(0)-automaton, which can be significantly bigger than the compacted one, can be produced with the slight modification of ML-Yacc found in `ml-yacc-graph/`.

7 Using the generated parser

7.1 Calling the parsing function

Suppose the generated parser `<name>Parser` is encapsulated in a front-end structure `<name>` as in figure 4. One can then call the generated parser interactively by typing

```
- <name>.parse();
```

and then typing an expression to be parsed and evaluated. Likewise,

```
- <name>.parse_file <filename>;
```

lets the parser read input from the named file.

The default print depth may be too low to reveal the result values in detail, if these are, say, expressions of an abstract syntax. The print depth can be raised by

```
- Control.Print.printDepth := 20;
```

at the top level, or from a program when `$smlnj/compiler.cm` is mentioned in the project description file. To view the parse trees found for a given input, set the flag `trees` by typing

```
- <name>.trees := true;
```

Subsequent calls to the parsing function will produce a file `<spec>.grm.sourceTrees.dot` containing a description of the trees in the format of the `graphviz/dot` program[5], and a translation `<spec>.grm.sourceTrees.ps` to PostScript. The ghostview previewer `gv` is then called to present the trees on the screen.

Internal nodes of the trees are shown as `A.n` where `A` is a symbol of the grammar. If `A` is a nonterminal, `n` is the number of the grammar rule used at this node. If `A` is a terminal, `n` refers to the n-th semantic value given to this terminal in the lexer rule which matched the string shown below the node. Isomorphic trees differing only in the attached numbers arise when the grammar or lexicon admit semantic ambiguities.

7.2 Error messages and debugging

The following exceptions may be raised by the user parser:

(i) If no categories are predicted for the remaining input while filling the chart, a message saying so is given and the exception `ParseError` is raised.

(ii) If no source trees can be extracted from the chart, the chartparser reports a syntax error (using its argument `error:string * pos * pos -> unit`) and then raises `ParseError`.

(iii) If, while filling the chart, a non-binary grammar rule or a binary one with terminal and nonterminal on its right hand side is met, this is reported to the top-level and the exception `InternalError` is raised.

(iv) If the evaluation of trees extracted from the chart hits a tree that does not correspond to a source grammar tree, the exception `InternalError` is raised.

Of course, internal errors should never arise. The user is responsible for handling a parse error.

For debugging purposes, one can set the flags `covertrees` and `debug` by

```
- <name>.covertrees := true;
- <name>.debug := true;
```

provided they are members of the front end structure `<name>`. When the flag `covertrees` is set to `true`, the parser produces files `<spec>.grm.coverTrees.dot` and `<spec>.grm.coverTrees.ps` with graphical representations of the input's parse trees with respect to the 2LR-cover grammar. A simple translation of these gives the trees with respect to the source grammar, which are show when the flag `trees` is set.

When the flag `debug` is set to `true`, a call to the parsing function will present information on how the parse chart is filled and how the extracted trees are evaluated. For long input, one may set

```
- <name>.progress := true;
```

to get an estimation on how much of the chart has been filled, provided this flag of `<name>Parser` is made available in the front end.

The cover grammar can be found in the file `<spec>.grm.desc`, which is produced when the grammar specification has a `%verbose` declaration. The file also contains the left-to-right filter relations in a readable format, which may be much larger than the coded version in the `<spec>.grm.sml`.

8 Example

To demonstrate the usage of ML-Lex and ML-GLR with an ambiguous grammar, we use a small fragment of German (see `examples/grm/ger.*`). To get an overview, the description file is presented first.

8.1 Building lexer and parser with the CM-tool `MLGLR`

In CM's description file `ger.cm` in figure 7, we mention the library `ml-glr-lib.cm`, the lexer and grammar specifications `ger.lex` and `ger.grm`, and files defining an abstract syntax, an interface used in the lexer and parser, and a file containing a front end structure `Ger` with the functions by which the generated parser may be called.

```
Library
  structure Ger
  structure GerParser  (* to avoid ?-components in long type names *)
is
  $/basis.cm
  $/ml-glr-lib.cm
  $smlnj/compiler.cm

  ger.absyn.sml        (* abstract syntax *)
  ger.lex              (* lexer specification *)
  ger.grm:MLGLR        (* grammar specification *)

  interface.sml        (* interface functor *)
  ger.interface.sml    (* interface structure *)
  ger.parse.sml        (* front end structure Ger *)
```

Figure 7: Description file `ger.cm`

Since the grammar specification is marked by `:MLGLR`, the compilation manager will use `ml-glr` rather than `ml-yacc` to handle this file, which will produce three files `ger.lex.sml`, `ger.grm.sml` and `ger.grm.lnk.sml` and compile them in combination with the remaining files mentioned.

The interface functor shown in figure 3 can be used by parsers with an additional argument; since we don't want such an additional argument, we specialize the abstract types `arg` and `default` as shown in figure 8.

```
structure Interface = Interface(type arg = unit val default = ())
```

Figure 8: Interface structure `ger.interface.sml`

8.2 Grammar and lexer specification

We want to generate a parser which translates simple German sentences to expressions of an abstract syntax `Absyn:ABSYN` with terms `trm` and formulas `fml`. The structure `Absyn` contains just the datatypes shown in the signature `ABSYN` in figure 9. For simplicity, we don't separate terms into constants and variables.

```
signature ABSYN =
  sig
      datatype trm = George | Mary | Emil | u | v
           and fml = Dog of trm | Cat of trm | Mouse of trm
                   | sleep of trm | eat1 of trm
                   | love of trm * trm | eat of trm * trm
                   | Or of fml * fml | And of fml * fml
                   | Iota of trm * fml * fml
                   | Many of trm * fml * fml
                   | All of trm * fml * fml
                   | Some of trm * fml * fml
  end

structure Absyn : ABSYN = struct ... end
```

Figure 9: Abstract syntax `ger.absyn.sml`

The source grammar specification `ger.grm` is shown in figures 10 and 11. In its user declarations section, structure `Absyn` is opened, so that in the ML-GLR declarations and rules section we can use short identifiers like `Or` instead of the long ones like `Absyn.Or`.

The ML-GLR declarations section provides a name `Ger` which will be used to build structures `GerVals`, `GerLex`, and `GerParser`. We want to do so by closed functors, i.e. all values and types needed will be defined in argument structures of the functors and not taken from the environment. Hence the `%header`-declaration for the functor `GerValsFun` and the `%lexheader`-declaration for `GerLexFun` need an `Absyn:ABSYN` argument. As some of the lexer's types and values come from the structure `Interface` of figure 8, `GerLexFun` also needs an `Interface:INTERFACE` argument. We want our lexer to deliver tokens containing constructors of the abstract syntax, so the signature `Ger_TOKENS` must know about their types. Therefore, using the `%token_info_sig` declaration we advise ML-GLR to include `ABSYN` into `Ger_TOKENS`.[7]

The declarations shown in figure 10 declare terminal categories `DET`, `QUANT`, `N`, `EN`, `TV` and `IV`, suitably refined by suffixes coding gender, number and case, where letters `m`, `f` stand for masculine,

[7] If we do not wish to build `Ger` using closed functors, things simplify as follows: we may drop the argument `Absyn:ABSYN` and the related sharing constraints from the `%header` declarations. Since then `GerValsFun` has no additional argument, we can omit the `%header`-declaration for it. In the `%lexheader`-declaration, the argument `Absyn:ABSYN` is to be omitted; so the `%token_sig_info` is obsolete as well. But then the signature `Ger_TOKENS` mentions the types of `Absyn` that occur in the `%term` declarations; hence one has to use long type names like `Absyn.trm` rather than `trm`, which will then appear in `Ger_TOKENS` as well, making this a signature depending on the structure `Absyn` in the context.

feminine, `s`, `p` stand for singular, plural, and `n`, `a` stand for nominative, accusative. For the nonterminal category `NP`, we don't need a refinement by gender. Note that nonterminals have values of higher-order types.

```
open Absyn
%%
%name Ger
%header (functor GerValsFun(structure Token: TOKEN
                            structure Grammar: GRAMMAR
                              sharing type Grammar.term = Token.term
                            structure Absyn: ABSYN) : Ger_VALS)
%token_sig_info (include ABSYN)
%lexheader (functor GerLexFun(structure Tokens: Ger_TOKENS
                              structure Absyn: ABSYN
                              structure Interface: INTERFACE) : LEXER)
%term
       DETmsn of trm * fml * fml -> fml   | DETmsa of trm * fml * fml -> fml
     | DETfsn of trm * fml * fml -> fml   | DETfsa of trm * fml * fml -> fml
     | DETpn                              | DETpa
     | Nmsn of trm -> fml                 | Nmsa of trm -> fml
     | Nfsn of trm -> fml                 | Nfsa of trm -> fml
     | Npn of trm -> fml                  | Npa of trm -> fml
     | EN of trm
     | TVs of trm * trm -> fml            | TVp of trm * trm -> fml
     | IVs of trm -> fml                  | IVp of trm -> fml
     | QUANTmsn of trm * fml * fml -> fml | QUANTmsa of trm * fml * fml -> fml
     | QUANTfsn of trm * fml * fml -> fml | QUANTfsa of trm * fml * fml -> fml
     | QUANTp of trm * fml * fml -> fml
     | UND | ODER                         | DOT | EOF
%nonterm
       NPsn of (trm -> fml) -> fml        | NPpn of (trm -> fml) -> fml
     | NPsa of (trm -> fml) -> fml        | NPpa of (trm -> fml) -> fml
     | NPa of  (trm -> fml) -> fml        | S of fml

%eop EOF DOT
%pos int
%right ODER
%right UND
```

Figure 10: Grammar specification `ger.grm`, user declarations and ML-GLR declarations

The grammar rules section is shown in figure 11. The first two rules for atomic sentences built with a transitive verb differ only in the semantic annotation, making such sentences semantically ambiguous. The same conjunctions `UND` and `ODER` conjoin both sentences and noun phrases, but are interpreted differently, as can be seen from the semantic annotations in the rules. By the precedence declarations given, `UND` binds tighter than `ODER`, and both are right-associative. The latter implies that `NPpn UND NPpn UND NPpn` has the right-recursive analysis only.

Finally, we need a specification file `ger.lex` for the lexer, see figure 12. In its user declaration section, which is copied to the body of the functor `GerLexFun` that `ml-lex` produces from this file, we introduce abbreviations for the `Tokens` structure to be produced by `ml-yacc` and the `Interface` structure, and define some types and values as demanded by the signature `LEXER`.

Note that in the rules section, each match returns a list of tokens; where these lists are not singletons, the same word has different readings.

The lexer generator `ml-lex` generates from `ger.lex` a file `ger.lex.sml` containing the functor `GerLexFun`, which is parameterized by structures `Tokens`, `Absyn` and `Interface`. The parser

```
%%                                                              (* wide scope: *)
S     : NPsn TVs NPa   (NPsn (fn x => NPa (fn y => TVs(x,y)))) (*   subject   *)
      | NPsn TVs NPa   (NPa (fn y => NPsn (fn x => TVs(x,y)))) (*   object    *)
      | NPa TVs NPsn   (NPsn (fn x => NPa (fn y => TVs(x,y)))) (*   subject   *)
      | NPpn TVp NPa   (NPpn (fn x => NPa (fn y => TVp(x,y)))) (*   subject   *)
      | NPa TVp NPpn   (NPpn (fn x => NPa (fn y => TVp(x,y)))) (*   subject   *)
      | NPsn IVs       (NPsn IVs)
      | NPpn IVp       (NPpn IVp)
      | S UND S        (And(S1,S2))
      | S ODER S       (Or(S1,S2))
NPsn :  EN             (fn P => P (EN))
      | DETmsn Nmsn    (fn P => Iota(v,Nmsn(v),P(v)))
      | DETfsn Nfsn    (fn P => Iota(v,Nfsn(v),P(v)))
      | NPsn ODER NPsn (fn P => Or(NPsn1 P, NPsn2 P))
      | QUANTmsn Nmsn  (fn P => QUANTmsn(v,Nmsn(v),P(v)))
      | QUANTfsn Nfsn  (fn P => QUANTfsn(v,Nfsn v, P(v)))
NPsa : EN              (fn P => P (EN))
      | DETmsa Nmsa    (fn P => Iota(u,Nmsa(u),P(u)))
      | DETfsa Nfsa    (fn P => Iota(u,Nfsa(u),P(u)))
      | NPsa ODER NPsa (fn P => Or(NPsa1 P, NPsa2 P))
      | QUANTmsa Nmsa  (fn P => QUANTmsa(u,Nmsa u, P(u)))
      | QUANTfsa Nfsa  (fn P => QUANTfsa(u,Nfsa u, P(u)))
NPa : NPsa (NPsa) | NPpa (NPpa)
NPpn :  DETpn Npn      (fn P => All(v,Npn v, P(v)))
      | QUANTp Npn     (fn P => QUANTp(v, Npn v, P(v)))
      | NPsn UND NPsn  (fn P => And(NPsn1 P, NPsn2 P))
      | NPpn UND NPsn  (fn P => And(NPpn P, NPsn P))
      | NPsn UND NPpn  (fn P => And(NPsn P, NPpn P))
      | NPpn UND NPpn  (fn P => And(NPpn1 P,NPpn2 P))
      | NPpn ODER NPpn (fn P => Or(NPpn1 P, NPpn2 P))
      | NPpn ODER NPsn (fn P => Or(NPpn P, NPsn P))
      | NPsn ODER NPpn (fn P => Or(NPsn P, NPpn P))
NPpa :  DETpa Npa      (fn P => All(u, Npa u, P(u)))
      | QUANTp Npa     (fn P => QUANTp(u, Npa u, P(u)))
      | NPsa UND NPsa  (fn P => And(NPsa1 P, NPsa2 P))
      | NPpa UND NPsa  (fn P => And(NPpa P, NPsa P))
      | NPsa UND NPpa  (fn P => And(NPsa P, NPpa P))
      | NPpa UND NPpa  (fn P => And(NPpa1 P, NPpa2 P))
      | NPpa ODER NPpa (fn P => Or(NPpa1 P, NPpa2 P))
      | NPpa ODER NPsa (fn P => Or(NPpa P, NPsa P))
      | NPsa ODER NPpa (fn P => Or(NPsa P, NPpa P))
```

Figure 11: Grammar specification `ger.grm`, rules section

generator `ml-glr` generates from the grammar specification `ger.grm` a file `ger.grm.sml` containing the functor `GerValsFun`, and a link-file `ger.grm.lnk.sml` defining the lexer and parser structures, which is shown in figure 13.

8.3 Front end to the parser

Some functions to call the generated `GerParser` are defined in the front end file `ger.parse.sml` in figure 14 and 15. (A functorized version of the front end is given in `examples/grm/ger.parseF.sml`.)

The lexer reads until it finds an `eop` symbol, then the parser builds a parse chart, extracts the trees with start symbol at their root, evaluates them, returns the list `result` of values and exits.

```
structure T = Tokens
structure I = Interface
open I
open Absyn
type pos = I.pos
type svalue = T.svalue
type ('a,'b) token = ('a,'b) T.token
type lexresult = (svalue,pos) token list * string
fun eof() = (T.EOF(!lin,!col)::nil,"")
%%
%header (functor GerLexFun(structure Absyn : ABSYN
                           structure Tokens: Ger_TOKENS
                             sharing type Tokens.trm = Absyn.trm
                             sharing type Tokens.fml = Absyn.fml
                           structure Interface: INTERFACE) : LEXER);
%full
comment=%(.*)\n;
whitespace=[\t\ ]+;
%%
{whitespace}          => (coln(yypos+(size yytext));lex());
\n | {comment}        => (eol(yypos,yytext);line(); lex());
"eine"                => (coln(yypos+4);(T.QUANTfsn(Some,!lin,!col)::
                                         T.QUANTfsa(Some,!lin,!col)::nil,yytext));
"viele"               => (coln(yypos+5);(T.QUANTp(Many,!lin,!col)::nil,yytext));
"Katze"               => (coln(yypos+5);(T.Nfsn(Cat,!lin,!col)::
                                         T.Nfsa(Cat,!lin,!col)::nil,yytext));
"Mäuse"               => (coln(yypos+5);(T.Npn(Mouse,!lin,!col)::
                                         T.Npa(Mouse,!lin,!col)::nil,yytext));
"frißt" | "frisst"    => (coln(yypos+5);(T.TVs(eat,!lin,!col)::
                                         T.IVs(eat1,!lin,!col)::nil,yytext);
                                  ...

"stop."               => (coln(yypos+5);(T.EOF(!lin,!col)::nil,yytext));
"."                   => (coln(yypos+1);(T.DOT(!lin,!col)::nil,yytext);
.            => (error ("ignoring unexpected character "^yytext,!lin,!col);lex());
```

Figure 12: Lexer specification `ger.lex`

In the following sample interactive session, the input sentence has two parse trees (differing in which of the first two `s`-rules is used), and hence returns two values.

```
- CM.make "examples/grm/ger.cm";
...
[scanning $/ml-glr-lib.cm]
[loading examples/grm/(ger.cm):ger.absyn.sml]
...
[loading examples/grm/(ger.cm):ger.parse.sml]
...
[New bindings added.]
val it = true : bool

- Ger.parse();
eine Katze frißt viele Mäuse.
val it =
  [Some (v,Cat v,Many (u,Mouse u,eat (v,u))),
   Many (u,Mouse u,Some (v,Cat v,eat (v,u)))] : GerParser.result list
```

```
(* This file was generated by ML-GLR, using %header and %lexheader in .grm *)

structure GerVals =
   GerValsFun(structure Token = ChartParser.Token
              structure Grammar = ChartParser.Grammar
              structure Absyn = Absyn)

structure GerLex =
   GerLexFun(structure Tokens = GerVals.Tokens
             structure Absyn = Absyn
             structure Interface = Interface)

structure GerParser =
   Join(structure ParserData = GerVals.ParserData
        structure Lex = GerLex
        structure ChartParser = ChartParser)
```

Figure 13: The generated file `ger.grm.lnk.sml`

Numbers n of rules used are attached as $.n$ to nonterminals in the graphical representations of the trees, their precedences p are attached as $\_p$ to the rule number. At terminals, numbers k of values used are attached as $.k$ to the terminal, except for $k = 0$.

Parse trees: rule numbers take semantic annotation into account

Remark 8.1 ML-GLR allows one to use several start symbols and select the parse trees for one of those as follows: for each start symbol `s`, add a dummy terminal `ts` and a rule `start : ts s`, and in order to parse with start symbol `s`, put `([Token.ts(pos,pos)],"dummy")` in front of the token stream. (Parse trees will branch according to the additional rule at the root.)

Remark 8.2 ML-Lex reports the initial position of a file wrongly as 2 instead of 0. The `Interface` functor of figure 3 contains a simple way to correct this in one's own position counting.

One can permanently fix this bug by changing the sources of ML-Lex: where `makeLexer` is defined in `ml-lex/lexgen.sml`, replace `val yygone0=1` by `val yygone0=~1`. Alternatively, when your lexer specification is stable, you can do the same modification in the generated file `<spec>.lex.sml` and then use this instead of `<spec>.lex` in the description file.

Another solution is to use the `%posarg` declaration of ML-Lex. In this case, ML-Lex will produce a `makeLexer` function with input type `(int -> string) * int`, where the integer argument is the position in the input file where reading starts. So one needs to adjust the signature `LEXER` and use `<name>Parser.makeLexer (reader,0)` in the front end of one's parser. However, the

```
structure Ger : sig
                    val parse : unit -> GerParser.result list
                    val parse_seq : unit -> GerParser.result list list
                    val parse_file : string -> GerParser.result list list
                    val debug : bool ref
                    val trees : bool ref
                end =
  struct
    val debug = GerParser.debug
    val trees = GerParser.trees
    val _ = Control.Print.printDepth := 20
    val _ = Control.Print.printLength := 100
    exception GerError

    val EOF = GerVals.Tokens.EOF(!Interface.lin,!Interface.col)
    val DOT = GerVals.Tokens.DOT(!Interface.lin,!Interface.col)

    fun invoke lexer = GerParser.parse(lexer,Interface.error,Interface.default)

    (* Parse a single sentence, show its trees and return its values. *)
    fun parse () =
        let val _ = Interface.init();
            fun reader (_:int) =
                case (TextIO.inputLine TextIO.stdIn) of SOME s => s | NONE => ""
            val _ = (trees := true)
        in loop DOT (GerParser.makeLexer reader) end
    and loop stopToken lexer =
        let val (result,lexer) = invoke lexer
                handle ParseError => raise GerError
            val ((nextTokens,nextWord),lexer) = GerParser.Stream.get lexer
        in
            if GerParser.sameToken(List.hd nextTokens,stopToken)
            then result else loop stopToken lexer
        end handle ParseError => raise GerError
```

Figure 14: Front end structure `Ger` with function to call the generated parser, `ger.parse.sml`

`%posarg` declaration causes `ml-ulex --ml-lex-mode` to raise errors, and one needs to use the original `ml-lex`.

9 Possible Improvements

We mention a number of possible improvements that may be worthwhile.

(i) The 2LR-cover grammar of a grammar G is a binary grammar whose nonterminals are suffixes of right hand sides of rules of G or nonterminals of G paired with states of the compacted LR(0)-automaton of G. We don't know whether this construction really has an advantage over the standard way to turn G into a binary grammar and then use a variant of the CYK-algorithm with top-down filtering. (See Leiß and Klein [7] for a theoretical description.) The top-down-filter, which is just the transitive closure of the left-corner relation, can easily be computed using ML-Yacc's `src/look.sml` and the chain relation from our acyclicity test.

Provided the semantic actions coming with a derivation $A \Rightarrow^+ A$ amount to the identity, this apparently simpler variant would not exclude acyclic grammars. Whether it is more efficient

```
(* Parse a sequence of sentences separated by DOT, accumulate their values and
   return them at EOF, with an empty value list for sentences with parse errors. *)

fun parse_seq () =
    let val _ = Interface.init()
        fun reader (_:int) =
            case (TextIO.inputLine TextIO.stdIn) of SOME s => s | NONE => ""
        val _ = (trees := true)
    in loop_seq ([], GerParser.makeLexer reader) end
and parse_file name =
    let val _ = Interface.init()
        val stream = TextIO.openIn name
        fun reader i = TextIO.inputN(stream,i)
        val _ = (trees := false)
    in loop_seq ([], GerParser.makeLexer reader) before TextIO.closeIn stream end
and skip lexer eop = (* skip the remaining words up to the given eop-symbol *)
    let val ((nextTokens,_),lexer) = GerParser.Stream.get lexer
    in if GerParser.sameToken (hd nextTokens, eop)
       then lexer else skip lexer eop
    end
and loop_seq (acc, lexer) =
    let
        val ((nextTokens',_),_) = GerParser.Stream.get lexer
    in
        if GerParser.sameToken(List.hd nextTokens',EOF) then rev(acc)
        else
            let val (result,lexer) = invoke lexer
                val ((nextTokens,nextWord),lexer) = GerParser.Stream.get lexer
            in
                loop_seq (result::acc, lexer)
            end
            handle ParseError => (TextIO.print "Parse Error; ignoring this sentence.\n";
                                  let val lexer = skip lexer DOT
                                  in loop_seq ([]::acc,lexer) end)
    end
end
```

Figure 15: Functions to call the generated parser, `ger.parse.sml`

than the one of Nederhof/Satta is unclear, both with respect to the parser generation time and the parse time. The same applies when the LR(0) automaton *with lookahead* is used in Nederhof/Satta's[8] construction. It seems unplausible that the additional determinism of the underlying push-down automaton outweights the increase in the size of the cover grammar.

(ii) Nederhof and Satta[8] proposed that an implementation should insert trees into the parse chart and build a shared forest. Our implementation inserts grammar rules with position information into the chart's fields, and finally applies a tree extraction algorithm building separated trees. For highly ambiguous grammars, this takes too much space. The shared forest representation would represent the set of parse trees in polynomial space.

(iii) Since ML-Yacc's error-correction facility is based on its underlying LALR(1)-automaton, it cannot be used for ML-GLR. For ungrammatical inputs, ML-GLR just raises `ParseError` if no categories are predicted or if no tree fitting to the start symbol can be extracted from the chart. Of course, one can inspect the parse chart to find out why the input is rejected.

(iv) Associativity and precedence declarations are checked on the source grammar trees that

are obtained by transforming the cover grammar trees extracted from the parse chart. For highly ambiguous grammars, it would probably be more efficient to check associativity and precedences on the cover grammar trees.

(v) The chart parser expects an ambiguous lexer, whence we have fixed `type lexresult = (svalue,pos) token list * string` in the signature `LEXER`. In some situations, one may want to use a deterministic lexer with `type lexresult = (svalue,pos) token`. It might therefore be better if `lexresult` were abstract in `LEXER` and the user could provide a conversion

```
convert :  lexresult -> (svalue,pos) token list * string
```

to `Parser.makeLexer` in the front end.

References

[1] Standard ML of New Jersey. SML/NJ, Version 110.67, November 2007. `http://www.smlnj.org/dist/working/110.67/`.

[2] Andrew W. Appel and David R. Tarditi. ML-Yacc User's Manual, Version 2.4, April 2000. In: Distribution files of SML of New Jersey Version 110.

[3] Andrew W. Appel and David R. Tarditi. ML-Lex. A lexical analyzer for Standard ML, Version 1.6, October 1994. In: Distribution files of SML of New Jersey Version 110.

[4] Matthias Blume. CM. The SML/NJ Compilation and Library Manager (for SML/NJ version 110/40 and later). User Manual. Lucent Technologies, Bell Labs, May 21 2002. See also `http://www.smlnj.org/doc/CM/`.

[5] Emden Gansner and Stephen North. Graphviz – Graph Visualization Software. See `http://www.graphviz.org`.

[6] Stephen C. Johnson. Yacc: Yet Another Compiler Compiler. AT & T Bell Laboratories. Murray Hill, New Jersey. See also: `http://dinosaur.compilertools.net/yacc/index.html`.

[7] Hans Leiß and Georg Klein. A tabular GLR-Parsergenerator Based on ML-Yacc. Technical report, CIS, Universität München, 2008 (in preparation).

[8] Mark-Jan Nederhof and Giorgio Satta. Efficient Tabular LR Parsing. In *34th Annual Meeting of the ACL*, pages 239–246. Association for Computational Linguistics, 1996.

[9] Roger Price. User's Guide to ML-Lex and ML-Yacc. See `http://rogerprice.org/`.

[10] Masaru Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Press, 1986.

Structure of the source code

The code to analyse the grammar specification is in file `src/glr.sml`. It is mainly ML-Yacc's `src/yacc.sml`, modified to produce somewhat more readable output files `<spec>.grm.sml`, different `.desc`-files, and of course replaces the calls to build the LALR(1)-automation and LR-table by calls to build the compacted LR(0)-automaton and the 2LR-cover-grammar.

- Some signatures had to be moved from `src/sig.sml` to `lib/base.sig`, since we made `Grammar` a core structure, rather than `LrTable` of ML-Yacc.
- The datatype `Grammar.grammar` is used both for source grammars and cover grammars, and extends the corresponding structure of ML-Yacc.
- The `Token` and `ChartParser` structures are based on the structure `Grammar`.

We need a slightly modified ML-Yacc to parse a source grammar. The modified specification `yacc.grm` as well as the files `yacc.sml.sig` and `yacc.grm.sml` generated from it by ML-Yacc are included in `/src/yacc.*`. The file `ml-glr.cm` uses these files rather than the corresponding one in the ML-Yacc distribution.

The reason for the modification is that we want to treat both the user's source grammar and the 2LR-cover grammar generated from it as `Grammar.grammar`, but also need to remember numbers of source grammar rules in the cover grammar; hence, we added a field `rulenumSG` to the values that ML-Yacc generates for rules of a grammar specification.

Another reason for the modification was that we need to check declarations like `%lexheader` that are admitted in grammar specifications for ML-GLR, but not in those for ML-Yacc.

TO DO

(i) Give an example for an `ARG_PARSER`, like `examples/cmpl.arg.cm`? (p.19).

(ii) ML-GLR ought to stop if it detects cycles in the source grammar.

(iii) Explain the chartparser: when finding eop, fill the chart, extract trees (and show them if the flag trees is set), calculate the values.

(iv) For a rule with empty left hand side, `defaultPos` ought to hold the leftmost position of the lookahead terminal which causes the reduction. Is this preserved from ML-Yacc to ML-GLR?

(v) Explain the line/column-counting in `Ger.lex` and `interface.sml`.

(vi) `src/glr.sml`: check if `printLrFilter` is $O(|LR0(G)|)$, no longer $O(|G|^2)$.

(vii) `src/glr.sml`: identify the two `printNtRel`-functions.

Wish list:

(i) Implement the $O(|G| * |w|^3)$-CYK using reachability relations; how to limit the semantics of $A \Rightarrow^+ A$-derivations to the identity?

(ii) Implement grammar symbols with features and rules with evaluation and side conditions

```
np -> det n   (fn P => det n P)     {det.nom = n.nom ...}
```

in the grammar rules. Instead of nonterminals `A`, feature structures `A{label=value,...}` would be stored in the chart, but not mentioned in the rules, except for the side conditions.