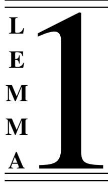


Copyright © : Lemma 1 Ltd 2005

Lemma 1 Ltd.
c/o Interglossa
2nd Floor
31A Chain St.
Reading
Berks
RG1 2HX

ProofPower

—

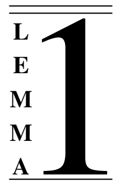
SLRP User Guide

Abstract

This document is the user guide for SLRP — a simple parser generator for Standard ML.

Version: *Revision* : 1.17
Date: 21 September 2005
Reference: LEMMA1/XPP/USR032
Pages: 60

Prepared by: R.D. Arthan
Tel: +44 118 958 4409
E-Mail: rda@lemma-one.com



0.1 Contents

0.1 Contents . . . 2
0.2 List of Figures . . . 4
0.3 References . . . 4
0.4 Changes History . . . 5
0.5 Changes Forecast . . . 5

1 INTRODUCTION 6
1.1 Overview of SLRP . . . 6
1.2 Overview of this Document . . . 7

2 GETTING STARTED WITH SLRP 9

3 WRITING A COMPLETE PARSER 12
3.1 Lexical Analyser . . . 13
3.2 Adding Actions . . . 16

4 HOW THE PARSERS WORK 19
4.1 LR(0) Parsing . . . 19
4.2 Conflict Resolution . . . 20
4.3 LR(0) States . . . 21
4.4 SLRP Parser Driver Implementation Details . . . 22

5 AMBIGUOUS GRAMMARS 24
5.1 Debugging a Grammar . . . 24
5.2 Reduce/Reduce Conflicts . . . 26
5.3 Shift/Reduce Conflicts . . . 28
5.3.1 If-then-else . . . 28
5.3.2 Operator Precedence . . . 29
5.3.3 An Inherently Ambiguous Language . . . 31
5.3.4 Transforming a non-LALR(1) Grammar into an LALR(1) one . . . 32

6 ERROR HANDLING 35

A STANDARD ML CODE EXAMPLES 36
A.1 Compiling SLRP Parsers . . . 36
A.2 A Lexical Analyser . . . 37
A.3 Adding Actions . . . 41
A.4 Operator Precedence Conflict Resolution . . . 44
A.5 Error Handling . . . 45

B COMMAND LINE INTERFACE 47
B.1 Command Line Syntax . . . 47
B.2 Input and Output File Conventions . . . 47
B.3 Code Generation Options . . . 47
B.4 Listing Options . . . 48

C SLRP INPUT FORMAT **49**

D STANDARD ML LIBRARY **50**

E PARSER DRIVER API **51**

F GENERIC PARSER API **57**

G API INDEX **60**

0.2 List of Figures

1 A Grammar for Arithmetic Expressions . . . 9
2 Generic Parser Input . . . 11
3 The Parse Tree for the Input in Figure 1 . . . 11
4 An Action Grammar for Arithmetic Expressions . . . 16
5 A Grammar and the Graph of its LR(0) Automaton . . . 19
6 LR(0) State Table and Conflict Listing . . . 22
7 Fragments of an Incorrect Grammar for C . . . 25
8 A Grammar With Reduce/Reduce Conflicts . . . 26
9 Reduce/Reduce Conflicts — Extracts from the Listing . . . 27
10 A Grammar Requiring Operator Precedence Rules . . . 29
11 Operator Precedence — Extracts from the Listing . . . 30
12 An Inherently Ambiguous Language . . . 31
13 Resolving Ambiguities By Widening the Language . . . 31
14 A non-LALR(1) Grammar . . . 33
15 Making a Grammar LALR(1) . . . 34
16 Compiling the SLRP Library Code . . . 36
17 Compiling the Code Generated by SLRP . . . 36
18 Constructing Lexical Values . . . 37
19 Recognising Punctuation Symbols . . . 38
20 Recognising an Identifier . . . 38
21 Recognising a Literal . . . 39
22 Dealing with Lexical Errors . . . 39
23 Constructing the Reader . . . 40
24 Constructing the Parser . . . 40
25 The Reduction Functions for Commands . . . 41
26 The Reduction Functions for Expressions I . . . 41
27 The Reduction Functions for Expressions II . . . 42
28 The Reduction Functions for Expressions III . . . 42
29 Constructing the Parser . . . 43
30 The Operator Precedence Parser . . . 44
31 A Simple Extension to the Default Error Routine . . . 45
32 A Simple Error Recovery Scheme . . . 46

0.3 References

[1] BS6154:1981. *Method of defining syntactic metalanguage.* British Standards Institution, 1981.

[2] DS/FMU/IED/DTD018. *Detailed Design for SLR Parser Driver.* R.D. Arthan, Lemma 1 Ltd., `http://www.lemma-one.com`.

[3] DS/FMU/IED/IMP018. *Implementation of SLR Parser Driver.* R.D. Arthan, Lemma 1 Ltd., `http://www.lemma-one.com`.

[4] DS/FMU/IED/PLN009. *BS6154:1981 Method of defining syntactic metalanguage.* British Standards Institution, 1981.

[5] LEMMA1/DEV/WRK063. *Demonstration SLRP Parser for ANSI-C.* R.D. Arthan, Lemma 1 Ltd., `rda@lemma-one.com`.

[6] LEMMA1/DEV/WRK064. *SLRP Grammars for Ada 95, Java 1.1 and Pascal.* R.D. Arthan, Lemma 1 Ltd., `rda@lemma-one.com`.

[7] LEMMA1/HOL/USR029. *ProofPower HOL Reference Manual.* Lemma 1 Ltd., `rda@lemma-one.com`.

0.4 Changes History

Issues 1.1 – 1.15 Author's initial drafts.

Issue 1.16 First complete version.

0.5 Changes Forecast

None at this release.

1 INTRODUCTION

1.1 Overview of SLRP

SLRP is a simple but powerful parser generator for Standard ML. The input to SLRP is a grammar written in a version of BNF, [1]. The output from SLRP is a file of Standard ML code that can be used to construct a parser for the language specified by the grammar. Grammar rules may include semantic actions: Standard ML expressions to be evaluated when a rule is applied during parsing. SLRP can also output a listing describing its analysis of a grammar to assist you in designing the language or the grammar that defines it.

SLRP is similar in conception to well-known parser generators like `yacc` and `bison` for C and ML-Yacc for Standard ML. SLRP differs from these in that the parser it generates is table-driven rather than comprising a mixture of tables and decision-making code. The tables are interpreted by a simple polymorphic parser-driver function that takes as its arguments functions that you supply to carry out application-specific tasks such as reading the input stream and reporting errors. The parser driver function is supplied as part of a simple API that provides support for coding these functions. Another difference between SLRP and `yacc` or `bison` is that it supports dynamic rather than static resolution of parsing conflicts. This makes it easy to implement language features such as user-defined operator precedences.

SLRP does not include an automatic lexical analyser generator. It does support production of a generic parser from a grammar. This means that SLRP can automatically generate the semantic actions for you. The result is a complete working parser whose input is a sequence of terminal symbols in the same format as was used in the grammar and whose output is a parse tree. This makes it easy to experiment with the design of the language or the grammar without committing a lot of effort to writing code. The generic parser is supplied as part of an API that you can use or adapt to provide the parsing functionality that your application requires.

SLRP is currently available to run with the Poly/ML or Standard ML of New Jersey compilers. The output from SLRP is Standard ML code that should work with any Standard ML compiler. SLRP includes the ProofPower library of utility functions and the code it generates does depend on these. However, substitutes for the small repertoire of functions actually used (primarily concerned with table lookup) can easily be developed to work in other environments.

Like `yacc` and `bison`, SLRP implements a version of the LALR(1) technique. This means that it is capable of generating code to parse a wide range of practical languages. It is used for all of the main object languages supported by ProofPower: HOL, Z and the Compliance Notation (which includes most of the Ada '83 language). The SLRP distribution includes example grammars for the programming languages Ada 95, Java 1.1 and Pascal and a fairly complete worked example of a parser for C. SLRP is also used for the parsers for the object languages HOL and Z supported by ProofPower.

The SLRP source code and documentation is packaged using PPTex, the ProofPower document preparation and literate programming system. You do not need to use PPTex to use SLRP, but it does provide a convenient way of packaging the various source files and scripts you need to build a parser with SLRP.

1.2 Overview of this Document

This document is intended to be a self-contained introduction to using SLRP. It assumes some familiarity with the Standard ML programming language and with the basic concepts of using a context-free grammar written in BNF to specify a language.

The document is structured as follows:

Section 2 shows you how to get started by preparing a grammar in the form supported by SLRP and to use it to implement a first cut at a generic parser, in a few lines of ML, that reads strings of grammar symbols and outputs parse trees.

Section 3 shows how you can use the API supplied with the generic parser support code to implement a lexical analyser and how to add semantic actions to the grammar and so implement a complete working parser for a simple calculator program.

Section 4 gives a brief introduction to the theory behind the LALR(1) parsing technique and explains the concept of shift/reduce and reduce/reduce conflicts that tend to arise when working with more complex languages and grammars.

Section 5 discusses some of the common problems that arise, such as errors in the grammar; it gives some useful techniques for working with ambiguous grammars, e.g., for languages which are best expressed using numeric operator precedence.

Section 6 shows you how to develop a parser with more sophisticated error handling and error recovery.

Appendix A give the full Standard ML code for the examples in the earlier sections.

Appendix B describes the SLRP command line interface.

Appendix C gives the BNF grammar for the SLRP input format.

Appendix D describes the library of Standard ML utility types and functions that is supplied with SLRP.

Appendix E gives detailed reference documentation for the parser driver API.

Appendix F gives detailed reference documentation for the generic parser API.

The source of this document is itself a literate script containing example grammars and example code that can be automatically processed and compiled. Other examples may be found in the documents [5] and [6] supplied with SLRP.

2 GETTING STARTED WITH SLRP

The first task in using SLRP to build a parser is to prepare a text file containing a grammar for the language you want to parse. The file should be given a name ending in "`.txt`". The format of this file is specified in detail in appendix C. Figure 1 shows an example file containing a grammar for a language of arithmetic expressions:

```
Text dumped to file usr032a.grm.txt
Expression =     Sum;

Atom =           literal
            |    identifier
            |    `(`, Expression, `)`;

Application =    Atom
            |    `-`, Application
            |    `+`, Application
            |    identifier, `(`, Expression, `)`;

Product =        Application
            |    Product, `*`, Application
            |    Product, `/`, Application;

Sum =            Product
            |    Sum, `+`, Product
            |    Sum, `-`, Product;
```

Figure 1: A Grammar for Arithmetic Expressions

In this document, we assume that you are familiar with the basic concepts of what are generally called *context-free* or *BNF* grammars, or, in this document, just *grammars*. This example shows the SLRP conventions for writing a grammar: the grammar is written as a list of equations, which we call *productions*; the right-hand side of each production comprises zero or more *alternatives* separated by vertical bars and terminated by a semi-colon; each alternative comprises zero or more *grammar symbols* separated by commas. A *nonterminal symbol* is a grammar symbol that appears on the left-hand side of some production. A *terminal symbol* is a grammar symbol that does not appear on the left-hand side of any production. A *grammar rule* is an equation such as *Sum* = *Product* whose left and right-hand sides are both drawn from some production in the grammar.

Nonterminal symbols are written as identifiers, e.g., *Sum*, following the Standard ML rules for forming identifiers (using letters, numbers, underscores and the prime character and beginning with a letter). Terminal symbols may be written either as identifiers, e.g., *literal*, or as arbitrary strings enclosed in back-quote characters, e.g., ‘+‘.

In strings, you may use a backslash character as an escape character, if you need to include a backslash, a new-line or a back-quote character in the string.

SLRP adopts the convention that the first production gives the starting symbol for the grammar, in this case *Expression*. We call this symbol the *sentence symbol* and we use the term *sentences* to refer to the programs, or algebraic expressions, or specification language paragraphs, or whatever syntactic objects it may be that the sentence symbol represents.

Once you have prepared your grammar file, you can run SLRP to analyse the grammar and generate the parser code. You use the shell script `slrp` to do this, specifying the name of your file using the *-f* option. The file name must end in "`.txt`". In this case, the name is `usr032a.grm.txt`. If you want to generate the generic parser, you specify the option *-g*. For our example, the command would be as follows:

Bourne Shell

```
slrp -g -f  usr032a.grm.txt >usr032a.grm.run 2>&1
```

This will result in some messages on standard output (redirected to the file `usr032a.grm.run` in the above example call on `slrp`). It will also produce two output files: `usr032a.grm.sml` which contains the Standard ML code of the generated parser and `usr032a.grm.log` which contains a listing of the grammar and various additional information; for example, it includes a sorted list of the terminal symbols, which is useful for checking for mistyped nonterminal names.

You can now compile and run the generic parser for your language after first compiling the parser driver and supporting material using the ML commands shown in figures 16 and 17. The example code in figure 17 defines a function *ae_ parser1* that takes the name of an input file as its argument; it parses the sequence of terminal symbols in the input file and, if the input is correct, prints out a textual representation of the parse tree. An example input file, `usr032a.grm.tst`, for it is shown in figure 2

If you now execute `ae_parser1 "usr032a.grm.tst"`, you will see the print-out of the parse tree shown in figure 3. The parse tree is printed out as a left-to-right, bottom-up listing of the grammar rules used to parse the input; each application of a rule is given a label showing its position in the tree. Each terminal symbol in the print-out of the tree is followed by the line number on which the terminal symbol in question was encountered. The print-out can be understood as a top-down description of how the input has been parsed by reading it backwards: "*1: an Expression can be a Sum; 1.1: a Sum can be a Sum followed by a plus sign (the one on line 2) followed by a Product; 1.1.3: a Product can be an Application ...*".

Of course the functionality of the generic parser as it stands is unlikely to be what you want in your actual application. Both its "front-end", i.e., its lexical analyser, and its "back-end", i.e., the generic reduction actions that build a generic parse tree, will generally need to be modified to do what is really wanted. Section 3 explains how you can go about mutating the generic parser to meet your needs.

Text dumped to file usr032a.grm.tst

```
literal '*' '(' literal '-' literal
')' '+' identifier '('
        literal '/'
        literal
')'
```

Figure 2: Generic Parser Input

```
1.1.1.1.1.1.1: Atom = literal(1);
1.1.1.1.1.1: Application = Atom;
1.1.1.1.1: Product = Application;
1.1.1.1.3.1.2.1.1.1.1.1: Atom = literal(1);
1.1.1.1.3.1.2.1.1.1.1: Application = Atom;
1.1.1.1.3.1.2.1.1.1: Product = Application;
1.1.1.1.3.1.2.1.1: Sum = Product;
1.1.1.1.3.1.2.1.3.1.1: Atom = literal(1);
1.1.1.1.3.1.2.1.3.1: Application = Atom;
1.1.1.1.3.1.2.1.3: Product = Application;
1.1.1.1.3.1.2.1: Sum = Sum, '-'(1), Product;
1.1.1.1.3.1.2: Expression = Sum;
1.1.1.1.3.1: Atom = '('(1), Expression, ')'(2);
1.1.1.1.3: Application = Atom;
1.1.1.1: Product = Product, '*'(1), Application;
1.1.1: Sum = Product;
1.1.3.1.3.1.1.1.1.1: Atom = literal(3);
1.1.3.1.3.1.1.1.1: Application = Atom;
1.1.3.1.3.1.1.1: Product = Application;
1.1.3.1.3.1.1.3.1: Atom = literal(4);
1.1.3.1.3.1.1.3: Application = Atom;
1.1.3.1.3.1.1: Product = Product, '/'(3), Application;
1.1.3.1.3.1: Sum = Product;
1.1.3.1.3: Expression = Sum;
1.1.3.1: Application = identifier(2), '('(2), Expression, ')'(5);
1.1.3: Product = Application;
1.1: Sum = Sum, '+'(2), Product;
1: Expression = Sum;
```

Figure 3: The Parse Tree for the Input in Figure 1

3 WRITING A COMPLETE PARSER

The code generated by SLRP defines a polymorphic function called *slrp'gen_parser*. This function takes four parameters which identify functions as follows:

RESOLVER: a function to resolve what SLRP sees as possible ambiguities in the grammar, referred to as *conflicts*, (see sections 4 and 5 below for more information on this). If the grammar has no conflicts, the function *default_resolver* provided as part of the SLRP library will serve for this parameter.

CLASSIFIER: a function to map tokens read from an input stream, e.g., "TEMP_VAR" or "+" to the terminal symbol in your grammar that represents the lexical class of the token, e.g., *Identifier* or '+'. A function *classifier* is provided as part of the generic SLRP parser which will serve for this parameter unless you want to customise the data types used to represent the lexical classes.

ERROR ROUTINE: a function to deal with syntax errors. A function *default_error* is provided as part of the SLRP library to serve as an off-the-shelf value for this parameter. See section 6 for more information on error handling.

READER: a function to provide input to the parser; this is interface by which you supply a lexical analyser suiting the rules of the language you are parsing.

The parser function is also implicitly parametrised by a table generated by SLRP which maps grammar rules to ML functions called *reduction actions* or just *actions* for short. These actions are executed during parsing and give an operational semantics to the grammar.

To understand how this works, you can think of the parser identifying the sequence of tokens that make up its input by a series of *reductions* that transform sequences of grammar symbols. The parser tries to match parts of the input with the right-hand side of a grammar rule. When a match is found the parser reduces the matching portion of the input to the name of the nonterminal symbol on the left-hand side of the matching rule. It keeps trying these reduction steps until it has reached a sequence comprising just the sentence symbol. Each time a rule is used in a reduction the corresponding action is evaluated and the result or side-effect of ths gives a semantics to the rule.

The reductions are analogous to a forwards reading of the parse tree in figure 3: "*1.1.1.1.1.1.1: The literal (on line 1) can be reduced to an Atom; 1.1.1.1.1.1: an Atom can be reduced to an Application ...*". The evaluation of the action functions can be viewed as as a similar reading together with a description of a calculation of a value: "*... an Atom [with value 99] can be reduced to an Application [with value 99] ...*".

For the generic parser we constructed in section 2, the action code is automatically generated by SLRP and the semantics is calculation of a parse tree. SLRP lets you write Standard ML

expressions alongside the alternatives in a grammar to serve as the action code; when you do this, your expressions are evaluated as the reduction actions, giving each alternative the semantics implied by your action code.

The purpose of this section is to show by example how to write a lexical analyser to serve as the READER function and how to include application-specific actions in a grammar. Our example will be based on the language of arithmetic expressions of figure 1; the result will be a calculator program that reads, parses and evaluates the sentences of the language. Section 3.1 deals with lexical analysis and section 3.2 deals with implementing the actions.

3.1 Lexical Analyser

The structure *GenericSlrpParser* contains the generic parser API which provides a framework to help you construct a lexical analyser. Reference documentation for the generic parser API is given in appendix F of this document.

In this section we show how you can use this framework to implement a lexical analyser for the language of arithmetic expressions of figure 1. The framework helps you with things such as reading an input stream and keeping track of line numbers, letting you concentrate on the lexical matters that are specific to your language.

The framework is provided both as an API and as source code. The source code is only a few hundred lines of ML. If you are writing a parser for use in a larger application you may well want, eventually, to take a copy of the source, strip out the parts you do not use, and customise what remains to the requirements of your application. In this section, we will concentrate on using the API, since that is the first step in understanding of the framework.

In the lexical analysis framework, a polymorphic type *'lc LEX_VALUE* is used for communication between the lexical analyser and the parser. It is defined as follows:

```
type 'lc LEX_VALUE = 'lc * (string * int);
```

Here *'lc* stands for the ML representation of the various terminal symbols (lexical classes) in the grammar (together with a special end-of-sentence symbol). In the generic parser we constructed in section 2, *'lc* is instantiated to a data type *LEX_CLASS* representing the identifiers and string quotations that make up the set of terminal symbols in the SLRP input format. For our language of arithmetic expressions, we will continue to use this data type for the lexical classes. It is defined as follows:

```
datatype LEX_CLASS =
            LCIdentifier of string
      |     LCString of string
      |     LCEos;
```

The listing file `usr032a.grm.log` generated by SLRP shows us that we have the following lexical classes to deal with: *identifier*, *literal*, ‘(‘, ‘)‘, ‘ * ‘, ‘ + ‘, ‘ − ‘, ‘/‘. Here are the lexical rules chosen for the example language: identifiers are formed from letters, numbers and underscores and start with a letter or an underscore; literals are strings of decimal digits; the punctuation symbols are as given above together with semicolon, the hash symbol and the equals symbol (for later use) and the parser must warn about and skip over other characters. The input stream comprises a sequence of arithmetic expressions separated by semicolons. Comments comprise any sequence of characters beginning with a hash character and terminated by an end-of-line character. The ML definitions given in figure 18 reflect some of these design decisions.

To construct a lexical analyser using the framework, you write what we call recogniser functions for the various lexical classes. These are defined in terms of the following data types, a recogniser function being a function from *'lc LEX_STATE* to itself.

```
datatype CONTINUATION_STATUS =
            InComment
      |     InString of string list;
datatype 'lc LEX_STATUS =
            Unknown
      |     Known of 'lc LEX_VALUE
      |     Comment
      |     Continuation of int * CONTINUATION_STATUS;
type 'lc LEX_STATE = (string list * 'lc LEX_STATUS);
```

The idea is that a recogniser function is passed as the first component of its parameter a buffer comprising a list of strings representing the unread input characters (one character per string) from the current input line. If it recognises what it finds, the function should consume the appropriate number of characters from the buffer and return the remaining characters in the buffer together with a status value indicating what has been found. The generic tools look after issues such as skipping white space between tokens and ignoring tokens that have been recognised as comments.

The data types above allow for lexical values, e.g., comments or strings in some languages, that can be continued over multiple lines. We do not need to use these features in our example.

Figures 19 to 22 show the code for recognising the lexical classes we are interested in. Because we want to accept a sequence of arithmetic expressions separated by semicolons, the recogniser for punctuation maps semicolon to the end of sentence symbol; it also deals with comments, and, for use when we extend the language a little in section 3.2, with equals signs. The recognisers for identifiers and numeric literals are straightforwards representations in ML of finite state machines implementing our lexical rules. Finally the recogniser for unknown symbols issues an error message and classifies the erroneous symbol as a comment, so that the lexical analyser will skip over it.

In figure 23, we use our recogniser functions and the lexical analysis framework API to con-

struct our READER and then in figure 24 we construct the parser by supplying the appropriate parameters to *slrp'gen_parser* to give a parsing function that, given a string containing the sentences to be parsed, returns a function which when called will return the results of parsing the successive sentences in the text.

In figure 23, the first step is to use the function *rec_first* from the API to combine our individual recognisers. It works by first of all skipping over space characters and then when a non-space character is found it calls the individual recognisers in turn until it finds one that signals success (i.e., that does not return status *Unknown*).

The READER parameter to *slrp'gen_parser* is defined to have type an instance of the polymorphic type $'st \; -> \; 'tok \; * \; 'st$. Here $'st$ is the type of some internal state of the reader and $'tok$ is the type of the lexical token returned by a call on the READER. In the lexical analysis framework, $'st$ is instantiated to $string \; list \; * \; bool$. I.e., the state comprises a buffer of single-character strings representing the as yet unconsumed contents of an input line together with a flag indicating whether the input source for the next line is available.

The function *gen_reader* in the API will automatically produce a READER given parameters identifying the lexical class representing end-of-input, a recogniser function and an input source. The input source is represented by a type *IN_CHAR_STREAM* defined in the API, which also provides functions to construct such input streams given a source of input text such as an ML string or a file name. This provides all that is needed to construct a READER for a parser that is expecting to find exactly one sentence in the input source. Our example is a little more complicated, because we want to read a sequence of sentences separated by semicolons. Our approach in figure 23 is to wrap round a call of the READER function produced by *gen_reader* some code that records the state of the READER in an assignable variable.

Given the READER function constructed in figure 23, figure 24 shows the construction of the finished parser. Note that the type *IN_CHAR_STREAM* is a record type including a component called *close* which we should call to free up resources, such as open files, associated with the input stream. We do not need to close a stream explicitly if we read to the end of the stream, but we do need to close it explicitly if the parsing is abandoned before the end of stream is read.

To test the parser, we can execute the following code which will print out the parse trees for the two expressions "$1 + 2$" and "$3 * 4$".

```
let   val p = ae_parser2 "1 + 2; 3 * 4";
in    print_tree(p());
      print_tree(p())
end;
```

3.2 Adding Actions

In this section, we will finish off our example by adding actions to the grammar. We also extend the language a little to introduce commands that bind numbers to names. The actions will compute the value of expressions, so that our parser becomes a simple interactive calculator with a memory of named numbers.

The grammar for the extended language is given in figure 4. The grammar has been extended in two ways: first, the sentence symbol is now *Command*, which is either an expression as before, or a definition of a named number, comprising an identifier and an expression giving its value; second, we have added *actions* giving semantics to the grammar rules as discussed on page 12. The actions are given by supplying a Standard ML expression enclosed in brackets at the end of each alternative.

```
Text dumped to file usr032b.grm.txt
Command =        Expression                       (red_command1 x1)
              |  identifier, `=`, Expression      (red_command2 x1 x3);

Expression = Sum                                  (red_expression x1);

Atom =           literal                          (red_atom1 x1)
              |  identifier                       (red_atom2 x1)
              |  `(`, Expression, `)`             (red_atom3 x2);

Application =    Atom                             (red_application1 x1)
              |  `-`, Application                 (red_application2 x2)
              |  `+`, Application                 (red_application3 x2)
              |  identifier, `(`, Expression,     `)`
                                                  (red_application4 x1 x3);

Product =        Application                      (red_product1 x1)
              |  Product, `*`, Application        (red_product2 x1 x3)
              |  Product, `/`, Application        (red_product3 x1 x3);

Sum =            Product                          (red_sum1 x1)
              |  Sum, `+`, Product                (red_sum2 x1 x3)
              |  Sum, `-`, Product                (red_sum3 x1 x3);
```

Figure 4: An Action Grammar for Arithmetic Expressions

Whenever the parser reduces an instance of an alternative in an action grammar, it evaluates the action as an ML expression. The result of evaluating an action is called the *semantic value* of the instance of the nonterminal symbol on the left-hand side of the grammar rule. The action is evaluated inside an ML `fn`-expression whose pattern binds the variables, *x1*, *x2*,

x3, ... to representations of the semantic values of the 1st, 2nd, 3rd, ... grammar symbols in the alternative (reading it left-to-right). Here, the semantic value of a terminal symbol is a representation of the lexical value of the input token.

The semantic values are represented in ML as instances of the following polymorphic type:

```
datatype ('tok, 'lc, 'pp) INPUT_STACK_ITEM =
            Token of 'tok * 'lc
    |       Parsed of 'pp;
```

Here *'pp* (the name stands for "partially-parsed") is the type of the semantic value of a nonterminal symbol and *'tok* and *'lc* are the types for the lexical value of a token and its lexical class. For our calculator, *'pp* will be integers representing the results of evaluating arithmetic expressions; *'tok* and *'lc* will be the types *AE_LEX_VALUE* (defined in figure 18) and *LEX_CLASS* (discussed in section 3.1) respectively. In fact, if you're using the lexical analysis framework, all you usually need to know about the operand of the constructor *Token* is that it has the form $((_,(s,_)),_)$ where s is the text of the input token.

In our action grammar, we have adopted the convention of having a separate ML function for each alternative in the grammar. We call these the *reduction* functions. Each reduction function is passed as parameters the xN that correspond to nonterminals, literals or identifiers in the alternative. So, for example, the function *red_sum2* is passed the semantic values of the two operands of the addition.

Figures 25 to 28 give the reduction functions which give the calculator semantics to our language. The reduction functions do pattern-matching on their parameters to pick out the semantic values of the symbols on the right-hand side of the corresponding grammar rule: a parameter corresponding to a nonterminal has a pattern constructed with *Parsed* and a parameter corresponding to a terminal symbol has a pattern constructed with *Token*. Several of the reduction functions are just defined to be equal to an auxiliary function *red_accept* which simply returns the semantic value of its parsed parameter (this is often useful for rules where the right-hand side comprises a single nonterminal).

For example, *red_application4* in figure 26 corresponds to the rule:

> *Application = identifier, '(', Expression, ')'*

Its first parameter is expected to be a token and its second a parsed value. It implements a semantics supporting two named functions "abs" and "sgn" which compute absolute value and the sign function. It checks to see that the name is one of these two and if so returns the value of the specified function applied to the number appearing in the second parameter. If the name is not one of the two supported functions it outputs an error message and raises an exception.

The reduction functions for commands shown in figure 25 also have side effects: they both cause something to be printed on standard output and update a table of named numbers (used

in *red_atom2* to look up the value of an expression comprising an identifier — *red_command1* adopts the ML-like convention of binding a top-level expression to the variable *it*).

In general, care should be taken when actions have side-effects, since the order in which the actions are evaluated can be tricky to understand. In this case, all we need to know is that the calling order corresponds to some order for constructing a parse tree from the bottom up. Since the production for the sentence symbol, *Command*, is not recursive, the actions in its alternatives will be evaluated once only and after evaluating any other actions for each sentence parsed.

Finally, figure 29 shows the construction of the finished parser with semantic actions. It reuses the lexical analyser code described in section 3.1. So that it can be tested interactively, we formulate the parser to take its input from an *instream*, such as *std_in*.

If you compile the code in the figures in sections refCompiling to A.3 and run the following ML command, you will begin an interactive session with our simple calculator program.

ae_parser3 std_in;

If you put a semicolon-separated list of commands in a file, say `usr032b.grm.tst`, then the following command will execute the commands.

ae_parser3 (*open_in* "*usr032b.grm.tst*");

Note that the calculator will not allow the file to end with a semicolon. You may find it an instructive exercise to try redesigning the calculator to treat semicolons as terminators rather than separators — there are many ways of going about it.

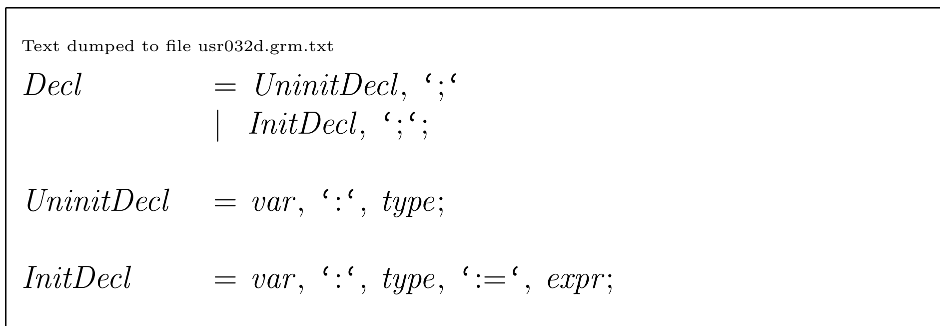
4 HOW THE PARSERS WORK

Many useful grammars give rise to problems when you first enter them into a tool like SLRP; moreover, some applications have requirements, e.g., for error-recovery, that are not covered by the simple techniques we have been using so far. To help you understand and solve the problems that can arise, this section gives a more detailed description of how SLRP and the parsers it generates work.

4.1 LR(0) Parsing

For any grammar, parsing a sentence according using the grammar corresponds to finding a possible computation of a machine called the LR(0) automaton for the grammar. The behaviour of this machine is governed by a certain rooted directed graph each of whose edges is labelled either with a grammar symbol or with a special symbol $ meaning end-of-input.

An example grammar and the corresponding graph are given in figure 5. The root of the graph is the node drawn in black. The nodes of the graph are referred to as *states*. There is a distinguished state "accept", which is labelled −1 in the figure.



Text dumped to file usr032d.grm.txt

Decl = *UninitDecl*, ‘;‘
| *InitDecl*, ‘;‘;

UninitDecl = *var*, ‘:‘, *type*;

InitDecl = *var*, ‘:‘, *type*, ‘:=‘, *expr*;

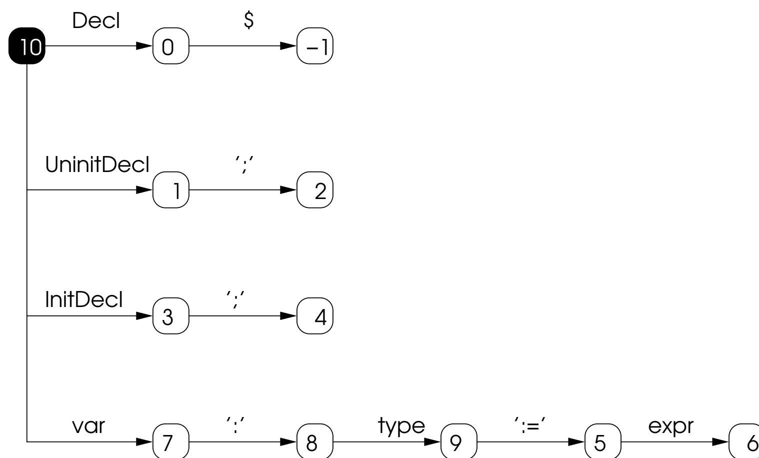


Figure 5: A Grammar and the Graph of its LR(0) Automaton

The LR(0) automaton is a machine whose input comprises the graph and a string of terminal symbols. The states of the automaton are the nodes of the graph, the initial state being the root of the graph. The automaton "consumes" its input string, symbol by symbol working from left to right, but it need not consume a new symbol on every transition. At each stage of its computation it carries out one of the following types of transition:

Accept: if the current state s is "accept", then the input string is a sentence of the language, the sequence of reductions which have been carried out determine a parse tree for it, and the automaton stops, accepting the input string;

Shift: if there is an edge labelled with the current input symbol and leading from the current state s to some state s', then the automaton can consume the input symbol and move to state s';

Reduce: if there is a state, s', and a grammar rule, $X = \alpha$, such that:

1. the path in the automaton that led to the current state s passes through s'
2. the edge labels on the path from s' to s spell out the sequence of symbols α
3. there is an edge labelled with X leading out of s' to some state s''

then the automaton can move to state s'' (without consuming an input symbol);

Error: if none of the above types of transition is possible, the input string is not a sentence of the language and the automaton stops, rejecting the input string.

For example, consider the input "*var, ':', type, ':=', expr*" for the example in figure 5. From the start state 10, the first three input symbols force the automaton to shift three times and enter state 9. In this state it has a choice: either it can reduce to state 1 (taking s' to be the start state 10, and using the rule for *UninitDecl*), or it can shift into state 5. Such a choice is referred to as a *conflict* and means that the LR(0) automaton does not give a deterministic parser for the grammar.

4.2 Conflict Resolution

An LR(1) automaton is an LR(0) automaton equipped with a table resolving any conflicts by telling it what transition to make in each state for each possible input symbol. This table is called the *lookahead* table. If a suitable lookahead table exists for the grammar then the grammar is unambiguous and the LR(1) automaton will provide a deterministic parser for it. At each stage of a parse, the action of the parser is driven by by the next unconsumed input symbol, which we refer to as the *lookahead token.*

There are several ways of attempting to calculate lookahead tables. The method used in SLRP is first to apply a simple method called SLR(1) to compute the LR(0) graph and to

calculate a first approximation to the lookahead table. This may not resolve all the conflicts, in which case SLRP applies a rather more sophisticated method called LALR(1) in an attempt to resolve the remaining conflicts.

Even the LALR(1) method will sometimes fail to resolve all conflicts. There can still be states and input symbols for which the lookahead tables offer the automaton more than one choice. It turns out that there will never be a choice between two different ways to shift: the possibilities for a given state and input symbol may include at most one shift transition and any number of reduce transitions.

SLRP implements a traditional heuristic to resolve the situations where there is a choice between several reduce transitions (reduce/reduce conflicts): the reduction corresponding to the nonterminal that appeared first in the grammar is taken as the resolution and the rest are removed. If several reductions for the same nonterminal are possible, then the alternative that appeared first is taken as the resolution.

After this heuristic has been applied, there will be at most one conflict for each state and input symbol and that will be a shift/reduce conflict. Section 5 below discusses how you can check that the built-in static resolution of reduce/reduce conflicts is appropriate for your application and how you can work with grammars that have shift/reduce conflicts by supplying ML code to resolve conflicts dynamically.

4.3 LR(0) States

If your grammar gives rise to conflicts you will often need to study the listings it produces to understand what is causing the problem. The listing of the state table can be very useful for this purpose. The state table comprises a list of sets of *items*, an *item* being a grammar rule together with a marker identifying a position the parser could reach in parsing an instance of that rule.

The LR(0) state table is constructed by an algorithm which groups the items into sets. The resulting sets of items are assigned numbers which represent the states in the implementation of the LR(1) automaton. Note that the algorithm does not generally produce an equivalence relation on items, i.e., an item may appear in several different states.

Figure 6 shows some extracts from the listing of a state table for a grammar for the C programming language. Items are shown in the listing as grammar rules with a dot to indicate the position marker. For example, the two items in state 207 show the position in a parse when the then-part of an if statement has just been identified. This is discussed in more detail in section 5.3.

```
+++ State Table +++
...
207: selection_statement = ‘if‘, ‘(‘, expression, ‘)‘, statement.
     selection_statement = ‘if‘, ‘(‘, expression, ‘)‘, statement., ‘else‘, statement
...
343: statement = .labeled_statement
     statement = .compound_statement
     statement = .expression_statement
     statement = .selection_statement
     statement = .iteration_statement
     statement = .jump_statement
     labeled_statement = .identifier, ‘:‘, statement
     labeled_statement = .‘case‘, constant_expression, ‘:‘, statement
     labeled_statement = .‘default‘, ‘:‘, statement
     compound_statement = .‘{‘, ‘}‘
...
+++ Conflicts +++
...
1 conflict detected (1 shift/reduce, 0 reduce/reduce)

State 207 on symbol ‘else‘
     Reduce by selection_statement = ‘if‘, ‘(‘, expression, ‘)‘, statement | ...
     Shift to 343
```

Figure 6: LR(0) State Table and Conflict Listing

4.4 SLRP Parser Driver Implementation Details

The LR(1) automaton is implemented in SLRP by the grammar-independent parser driver code in [2, 3]. Reference documentation for the parser dirver API is given in appendix E of this document.

The parser driver API implements a generic LR(1) automaton as an ML function *slrp'parse*. In the implementation, all the information the automaton needs about the LR(0) graph, the grammar and the lookahead table are encoded in a compact way in two tables called the action table and the goto table. The implementation maintains two stacks to represent the path in the graph that led from the root to the current state (one stack for the states and one for the edges).

The function *slrp'parse* takes 8 parameters. The first four of these parameters are values that are computed by SLRP: the first of these gives the initial state of the automaton; the next two give action table and goto table; the fourth parameter gives reduction table containing the semantic action functions that are applied whenever a reduction transition is taken.

The function *slrp'gen_parser* is then generated by SLRP as the result of applying *slrp'parse* to an initial state value and a trio of tables that it computes for your grammar (see appendix E for more information). The remaining four parameters of *slrp'parse* become the parameters to *slrp'gen_parser* as discussed at the beginning of section 3 above.

5 AMBIGUOUS GRAMMARS

This section is intended to help you understand and deal with shift/reduce and reduce/reduce conflicts in your grammar. If either kind of conflict occurs, you have essentially two options: either change the grammar to eliminate the conflicts or take steps to ensure that the parser that is generated does what is wanted despite the conflicts. The following points should be kept in mind when working on a grammar with conflicts.

- You may be fighting against one of the following possibilities:
  - your language may be inherently ambiguous, i.e., there may be no way of describing it with an unambiguous grammar;
  - your grammar may be ambiguous and it may be difficult or impossible to find a tractable grammar for the same language that is not ambiguous;
  - your grammar may be unambiguous but may not belong to the class of LALR(1) languages that SLRP can support without some hand-coded assistance.

  In practice, there is almost always a work-around for these problems, e.g., by writing code to resolve conflicts dynamically or by using a grammar for a tractable superset of the language and imposing extra hand-coded checks.
- A shift/reduce conflict will cause the parser to fail at run-time if it encounters an input that gives rise to the conflict unless you have supplied a RESOLVER function that tells the parser what to do.
- Reduce/reduce conflicts are all resolved statically using the heuristic described in section 4.2 above. You must check that this heuristic is appropriate for your language.
- A grammar with a few well understood conflicts that you can resolve dynamically is often a better solution than a more complicated grammar without the conflicts.

5.1 Debugging a Grammar

Conflicts often occur just because of a mistake in grammar. Common causes include: accidentally omitting an important punctuation symbol, accidentally duplicating an alternative in a production and giving several ways of accepting an empty sequence of symbols. Figure 7 shows some fragments of the grammar for C as given in [6] seeded with a few minor errors along these lines. These modifications introduce 200 conflicts into the grammar.

There is no general rule for debugging a grammar. The following suggestions may be helpful:

- The first thing to do is to study the conflict and state table listings to see which symbols and which rules are causing the problem.

- In the case of a large grammar like the one for C, try analysing sublanguages such as expressions, statements and declarations separately. This will often reduce the amount of information you need to process to find the source of the problem, particularly if there are several unrelated errors in the grammar.
- If the problem seems to be related to missing or ambiguous punctuation in the grammar, then try temporarily introducing distinctive delimiters for the main syntactic categories, e.g., for C, try putting distinctive keywords in at the points where the statement and declaration nonterminals are used.
- Try approximating the language from below: start from a small sublanguage that works and then put in language features one by one.
- Try approximating the language from above: design a superset which admits a simpler description and then refine it in stages to remove unwanted constructs.
- Check for other errors: e.g., SLRP reports if there are nonterminals in your grammar that are useless because they are not reachable from the sentence symbol or because they generate an empty language. These problems could highlight a mistake giving rise to conflicts elsewhere.

```
declaration            (* should have a semicolon terminator *)
        = declaration_specifiers
        | declaration_specifiers, init_declarator_list
        ;
declaration_list       (* an empty declaration list should not be allowed *)
        =
        | declaration_list, declaration
        ;
statement_list         (* an empty statement list should not be allowed *)
        =
        | statement_list, statement
        ;
relational_expression
        = shift_expression
        | relational_expression, '<', shift_expression
        | relational_expression, '>', shift_expression
        | relational_expression, '<=', shift_expression
        | relational_expression, '<=', shift_expression
                       (* typo: should be '>=' - duplicates previous alternative *)
        ;
```

Figure 7: Fragments of an Incorrect Grammar for C

5.2 Reduce/Reduce Conflicts

As discussed in section 4.2, SLRP resolves reduce/reduce conflicts statically using a heuristic based on the order of the rules in your grammar: if the conflict is between two different nonterminals it chooses the one that came first in the grammar, and similarly if it is between two different alternatives for the same nonterminal it chooses the first one.

An example that gives rise to several reduce/reduce conflicts is given in figure 8. This example shows a modification to part of the grammar of figure 4 which produces reduce/reduce conflicts. The heuristic will cause the alternatives with two minus signs or two plus signs to be used in favour of the ones with just one sign.

```
Text dumped to file usr032e.grm.txt
Application =   Atom                                (red_application1 x1)
            |   ‘−‘, ‘−‘, Application               (red_accept x3)
            |   ‘+‘, ‘+‘, Application               (red_accept x3)
            |   ‘−‘, Application                    (red_application2 x2)
            |   ‘+‘, Application                    (red_application3 x2)
            |   identifier, ‘(‘, Expression, ‘)‘
                                                    (red_application4 x1 x3);
```

Figure 8: A Grammar With Reduce/Reduce Conflicts

The heuristic for removing reduce/reduce conflicts works nicely with some styles for presenting a grammar, but care should be taken to check that it does actually implement the language that you want. It is wise to check the state table and the conflicts in the listings carefully. The relevant parts of the listings for the above example are shown in figure 9.

In the example, there are 12 conflicts — the two that are shown in figure 8 are both repeated for each of the symbols ‘*’, ‘/’, ‘+’, ‘-’ and ‘)’ that can validly appear immediately following an Application. The state table shows that the only thing the parser can be doing when the reduce/reduce conflict would occur is recognising an Application following two plus or minus signs. From this you can conclude that it is always safe to reduce by the rule that comes first, and so this tiny optimisation to our calculator is acceptable.

```
+++ State Table +++
...
12:     Application = ‘−‘, ‘−‘, Application.
        Application = ‘−‘, Application.
13:     Application = ‘+‘, ‘+‘, Application.
        Application = ‘+‘, Application.
14:     Application = identifier., ‘(‘, Expression, ‘)‘
...
+++ Conflicts +++
...
12 conflicts detected (0 shift/reduce, 12 reduce/reduce)

State 12 on symbol LCEos
        Reduce by Application = ... | ‘−‘, Application | ...
        Reduce by Application = ... | ‘−‘, ‘−‘, Application | ...
 ...
State 13 on symbol LCEos
        Reduce by Application = ... | ‘+‘, Application | ...
        Reduce by Application = ... | ‘+‘, ‘+‘, Application | ...
...
```

Figure 9: Reduce/Reduce Conflicts — Extracts from the Listing

5.3 Shift/Reduce Conflicts

When SLRP detects shift/reduce conflicts, these always have to be resolved dynamically. I.e., you must supply a RESOLVER function to tell the parser what to do when the conflict occurs. The function can either give the parser one of three options: Shift, Reduce or Error, or it can raise an exception, The function *default_resolver* supplied in the SLRP API always raises the exception *PARSER_ERROR*, which is intended to signal a design error.

The API uses the following data types for the RESOLVER function:

$$
\begin{array}{lcl}
\textit{datatype RESOLUTION} & = & \textit{DoShift} \\
 & & \mid \quad \textit{DoReduce} \\
 & & \mid \quad \textit{DoError};
\end{array}
$$

$$
\begin{array}{l}
\textit{type } ('tok, 'lc, 'pp)\textit{RESOLVER} \\
\qquad = ('tok * 'lc) * ('tok, 'lc, 'pp)\textit{INPUT_STACK} * ((\textit{string} * \textit{int}) * \textit{int}) \\
\qquad\qquad -> \textit{RESOLUTION};
\end{array}
$$

The RESOLVER function is called when a lookahead token giving rise to a shift/reduce conflict is encountered, its arguments have the form $(x, stack, r)$ where x is a pair containing the lexical value and lexical class of the lookahead token, *stack* is the parser driver input stack, and r describes the possible reduction by giving the nonterminal name, the index of the alternative (0-based indexing) and the state to be entered. Most applications do not need all this information, but there are situations where it is useful. As we shall see shortly, the parser driver API provides a function to make one common approach (based on operator precedence) a little easier.

5.3.1 If-then-else

For a first example, consider the example in figure 6 which shows extracts from the listing of the state table for the grammar for C given in [5] together with the description of the conflict that grammar gives rise to. This conflict is caused by a deliberate ambiguity in the grammar, which does not specify whether the else-part in the statement "`if(a) if(b) f(); else g();`" belongs to the inner if-statement or the outer one. The language rule is that the else-part should belong to the inner if-statement and that means the resolution should always be to shift. As this is the only conflict in the grammar, the RESOLVER function in [5] could hardle be simpler:

fun c_resolver _ = DoShift;

Note that if the C language rule said that an ambiguous else-part belonged to the outer if-statement, then it would not be satisfactory to use the grammar of [5] and a RESOLVER that always reduces — the resulting parser would reject valid statements such as "`while(a) if(b) f(); else g();`". To achieve the desired effect, you would either have to rework the

grammar or supply a more complicated RESOLVER that examined the input stack to check if it is appropriate to reduce.

5.3.2 Operator Precedence

Our next example shows a widely used technique. The grammar in figure 10 actually specifies the same language of arithmetic expressions as the grammar of figure 1 considered in section 2 above. However, unlike the earlier grammar, it is ambiguous: it no longer has the precedence rules for the arithmetic operators wired into it; it allows an expression such as "$1 * 2 + 3 * 4$" to be parsed in several different ways.

```
Text dumped to file usr032f.grm.txt
Expression  =      Binary;

Atom  =            literal
                |  identifier
                |  '(', Expression, ')';

Application  =     Atom
                |  '−', Application
                |  '+', Application
                |  identifier, '(', Expression, ')';

Binary  =          Binary, '+', Binary
                |  Binary, '−', Binary
                |  Binary, '∗', Binary
                |  Binary, '/', Binary
                |  Application;
```

Figure 10: A Grammar Requiring Operator Precedence Rules

Figure 11 shows an extract from the SLRP listing for the grammar. The 16 conflicts arise from the sixteen different pairs of the infix operators, '+', '-', '*' and '/'. The conflicts arise in four different states, all of which contain exactly the same set of items (the ones shown in the extract). From the state table, we can see that the conflict occurs when the parser has just got to the end of something that could be a *Binary* and the lookahead token is one of the infix operators. From the conflcts listing, we see that the possible reduction is via one of the alternatives *Binary, '+', Binary. Binary, '-', Binary* etc. This means that the conflict can be resolved by comparing the topmost terminal symbol on the stack with the lookahead token: for example, if the topmost terminal symbol is '*' and the lookahead token is '+', then we having something of the form $a * b + c$, and we should reduce so that the multiplication takes precedence over the addition.

```
+++ State Table +++
...
12:     Binary = Binary., ‘+‘, Binary
        Binary = Binary, ‘+‘, Binary.
        Binary = Binary., ‘−‘, Binary
        Binary = Binary., ‘∗‘, Binary
        Binary = Binary., ‘/‘, Binary
...
+++ Conflicts +++

...
16 conflicts detected (16 shift/reduce, 0 reduce/reduce)

...
State 12 on symbol ‘∗‘
        Reduce by Binary = Binary, ‘+‘, Binary | ...
        Shift to 20
...
```

Figure 11: Operator Precedence — Extracts from the Listing

The API provides the function *simple_resolver* to simplify the coding of a RESOLVER based on operator precedence, you provide a function to compare the two tokens and return the appropriate resolution and it constructs the RESOLVER for you. Your function is passed a pair (*stk_tok*, *la_tok*) where *stk_tok* represents the topmost terminal symbol on the stack and *la_tok* represents the lookahead token. The RESOLVER that results will return *DoError* if there is no unreduced terminal symbol on the stack.

```
val simple_resolver:
        (('tok * 'lc) * ('tok * 'lc) -> RESOLUTION) -> ('tok, 'lc, 'pp) RESOLVER;
```

Figure 30 shows this function in use to implement our example grammar. If you compile this and execute the following, you can check whether it is building the right parse trees.

```
let     val p = ae_parser4 "2*3 + 4; 2 - 3*4; 1 + 2 + 3";
in      print_tree(p());
        print_tree(p());
        print_tree(p())
end;
```

5.3.3 An Inherently Ambiguous Language

A language is called *inherently ambiguous* if there is no unambiguous context-free grammar that describes it. Such languages exist, but are not very common in practical applications. An example is given by the language defined by the ambiguous grammar in figure 12. The language comprises strings comprising some 'a's followed by some 'b's followed by some 'c's subject to the constraint that there should either be the same number of 'a's and 'b's or the same number of 'b's and 'c's or possibly both.

```
Text dumped to file usr032g.grm.txt
    S       = AB, C | A, BC;
    AB      = | `a`, AB, `b`;
    C       = | C, `c`;
    A       = | A, `a`;
    BC      = | `b`, BC, `c`;
```

Figure 12: An Inherently Ambiguous Language

If you are faced with an inherently ambiguous language, or, and much more likely, a language defined by a grammar that is difficult to transform into an unambiguous one, then your only option is to try to specify a wider language and supply code in the reduction functions or subsequent processing to detect invalid language constructs. This is demonstrated for our inherently ambiguous example in figure 13 where the grammar accepts any string comprising some 'a's followed by some 'b's followed by some 'c's and the reduction functions impose the desired checks on the numbers of 'a's, 'b's and 'c's.

```
Text dumped to file usr032h.grm.txt
    S       = A, B, C
                    ((fn Parsed  m => fn Parsed n => fn Parsed p =>
                    if      m <> n andalso n <> p
                    then    raise SYNTAX_ERROR
                    else    0) x1 x2 x3);
    A       =               ( 0 )
            |   A, `a`            ((fn Parsed m => m + 1) x1);
    B       =               ( 0 )
            |   B, `b`            ((fn Parsed n => n + 1) x1);
    C       =               ( 0 )
            |   C, `c`            ((fn Parsed p => p + 1) x1);
```

Figure 13: Resolving Ambiguities By Widening the Language

5.3.4 Transforming a non-LALR(1) Grammar into an LALR(1) one

In practice, if your grammar gives rise to conflicts, it can often be transformed into one that does not. Figure 14 shows an example which adapts our language of arithmetic expressions to support functions with more than one argument and to allow the user to define functions using syntax such as $mean(x, y) = (x + y)/2$.

The way the grammar is expressed in figure 14 must give rise to conflicts. The reason is that, in parsing a construct such as $f(x_1, x_2, \ldots) = \ldots$, while reading the comma after x_1, say, one needs to look ahead as far as the equals sign to guide the LR(0) automaton into parsing x_1 as an *IdentifierList* rather than as an *ExpressionList*.

Figure 15 shows a transformation to the productions for *Application* and *ExpressionList* which gives a grammar that defines the same language as that of figure 15 but with no conflicts. The transformed grammar distinguishes between lists of identifiers and lists of more complex expressions. It makes it explicit how the language of the grammar of figure 14 can be recognised by an LR(0) automaton guided only by one lookahead token.

The alternative to the solution in figure 15 would be to remove the non-terminal *IdentifierList* altogether and use *ExpressionList* in its place in the production for *FormalParams*. The action code for *FormalParams* would then have to check the semantic value of the *ExpressionList* and reject invalid ones. This would probably be a better solution in the case at hand, since it makes the grammar shorter and, assuming the semantic actions compute some kind of syntax tree, processing the tree to check for non-identifiers in a *FormalParams* will be no more work than converting the tree for an *IdentifierList* into what is needed for the tree representing an *Application*..

Text dumped to file usr032i.grm.txt

$$
\begin{array}{lll}
Command = & & Expression \\
& | & identifier,\ `=`,\ Expression \\
& | & identifier,\ FormalParams,\ \ `=`,\ \ Expression; \\
\\
Expression = Sum; & & \\
\\
FormalParams = & & `(`,\ IdentifierList,\ `)`; \\
\\
IdentifierList = & & identifier \\
& | & IdentifierList,\ `,`,\ identifier; \\
\\
Atom = & & literal \\
& | & identifier \\
& | & `(`,\ Expression,\ `)`; \\
\\
Application = & & Atom \\
& | & `-`,\ Application \\
& | & `+`,\ Application \\
& | & identifier,\ `(`,\ ExpressionList,\ `)`; \\
\\
ExpressionList = & & Expression \\
& | & ExpressionList,\ `,`,\ \ Expression; \\
\\
Product = & & Application \\
& | & Product,\ `*`,\ Application \\
& | & Product,\ `/`,\ Application; \\
\\
Sum = & & Product \\
& | & Sum,\ `+`,\ Product \\
& | & Sum,\ `-`,\ Product;
\end{array}
$$

Figure 14: A non-LALR(1) Grammar

Text appended to file usr032j.grm.txt

$$
\begin{array}{lcl}
Application = & & Atom \\
& | & \text{`}-\text{`},\ Application \\
& | & \text{`}+\text{`},\ Application \\
& | & identifier,\ \text{`(`},\ NonIdExpressionList,\ \text{`)`} \\
& | & identifier,\ \text{`(`},\ IdentifierList,\ \text{`)`};
\end{array}
$$

$$
\begin{array}{lcl}
NonIdExpressionList = & & \\
& & NonIdExpression \\
& | & IdentifierList,\ \text{`,`},\ NonIdExpression \\
& | & NonIdExpressionList,\ \text{`,`},\ Expression;
\end{array}
$$

$$
\begin{array}{lcl}
NonIdExpression = & & \text{`(`},\ Expression,\ \text{`)`} \\
& | & \text{`}-\text{`},\ Application \\
& | & \text{`}+\text{`},\ Application \\
& | & identifier,\ \text{`(`},\ NonIdExpressionList,\ \text{`)`} \\
& | & identifier,\ \text{`(`},\ IdentifierList,\ \text{`)`} \\
& | & Product,\ \text{`}*\text{`},\ Application \\
& | & Product,\ \text{`/`},\ Application \\
& | & Sum,\ \text{`}+\text{`},\ Product \\
& | & Sum,\ \text{`}-\text{`},\ Product;
\end{array}
$$

Figure 15: Making a Grammar LALR(1)

6 ERROR HANDLING

As mentioned at the head of section 2, one of the parameters to the SLRP parser driver is an ERROR ROUTINE: an ML function you provide to deal with errors. This function is called whenever there is no entry in the action table telling the parser how to respond to the lookahead token. The function *default_error* is an error routine that implements the simplest error handling strategy of all: it just prints an error message and raises an exception. This can be a perfectly acceptable approach, e.g., see the code in figure 29, which catches the exception, and continues trying to parse the input if it is running interactively.

The type of the ERROR ROUTINE is an instance of the following type:

$$\begin{array}{l} type\ ('tok, 'lc, 'pp, 'st)ERROR_ROUTINE \\ \quad = 'tok * ('tok, 'lc, 'pp)INPUT_STACK * STATE_STACK * 'st \\ \qquad -> 'tok * 'st * int; \end{array}$$

The quadruple that is passed as the argument to the ERROR ROUTINE has the form (tok, is, ss, rs), where: tok is the lookahead token that caused the error, is is the input stack, ss is the stack stack and rs is the READER state. If the function returns a value (rather than raising an exception), then the return value is a triple (tok', rs', n), where tok' is a lookahead token to be used in place of the erroneous one, rs' is a new READER state and n is an integer. If the ERROR ROUTINE returns (tok', rs', n), the parser will pop n entries off its input and state stacks, and proceed as if tok' had just been read as the lookahead token resulting in a READER state rs'.

Figures 31 show an ERROR ROUTINE that raises an exception after printing a bit more information than *default_error*. It looks up the current state (the top of the state stack) in the action table and gets the list of lexical classes that could be dealt with in that state. (The action table entries are lists of lexical class/action pairs). It prints out an error message similar to that printed by *default_error*, and then prints out the list of "expected" lexical classes. This is quite a common technique, although it is sometimes a little misleading, since the lexical classes the parser is expecting in a given state may not be the only ones that could validly continue the input processed so far.

Figuer 32 shows a simple error recovery scheme for the language of arithmetic expressions. Here the error routine first prints out the error message and then it attempts to recover from the error, by discarding tokens until it encounters the end-of-stream marker (either a semicolon or the actual end of the input stream). If there is another token available, it reads it and returns that token and the current READER state to the parser telling it to unwind its stacks back to their initial state. If there is no more input available it raises an exception which is handled in the parser function and causes a tidy exit.

A STANDARD ML CODE EXAMPLES

A.1 Compiling SLRP Parsers

The Standard ML code in the figures in this section compiles all the library support needed for the generic parser for arithmetic expressions described in section 2 and then compiles and instantiates the code generated by SLRP. If you are working in a ProofPower ML database, then you can omit the first six files since they are already compiled into ProofPower.

```
SML
map use [
        "dtd108.sml",        (* Portability infrastructure *)
        "imp108.sml",
        "dtd002.sml",        (* System control and error reporting *)
        "imp002.sml",
        "dtd001.sml",        (* Standard ML utilities *)
        "imp001.sml",
        "dtd018.sml",        (* SLRP parser driver *)
        "imp018.sml",
        "dtd118.sml",        (* Generic SLRP parser support *)
        "imp118.sml"
        ];
open GenericSlrpParser;
```

Figure 16: Compiling the SLRP Library Code

```
SML
use"usr032a.grm.sml";              (* The generated parser code *)
val ae_parser1 : string -> unit =
        print_tree o parse_file slrp'gen_parser;
```

Figure 17: Compiling the Code Generated by SLRP

A.2 A Lexical Analyser

The Standard ML code in the figures in this section implements the lexical analyser for the language of arithmetic expressions as discussed in section 3.1 above.

```
SML
type AE_LEX_VALUE = LEX_CLASS LEX_VALUE;
type AE_LEX_STATE = LEX_CLASS LEX_STATE;
fun lv_identifier (s : string) : AE_LEX_VALUE = (
	(LCIdentifier "identifier", (s, get_line_number()))
);
fun lv_literal (s : string) : AE_LEX_VALUE = (
	(LCIdentifier "literal", (s, get_line_number()))
);
fun lv_punctuation c = (LCString c, (c, get_line_number()));
val lv_left_bracket : AE_LEX_VALUE = lv_punctuation "(";
val lv_right_bracket : LEX_CLASS LEX_VALUE = lv_punctuation ")";
val lv_times : AE_LEX_VALUE = lv_punctuation "*";
val lv_plus : AE_LEX_VALUE = lv_punctuation "+";
val lv_minus : AE_LEX_VALUE = lv_punctuation "-";
val lv_over : AE_LEX_VALUE = lv_punctuation "/";
val lv_equals : AE_LEX_VALUE = lv_punctuation "=";
val lv_semicolon : AE_LEX_VALUE = (LCEos, (";", get_line_number()));
val lv_end_of_input : AE_LEX_VALUE =
	(LCEos, ("<end-of-input>", get_line_number()));
```

Figure 18: Constructing Lexical Values

```
SML
fun rec_punctuation (("(" :: more, _)  : AE_LEX_STATE)
        : AE_LEX_STATE
        = (more, Known  lv_left_bracket)
|   rec_punctuation ( ")" :: more, _ ) = (more, Known lv_right_bracket)
|   rec_punctuation ( "*" :: more, _ ) = (more, Known lv_times)
|   rec_punctuation ( "+" :: more, _ ) = (more, Known lv_plus)
|   rec_punctuation ( "-" :: more, _ ) = (more, Known lv_minus)
|   rec_punctuation ( "/" :: more, _ ) = (more, Known lv_over)
|   rec_punctuation ( ";" :: more, _ ) = (more, Known lv_semicolon)
|   rec_punctuation ( "=" :: more, _ ) = (more, Known lv_equals)
|   rec_punctuation ( "#" :: _, _ ) = ([], Comment)
|   rec_punctuation (chs, _) = (chs, Unknown);
```

Figure 19: Recognising Punctuation Symbols

```
SML
fun is_alph_or_us ch = (
                "a" <= ch andalso ch <= "z"
        orelse "A" <= ch andalso ch <= "Z"
        orelse ch = "_"
);
fun is_digit ch = "0" <= ch andalso ch <= "9";
val is_alnum = is_alph_or_us fun_or is_digit;
fun rec_identifier (([], _) : AE_LEX_STATE) : AE_LEX_STATE = (
        ([], Unknown)
) | rec_identifier (chs as (ch :: more), _) = (
        let     fun aux acc [] = (implode (rev acc), [])
                |   aux acc (cs as (c::more)) = (
                        if      is_alnum c
                        then    aux (c::acc) more
                        else    (implode(rev acc), cs)
                );
        in      if      is_alph_or_us ch
                then    let     val (name, rest) = aux [ch] more;
                        in      (rest, Known (lv_identifier name))
                        end
                else    (chs, Unknown)
        end
);
```

Figure 20: Recognising an Identifier

```
SML
fun rec_literal (([], _) : AE_LEX_STATE)
        : AE_LEX_STATE = ([], Unknown)
|   rec_literal (chs as (ch :: more), _) = (
        let     fun aux acc [] = (implode (rev acc), [])
                |   aux acc (cs as (c::more)) = (
                        if      is_digit c
                        then    aux (c::acc) more
                        else    (implode(rev acc), cs)
                );
        in      if      is_digit ch
                then    let     val (digits, rest) = aux [ch] more;
                        in      (rest, Known (lv_literal digits))
                        end
                else    (chs, Unknown)
        end
);
```

Figure 21: Recognising a Literal

```
SML
fun rec_unknown (( ch :: more, _ ) : AE_LEX_STATE)
        : AE_LEX_STATE = (
        output(std_out, "Unrecognised input character \"" ^ ch ^ "\"\n");
        (more, Comment)
) | rec_unknown ([], _) = ([], Unknown);
```

Figure 22: Dealing with Lexical Errors

```
SML
val rec_token : AE_LEX_STATE -> AE_LEX_STATE =
        rec_first[rec_punctuation, rec_identifier, rec_literal, rec_unknown];
val reader_state : (string list * bool) ref = ref ([], true);
fun reader (strm : IN_CHAR_STREAM)
        : (AE_LEX_VALUE, string list * bool) READER = (
        let     val do_read = gen_reader LCEos rec_token strm
        in      fn state =>
                let     val (tok, state') = do_read state;
                in      reader_state := state';
                        (tok, state')
                end
        end
);
```

Figure 23: Constructing the Reader

```
SML
use"usr032a.grm.sml";
fun ae_parser2
        (text : string)  : unit -> LEX_CLASS GEN_PARSE_TREE = (
        let     val strm as {close, ...} = in_char_stream_of_string text;
                val do_parse =
                        slrp'gen_parser
                        default_resolver
                        classifier
                        (default_error string_of_lex_value)
                        (reader strm);
                val _ = reader_state := ([], true);
        in      fn () => do_parse (!reader_state)
                handle ex => (
                        close();
                        raise ex
                )
        end
);
```

Figure 24: Constructing the Parser

A.3 Adding Actions

The Standard ML code in this section implements the reduction functions that support the action grammar shown in figure 4 in section 3.2 above.

```
SML
val named_numbers : int S_DICT ref = ref [];
fun red_command1 (Parsed i) = (
	named_numbers := s_enter "it" i (!named_numbers);
	output(std_out, "it = " ^ string_of_int i ^ "\n");
	i
);
fun red_command2 (Token ((_, (s, _)), _)) (Parsed i) = (
	named_numbers := s_enter s i (!named_numbers);
	output(std_out, s ^ " = " ^ string_of_int i ^ "\n");
	i
);
```

Figure 25: The Reduction Functions for Commands

```
SML
fun red_accept (Parsed i) = i;
val red_expression = red_accept;
fun red_atom1 (Token ((_, (s, _)), _)) = nat_of_string s;
fun red_atom2 (Token ((_, (s, _)), _)) = (
	case s_lookup s (!named_numbers) of
		Value i => i
	|	Nil => (
		output(std_out, "Undefined name \"" ^ s ^ "\"\n");
		raise SYNTAX_ERROR
	)
);
```

Figure 26: The Reduction Functions for Expressions I

```
SML
val red_atom3 = red_accept;
val red_application1 = red_accept;
fun red_application2 (Parsed i) = ~i;
val red_application3 = red_accept;
fun red_application4 (Token((_, (s, _)), _)) (Parsed i) = (
	case s of
		"abs" => if i < 0 then ~i else i
	|	"sgn" => if i > 0 then 1 else if i = 0 then 0 else ~1
	|	_ => (
		output(std_out, "Unsupported function \"" ^ s ^ "\"\n");
		raise SYNTAX_ERROR
	)
);
```

Figure 27: The Reduction Functions for Expressions II

```
SML
val red_product1 = red_accept;
fun red_product2 (Parsed i) (Parsed j) = i * j;
fun red_product3 (Parsed i) (Parsed j) = i div j handle Div => (
	output(std_out, "Zero divisor\n");
	raise SYNTAX_ERROR
);
val red_sum1 = red_accept;
fun red_sum2 (Parsed i) (Parsed j) = i + j;
fun red_sum3 (Parsed i) (Parsed j) = i - j;
```

Figure 28: The Reduction Functions for Expressions III

```
SML
use"usr032b.grm.sml";
fun ae_parser3
        (instrm : instream)  : unit = (
        let     val strm as {close, ...} = in_char_stream_of_instream instrm;
                val do_parse =
                        slrp'gen_parser
                        default_resolver
                        classifier
                        (default_error string_of_lex_value)
                        (reader strm);
                val _ = reader_state := ([], true);
        in      while case !reader_state of ([], false) => false | _ => true do
                        (do_parse (!reader_state); ())
                        handle SYNTAX_ERROR =>  (
                                (if     not (ExtendedIO.is_term_in instrm)
                                then    (close(); raise SYNTAX_ERROR)
                                else    ())
                        )
        end
);
```

Figure 29: Constructing the Parser

A.4 Operator Precedence Conflict Resolution

```
SML
use"usr032f.grm.sml";
fun     precedence (((_, (stk_val, _)), _) , ((_, (la_val, _)), _))  = (
        let     fun num_prec "*" = 2
                |   num_prec "/" = 2
                |   num_prec _ = 1;
                val stk_prec = num_prec stk_val;
                val la_prec = num_prec  la_val;
        in      if      stk_prec <  la_prec
                then    DoShift
                else    DoReduce
        end
);


fun ae_parser4
        (text : string)  : unit -> LEX_CLASS GEN_PARSE_TREE = (
        let     val strm as {close, ...} = in_char_stream_of_string text;
                val do_parse =
                        slrp'gen_parser
                        (simple_resolver precedence)
                        classifier
                        (default_error string_of_lex_value)
                        (reader strm);
                val _ = reader_state := ([], true);
        in      fn () => do_parse (!reader_state)
                handle ex => (
                        close();
                        raise ex
                )
        end
);
```

Figure 30: The Operator Precedence Parser

A.5 Error Handling

```
SML
use"usr032b.grm.sml";
fun ae_error_reporter (tok, is, ss as (s::_), rs) = (
	let	val lcs =map (string_of_lex_class o fst)
			(PPArray.sub(slrp'actions, s));
		val location = case is of [] => "here"
		|	_ => "after " ^ format_stack string_of_lex_value is
		val sorted_lcs = Sort.sort Sort.string_order lcs;
		val msg1 = "*** Error in arithmetic expression:\n";
		val msg2 = string_of_lex_value tok ^
			" is not expected " ^ location ^ "\n";
		val msg3 = "Expected " ^
			format_list (fn x => x) sorted_lcs ", " ^ ", ...\n";
	in	output(std_out, msg1 ^ msg2 ^ msg3)
	end
);
fun ae_error_handler5 tok_is_ss_rs =
	(ae_error_reporter tok_is_ss_rs; raise SYNTAX_ERROR);
fun ae_parser5
	(instrm : instream) : unit = (
	let	val strm as {close, ...} = in_char_stream_of_instream instrm;
		val do_parse = slrp'gen_parser default_resolver
			classifier ae_error_handler5 (reader strm);
		val _ = reader_state := ([], true);
	in	while case !reader_state of ([], false) => false | _ => true do
			(do_parse (!reader_state); ())
			handle SYNTAX_ERROR =>
				(if	not (ExtendedIO.is_term_in instrm)
				then	(close(); raise SYNTAX_ERROR)
				else	())
	end
);
```

Figure 31: A Simple Extension to the Default Error Routine

```
SML
use"usr032b.grm.sml";
exception DONE;
fun ae_error_handler6 rdr = (fn (tok, is, ss, rs) =>
	let	val _ = ae_error_reporter (tok, is, ss, rs);
		val n = length is;
		fun is_eos (LCEos, _) = true | is_eos _ = false;
		val ref_tok : AE_LEX_VALUE ref = ref tok;
	in	(while not(is_eos(!ref_tok))
		andalso		case !reader_state of ([], false) => false | _ => true
		do	let	val (t, s) = rdr (!reader_state);
			in	ref_tok := t
			end);
		case !reader_state of
			([], false) =>	raise DONE
		|	_ =>
			let	val (t, s) = rdr (!reader_state);
			in	(t, s, n)
			end
	end
);
fun ae_parser6
	(instrm : instream)  : unit = (
	let	val strm as {close, ...} = in_char_stream_of_instream instrm;
		val rdr = reader strm;
		val do_parse =
			slrp'gen_parser
			default_resolver
			classifier
			(ae_error_handler6 rdr)
			(reader strm);
		val _ = reader_state := ([], true);
	in	(while case !reader_state of ([], false) => false | _ => true do
			(do_parse (!reader_state); ())
			handle SYNTAX_ERROR =>	(
				(if	not (ExtendedIO.is_term_in instrm)
				then	(close(); raise SYNTAX_ERROR)
				else	())
			)) handle DONE => output(std_out, "\nBye!\n")
	end
);
```

Figure 32: A Simple Error Recovery Scheme

B COMMAND LINE INTERFACE

B.1 Command Line Syntax

The shell script *slrp* runs the `slrp` parser generator. It is supplied as part of the ProofPower developer toolkit. It is called from the UN*X shell command line with the following syntax:

slrp $-f$ *grammar_file* $[-e\ \ eos]$ $[-g]$ $[-q\ \ quote_con]$ $[-n\ \ name_con]$ $[-l\ \ n]$ $[-t]$

B.2 Input and Output File Conventions

The *-f grammar_file* option identifies a file containing the input for `slrp`. The file name must have a ".txt" file name extension, e.g. *myparser.txt*, so "*.txt*" is appended to *grammar_file*, if not already present, e.g., *myparser* is treated as *myparser.txt*.

The SML code output by `slrp` is written to a file whose name is formed by replacing the ".txt" extension with ".sml", e.g., *myparser.sml*.

Listings (see section B.4 below), if any, are written to a log file whose name is formed by replacing ".txt" with ".log", e.g., *myparser.log*. A brief report on the results of the run is written to standard output.

B.3 Code Generation Options

The *-e eos* option gives the name of the lexical class denoting end of input. The default is "*LCEos*".

The *-g* option causes calls on the generic reduction functions to be included in the reduction tables for productions in the grammar which have no actions.

If *-q quote_con* is specified, quoted constants in the grammar are interpreted as Standard ML strings, and *quote_con* is the name of a constructor to apply to the strings where they appear in the action tables. If *-q quote_con* is not specified, but *-g* is specified, the constructor "*LCString*" is used. If neither of *-q quote_con* and *-g* is specified, quoted constants in the grammar are interpreted as Standard ML identifiers.

The option *-n name_con* has a similar effect to *-q quote_con* for terminal symbols given in the grammar as names rather than quoted constants. The default constructor for names when *-g* is specified is "*LCIdentifier*".

B.4 Listing Options

The option *-l* may be used to specify an integer n determining the level of log information to be written to the file named *log_file*, as shown in the table below. The option *-l 2* is the most useful level during development of a parser and is the default.

Level	Information Logged
0	No log file is produced.
1	This gives a summary of the conflict resolution process and a listing of any conflicts requiring dynamic resolution.
2	As 1 together with a listing of the grammar and its terminals and the state table and a listing of any conflicts before resolution.
3	As 2 together with a listing of the full action table.
4	As 3 together with a listing of the graph of the LR(0) automaton.
11	As 4 together with a listing of the Bermudez-Logothetis state transition grammar (forces LALR(1) calculations even for an SLR(1) grammar).
≥ 12	As 11 together with a list of all the LALR(1) lookahead sets. (forces calculation of all lookahead sets).

The option *-t* causes execution times for to be included in the report written to standard output at the end of the run.

C SLRP INPUT FORMAT

The BNF dialect used by `slrp` is a subset of British Standard BNF, [4], extended to allow fragments of ML code to be given with any alternative.

The subset is the one in which the only operators are concatenation and alternation and in which grouping with parentheses is not allowed. Empty alternatives are allowed. An alternative may optionally be followed by an *action*, which is just a fragment of Standard ML text. We refer to such an extended grammar as an *action grammar*. The BNF syntax for action grammars is as follows:

Grammar	=	*Prod*, ‘;‘
	|	*Prod*, ‘;‘, *Grammar*;
Prod	=	*Name*, ‘=‘, *Def*;
Def	=	*OptAlt*, *OptAction* | *OptAlt*, *OptAction*, ‘|‘, *Def*;
OptAlt	=	*Alt* |;
Alt	=	*Symbol* | *Symbol*, ‘,‘, *Alt*;
Symbol	=	*Name* | *Constant*;
OptAction	=	*Action* | ;

The terminal symbols in the above grammar are *Name* :, *Constant* and, *Action*. Names are formed using alphanumeric characters and underscores. Names can be used to denote individual non-terminal symbols or classes of terminal symbols.

Constants start and finish with a backprime character ‘‘’. A ‘‘’ may appear within a constant provided it is preceded by a backslash character, ‘\’. Constants are typically used to denote individual terminal symbols.

Actions start with a left bracket, ‘`(`, and finish with a right bracket, ‘`)`’. Any brackets appearing within an action must be well-balanced. Actions denote fragments of Standard ML code to be executed by the generated parser when a particular alternative has been recognised. The actions may be omitted, e.g., when using `slrp` to help design a grammar. Omitting the actions is appropriate when one is experimenting with a grammar, e.g., to convert a non-LALR(1) grammar into an LALR(1) one.

Comments may be included in a grammar using the Standard ML comment symbols: “ verb”(*”” and “`*)`”. Comments may be nested.

D STANDARD ML LIBRARY

The parser driver API and the generic parser API are implemented using a library of data types and functions borrowed from the ProofPower-HOL system. This is described in detail in chapter 2 of the ProofPower-HOL reference manual [7].

In fact, the parser driver function and the data types that it needs only depend on one data type from the library: the type *E_DICT* of efficient string indexed dictionaries. If only the parser driver function is required, it is be straightforward to compile the parser driver function given a suitable implementation of this data type only. This allows a parser generated by SLRP to be integrated at source level without having to import the ProofPower-HOL library.

E PARSER DRIVER API

This section contains the reference documentation for the parser driver API. It includes the Standard ML signature to which the API conforms together with brief narrative descriptions of the types and values defined in the signature. An index to the APIs described in this document is given in appendix G below.

SML

signature ***SlrpDriver*** = *sig*

Description This is the signature of the structure *SlrpDriver* which provides the parser driver function and associated data types and utility functions.

SML

type ***STATE*** = *int*;

type ***STATE_STACK*** = *STATE list*;

datatype ('*tok*, '*lc*, '*pp*) ***INPUT_STACK_ITEM***
= ***Token*** *of* '*tok* ∗ '*lc*
| ***Parsed*** *of* '*pp*;
type('*tok*, '*lc*, '*pp*) ***INPUT_STACK***
= ('*tok*, '*lc*, '*pp*) *INPUT_STACK_ITEM list*;

Description Parser states are represented as integers, although user code should generally not need to be aware of that. The parsing stack is represented in two parts: a state stack and a stack of partially parsed input. These represent (respectively) the nodes and edges making up the path from the root of LR(0) graph that has led to the current state (in reverse order).

The partially parsed input stack contains items of two sorts: *(a)* unreduced tokens, given by a pair consisting of a '*tok*' and a '*lc*, '*tok* being the type of the items in the input token stream, '*lc* being the type of the lexical classes used, and *(b)*, reduced phrases represented by a value of type '*pp*.

SML

```
datatype ACTION        =
                         Shift of STATE
                    |    Reduce of ((string * int) * int)
                    |    Dynamic of STATE * ((string * int) * int)
                    |    Accept
                    |    Error;
```

Description Actions are encoded in the datatype *ACTION*. These represent the transitions that can be made by the parser driver function.

Shift(s) means shift into state *s*, i.e., push *s* and the current input token onto the parsing stacks.

Reduce((N, a), m) means reduce to non-terminal named N using the a-th alternative for that non-terminal and popping m entries from the parsing stacks. Before popping the entries off the stacks, the function stored in the reduction table for the specified alternative is called passing the input stack as the parameter. In the tables generated by SLRP, this function will be an expression of the form *fn (xm::. . . ::x1::stk) => e*, where e is the action for the alternative. Thus when the action e is evaluated, each ***xJ*** will be bounded to the J-th symbol in the alternative and ***stk*** will be bound to the portion of the input stack that represents the left context of the non-terminal that is being reduced (e.g., for use in formatting diagnostic reports).

The *Dynamic* option is for a shift/reduce conflict which is to be resolved during parsing by a user-defined routine. It combines the information required for the shift and the reduce action. The parser driver calls a user-supplied RESOLVER function to decide which action to take.

Accept means that a sentence in the language has been successfully parsed. The parser driver returns the partially parsed input that makes up the single entry left on the input stack.

Error means that the input is not a sentence in the language. The parser driver calls a user-supplied error which can either raise an exception or attempt to recover from the error. See the description of the data type *ERROR_ROUTINE* below.

SML

```
type ('lc)ACTION_TABLE
        = ('lc * ACTION) list Array.array;
type GOTO_TABLE
        = (string * STATE) list Array.array;
type ('tok, 'lc, 'pp)REDUCTION_TABLE
        = (('tok, 'lc, 'pp)INPUT_STACK -> 'pp) Array.array E_DICT;
```

Description The action tables are two-dimensional arrays indexed by states and lexical classes. The action tables with which we work are held as one-dimensional arrays of lists of lexical class-(action-state pair) pairs. If the state index for these tables is out of range then the table has been generated incorrectly and the exception *PARSER_ERROR* is raised if this occurs. User error entries correspond to valid state indices for which the lexical class in question is not represented.

Similarly the goto tables (for the non-terminal symbols) are held in a one-dimensional array of pairs each comprising a non-terminal name and a state.

Finally, the user-defined reduction code (indexed by non-terminal names and alternative indices) is held in string-indexed dictionaries of lists of functions from (slices of) partially parsed input stacks to *'pp*.

Indexing errors with either the goto tables or the reduction tables indicate that the table is incorrect and cause the exception *PARSER_ERROR* to be raised.

SML

```
datatype RESOLUTION     =     DoShift
                        |     DoReduce
                        |     DoError;
type ('tok, 'lc, 'pp)RESOLVER
        = ('tok * 'lc) * ('tok, 'lc, 'pp)INPUT_STACK * ((string * int) * int)
              -> RESOLUTION;
```

Description The user-defined dynamic conflict RESOLVER function takes an input stack and the reduction information contained in the dynamic action and returns a value indicating whether the parser should shift, reduce or report an error:

SML

```
type ('tok, 'lc)CLASSIFIER          = 'tok -> 'lc;
```

Description The CLASSIFIER function that gives the lexical class of an input token is passed to the parser driver function as a value of this type.

SML

$type\ ('tok, 'lc, 'pp, 'st)\boldsymbol{ERROR_ROUTINE}$
$\quad = 'tok * ('tok, 'lc, 'pp)INPUT_STACK * STATE_STACK * 'st$
$\qquad -> 'tok * 'st * int;$

Description The ERROR ROUTINE that can either report errors or attempt to recover from them (or both) is passed to the parser driver function as a value of this type.

The ERROR routine must either raise an exception or return a value, (tok, st, n). The latter case instructs the driver to continue as if the current input token is tok, the current READER state is st after popping n entries from its parsing stacks.

SML

$type\ ('tok, 'st)\boldsymbol{READER}$
$\qquad = 'st -> ('tok * 'st);$

Description The READER function that is used to generate the input stream to be parsed is passed to the parser driver function as a value of this type.

The type parameter $'st$ represents some kind of internal state of the input stream. For example, $'st$ might be instantiated to $('tok)list$ for a parser which was to be used in a context where all the input is to be available before parsing begins — in this case the reader would be a function which returns the head and tail of the argument if the list was not empty and returns the end-of-sentence symbol and an empty list otherwise. Alternatively, for a reader which read text from a file, $'st$ might be instantiated to *instream* or *unit*.

SML

```
val slrp'parse:
        STATE ->
        ''lc ACTION_TABLE ->
        GOTO_TABLE ->
        ('tok, ''lc, 'pp) REDUCTION_TABLE ->
        ('tok, ''lc, 'pp) RESOLVER ->
        ('tok, ''lc) CLASSIFIER ->
        ('tok, ''lc, 'pp, 'st) ERROR_ROUTINE ->
        ('tok, 'st) READER -> 'st -> 'pp;
```

Description This is the type of the parser driver function. The first parameter is the initial parser state. The meaning of the other parameters is given in the description of the corresponding data types above.

This function is not normally called directly from user code. Instead, the `slrp` parser generator is used to generate a file of ML code which first binds the ML variables *slrp'initial_state*, *slrp'actions*, *slrp'gotos*, and *slrp'reducers* to appropriate values and then binds the ML variable *slrp'gen_parser* as follows:

```
fun slrp'gen_parser x =
        slrp'parse slrp'initial_state slrp'actions slrp'gotos (slrp'reducers()) x;
```

The user code then applies *slrp'gen_parser* to a RESOLVER, a CLASSIFIER, an ERROR ROUTINE and a READER to give a parser for the language defined by the grammar supplied as the input to `slrp`.

Note: the bindings generated for *slrp'reducers* and *slrp'gen_parser* are *fun* bindings rather than *val* bindings to circumvent the limitations imposed by the value binding restriction introduced in Standard ML '97.

SML

```
exception SYNTAX_ERROR;
exception PARSER_ERROR of string;
```

Description These two exceptions may be raised by the parser driver function and associated functions.

SYNTAX_ERROR is raised by *default_error*. It is intended to signal a syntax error in the input to the parser.

PARSER_ERROR “should not happen”: it indicates an error in the parser driver function logic or its tables. It is also raised by the *default_resolver* function (since that is intended for use in situations where there are no shift/reduce conflicts, so that it is a design error if the parser driver function encounters a dynamic action requiring it to call the RESOLVER function).

SML
```
val format_stack : ('tok -> string) -> ('tok, 'lc, 'pp)INPUT_STACK -> string;
```

Description This utility function is for use in formatting diagnostic messages. It formats the parsing stack in reverse order (i.e., in the same order as the input stream) as a string. It formats the reduced entries as three dots and uses the supplied token printer to format the unreduced entries.

SML
```
val default_error:
        ('tok -> string) -> ('tok, 'lc, 'pp, 'st)ERROR_ROUTINE;
```

Description Many applications of the parser generator will be served by the following *default_error* function which is parameterised by a function to print input tokens. It writes an error message on the standard output and then raises *SYNTAX_ERROR*. The messages it produces have one of the following forms:

**** ERROR Syntax error ****
<token> not expected after: <stack print out>

**** ERROR Syntax error ****
<token> not expected here

Here "< *token* >" and "< *stackprintout* >" are the result of printing the offending input token and the parsing stack using the supplied input token printer (and *format_stack*). The second form is produced if there is nothing on the stack to print out (i.e., the error has occurred on the first token read, or the stack contains only reduced entries).

SML
```
val default_resolver: ('tok, 'lc, 'pp) RESOLVER;
```

Description The default resolver is one which raises *PARSER_ERROR* if it is called. It is intended for use when there are no conflicts.

SML
```
val simple_resolver:
        (('tok * 'lc) * ('tok * 'lc) -> RESOLUTION) -> ('tok, 'lc, 'pp) RESOLVER;
```

Description For grammars which do contain shift/reduce conflicts, one of the commonest forms of conflict resolution simply compares the latest input token with the topmost token on the stack. *simple_resolver* helps you such a resolver. Its argument is a function which compares two tokens (which are given as a pair in input order, i.e., the latest input token comes second). The resulting resolver will report a syntax error if there are no tokens on the stack, a situation which corresponds, for example, to an input whose first symbol is an infix operator.

SML
```
end; (* of signature SlrpDriver *)
```

F GENERIC PARSER API

This section contains the reference documentation for the generic parser API. It includes the Standard ML signature to which the API conforms together with brief narrative descriptions of the types and values defined in the signature. An index to the APIs described in this document is given in appendix G below.

SML

```
signature GenericSlrpParser = sig
include SlrpDriver;
```

Description This is the signature of a structure containing the generic `slrp` parser API.

SML

```
datatype LEX_CLASS =
            LCIdentifier of string
      |     LCString of string
      |     LCEos;
type 'lc LEX_VALUE = 'lc * (string * int);
val classifier : ('lc LEX_VALUE, 'lc) CLASSIFIER;
val string_of_lex_class : LEX_CLASS -> string;
val string_of_lex_value : LEX_CLASS LEX_VALUE -> string;
val line_number_of_lex_value : 'lc LEX_VALUE -> int;
```

Description The types are the types of lexical classes (*'lc* in the parser driver API) and of lexical values (*'tok* in the parser driver API) for the generic `slrp` parser. *classifier* is the CLASSIFIER function that maps the latter onto the former. The integers in the lexical values are source line numbers. The function *string_of_lex_value* returns a textual representation of a lexical value and is what is used as the parameter to the generic ERROR ROUTINE function *default_error* in the parser. *line_number_of_lex_value* returns what its name suggests.

SML

```
exception LexFail of int * string;
datatype CONTINUATION_STATUS =
            InComment
     |      InString of string list;
datatype 'lc LEX_STATUS =
            Unknown
     |      Known of 'lc LEX_VALUE
     |      Comment
     |      Continuation of int * CONTINUATION_STATUS;
type 'lc LEX_STATE = (string list * 'lc LEX_STATUS);
val rec_first :
     ('lc LEX_STATE -> 'lc LEX_STATE) list ->
     'lc LEX_STATE -> 'lc LEX_STATE;
val get_line_number : unit -> int;
```

Description These types etc. are used to construct the line-at-a-time part of the lexical analyser for the generic `slrp` parser. In conjunction with the function *gen_reader*, q.v., they may be used to construct lexical analysers for an actual language rather than for the `slrp` representation of its terminal symbols.

SML

```
type IN_CHAR_STREAM;
val in_char_stream_of_instream : instream -> IN_CHAR_STREAM;
val in_char_stream_of_file : string -> IN_CHAR_STREAM;
val in_char_stream_of_string : string -> IN_CHAR_STREAM;
val gen_reader : 'lc ->
     ('lc LEX_STATE -> 'lc LEX_STATE) ->IN_CHAR_STREAM ->
      ('lc LEX_VALUE, string list * bool) READER;
val reader : IN_CHAR_STREAM ->
      (LEX_CLASS LEX_VALUE, string list * bool) READER;
```

Description This type and associated functions provide the READER function that the parsing function generated by `slrp` uses to read a sequence of lexical tokens.

SML

```
type 'lc SLRP_GEN_PARSER;
datatype 'lc GEN_PARSE_TREE =
            Leaf of 'lc LEX_VALUE
    |       Node of (string * int) * 'lc GEN_PARSE_TREE list;
val generic_reducer  : string * int ->
    ('lc LEX_VALUE, 'lc, 'lc GEN_PARSE_TREE)
            INPUT_STACK_ITEM list ->
    ('lc LEX_VALUE, 'lc, 'lc GEN_PARSE_TREE)
            INPUT_STACK_ITEM list ->
    'lc GEN_PARSE_TREE;
val output_tree : ('lc LEX_VALUE -> string) -> (string -> unit) ->
    'lc GEN_PARSE_TREE -> unit;
val print_tree : LEX_CLASS GEN_PARSE_TREE -> unit;
val parse_file : LEX_CLASS SLRP_GEN_PARSER -> string ->
    LEX_CLASS GEN_PARSE_TREE;
val parse_string : LEX_CLASS SLRP_GEN_PARSER -> string ->
    LEX_CLASS GEN_PARSE_TREE;
```

Description The type *SLRP_GEN_PARSER* is the instance of the type of the parser function *slrp'gen_parse* generated by `slrp`.

The remaining type and associated functions support the generic action functions generated by `slrp` when called with the *-g* option and implement the resulting parser. When used with the supplied code for *generic_reducer*, the parser will compute a value of type *GEN_PARSE_TREE*. *print_tree* and output_tree may be used to print out a textual representation of such a value .

parse_file name attempts to parse sequence of lexical tokens represented in the named file. *parse_string string* does the same job for a sequence of tokens given in an ML string. The lexical format for the token sequences is the same as that used in the `slrp` grammar. I.e., they are either ML-style alphanumeric identifiers or arbitrary strings given in back-quotes. ML-style comments are allowed and ignored.

SML

```
end; (* of signature GenericSlrpParser *)
```

G API INDEX

Accept ... 52
ACTION_TABLE ... 53
ACTION ... 52
CLASSIFIER ... 53
classifier ... 57
Comment ... 58
CONTINUATION_STATUS ... 58
Continuation ... 58
default_error ... 56
default_resolver ... 56
DoError ... 53
DoReduce ... 53
DoShift ... 53
Dynamic ... 52
ERROR_ROUTINE ... 54
Error ... 52
format_stack ... 56
GenericSlrpParser ... 57
generic_reducer ... 59
GEN_PARSE_TREE ... 59
gen_reader ... 58
get_line_number ... 58
GOTO_TABLE ... 53
InComment ... 58
INPUT_STACK_ITEM ... 51
INPUT_STACK ... 51
INPUT_STACK ... 53
InString ... 58
in_char_stream_of_file ... 58
in_char_stream_of_instream ... 58
in_char_stream_of_string ... 58
IN_CHAR_STREAM ... 58
Known ... 58
LCEos ... 57
LCIdentifier ... 57
LCString ... 57
Leaf ... 59
LexFail ... 58
LEX_CLASS ... 57
LEX_STATE ... 58
LEX_STATUS ... 58
LEX_VALUE ... 57
line_number_of_lex_value ... 57
Node ... 59
output_tree ... 59
Parsed ... 51
PARSER_ERROR ... 55
parse_file ... 59
parse_string ... 59
print_tree ... 59
READER ... 54
reader ... 58
rec_first ... 58
Reduce ... 52
REDUCTION_TABLE ... 53
RESOLUTION ... 53
RESOLVER ... 53
Shift ... 52
simple_resolver ... 56
slrp'parse ... 55
SlrpDriver ... 51
SLRP_GEN_PARSER ... 59
slrp ... 47
STATE_STACK ... 51
STATE ... 51
stk ... 52
string_of_lex_class ... 57
string_of_lex_value ... 57
SYNTAX_ERROR ... 55
Token ... 51
Unknown ... 58
xJ ... 52