RAL-89-082

Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-89-082

SML-Yacc: A Parser-Generator System in Standard-ML. A User Guide

B M Matthews and S K Robinson

August 1989

SML-Yacc : A Parser - Generator System in Standard-ML
A User Guide

Brian M. Matthews

Stuart K. Robinson

Rutherford Appleton Laboratory

1. Introduction

This paper describes SML-Yacc – a parser-generator in Standard ML. The system expects a grammar specification as input and produces a parser for that language in Standard ML as output. The user may include Standard ML expressions (*parse actions*) within the specification to be executed on recognising a phrase of the language and thereby define the semantics of the language that the grammar specifies. For example, the expressions may be used to build up a parse-tree in some representation defined by the user. Parser-Generators can be used in the construction of compliers and they can also be used to produce a variety of other software tools such as automatic type-checkers, automatic type-setters, and cross-compilers, which rely on the syntactic structure of the input.

The overall structure of the process which must be undertaken to produce a parser is given in Fig 1. The user supplies SML-Yacc with a grammar specification plus associated parse actions which describe the language and its intended semantics. SML-Yacc generates a parse table for this language, which is then combined with a driver routine to form the parser. The user then provides this parser with input which has been preprocessed by a user defined lexical analysis function and the parser produces, if the input is syntactically correct, the corresponding translated output as dictated by the parse actions.

Many people are familiar with Yacc, the standard parser-generator utility on UNIX† . Consequently we have endeavoured to design the interface to the SML-Yacc system to be as similar to that of Yacc as possible. This should make the conversion of Yacc grammars simpler.

1.1. History

The primary motivation for the production of SML-Yacc was as a Rutherford Appleton Laboratory contribution to the Alvey funded FORSITE project, a collaboration between the University of Oxford, Racal ITD Ltd, and the University of Surrey. However, the resulting system is intended for distribution to the Standard ML community at large to enhance the range of tools available in this language.

Parser-generator systems have been available since the late 1950's as aids to the complex task of writing translation tool (eg compilers) for programming languages. The use of such a system can significantly reduce the time required to produce such a tool, although the efficiency of the tool produced may not be as high as one which has been hand-coded and optimised.

The Yacc parser-generator is a standard tool provided on Unix and as such is a widely known and popular system for this task. It is written in the UNIX standard language 'C' and also expects parse actions and produces output code in 'C'. The near standardisation of UNIX in the UK academic community makes this system a natural choice on which to base our own parser-generator, and thus SML-Yacc input specifications contain many similar constructs to those found in Yacc. Existing specifications should thus be adaptable into the new format in a straightforward manner.

Standard ML is a strict polymorphic functional language developed at Edinburgh University. It was

† UNIX is a registered trademark of AT&T in the USA and other countries.

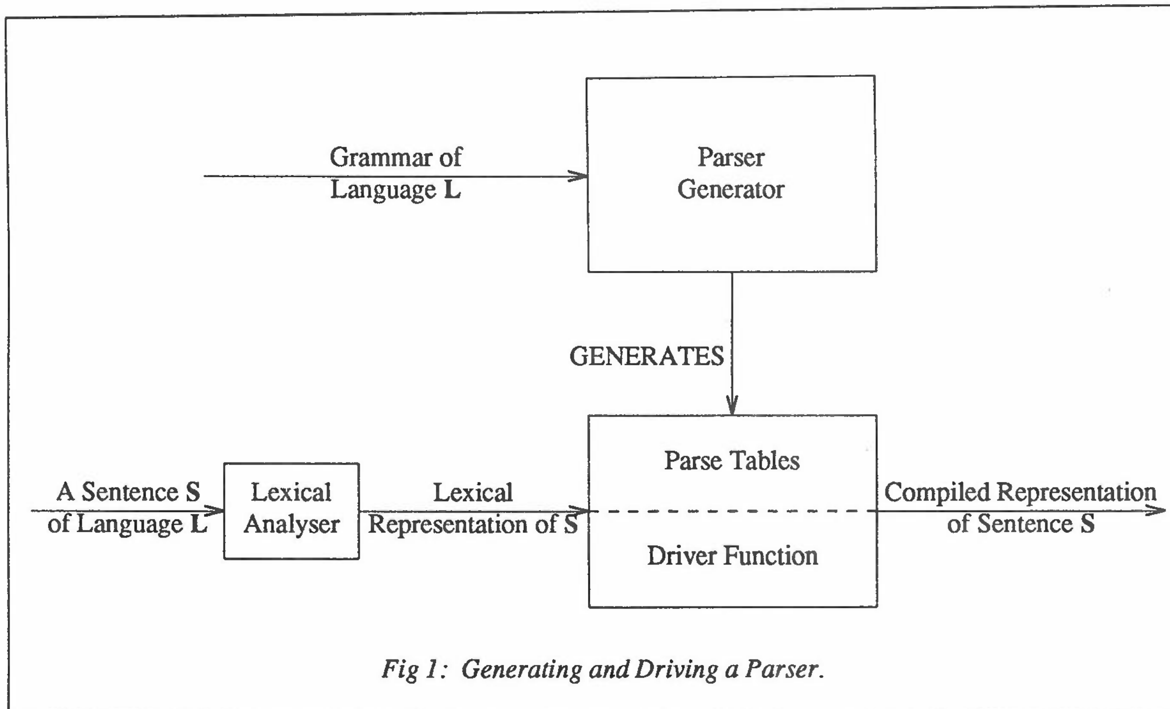

Fig 1: Generating and Driving a Parser.

originally designed as a Meta-Language for the LCF theorem proving system, but since then it has been developed independently as a general purpose programming language. Although it does have some imperative features, we were interested in experimenting with parser-generation techniques in a functional programming paradigm.

1.2. Technical Background and Related Work

For the theoretical background to context-free grammars, finite automata and parsing techniques see [HopUll79]. SML-Yacc is a LALR(1) parser generator. For the technical details of this and for a description of the algorithm upon which SML-Yacc is based see [AhoUll77, AhoSetUll86]. For an alternative approach to LALR(1) parser generation see [Trem85]. These books also give a good motivation to lexical analysis and parsing.

The Yacc system [John75] is a well-known parser-generator system written in C as part of the toolkit supplied with UNIX. [Pey83] describes an attempt to implement Yacc in the untyped functional language SASL. Other systems include [Roth87] using New-Jersey ML at Edinburgh, mllama [Suf89] written in Pascal with ML front and back ends as part of the Zebra (a Z specification type-checking tool), and [Udd88] using Lazy ML at Chalmers University Gothenburg, which extends the basic Yacc functionality in the context of attribute grammars permitting both synthesised and inherited attributes. This can be carried out in a Yacc-like framework because of the lazy evaluation strategy of the implementation language.

For an introduction to Standard ML see [Wik87].

1.3. Organisation of this Manual

In §2 we describe the format of the input grammar specifications used in SML-Yacc, together with user defined functions. In §3 we describe the parse actions, and how to use the parse funtion output by SML-Yacc. There are some extra features available to SML-Yacc allowing the user more control over the parser produced; these are described in §4. We then follow this by a fully worked out example in §5, installation instructions in§6, and a description of the grammar of the input specifications to SML-Yacc in a SML-Yacc acceptable form. We conclude with a note on maintainance.

2. Input Features

2.1. Grammars

The language is specified using a grammar, which is then input to SML-Yacc. The grammar consists of a set of *terminal symbols* representing the fundamental units of the input, a set of *non-terminal symbols* which represent the higher structures of the language, and a set of *productions* which relate a single non-terminal symbol (the *left side*) to a sequence of terminal and non-terminal symbols (the *right side*). The theory of grammars can be found in the above references, and the detailed syntax for defining input grammars for SML-Yacc is covered in subsequent sections of this paper.

SML-Yacc will accept Context-Free Grammars which are acceptable to a LALR(1) parsing algorithm. Parsers for LALR(1) languages are often implemented using two components: a set of *parse tables* which contain the information of the the structure of the particular language; and a *driver function*, which is the same for all LALR(1) parsers, which uses the parse tables to process the input. SML-Yacc produces the parse tables for a language by analysing the grammar, and also provides the driver function to operate the parser.

LALR(1) grammars define a wide class of languages and most common programming language constructs can be encoded in this framework. However, there are some language constructs which cannot be defined, leading to ambiguities in the language; that is constructs where more than one legal parse is possible. One of the strengths of SML-Yacc is the facility to resolve some language ambiguities by precedence rules set by the user (see below).

Upon input, several checks are made upon the grammar specification:

1 All non-terminals used must be defined. That is, if a non-terminal occurs in the right side of any rule, there must be a rule which has that non-terminal as its left side.

2 All non-terminals defined must be used. That is, if a non-terminal occurs on the left side of a rule, then there must be some rule in which it occurs as part of the right side. Although not strictly necessary for the correct implementation of the parser-generator, this check is made to improve efficiency and to promote good style from the user by identifying unnecessary rules.

3 Rules which promote non-terminating loops are detected, reported and removed. The simplest example is the definition of non-terminal S with the single production $S \rightarrow \alpha S \beta$, but non-termination can also arise from certain sequences of rules which mutually refer to each other.

4 Token identifiers and non-terminal identifiers must be taken from disjoint sets. That is no token identifier is found on the left side of a production.

In checks 2 and 3 above, the offending productions in the grammar are signalled with an appropriate error message, and then removed from the grammar. Analysis then proceeds using the reduced grammar to detect further errors.

2.2. User Supplied Functions

The user is expected to supply to SML-Yacc a type **value**, a function **lex:** *unit → (int * value)* and a function **yyerror:***(value list * (int * value)) → string*. These three Standard ML objects provide the type of the result of the parse actions given to productions, a lexical analysis function, and a function which outputs an error message respectively. The two functions are given as arguments to the **yyparse** outlined in §3.3 below. The function **yyerror** is outlined in §4.3 below.

2.2.1. Type value

All the parse actions assigned to productions take their arguments from and return their results to a common value stack. The strong type discipline of Standard ML insists that all these arguments and results are of the same type. The type **value** is declared to be this type in the driver function, but not defined. Consequently, the user is required to give this type in advance, and to ensure that all given actions use this type for all arguments, and results.

2.2.2. Lexical Analysis

The lexical analyser performs a local analysis of the input, breaking up the input symbol stream into tokens, which are used by the parser to analyse the grammatical structure of the input more deeply. The lexical analyser also can be used to perform routine tasks such as stripping out white space from the input, removing comments, keeping a tally of the current line number for error diagnosis, and maintaining symbol tables.

In order to perform lexical analysis, the user is expected to provide a function **lex:** *unit → (int * value)* which returns the next token from the input stream when called with a null argument. This input token is in the form of a pair. The first element of the input token is the ineger code (or token class-code) for the token, and the second is the value of that token, which has already been coerced to be of type **value**. The value of the integral class-codes for the tokens can be found in a list, **Token_table:***(string * int) list* (ordered on the second argument in reverse order), generated by SML-Yacc, and available to the lexical analyser.

Thus for example, we might outline a lexical analysis function skeleton as follows.

```
datatype value = ...  (* defines the type value *)

exception no_token : unit ;
fun search nil input = raise no_token
   | search ((s,n)::list) input = if input = s then n
                                  else search list input ;

fun read (():unit):string = ... (* reads a tokens string from the input *)

fun make_value (s:string):value = ... (* converts a string to the appropriate value *)

fun lex token_table () =   (* Here we give the lexical analysis function *)
    let val input = read ()
    in
    (search token_table input, make_value input)
    end

(* We now load the parse tables as generated by SML-Yacc. *)

val Token_table = ...

(* and build the final lexical analysis function. *)

val Lexical = lex Token_table ;
```

This is rather crude, and the user would probably produce something more efficient, and tailored to his or her own requirements.

2.3. The Form of the Specification File

The overall format of a SML-Yacc input specification of a language is as follows.

```
declarations
%%
rules
%%
programs
```

The *declarations* section contains the definitions of various token identifiers and symbols. The *rules* section contains the productions (with optional parse actions) of the input grammar. The *programs* section contains declarations of user-defined ML functions and datatypes which the user wishes to use in conjunction with the parser that has been defined. The lexical analysis function can be placed here, as can any

user-defined parse error handling functions.

All declarations (apart from the start symbol declaration (see below)), can be omitted if they are not required, as can the programs section. If the latter is done the second delimiter '%%' can also be omitted. Thus the smallest legal SML-Yacc specification is one of the form:

```
%start goal   (* see later *)
%%
rules
```

Blanks, newlines and tabs are ignored for input purposes. Comments can be placed at any point in the specification, being delimited by '(*' and '*)' as in Standard-ML.

2.4. Rule Format

The rule section consists of one or more grammar productions of the form:

```
n : BODY ;
```

where 'n' represents a non-terminal, and BODY represents a sequence of right sides separated by bars ("|"). The right sides themselves are sequences of token identifiers (terminals), literals and non-terminals, followed by an optional parse action at the end. The BODY may be empty, or just consist of a parse action, in which case it represents the empty string. The colon, bar and semi-colon are SML-Yacc punctuation (metasymbols).

Identifiers for tokens and non-terminals can be of arbitrary length, and contain any characters apart from those with a special meaning in SML-Yacc, that is:

```
';'   ':'   '|'   '%'   '%%'
'(*'  '*)'  '" '  '%('  '%)'
```

The recommended convention is to identify tokens by upper-case strings and non-terminals by lower-case strings. Literals are denoted by enclosing characters in double quotes, eg ":", and may contain any character including the SML-Yacc reserved characters and the following escape sequences:

\n	newline
\t	tab
\ddd	single character with octal Ascii code 'ddd'
\"	quote character "
\\	backslash character \

Literals are regarded as terminal tokens, but they do not have to be entered in the declarations section of the specification (see below) .

Several productions may have the same non-terminal symbol on the left side. In SML-Yacc we do not insist that all the productions of a given non-terminal be presented together and separated by '|'. Hence the following is allowed:

```
a  :  b;
   .
   .
   .
a  :  c;
```

However, we do recommend, for the sake of readability, that this be written as:

```
a  :  b | c ;
```

Further, as SML-Yacc ignores new-lines in the input specification, each right side can occupy a different line. This style is recommended, with the ';' on a further line, as this improves readability and makes any subsequent modification of the specification easier. Thus the above example would be written:

```
a   :  b
    |  c
    ;
```

2.5. Declarations

Each identifier used to represent a terminal token must be declared in the declarations section. This enables SML-Yacc to distinguish between terminal and non-terminal identifiers. If an identifier is not declared it will be regarded as a non-terminal and cause an error message from input check 1 above. Not all tokens need be declared however: literals delimited by double quotes are also regarded as terminal identifiers. Check 4 above prevents tokens and non-terminals from having the same identifier. Token identifiers are declared by a statement of the form:

```
%token identifier1 identifier2 identifier3 ...
```

in the declarations section of the specification. There may be several such statements in the specification.

One non-terminal symbol, known as the start symbol, must be declared explicitly, by the statement:

```
%start symbol
```

in the declarations section. This symbol represents the most general structure which can be built up from the grammar, and is used by SML-Yacc as an indication that the parsed input is a complete sentence of the language specified by the input grammar.

A special endmarker token, *EOF*, with the token class code ~1 (the only negative class code) is the signal to the parser produced that the input has been completed. It must be delivered by the user supplied lexical function when the input has ended. If the tokens up to the endmarker form a sentence derivable from the start symbol according to the specified grammar rules, then the input is accepted; otherwise an error results.

The user can supply ML code to be used by the resulting parser in the initial declarations section. This is done by placing the code within the delimiters '%(' and '%)'. This code is then copied to the beginning of the generated parser output file *filename*.**tab** (see below). This is a useful place in which to put the definition of the type **value**, and the functions used by the parse actions (see below). Code can also be placed after the final '%%', which will be copied to the end of the output file *filename*.**tab**, after the code for the generated parser.

3. Output Features

When SML-Yacc is run on a correct input specification of a grammar, a parser is produced which will accept any input derivable from the grammar's start symbol. The user can specify actions which are to be performed when the parser accepts those sections of the input which match nonterminal structures in the specifying grammar, to produce a processed version of the input.

3.1. Actions

The action to be performed on recognising a production's right side can be specified by entering into the input specification a Standard ML expression enclosed in parentheses at the end of the relevant right side. For example, given the production

```
a  :  "(" b ")"   ([1,2,3]) ;
```

then, on successfully parsing "(" b ")", the list [1,2,3] will be the resulting value of the production (assuming that we have already declared **type** *value = int list ;*).

Note that the Standard ML terminator symbol ';' should not be included in an action.

An action may contain pseudo-variables to pass values from the input to the output. These are written v_1, v_2, v_3, ... and are bound to the values returned by the grammar symbols (terminals and non-terminals) in the right side of the relevant production. The number 'i' of the pseudo-variable v_i, corresponds to the position of the grammar symbol whose value it takes, counting from left-to-right, in the right side of the production. For example, if we have the production

```
b  :  BODY   ( 1 ) ;
```

then the expression '1' is perceived to be the 'result' of the parse of BODY, and becomes the value of 'b'. This value can be used when b is used within other productions by use of pseudo-variables. Thus if we also have the production

```
a  :  "(" b ")"   ( v_2 ) ;
```

then v_2 takes the value of the second element in the rule, which is then passed on as the value for a. In the case where b is parsed using the above production, then this particular use of a returns the value '1'.

Standard ML tuple expressions and expression sequences need not be enclosed by additional parenthesis. Thus:

```
a  :  a c   (1,2) ;
```

is a valid action. Actions cannot be placed in the middle of the right side of a production, such as:

```
a  :  b ( 1 ) c   ( v_2 ; v_3 ) ;
```

Such actions can be simulated by adding extra rules:

```
a  :  b d c   ( v_2 ; v_3 ) ;
d  :          ( 1 ) ;          (* An Empty Production *)
```

but care must be taken. In changing the grammar, it may cease to be a grammar suitable for LALR(1) parsing and conflicts may arise. Note that a Standard ML expression sequence is being used here, and so the separator ';' is being used legally in the first action.

One task of the lexical analysis function is to return the required value of the appropriate token to be used in actions. As described earlier, all values (ie. those passed by the lexical analyser as the values of the tokens, and those returned from the users parse actions as the value of non-terminals) must be of the same type, **value**. Should they not be, the Standard ML type checker will produce an error when it comes to load up the resulting parser.

3.2. Actions on Empty Productions

The driver function for the parsers produced by SML-Yacc maintains a stack of values which have been parsed. When the parser reads a token, its value is pushed onto the value stack. When the end of a right side is recognised, the parser pops a number of values equal to the number of symbols in the right side, performs the given action for the right side on the popped values, and places the result of the action on top of the value stack.

For example, if we have the following section of a grammar definition:

```
%(
type value = int ;
%)
.
%token Num
.
%%
.
add    :    Num "+" Num      (v_1 + v_3) ;
.
```

and the input contains the sequence *'6 + 5'*, we may recognise the end of the 'add' production with the following value stack (0 is an arbitrary value returned for the "+" symbol by the user's lexical analyser).

5
0
6
3

The parser will then pop the top three values off leaving

3

and then, with the binding v_1 = 6, v_2 = 0, v_3 = 5 (reverse order, since that is the order they have been read in), it will perform the action (v_1 + v_3), evaluating to 11 which is then put on the top of the value stack as the value of this use of 'add', yeiding:

11
3

Thus a value is required both for every terminal and for every non-terminal used in the grammar. If no action is provided, then the parser uses the value of the first symbol onto the right side on the value stack as the value of the left side. Thus if we have the production:

```
a    :    b c ;
```

then, after a has been recognised, the value of b will be left on the top of the stack.

The rule that every non-terminal used has a value applies equally to empty productions. On 'recognising' an empty production, no extra values have been placed on the value stack. If an action is provided for the production then the result of the action (which cannot contain pseudo-variables) is used as the value of the left-side non-terminal. For example, a common use for the empty production is in sequences of expressions or list construction:

```
%(
type value = int list;
%)
.
%%
.
list   :   singleton list   (v_1 @ v_2)
       |                    ([])          (* Empty Production *)
       ;
.
```

Here the empty production returns the value '[]' which is placed on the value stack.

If no action is provided on an empty production, a problem arises. We need a value to return, but have no such value available. To overcome this problem, the user can declare a 'dummy value' to be used in such circumstances in the declarations section of the SML-Yacc grammar. This is done using the keyword '**%dummy**' followed by a element of type **value** to be returned by such empty productions which have no action provided. Thus given:

```
%(
type value = int;
%)
%dummy 0
.
%%
.
empty :          (* Empty Production *) ;
.
```

then when the production empty is recognised, the value '0' will be returned and placed on the value stack. If an empty production is declared with no action provided, and no default value is supplied using %dummy, then during the analysis of the grammar, SML-Yacc will fail and output the error message "Error: No dummy value supplied for empty production".

3.3. Using SML-Yacc and the Parser Function

SML-Yacc provides a function **Yacc :** *(string * string * string * string) → unit* to generate parsers from grammar specifications. Yacc takes four parameters used as follows.

Argument	Parameter	Function
1st	"*filename*"	The file name containing the input specification to be processed.
2nd	"v" or ""	If set to "v" then SML-Yacc produces a verbose output outlining the states of the resulting parser in a file called *filename.v.tab.*
3rd	"y" or ""	If set to "y" then SML-Yacc produces a file called *filename.err* to which any error and parser conflict reports are sent.
4th	"y" or ""	If set to "y" then SML-Yacc produces a file called *filename.out* to which all output, normally displayed on the screen, is sent.

The system produces, as a side effect, a file *filename.tab* containing the generated set of parse tables, which include the following.

Token_table: *(string * int) list* , which is a list of all acceptable token identifiers and their numeric internal class codes. This list is organised as a reversed ordered list on the *second* (numeric) argument. This list is essential for the lexical analysis, as **lex** needs to return the token-class codes for tokens.

Reduce_functions: *(value list → value list) list* which are the user supplied semantic actions embedded in value stack manipulating functions.

The other tables contain the parser action information in a compacted form. When the parser is loaded, the functions in the file **make.ml** are used to build the parse table used by the driver function, which is to be found in the file **driver.ml**.

In order to use these tables, the user should first load his definitions of **lex**, the datatype **value** (this must be provided as SML-Yacc provides no default value type), the error function **yyerror** (see below), and any functions utilised in parse actions, then the file containing the tables, *filename*.**tab**. This file automatically loads the make and driver functions from the files **make.ml** and **driver.ml** respectively. Of course, if the type value and the user defined functions are declared in the declarations section of the grammar specification between the '%(' and '%)' delimiters, then they are already in the file *filename*.**tab**, and thus only this file needs to be loaded.

In this way the user is provided with a function:

yyparse: *(unit → (int * value)) → ((value list * (int * value)) → string) → unit → value.*

This can then be supplied with the user's lexical analysis function, **lex** and the user's error message function, **yyerror**, to deliver a function of type *unit → value*. It is assumed that the user's lexical analysis function handles input from the user's file, or from the terminal.

After a successful parse, the final value will appear on top of the value stack and is presented to the user as the result of the parse. Usually, this will still have the value constructor attached which will have to be stripped away from the result before further use.

Note that all parse actions must have the type *value*^{n} *→ value.* Thus, if functions defined on a variety of types are used, then a union type must be defined for value to collect all the types together. Constructors

and deconstructors must be extensively used to coerce the function arguments and results to the appropriate types. This is a tedious necessity imposed on us by the strict type discipline of Standard ML. If only one type is used as a value, then the type value can be set by simple aliasing.

3.4. Example: Actions and the Parser Function

Suppose we wish to construct a parse tree for a simple language described by the following grammar specification:

```
%start sum
%token NUMBER IDENTIFIER (* the values are passed from the lexical analyser *)
%%

sum    :   atom "+" atom    (action) ;
atom   :   IDENTIFIER       (v_1)
       |   NUMBER           (v_1)
       ;
```

The action on the first right side is undefined at present.

The required parse tree is of the following type:

```
datatype exp = mk_plus of (aexp * aexp)
and aexp = mk_int of int
         | mk_real of real
         | mk_string of string ;
```

NUMBER can either be an integer or a real, and IDENTIFIER is of type string. Since all values must have the same type, we must declare a datatype that can embed objects of type *exp* and *aexp*. We will also need deconstructors for these types.

```
datatype value = mk_val1 of exp | mk_val2 of aexp ;
fun umk_val1 (mk_val1 e) = e ;
fun umk_val2 (mk_val2 e) = e ;
```

To get the parse tree of type **exp** as we desire, the action of the first production in the grammar above might be:

```
(mk-val1 (mk-plus (get (umk_val2 v_1), get (umk_val2 v_2)))
```

where **get**:*aexp -> real* is defined to be a function which returns a real number by looking up the assigned value of a string in some predefined association list such as (without going into all the details)

```
fun get (mk_int i) = real i
  | get (mk_real r) = r
  | get (mk_string s) = lookup Assoc_List s ;
```

After the grammar specification has been processed by SML-Yacc, we can use the parse function **yyparse** in our own program as follows. **yyerror** is the error message function as normal, and I assume that the user has supplied a function **makelex** which given an instream, returns an appropriate lexical analysis function.

```
fun my_parse (file : string) =
    let
        val instream = open_in file
        val lex = makelex instream
        val value = yyparse lex yyerrror ()
        val ((mk-val1 parse_tree) = value
    in
        parse_tree
    end;
```

The variable parse_tree will take the value of the parse tree which appears on top of the value stack after a successful parse and is given as the result of my_parse.

4. System Features

Some features of SML-Yacc's analysis of the input grammar are explicit and partially under the user's control.

4.1. Ambiguity

If there is a set of productions in the input grammar which can structure an input string in two different ways, then the grammar is ambiguous (and so not LALR(1)). There are two valid parses and we must make a choice in the parser as to which production to apply. The choice may be incorrect and not lead to a valid parse of the whole sentence, or not carry the semantic meaning we intend.

For example, consider the following productions for subtraction, where NUM is a token with a numerical value:

```
exp  :  exp "-" exp    (v_1 - v-2)
     |   NUM
     ;
```

Consider the input: NUM – NUM – NUM. This can be structured in two ways by the above productions:

```
( NUM - NUM ) - NUM     (left association)
```

or

```
NUM - ( NUM - NUM )     (right association)
```

On applying the parsing algorithm, the input can read up to 'NUM – NUM' which matches the right side of the production, storing each value on the value stack. On reading the next '–', it could then be *reduced* by applying this production, and replacing the top three items on the stack by 'exp' and then carry on to read '– NUM'. This is equivalent to left association. Alternatively, a *shift* action could performed on reading the second '–', going on to read all the input until it has read the entire sentence. It is then reduced from the right. This is equivalent to right association. This situation, where there is a choice of parsing strategies is known as a *shift / reduce conflict.* If the parse action associated with the above production is normal arithmetic subtraction, it can be seen that these two different parsing strategies could lead to very different results.

The input grammar can also contain situations where there are two alternative reduce operations available. For instance consider the following three rules.

```
x  :  A B C;
y  :  B C;
z  :  A y;
```

On reading 'ABC' two parse strategies are available. It can either be reduced using x, or it can be reduced using y followed by a reduction using z. Such situations are known as *reduce / reduce conflicts.*

Productions where these conflicts could potentially occur can be detected during the analysis of the grammar by the parser-generator. Some parser-generators, on detecting such conflicts, would halt and report error messages. However, by default, SML-Yacc will resolve them in a standard fashion according to the following rules:

1 Shift/Reduce conflicts are resolved by taking the shift action - reductions are deferred if possible. This leads to right association.

2 Reduce/Reduce conflicts are resolved by preferring one reduction over the other. The choice, which is arbitrary, is signalled to the user. This is a very unsatisfactory solution to this problem, but such conflicts can be regarded as near fatal. The situations where they occur are often artificial as in the above where rules y and z could be dispensed with altogether.

In general it is better to try to rewrite the grammar to remove all conflicts. In particular, this should be done in cases of reduce/reduce conflict. SML-Yacc will produce an error message indicating where in the grammar the conflict occurs, before resolving the conflict by the above rules and recommencing the analysis of the grammar. If many conflicts occur, or the grammar is long, we recommend that these error messages

are redirected to a file for later analysis. This can be done using the error redirection argument of the top-level function **Yacc** as indicated above.

The resolution of shift/reduce conflicts by a shift is probably more useful than resolving by a reduction. This can be seen by considering the following rules.

```
statement   :   IF "(" condition ")" statement
            |   IF "(" condition ")" statement ELSE statement
            ;
```

This is a construction which often occurs in programming languages, where a conditional may or may not have an action on falsity. Consider how the following sentence is parsed using these productions:

IF (C) IF (D) S ELSE T

The ELSE token could either apply to the outermost or the innermost conditional. A shift / reduce conflict occurs after reading statement S. If reduce is applied at that point, then the ELSE read, the resulting form will be:

IF (C) { IF (D) S } ELSE T

However if the shift option is taken at this point to read the ELSE then the resulting form will be:

IF (C) { IF (D) S ELSE T }

This is the form which most programmers would understand by this sentence.

4.2. Precedence

Ambiguities can be resolved in another way which overrides the default mechanism, and allows the user a great deal of freedom in the way in which operators are parsed and used.

The user can declare precedence levels and associativities for operators in the following fashion. In the declarations section of the input specification, the operator token identifiers or token literals can be declared using the statements **%left**, **%right**, and **%nonassoc** instead of **%token**. %left / %right declare the operator to be left / right associative respectively with operators of equal precedence, and the top to bottom ordering of the declarations gives the precedence of the operators, with the least binding at the top and the strongest binding at the bottom. Operators declared in the same statement have the same precedence.

The %nonassoc declaration is useful for detecting illegal constructs. It means that the operator does not associate. For example, consider the conditional statements of FORTRAN, where the relational operators may not associate with themselves. The declaration:

%nonassoc .LT. .GT. .LE. .GE. .EQ. .NE.

would would cause the generated parser to detect such phrases as

A .GT. B .GT. C

and flag a syntax error with an appropriate message.

Another facility available with precedence is the **%prec** keyword which allows the precedence of an operator to be locally changed in one particular grammar rule. This is particularly useful, for example, in the case of a language with a unary minus where the binding is much stronger than the binary minus operator, but the same symbol is used.

We give as an example of these precedence rules, the following specification of a language of arithmetic:

```
%token ID

%right "="
%left "+" "–"
%left "*" "/"
%left NEGATIVE
%%

arith_exp   :   "(" arith_exp ")"
            |   arith_exp "=" arith_exp
            |   arith_exp "+" arith_exp
            |   arith_exp "–" arith_exp
            |   arith_exp "*" arith_exp
            |   arith_exp "/" arith_exp
            |   "–" arith_exp              %prec NEGATIVE
            |   ID
            ;
```

Here '+' and '–' are both left associative and have a lower precedence than '*' and '/'. However, in the seventh right side of 'arith_exp', '–' precedence has been locally changed to have the precedence of the dummy token NEGATIVE, which has the strongest binding of any of the operators. Hence the input

x = y + z * w – – a / b + c + d

would be structured as:

x = ((((y + (z*w)) – ((–a) / b)) +c) +d)

In these rules, expressions with brackets would parse as would be expected; they would reduce to one arith_expr as if they had top precedence. Declared precedences and associativities override the default activity in conflict resolution in favour of the correct associativity when precedences are equal (left gives a reduce , right a shift) or in favour of the highest precedence when precedences differ.

The %prec keyword and its token should appear in a production after all the grammar symbols, but before any action on the production.

Conflicts resolved in such a manner are not reported and care should be exercised that errors in the specification are not being disguised by the misintended use of these constructions.

4.3. Error Handling

SML_Yacc provides the user with a basic error handling facility.

Under normal circumstances, when the parser generated by SML_Yacc discovers an error in parsing, the system displays an error message, aborts the parse and returns a Standard ML exception **PARSE_ERROR:** *unit.* The driver function's error handling routine will provide the initial message

"Parse Error : Unexpected Input Token".

The user must provide a function **yyerror:***(value list * (int * value)) → string* which should give a more complete error message to follow this. The first argument is the current value stack, giving information on what has been recognised; the second argument is the current input token (of the same form as the output from the (user provided) **lex** function considered earlier). A basic function would thus be

fun yyerror (vals,input_token) = "ERROR"

but the user will probably want to produce a more detailed error message providing details about the current state of the parse. In addition, he or she may wish to provide extra information provided by the lexical analysis, such as the current line number.

Further, the user can provide recovery facilities to handle situations where errors are foreseen. This is carried out by adding, at a non-terminal where the user believes that errors are likely to occur, the extra

production:

non-terminal : **error** α ;

where **error** is a predefined reserved token identifier, and α is a possibly empty string of terminal and non-terminal symbols. When an error occurs the parser will look back down its stack of previously accepted non-terminals. If it finds one with the error token at the start of one of its right sides, it will then try to match the input with the string α. If the input is not acceptable to α, it is silently consumed and the next input token is taken. If the token is acceptable to α the parser will attempt to resume parsing. However, it remains in an error state until three consecutive, acceptable tokens are found in order to prevent a cascade of error messages. For example if the user wishes to consume the input until some end of statement marker is reached, such as a semi-colon in Standard ML, then a production such as:

statement : error ";" ;

can be added. The input will then be consumed silently until a semi-colon marking the end of a statement is reached. The parser will then attempt to resume. If the subsequent input is valid to follow a statement, then the parse will proceed as normal. Otherwise it will silently consume input until three tokens are successfully read. The string of non-terminal and terminal symbols may be empty as in:

statement : error ;

In this case the parse will resume on any token which can legitimately follow the non-terminal statement. This is a very uncontrolled form of error recovery.

5. An Example

We give here a complete example of a simple language which defines a calculator with assignable registers.

> A simple calculator.
>
> This grammar will parse, and evaluate line-by-line, simple arithmetic in floating point numbers. It can also assign values to registers and use them in later expressions. Allows up to 52 registers, denoted by single upper or lower case letters of the alphabet.
>
> Input is terminated by entering a single exclamation mark character, "!".
>
> As we are using ML's limited floating point capacity, the answers may not be entirely accurate!

```
%(

(* The datatype value is a union type of strings (for registers) and reals *)

datatype value = mk_string of string
               | mk_real of real
               | dummy ;

fun unmk_string (mk_string s) = s ;
fun unmk_real (mk_real r) = r ;

(* Functions used in the semantic actions *)

fun curry f x y = f(x,y) ;
val plus:(real -> real -> real) = curry (op +) ;
val minus:(real -> real -> real) = curry (op -) ;
fun prod (r1:real) r2  = r1 * r2 ;
val quot = curry (op /) ;
fun power r1 r2 = exp (r2 * ln r1 ) ;
val neg:(real -> real) = ~ ;

val Assignments = ref ([]:(string * real) list);
exception UnAssigned : string ;

fun assign s r =
   let fun insert [] = [(s,r)]
         | insert ((a,b)::l) = if a = s then (s,r)::l else if a < s then
         (a,b)::insert l
            else (s,r)::(a,b)::l
   in (Assignments := insert (!Assignments) ; dummy)
   end ;

fun lookup s =
   let fun search [] = raise UnAssigned with s
         | search ((a,r)::l) = if a = s then r else if a < s then search
         l
            else raise UnAssigned with s
   in search (!Assignments)
   end ;

fun Output (r:real) = (output(std_out,"The Expression has value "
```

```
                    ^(makestring r)^"\n");dummy);

%)

(* tokens used *)

%token EOL BRA KET NUM IDENT EQ

%left MINUS
%right PLUS
%left QUOT
%right PROD
%right EXP
%right UMINUS (* a dummy token used for the unary minus with %prec *)

%dummy dummy

%start session

(* Now we have the definition of the grammar *)
%%

session  :                              (* empty *))
         |  session line EOL
         ;

line     :  expr                        (Output (unmk_real v_1))
         |  assign
         |  error                       (* Error recovery will take place on a new line *)
         |                              (* empty *)
         ;

assign   :  IDENT EQ expr               (assign (unmk_string v_1) (unmk_real v_3))
         ;

expr     :  BRA expr KET                (v_2)
         |  bexpr
         ;

bexpr    :  NUM
         |  IDENT                       (mk_real (lookup (unmk_string v_1)))
         |  expr PLUS  expr             (mk_real (plus (unmk_real v_1) (unmk_real v_3)))
         |  expr MINUS expr             (mk_real (minus (unmk_real v_1) (unmk_real v_3)))
         |  expr PROD  expr             (mk_real (prod (unmk_real v_1) (unmk_real v_3)))
         |  expr QUOT  expr             (mk_real (quot (unmk_real v_1) (unmk_real v_3)))
         |  expr EXP   expr             (mk_real (power (unmk_real v_1) (unmk_real v_3)))
         |  MINUS expr %prec UMINUS     (mk_real (neg (unmk_real v_2)))
         ;

%%

(*
      When passed through SML-Yacc, the above grammar results in 28 parse states, with no conflicts.
*)
```

```
(*
        We now define the lexical analysis functions.

        First some Utility functions for manipulating input, characters and floating point numbers.
*)

val Instream = ref (open_in "") ;
val line_num = ref 1 ;
val file_name = ref "" ;

fun digit c = c >= "0" andalso c <= "9" ;
fun lower c = c >= "a" andalso c <= "z" ;
fun upper c = c >= "A" andalso c <= "Z" ;
fun digitval d = real(ord d - ord "0") ;

fun getdecimal dec r =
      (case lookahead(!Instream) of
      " " => r
      | c  => if digit c
            then getdecimal (10.0 * dec)
            (r + (1.0/dec * digitval(input(!Instream,1))))
            else r
      ) ;

fun getreal r =
      (case lookahead(!Instream) of
      " " => r
      |"." => (input(!Instream,1) ;r + getdecimal 10.0 0.0)
      | c  => if digit c
            then getreal (10.0*r + digitval(input(!Instream,1)))
            else r
      ) ;

(*
        Then we give the lexical analysis function proper. Note that we lookup the token number in an list
        input as an argument. This can be assigned the Token_table as output by SML-Yacc.
*)

local
fun search ((a,r)::l) s =
      if a = s then r
      else search l s
   | search [] s = raise UnAssigned with s
in
fun lex token_list () =
      if lookahead(!Instream) = " " orelse lookahead(!Instream) = ""
      then
      (input(!Instream,1) ; lex token_list () ) else
         let val token = input(!Instream,1)
             val (tn,vn) = (case token of
                   "!" => ("eof",dummy) |
                   "+" => ("PLUS",dummy) |
                   "-" => ("MINUS" ,dummy)|
                   "*" => ("PROD",dummy) |
```

```
                    "/" => ("QUOT",dummy) |
                    "^" => ("EXP" ,dummy) |
                    "(" => ("BRA",dummy) |
                    ")" => ("KET",dummy) |
                    "=" => ("EQ",dummy) |
                    "." => ("NUM",mk_real(getdecimal 10.0 0.0)) |
                    "\n"=> (line_num := (!line_num + 1);("EOL",dummy)) |
                    _   => if upper token orelse lower token
                           then ("IDENT",mk_string token) else
                           if digit token
                           then ("NUM",mk_real(getreal (digitval token)))
                           else raise UnAssigned with token
                   )
          in
          (search token_list tn,vn)
          end
end                 (* of local *);

(*
        Now the yyerror function.
*)

local
fun search ((a,r)::l) s =
      if r = s then a
      else search l s
   | search [] (s:int) = raise UnAssigned with (makestring s)
in fun yyerror token_list (_,(token,value:value)) =
      "Error at symbol " ^
      (case search token_list token of
            "NUM" => makestring(unmk_real value) |
            "IDENT" => unmk_string value |
            "SUM" => "+" |
            "MINUS" => "-" |
            "PROD" => "*" |
            "QUOT" => "/" |
            "EXP" => "^" |
            "BRA" => "(" |
            "KET" => ")" |
            "EQ" => "=" |
            _ => "" )  ^
      " on line number " ^ (makestring (!line_num))
end                 (* of local *);


(*
```

Finally we tie everything together in the function Calculator, which opens the appropriate input file, evaluates lex and yyerror with the argument Token_table generated by SML-Yacc, and then evaluates the input file.

If Calculator is called with the argument "", then the parser works interactively, evaluating lines as the user types them in from the terminal.

```
*)
```

- Example -

```
fun Calculator (filename:string) =
   let val _ = if filename = "" then (Instream := std_in)
             else (Instream := open_in filename)
       val _ = (line_num := 1 ; Assignments := [])
       val Lex = lex Token_table
       val Yyerror = yyerror Token_table
       val value = yyparse Lex Yyerror ()
   in
   (if filename = "" then ()
   else close_in (!Instream) ;
   value )
   end ;
```

6. Installation Instructions

SML-Yacc was written using FAM Version 3.3 of Standard ML from Edinburgh. The SML-Yacc parser-generator system tries to keep to the definition of the core Standard ML language as closely as possible. There are however a few places where difficulties may occur in porting the system to other versions of ML. Polymorphic references are used in one or two places (such as declaring a reference to be bound to the empty list). Non-functional arrays are used in the final output tables. If any difficulty arises due to these, then the files **top.ml** and **make.ml** contain alternative functional definitions of the datatype contained in comments, which can be used directly to replace the use of the imperative arrays with the loss of some speed. No use is made of Standard ML's Module system.

When the SML-Yacc system has been loaded from tape, the user will be provided with two directories, *examples* and *system.* The directory *examples* contains example grammars for use with SML-Yacc, including the calculator example above. The directory *system* will contain the ML source code for the SML-Yacc system. In order to install the system, the following modifications should be made.

Change the line in the file **load** from

```
val source = "/u/bmm/forsite/system/";
```
to
```
val source = "your_directory/system/";
```

and in the file **IO.ml** from

```
val source = "/u/bmm/forsite/system/";
```
to
```
val source = "your_directory/system/";
```

To load and run the system, enter ML (we recommend with an expanded heap space – say 4Mb), and enter the expression

```
use["your_directory/system/load"] ;
```

and the system should load up. If you are already in the directory *your_directory*/system this part of the pathname can be omitted. An image can then be saved to avoid loading the entire system up again, as this is rather time consuming. This can be done (in the above version of Standard ML) by entering the expression:

```
Save_SML_Yacc () ;
```

to Standard ML after loading the SML-Yacc system. This creates an image and stores it in the file SML_Yacc.x. This image can then be reinvoked by calling Standard ML with the image file SML_Yacc.x.

7. The Grammar of the SML-Yacc.

We give the grammar of the input specifications used in SML-Yacc in a format suitable for processing by SML-Yacc. SML-Yacc uses a cut-down variant of this grammar for its own parser. However, the the declarations are handled by a preprocessor, and there are a variety of modifications to handle special circumstances.

```
(*****************************************************************)
(*          Grammar of the input specifications for SML-Yacc.           *)
(*****************************************************************)

%dummy 0                (* a dummy dummy value!! *)

%token TOKEN NONTERMINAL ACTION EOF VALUE ML_CODE
%start specification

%%

specification   :   declarations "%%" grammar end
                ;

(* declarations ensure that the %start declarations occurs once in any spec *)

declarations    :   nonstart_decs "%start" NONTERMINAL nonstart_decs
                ;

(* nonstart_decs ensure that the %dummy occurs at most once in any spec *)

nonstart_decs   :   token_decs "%dummy" VALUE token_decs
                |   token_decs
                ;

token_decs      :   "(%" ML_CODE "%)" token_decs
                |   "%token" token_list token_decs
                |   "%left" token_list token_decs
                |   "%right" token_list token_decs
                |   "%nonassoc" token_list token_decs
                |       (* empty *)
                ;

token_list      :   token_list TOKEN
                |   TOKEN
                ;

(* end allows for the code to be placed at the end of the spec *)

end             :       (* empty *)
                |   "%%"
                |   "%%" ML_CODE
                ;

grammar         :   prods
                ;
```

```
prods           :   prods prod
                |       (* empty *)
                ;

prod            :   NONTERMINAL ":" clauses
                ;

clauses         :   action_clause ";"
                |   action_clause "|" clauses
                ;

action_clause   :   clause ACTION
                |   clause  (* no action *)
                ;

clause          :       (* empty clause such as this one *)
                |   clause symbol
                |   clause symbol "%prec" TOKEN
                ;

symbol          :   NONTERMINAL
                |   TOKEN
                ;

%%
```

8. Maintenance and Distribution

SML-Yacc will be maintained and distributed by the Systems Engineering Division, Informatics Department at the Rutherford Appleton Laboratories. An improved version, which runs more efficiently and with extra features such as improved error handling is under consideration for future development.

A lexical analyser generator similar to the Unix utility **lex** is under development, using the SML-Yacc system. This system, known as **Yalag** (Yet Another Lexical Analyser Generator), should be compatible with the SML-Yacc system, and should make the construction of the lexical analysis function **lex** more straightforward. It is hoped that it can be distributed with the SML-Yacc system.

Problems, errors and suggestions for future improvements should be directed to:

B. M. Matthews,
Systems Engineering Division,
Informatics Department,
Rutherford Appleton Laboratory,
Chilton,
Nr Didcot,
OXON OX11 0QX.

e-mail : bmm@uk.ac.rl.inf

Acknowledgements

This work was carried out under the Alvey Software Engineering support program at Rutherford Appleton Laboratory. Our thanks to Dr David Duce and Dr Rob Witty for their support and encouragement during this project. Mikael Hedlund was involved in its early stages. Bernard Sufrin gave helpful advice. Thanks to John Kalmus for proof reading this document.

References

AhoUll77. Alfred V Aho and Jeffrey D Ullman, *Principles of Compiler Design*, Addison-Wesley (1977).

AhoSetUll86. Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley (1986).

HopUll79. John E Hopcroft and Jeffrey D Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley (1979).

John75. Stephen C Johnson, ''Yacc – Yet another Compiler-Compiler,'' Bell laboratories Technical Report (1975).

Pey83. Simon L Peyton-Jones, ''Yacc in SASL,'' INDRA Working Paper, No. 1533 (1983).

Roth87. Nick Rothwell, ''Parsing Utilities for Standard ML,'' Draft note, University of Edinburgh (1987).

Suf88. Bernard Sufrin, ''mllama - ML Translator Generator,'' Unpublished Notes (1988).

Trem85. Jean-Paul Tremblay and Paul G Sorenson, *The Theory and Practise of Compiler Writing*, McGraw-Hill (1985).

Udd88. Goran O Uddeborg, ''A Functional Parser Generator,'' Programming Methodology Group report, Chalmers University, Goteborg, Sweden (1987).

Wik87. Ake Wikström, *Functional Programming using Standard ML*, Prentice-Hall (1987).