Fachrichtung 6.2 – Informatik
Naturwissenschaftlich-Technische Fakultät I
– Mathematik und Informatik –
Universität des Saarlandes

Jacke – A Parser-Generator Tool for Standard ML

Jan Schwinghammer
jan@ps.uni-sb.de
Programming Systems Lab
Saarland University, Germany

Fortgeschrittenen-Praktikum

Angefertigt unter der Leitung von
Prof. Dr. Gert Smolka

Vorgeschlagen und betreut von
Leif Kornstaedt und Andreas Rossberg

February 28, 2002

Jacke – A Parser-Generator Tool for Standard ML

Jan Schwinghammer
jan@ps.uni-sb.de
Programming Systems Lab
Saarland University, Germany

Introduction

In this report, design, usage and implementation of the Jacke tool are described. Jacke is a parser generator for Standard ML [4], generating LALR parsers from grammar specifications which can be embedded into ordinary ML code. It features compositional syntax to specify grammar productions, closely resembling ML notation.

Outline. In the following section, a brief overview of the functionality of Jacke is given. Next, usage of the tool is explained in detail, and an example is presented. Finally the implementation is described, though it is not necessary to read this part in order to use Jacke. The present version does not contain all the features of the original proposal [5], possible extensions are discussed at the end of the next section on page 3.

Acknowledgment. The implementation of Jacke has been supervised by Leif Kornstaedt and Andreas Rossberg. The software makes use of parts of ML-Yacc [6], which is copyrighted by David R. Tarditi and Andrew W. Appel.

Functionality of the Tool

Jacke is a parser-generator tool, similar to the Yacc tools, but with SML consistent syntax. One of the main design goals of Jacke is simplicity. It features the following properties:

- grammar definition and parser declarations embeddable into SML source code
- extended, "compositional" Backus-Naur-Form for the specification of grammar productions and semantic actions, similar to the use of parser combinators
- ability to deal with multiple start symbols

Contents

Introduction **1**

Functionality of the Tool **1**
Overview 2
Limitations and possible Extensions 3

Detailed Description **4**
Syntax and Semantics 4
Example 6

Implementation **8**
Lexer and Parser 9
Normalization 9
Generation of LALR Tables 10
Code Generation 10
Performance 11

Conclusion **11**

References **12**

Like ML-Yacc [6], Jacke generates (SML code for) parsers for LALR languages [1]. Context-free grammars are used to specify the syntax of the language to be parsed. Such a grammar consists of disjoint sets T and NT, a set P, and a *start symbol* $s \in NT$. Elements in T are called *terminals*, or *tokens*, elements in NT are called *nonterminals*. A *symbol* is either a terminal or nonterminal, the set of symbols is denoted by $S = T \uplus NT$. P contains the *productions*, or *rules*, of the grammar, which are of the form $(A, \sigma_1 \ldots \sigma_k)$, where $A \in NT$ and $\sigma_i \in S$. A string $\alpha_1 \ldots \alpha_{k-1}\sigma_1 \ldots \sigma_m \alpha_{k+1} \ldots \alpha_n$ of symbols is *derivable* from the string $\alpha_1 \ldots \alpha_{k-1} A \alpha_{k+1} \ldots \alpha_n$ if there is a production $(A, \sigma_1 \ldots \sigma_m) \in P$. The *language* defined by the grammar is the set of strings of tokens derivable from s in zero or more steps.

Just as ML-Yacc, Jacke uses an attribute grammar scheme with synthesized attributes. Each symbol in the grammar is associated with such an attribute (i.e., an ML value), and each production $(A, \sigma_1 \ldots \sigma_k)$ has a semantic action associated with it. Parsers perform bottom-up, left-to-right evaluations of parse trees, and so the attribute associated with A can be computed from those associated with the σ_i, using the corresponding semantic action. The result of a successful parse is the attribute associated with the start symbol s of the grammar.

In contrast to ML-Yacc, the specification of the parser is embeddable into ordinary SML code. Moreover, we think ML-Yacc is too restrictive in several aspects by requiring that

- nonterminals must be explicitly defined, and a type needs to be annotated for each nonterminal
- the type of positions must be specified and by
- only allowing for the definition of grammars with a single start symbol

The dependance of generated parsers and lexers is complicated by the fact that ML-Yacc generates "token functions", which a lexer must call instead of simply supplying the tokens. Moreover, the output of ML-Yacc is a whole collection of SML functors which must be instantiated before the actual parser is produced.

Jacke addresses all of these issues. Nonterminals need not be defined explicity, and in particular it is *not* necessary to annotate types for the associated semantic attributes. In parsers produced by Jacke, all symbols carry left and right position information of arbitrary, user-definable type. This information is available to semantic actions and is also used in syntactic error messages, but the actual parser is polymorphic in this type. Further, several parsers may be defined in a single file, corresponding to the definition of multiple start symbols. Finally, a simplification is obtained by providing a token datatype. The lexer only needs to yield the sequence of these tokens, along with their respective positions.

Overview

Token declarations in Jacke look very much like SML datatypes. Jacke translates them to a corresponding datatype `token`. *Associativity declarations* mimic SML infix directives. For example,

```
token PLUS | MINUS | TIMES | NUM of int

assocl TIMES
assocl PLUS MINUS
```

declares four tokens, where the `NUM` token carries an integer as additional semantic attribute. Further, `TIMES`, `PLUS` and `MINUS` are declared to be associative to the left, and `TIMES` is given highest precedence.

The actual productions of the grammar (which are usually mutually recursive) take the form of *rule bindings*, resembling SML function definitions. They may include a type annotation on the left hand side, documenting the type of the intended return value. The right hand sides of rule bindings, commonly just a *sequence* ('`,`') of symbols, are extended to *BNF expressions*. These are

result transformations (‘`=>`’) and *alternatives* (‘`|`’), which are compositional in structure, i.e. they may be nested arbitrarily. To continue the example of arithmetic expressions, consider the rule bindings

```
rule exp :int =
    NUM
  | n1 as exp, oper, n2 as exp  => (oper (n1,n2))
and oper =
    PLUS   => (op+)
  | MINUS  => (op-)
```

Conceptionally, each BNF expression returns a value, synthesized from the attributes of the subexpressions. For token identifiers without token attributes, this is `()`. For tokens with attributes (those having an ‘`of`’ part, such as `NUM` above), this is just their attribute. Sequential expressions return the tuple of their component results, alternative expressions return the value of the rule that was successful. The combinator ‘`=>`’ allows results to be transformed into a different value, using SML expressions.

The ‘`as`’ expression allows a subexpression result to be named, so that it is available in following result transformations. As short hand, expressions consisting of a symbol identifier *symid* only (such as `oper` above) stand for the expression *symid* `as` *symid*. This often allows for more compact rules and is consistent with the scheme used to name results of subexpressions in New Jersey's ML-Yacc. Finally, there is the expression ‘`skip`’, representing ϵ. The value associated with ‘`skip`’ is `()`.

Rules may be given explicit precedences, allowing to deal with conflicts occurring during generation of the LALR tables. For example, one would write

```
rule exp =
    n1 as exp, oper, n2 as exp  => (oper (n1,n2))  prec PLUS
```

when implementing an evaluator for arithmetic expressions.

The start symbols of the grammar are declared using *parser bindings*. Only these declarations actually generate SML functions. A parser binding just denotes the corresponding start symbol, and conversely, only these declarations specify start symbols of the grammar. For example,

```
parser eval = exp
```

will generate the actual parsing function `eval` for the above grammar with start symbol `exp`, taking a *lexer* as input to produce the parse result. The type of the lexer, α `lexer`, is `unit` $\rightarrow$ `token option` $\times \alpha \times \alpha$, where α is the type of positions.

Limitations and possible Extensions

Although it is most likely sufficient for practical purposes, having no *scoping* for the input to Jacke is not satisfying from an aesthetical point of view. However, the advantage of this approach is that only few assumptions on the syntax of SML are made apart from some basic keywords, string literals, comments, and bracket structure. In particular, the tool should work well with most extensions of Standard ML.

A useful extension is to also allow productions to contain *inherited attributes*. This was specified by Andreas Rossberg in the original proposal [5] as follows. A rule may contain an optional argument pattern which represents the inherited atribute. To pass these attributes, each occurrence of the left hand side symbol of this rule in the grammar must be followed by a corresponding *attribute application*. Thus, one might specify a rule such as

```
rule exp E =
    LET, ID, EQUAL, n1 as exp E, IN, n2 as exp (insert(E,ID,n1)) => (n2)
```

using environments (which are extended during the parse) for the evaluation of an expression syntax with `let`. Used as a start symbol, a parameterised symbol would create a parsing function with an additional (curried) argument.

Further, *rule templates* are conceivable, i.e. generic rules parameterised over BNF expressions. Such templates are available in GUMP, see [3].

Detailed Description

In this section, usage of Jacke to develop LALR parsers for SML is described in detail.

Syntax and Semantics of Specifications

The syntax of specifications extends ML syntax [4], as defined by the following grammar.

dec ::=	...	extends declarations by
|	`token` *tokbind*	token declaration
|	`assocl` $symid_1 \ldots symid_n$	left associative symbols ($n \geq 1$)
|	`assocr` $symid_1 \ldots symid_n$	right associative symbols ($n \geq 1$)
|	`nonassoc` $symid_1 \ldots symid_n$	non associative symbols ($n \geq 1$)
|	`rule` *rulebind*	production rule
|	`parser` *parsbind*	start symbol

tokbind ::= *symid* ⟨`of` *type*⟩ ⟨`|` *tokbind*⟩

rulebind ::= *symid* ⟨`:` *type*⟩ `=` *bnfexp* ⟨`and` *rulebind*⟩

parsbind ::= *vid* ⟨`:` *type*⟩ `=` *symid*

bnfexp ::=	`skip`	ϵ
|	*symid*	terminal or non-terminal symbol
|	`(` *bnfexp* `)`	
|	*vid* `as` *bnfexp*	naming
|	*bnfexp* `,` *bnfexp*	sequence
|	*bnfexp* `prec` *symid*	precedence annotation
|	*bnfexp* `=>` `(` *exp* `)`	result transformation
|	*bnfexp* `|` *bnfexp*	alternative

From a given input file *file*, Jacke produces a file *file*`.out` which contains the toplevel SML declarations, datatype declarations for the specified tokens, and finally, for each specified parser, SML code which implements the corresponding parsing function.

Toplevel declarations are simply copied in the output. Each *token declaration*

`token` $symid_1$ ⟨`of` $type_1$⟩ | ... | $symid_n$ ⟨`of` $type_n$⟩

is translated to a corresponding datatype declaration

`datatype token =` $symid_1$ ⟨`of` $type_1$⟩ | ... | $symid_n$ ⟨`of` $type_n$⟩

If several differing token declarations are found in the input file, Jacke will issue a warning message and regard the last declaration as the one specifying the terminal symbols of the grammar. Nevertheless it makes sense to have more than one token declaration in the source file, e.g. when writing code for both a signature and a matching structure.

From all the *associativity declarations* and *rule bindings* found in the input file, a single structure containing the generated LALR tables and some further declarations is generated[1]. This means that these declarations are global, and in particular there is no scoping of these declarations possible. Consequently, multiple associativity declarations for a single token lead to an error message. However, precedences are assigned to terminal symbols according to the *order* in which associativity declarations occur in the input.

Nonterminals of the grammar are implicitly defined by the rule bindings. An error message is generated if identifiers occur in some right hand side that are neither tokens nor appear on the left hand side of any rule. Start symbols are defined through *parser declarations*. Each parser declaration will produce SML code implementing a parser for the language described by the grammar with that specific start symbol. This generated function will take a lexer as input to produce the result of the parse, i.e. the semantic attribute of the start symbol.

Rule bindings associate each nonterminal A with an BNF expression. BNF expressions provide the notation for specifying the productions with left hand side A. At the same time, they describe the synthesized attribute to be associated with A. Internally, Jacke rewrites these BNF expressions into the form of ordinary grammar productions.

`skip` is just the empty sequence, with associated attribute `()`. *symid* stands for the symbol *symid* and returns the value associated with *symid*. Similarly, *vid* `as` *bnfexp* stands for *bnfexp*, returning its attribute. As short hand, expressions consisting of a symbol identifier *symid* only stand for the expression *symid* `as` *symid*. This *naming scheme* is subject to the condition that names in a sequence of BNF expressions are distinct. The value associated with a sequence $bnfexp_1$ `,` $bnfexp_2$ is the tuple of the values of the elements.

In the result transformation *bnfexp* `=> (` *exp* `)`, the SML code *exp* may use identifiers which are bound by *bnfexp*. These *environments* are defined in the following way.

$$\begin{aligned}
env(symid) &= \{symid\} \\
env(vid\ \texttt{as}\ bnfexp) &= \{vid\} \\
env(\texttt{(}bnfexp\texttt{)}) &= env(bnfexp) \\
env(bnfexp_1\ \texttt{,}\ bnfexp_2) &= (env(bnfexp_1) \cup env(bnfexp_2)) - (env(bnfexp_1) \cap env(bnfexp_2))
\end{aligned}$$

For the remaining cases, $env(bnfexp) = \emptyset$ is just the empty set, i.e. no identifiers are bound. This could be relaxed to some extent, for example by defining $env(bnfexp_1\ \texttt{|}\ bnfexp_2) = env(bnfexp_1) \cap env(bnfexp_2)$. For simplicity, this is not done. Apart from the attributes of identifiers *id* in $env(bnfexp)$, the semantic action *exp* may use position information that is made available by the lexer. Access is provided via the identifiers *id*`left` and *id*`right`, respectively[2].

An alternative $bnfexp_1$ `|` $bnfexp_2$ denotes a rule for each case. More exactly, it stands for a fresh nonterminal A and an extended grammar, with the two new rule bindings

$$\texttt{rule}\ A = bnfexp_1 \quad \text{and} \quad \texttt{rule}\ A = bnfexp_2$$

Finally, Jacke allows to assign precedences to rules by the BNF expression

$$bnfexp\ \texttt{prec}\ symid$$

Here, the precedence of *symid* is assigned to *bnfexp*. Precedences and associativities are used to resolve shift/reduce conflicts which may occur during generation of LALR tables.

[1] The structure is output at the position of the first rule binding in the source.

[2] *id*`.left`, or `left(`*id*`)`, certainly would be preferrable, but this would also mean writing *id*`.result` or `result(`*id*`)`, respectively, whenever referring to the semantic value of *id*.

This *precedence scheme* is the same as used in Yacc, and is in fact directly inherited from ML-Yacc: each rule is assigned the precedence of its rightmost terminal. A shift/reduce conflict can be resolved if both rule and lookahead symbol have precedences. The result is a shift if the symbol has higher precedence, and a reduction if the rule has the higher precedence. If the respective precedences are equal, associativity information for the lookahead symbol is used to resolve the conflict.

In the case of an left associative token, the reduction is chosen, in the case of a right associative token the conflict is resolved by shifting. Finally, if the token is declared nonassociative, or else if token or rule have not been assigned a precedence, a warning is produced and the shift is chosen.

Reduce/reduce conflicts are resolved as in ML-Yacc by always choosing the rule appearing first in the input. However, this should be utilised **with great care** only, since *new* rules are generated by Jacke when rewriting BNF expressions.

Jacke generates code that makes the following assumptions on the scope. The rule and parser declarations must be in the scope of the token declaration. In the current implementation, the generated parsers will call an error function `parseError` if a *syntactical error* is detected during the parse. This function takes as input the current position:

`parseError` : *position type* → *parse result type*

The generated code will assume this function in the environment. Note that this design decision is necessary since the parser does not know the type of positions, but usually the user should know about the position where the error has been detected.

Observe that `skip` is not available as name to semantic actions. However, this can always be simulated by writing

name `as skip => (...)`

In particular, it is possible to obtain position information in this way. Also note that (type) errors in the generated code, in particular those in the functions implementing the semantic actions, are most likely caused by mismatching types of the rule attributes. To facilitate debugging of parser specifications, the corresponding positions in the source are printed in the output. However, due to the normalization with its generation of additional rules, this is not always possible.

Example – A Parser for Pure Lambda Calculus

The following sample code defines an SML structure containing an abstract syntax data type, and a second structure containing a parser for terms of the lambda calculus. The result of a parse will be an abstract syntax tree for the input token sequence.

```
structure AbstractSyntax =
struct
    datatype abs_syn =
        Var of string | App of abs_syn * abs_syn | Lam of string * abs_syn
end

structure Parse =
struct
    structure A = AbstractSyntax

    token VAR of string | LPAR | RPAR | DOT | LAM
```

```
    assocl VAR LPAR LAM

    rule exp =
        VAR                   => (A.Var VAR)
      | e1 as exp, e2 as exp  => (A.App (e1,e2)) prec VAR
      | LAM, VAR, DOT, exp    => (A.Lam (VAR,exp))
      | LPAR, exp, RPAR       => (exp)

    parser parse = exp
end
```

From this code, Jacke will generate a file containing the following SML objects.

```
(* just copied *)
structure AbstractSyntax :
  sig
    datatype abs_syn
      = App of abs_syn * abs_syn | Lam of string * abs_syn | Var of string
  end

structure Parse :
  sig
    structure A : <sig>

    (* generated from Jacke declarations *)
    structure JackeDeclarationsStruct__1013197552 : <sig>

    (* generated from token declaration *)
    datatype token = DOT | LAM | LPAR | RPAR | VAR of string

    (* generated from parser declaration *)
    val parse : (unit -> token option * 'a * 'a) -> AbstractSyntax.abs_syn
  end
```

The additional structure contains the LALR tables as well as some internal definitions. The generated code for the `parse` function makes use of a previously defined error function in the environment, which takes a position as input. In the case of syntactical errors occuring during a parse, this function is called with argument the position where the error has been detected. Moreover, the code for the parsing engine for the generated LR parsers is in the files `LrTable.sml` and `LrParser.sml`, which must be in the environment before compiling the generated file.

All that is left is to provide a lexing function *nextToken* to the parser,

$$\mathit{nextToken} : \texttt{unit} \rightarrow \texttt{token option} \times \mathit{position\ type} \times \mathit{position\ type}$$

When applied to `()`, this function should yield the next token of the input, if any. If, for example, using the ML-Lex lexer generator, the following would be a suitable specification file.

```
structure P = Parse
type posType = int
type lexresult = P.token option * posType * posType
val linenum = ref 1
val error = fn x => TextIO.print(x ^ "\n")
val eof = fn () => (NONE, !linenum, !linenum)
%%
```

```
%structure Lexer
alpha=[A-Za-z];
digit=[0-9];
ws = [\ \t];
%%
\n       => (linenum:=(!linenum+1); lex());
{ws}+    => (lex());
"\\"     => (SOME P.LAM,!linenum,!linenum);
{alpha}+ => (SOME (P.VAR yytext), !linenum,!linenum);
"("      => (SOME P.LPAR, !linenum, !linenum);
")"      => (SOME P.RPAR, !linenum, !linenum);
"."      => (SOME P.DOT, !linenum, !linenum);
.        => (error ("lexer: ignoring bad character "^yytext); lex());
```

This will produce a structure `Lexer` containing the function `makeLexer`. From this, it is easily possible to assemble a function taking a filename as input and parsing this file. This is demonstrated by the following piece of code.

```
val fileParser = fn f =>
        let val ins = TextIO.openIn f
            val lexer = Lexer.makeLexer (fn n => TextIO.inputN (ins,n))
            val result = Parse.parse lexer
            val _ = TextIO.closeIn ins
        in
            result
        end
        handle e => (TextIO.closeIn ins; raise e)
```

Consult the ML-Lex documentation [2] for further details on how to generate lexical analysers with this particular tool.

Implementation

In this section, the implementation of the tool and design decisions made for this implementation are described at a high level. The implementation itself can be roughly divided into four parts, as sketched in Figure 1: lexer and parser for Jacke itself, normalization phase, LALR table generation, and code generation. The code for the *parsing engine* for the generated LR Parsers is in the files

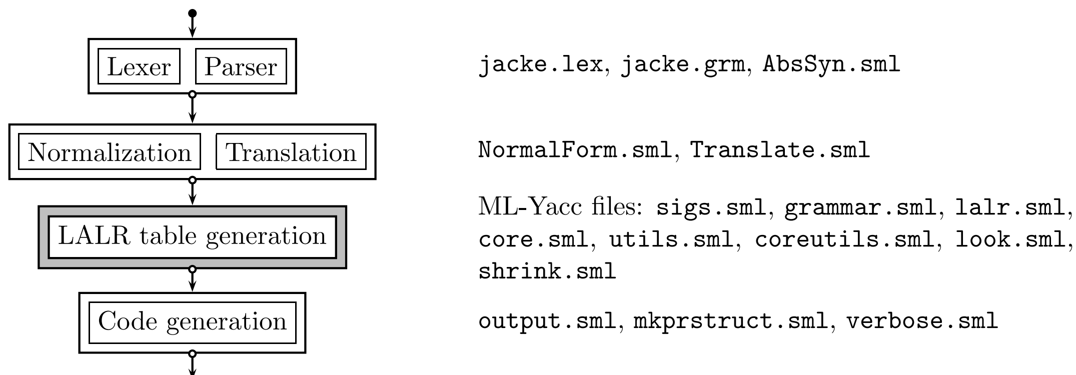

Figure 1: Overview of the implementation. Source files for the different phases.

`LrTable.sml` and `LrParser.sml`, which must be in the environment before compiling the generated file *file*`.out`. The present implementation of Jacke reuses code of the ML-Yacc parser generator [6]. In particular, the generation of the tables is taken without any modifications. In the following, the phases are described.

Lexer and Parser

Lexer and parser for Jacke are generated from `jacke.lex` and `jacke.grm` using ML-Lex and ML-Yacc, respectively. Abstract syntax datatypes are defined in `AbsSyn.sml`. This file also provides a semantic analysis, checking if

- all symbols in associativity declarations are terminal symbols (i.e., defined as token)
- terminal symbols (i.e., symbols occuring in the token declaration) and nonterminal symbols (i.e., symbols occuring on the left hand side of some rule) are disjoint sets
- all identifiers occuring on the right hand sides of rules are defined as symbols, i.e. either terminal or nonterminal
- all start symbols, i.e. identifiers appearing in parser declarations, are defined as (nonterminal) symbols

Moreover, a warning is issued if several *differing* token declarations are found in the source file. For the subsequent processing of the input, only the positions of rule declarations are retained (more exactly, result transformations). This information is eventually output with the corresponding semantic action clause to facilitate debugging of grammar specifications.

Normalization

Jacke allows rules to be defined in a generalized BNF notation. Expressions given in this form are transformed to productions in "standard" BNF, i.e., to the form

$$\texttt{rule}\ r\ \texttt{=}\ n_1\ \texttt{as}\ S_1\texttt{,...,}\ n_k\ \texttt{as}\ S_k\ \texttt{=> (}\ \mathit{exp}\ \texttt{)}\quad \langle\ \texttt{prec}\ \mathit{symid}\ \rangle$$

where $k > 0$, or

$$\texttt{rule}\ r\ \texttt{= skip => ()}$$

Moreover, a new start symbol S and terminals `EOP` and $\texttt{D}_i$ are introduced[3], together with a rule

$$\texttt{rule}\ S\ \texttt{=}\ \texttt{D}_i\texttt{,}\ S_i\texttt{, EOP => (}\ S_i\ \texttt{)}$$

for each declared parser with respective start symbol S_i.

Given such a *normalized* grammar, terminals and nonterminals are internally coded as integers. Starting with first the $\texttt{D}_i$, followed by `EOP` and then the user-defined tokens, terminals are numbered $0, 1, 2, \ldots$ Similarly, nonterminals are numbered, beginning with the newly introduced start symbol.

Associativity and precedences are coded according to the internal ML-Yacc scheme: precedences are multiples p of 3, where greater integer means higher precedence. A number p stands for left-associative, $p+1$ for non-associativity and precedences of rules, and $p+2$ means right-associative.

With the help of these codings, an "ML-Yacc grammar", i.e., a value of the `Grammar.grammar` type of ML-Yacc, is constructed from the normalized grammar productions. This is used as input to the table generation modules of ML-Yacc, which use associativities and precedences as above to resolve shift/reduce conflicts.

[3] In fact, these symbols have fresh names, to avoid name clashes with user-defined symbols.

Generation of LALR Tables

The code for the generation of LALR tables from a (normalized) grammar description is taken from ML-Yacc. It produces action and goto tables which can be output using the functions in the structure `mkPrStruct`. These use the internal integer representation of symbols and associativities/precedences as well as ML-Yacc's defaults to resolve conflicts occurring during table generation.

More specifically, `MakeLrTable.mkTable` takes a `Grammar.grammar` value and returns a value of type `LrTable.table`, along with a list of errors and a verbose description of the table.

Code Generation

Code for the output file *file*`.out` is generated by the "main routine", the function `Output.output`. Apart from ML toplevel declarations, which are just copied, and the generated `token` datatype, the output file will contain code for

- LALR tables
- code for each individual parser, taking a lexical analyzer as input and "wrapping" it so that it works with multiple start symbols
- a datatype of *semantic values*, to hold the attributes associated with symbols on the parse stack, and a semantic actions function

To deal with *multiple start symbols*, the following standard trick is applied: the terminals are (internally) extended by a set of dummy tokens, one for each start symbol. To select the right one when parsing, the appropriate token must be the first symbol supplied by the lexer stream. This is achieved by wrapping the lexing function in the code for each parser. Likewise, an end-of-parse token is introduced and appended to the original lexing stream to indicate its end, as soon as the original lexing function yields a result `(NONE,`p_1`,`p_2`)`.

The *semantic values datatype* holds the attributes associated with symbols on the parse stack. Generally, if k start symbols are defined, this datatype declaration takes the form

$$
\begin{array}{lll}
\texttt{datatype } (ty_1,\ldots,ty_m)\texttt{ svalue} & = & \texttt{VOID} \mid \texttt{EOP of unit}\rightarrow\texttt{unit} \\
 & \mid & \texttt{D}_1 \texttt{ of unit}\rightarrow\texttt{unit} \mid \ldots \mid \texttt{D}_k \texttt{ of unit}\rightarrow\texttt{unit} \\
 & \mid & tok_1 \texttt{ of unit}\rightarrow tokenty_1 \mid \ldots \mid tok_l \texttt{ of unit}\rightarrow tokenty_l \\
 & \mid & rule_1 \texttt{ of unit}\rightarrow ty_1 \mid \ldots \mid rule_m \texttt{ of unit}\rightarrow ty_m
\end{array}
$$

where tok_i are the tokens, $rule_j$ the nonterminals, and $tokenty_i$ is either the specified attribute associated with tok_i or `unit` if none is specified. Note how *type variables* $ty_1,\ldots,ty_m$ are used to leave the typing of rules implicit in the user definitions, contrasting ML-Yacc where explicit type annotations for each nonterminal are required. In fact, this would not even work in our case since Jacke introduces new rules during normalization.

The *semantics action* function implements the semantic actions associated with each rule of the normalized grammar. For a rule

$$\texttt{rule } r \texttt{ = } n_1 \texttt{ as } S_1\texttt{,}\ldots\texttt{, } n_k \texttt{ as } S_k \texttt{ => (} exp \texttt{)}$$

the corresponding clause of function `actions` will match the k topmost values (and their positions) on the parse stack, and bind them to variables $n_1,\ldots,n_k$ to make them accessible in *code*. Next, the result is the left hand symbol of the rule along with the result of the semantic action, which is pushed back onto the stack by the actual parser when reducing by this rule. Thus, the clause takes

the form

```
(_, (SValue.S_k(n_k), n_kleft, n_kright))
 :: ...::(_, (SValue.S_1(n_1), n_1left, n_1right)) ::stack
   => let val result =
          let val n_1 = n_1() ··· val n_k = n_k()
          in r(fn () => (exp)) end
      in (sym_r, (result, n_1left, n_kright), stack) end
```

where sym_r is the integer coding for the symbol r. From this code fragment it can also be seen that token names must not have constructor status in this place: considering t as abbreviation for

t `as` t

the pattern $t(t)$ will appear in the above. For this reason, there is a structure generated that defines a function for every symbol name, before the token datatype declaration appears. This structure is then opened when defining the semantic action function. Moreover, the datatype of semantic values is defined in a structure `SValue`, so the pattern has in fact the form `SValue.`$t(t)$.

Performance: Delayed vs. Immediate Evaluation of Semantic Actions

Besides the above clause for the semantic action function, there is a second possibility: evaluation of semantic actions could be delayed until the completion of a successful parse. This is the default with ML-Yacc since it allows for error-recovery, where it must be possible to "undo" semantic actions. The clause for the the semantic action function will in this case take the form

```
(_, (SValue.S_k(n_k), n_kleft, n_kright))
 :: ...::(_, (SValue.S_1(n_1), n_1left, n_1right)) ::stack
   => let val result =
          r(fn () =>
              let val n_1 = n_1() ··· val n_k = n_k()
              in exp end )
      in (sym_r, (result, n_1left, n_kright), stack) end
```

Note that this code generation scheme of delayed evaluation can be quite easily adapted to deal with inherited attributes, as outlined on page 3.

Clearly, this comes at the price of higher memory consumption and impairs efficiency. The numbers in Figure 2 indicate this quite clearly. The numbers are (user) time in seconds, obtained as total sum of running times of five runs. The considered parser was a simple expression parser, evaluating the expressions in the semantic actions. The input consisted of 100000 (400000) tokens in each run. In the case of SML New Jersey and Moscow ML, the improvement in running time (and memory usage) are quite obvious. The anomalous performance of Alice needs further investigation.

	delayed	*immediate*
SML NJ	41 (182)	27 (117)
Alice	195 (389)	197 (760)
Moscow ML	120 (`out of memory`)	59 (255)

Figure 2: Comparing running times of delayed and immediate evaluation of semantic actions.

Conclusion

We presented Jacke, a parser-generator tool for SML. It features SML-consistent syntax, and definitions of grammars are embeddable into SML source code. The syntax for the specification of

grammar productions is compositional and extends Backus-Naur-Form. Jacke improves on the comparable tool ML-Yacc by generating parsers which are significantly easier to use. It also allows to omit some unnecessary specifications required by ML-Yacc, such as the explicit declaration of nonterminals and the types of their associated semantic attributes.

The specification language is quite flexible and several extensions of the tool are conceivable. For example, productions making use of inherited attributes could be included. Further, the tool could be extended by rule templates, i.e. generic rules parameterised over BNF expressions. These parameters could even be higher-order, allowing parameters to be rule templates themselves. Such templates are available in Gump [3].

Finally, the performance of parsers generated by Jacke and ML-Yacc, repectively, should be compared. This might be interesting since Jacke's treatment of tokens as datatype and the resulting wrapping of lexers leads to the evaluation of a case statement, for each call to the lexer. The size of this case expression depends linearly on the number of tokens. On the other hand, the complexity of parsers generated by ML-Yacc is largely due to a construction made in order to avoid this. Moreover, a reasonable ML implementation will optimise the case statement to a table lookup.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] A. W. Appel, J. S. Mattson, and D. R. Tarditi. A lexical analyzer generator for Standard ML, Version 1.6, Oct 1994. Available at `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/`.

[3] L. Kornstaedt. Definition und Implementierung eines Front-End-Generators für Oz. Master's thesis, Fachbereich Informatik, Universität Kaiserslautern, and Programming Systems Lab, Universität des Saarlandes, Sept. 1996.

[4] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[5] A. Rossberg. Proposal for Jacke and Hose, June 1999. Personal communication.

[6] D. R. Tarditi and A. W. Appel. ML-Yacc User's Manual, Version 2.4, April 2000. Available at `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/`.