Preprocessors

J. Grosch

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

Preprocessors

Josef Grosch

Aug. 4, 1992

Report No. 24

Gesellschaft für Mathematik und Datenverarbeitung mbH
Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1
D-7500 Karlsruhe

Preprocessors

1. Introduction

This manual describes the features and the usage of two kinds of preprocessors contained in the Karlsruhe Toolbox for Compiler Construction. The preprocessors *cg -xz* and *rpp* derive a parser specification and most of a scanner specification from an attribute grammar. The preprocessors *l2r*, *y2l*, and *r2l* convert input for *lex* and *yacc* into specifications for *rex* and *lalr* and vice versa. All preprocessors work as filter programs reading input from *standard input* and writing output to *standard output*. Some preprocessors can read from a file specified as argument, too.

2. Specification of Scanner and Parser with an Attribute Grammar

Writing specifications for the scanner generator *rex* [Gro87] and the parser generator *lalr* [GrV88] directly in the tool specific language is a practicable method. However, it has some disadvantages. Most of the tokens are specified twice and their internal representation or code, respectively, has to be selected and kept consistent, manually. Access to attributes using the yacc-style $i construct (see e.g. [GrV88]) is less readable and error-prone in case of changes. The following solution avoids these disadvantages.

Instead of using the tool specific language for *rex* and *lalr* directly, a language of higher level is used. It replaces the $i construct by named attributes and describes a complete parser and most of a scanner in one document. Two preprocessors transform such a specification into the languages directly understood by *rex* and *lalr*. Fig. 1 shows the data flow using arrows between boxes and circles. Boxes represent files or file types and circles represent tools or preprocessors. The intermediate file named *Scanner.rpp* by default contains the part of the scanner specification that can be extracted from the parser specification. Table 1 gives the meaning of the file types used in Fig. 1 and this manual.

Table 1: Meaning of file types

file type	meaning
.pars	scanner and parser specification (including S-attribution)
.scan	rest of scanner specification
.rpp	intermediate data for completion of scanner in .scan
.rex	scanner specification understood by *rex*
.lalr	parser specification understood by *lalr*
.ell	input for *ell* (= input for *lalr* with EBNF constructs [GrV88])
.lex	input for *lex*
.yacc	input for *yacc*
Source	generated module (or linked from library *reuse*)
Scanner	generated module
Parser	generated module
Errors	generated module (or linked from library *reuse*)

The formalism used to describe parsers (.pars) is an extension of the input language for the tools *ast* [Gro91] and *ag* [Gro89] (see section 2.1.). This leads to a rather uniform set of input languages for most of the tools and simplifies their use. The preprocessor *cg -xz* transforms a parser specification in *ag* notation into one in *lalr* notation and extracts most of a scanner specification. The parser specification in *lalr* notation is written on a file named <Parser>.lalr. <Parser> is substituted by the name of the parser module which defaults to *Parser*. The extracted scanner specification is written to a file named <Scanner>.rpp. <Scanner> is

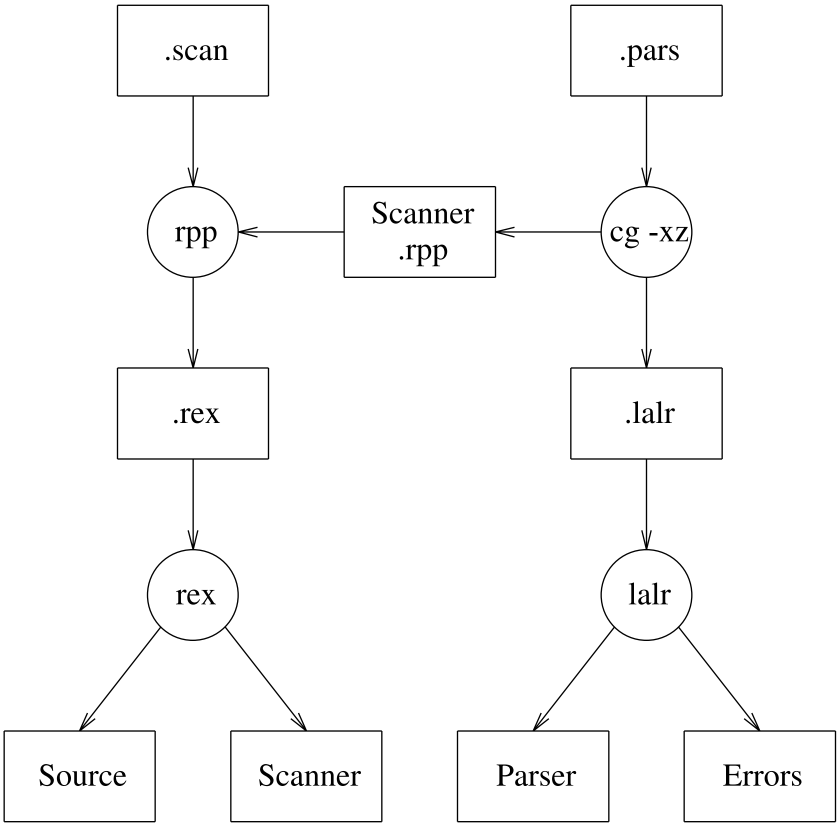

Fig. 1: Data flow during scanner and parser generation

substituted by the name of the scanner module which defaults to *Scanner*. The rest of the scanner specification must be written in the language directly understood by *rex*. It has to contain the part of a scanner specification that can not be derived automatically. This part is usually rather small and comprises the description of user-defined tokens like identifiers and numbers, comments, and the computation of attributes for the tokens. A few insertion points are marked to tell the preprocessor *rpp* where to include the generated parts (see section 2.2.).

2.1. Parser Specification

The input language of *ast* and *ag* can be used to generate a parser. The details of this language can be found in the manuals for these tools [Gro89, Gro91]. The reader should be familiar with these documents because the current manual describes primarily the extensions necessary for parser generation.

The language can describe concrete as well as abstract syntaxes. Nonterminal, terminal, and abstract symbols or node types are distinguished by the definition characters '=', ':', and ':=', respectively, and have to be declared by default. However, the option j of *cg -xz* allows undeclared terminal symbols and declares them implicitly. In any case, terminal symbols with attributes have to be declared explicitly.

Note, the following names are reserved for keywords and can not be used for grammar symbols. Unfortunately, some of them occur frequently as keywords in languages that are being defined. They should be turned into strings by surrounding apostrophes:

```
BEGIN          CLOSE      DECLARE   DEMAND    END
EVAL           EXPORT     FOR       FUNCTION  GLOBAL
IGNORE         IMPORT     IN        INH       INHERITED
INPUT          LEFT       LOCAL     MODULE    NONE
OUT            OUTPUT     PARSER    PREC      PROPERTY
REMOTE         REV        REVERSE   RIGHT     RULE
SCANNER        SELECT     STACK     SUBUNIT   SYN
SYNTHESIZED    THREAD     TREE      VIEW      VIRTUAL
VOID
```

The right-hand sides of node types without extensions are interpreted as right-hand sides of grammar rules (see e. g. Assign, Call0, Call, and If in the example below). The children of the right-hand side form a sequence of terminal and nonterminal symbols as usual. The names of those node types serve as rule names. If a symbol occurs several times on one right-hand side, it has to be preceded by different selector names (see e. g. the rule named *If*). Attributes in brackets are not interpreted as grammar symbols but as attribute declarations representing values to be evaluated during parsing.

Not every name of a node type is interpreted as nonterminal or terminal symbol. Only those node types that are used (referenced) on some right-hand side and the first node type which is regarded as start symbol are treated as grammar symbols. If a node type is defined as nonterminal then all associated extensions become alternative right-hand sides for this nonterminal symbol. If a node type is defined as terminal it remains a terminal symbol. If a node type is not defined and option j is not set an error message is issued. If option j is set then undefined node types are implicitly defined as terminals.

The grammar needs not to be reduced. This means it may contain superfluous terminal and nonterminal symbols. Symbols are superfluous if they are not referenced from any rule. Those symbols are simply ignored and reported as a warning.

Example: input of cg -xz

```
Stat           = <
   Assign      = Adr ':=' Expr .
   Calls       = Ident <
      Call0    = .
      Call     = '(' Actuals ')' .
   > .
   If          = IF Expr THEN Then: Stats ELSE Else: Stats 'END' .
> .
Expr           = <
   Plus        = Lop: Expr '+' Rop: Expr .
   Times       = Lop: Expr '*' Rop: Expr .
   '()'        = '(' Expr ')' .
   Adr         = <
      Name     = Ident .
      Index    = Adr '[' Expr ']' .
   > .
> .
```

Example: output of cg -xz

```
Stat : Adr ':=' Expr .
Stat : Ident .
Stat : Ident '(' Actuals ')' .
Stat : IF Expr THEN Stats ELSE Stats 'END' .

Expr : Expr '+' Expr .
Expr : Expr '*' Expr .
Expr : '(' Expr ')' .
Expr : Ident .
Expr : Adr '[' Expr ']' .

Adr  : Ident .
Adr  : Adr '[' Expr ']' .
```

The rule names and the selector names on the right-hand sides disappear (e. g. If). The extension formalism is expanded (e. g. Calls and Adr) – it is not mapped to chain rules. The expansion includes the inheritance of children and attributes (e. g. Calls). All node types that are used somewhere become nonterminal symbols (e. g. Expr and Adr).

The rule names of non-referenced node types may be omitted. They are necessary for example if modules are used in order to refer from an attribute computation to a grammar rule.

Example without rule names:

```
Stat   = -> [Tree: tTree] <
       = Adr ':=' Expr          { Tree := mAssign (Adr:Tree, Expr:Tree);     } .
       = Ident '(' Actuals ')'  { Tree := mCall (Ident:Ident, Actuals:Tree); } .
> .

Ident : [Ident: tIdent] .
```

Example with rule names:

```
Stat      = <
   Assign = Adr ':=' Expr .
   Call   = Ident '(' Actuals ')' .
> .

Ident     : [Ident: tIdent] .

MODULE Tree

DECLARE Stat = -> [Tree: tTree] .

Assign    = { Tree := mAssign (Adr:Tree, Expr:Tree);          } .
Call      = { Tree := mCall (Ident:Ident, Actuals:Tree);      } .

END Tree
```

Attribute declarations and attribute computations are written exactly as for *ast* and *ag*. Attribute computations may be placed everywhere within right-hand sides, not only at the end. These computations are executed from left to right as parsing proceeds. They may only make use of those attributes that have already been computed before or to the left, respectively. The extension or inheritance mechanism for right-hand sides, attribute declarations, and attribute computations is available. The default computations (copy rules) for synthesized attributes are available, too. A specification may be separated into several modules. There could for example be modules for the concrete syntax, for the attribute declarations, and for the attribute computations. It is even possible to distribute the mentioned kinds of information into several modules.

Attribute declarations and attribute computations with named attributes replace the explicit declaration of the type *tParsAttribute* and the $i construct. The attribute declarations are transformed automatically into a declaration of the type *tParsAttribute*. The attribute computations in the new style are more readable and robust against changes. The given attribute computations are checked for completeness and whether the resulting attribute grammar obeys the SAG property. Only attribute grammars with synthesized attributes can be evaluated during parsing.

Every terminal has a predefined attribute named *Position* of type *tPosition*. This type is a user defined struct (record) type and has to contain at least the members *Line* and *Column*. This attribute describes the source coordinates of every token and it is computed automatically by *rex* generated scanners. The values of all other attributes of terminals have to be supplied by the scanner by user specified computations. Still, attribute computations for those attributes (except *Position*) have to be specified in the parser specification, too. They are used to generate the procedure *ErrorAttribute*. This procedure is called by the parser whenever tokens are inserted in order to repair syntax errors. The procedure and therefore the mentioned attribute computations have to supply default values for the attributes of tokens.

The right-hand side of a node type or a grammar rule, respectively, may contain actions to be executed during parsing. These actions may be placed at the end of the right-hand side or anywhere between the right-hand side elements. In any case these actions are executed from left to right according to the progress of parsing. The syntax of the actions is the one defined for the attribute computations of *ag*. It is assumed that most of the actions will compute attributes. Actions which are not attribute computations are possible but have to be written as some kind of CHECK statement:

```
=> statement ;   or    => { statement_sequence } ;
```

The following extensions of the language of *ast* and *ag* are used for parser generation, only. A grammar may be optionally headed by names for the modules to be generated:

```
SCANNER Name PARSER Name
```

The first identifier specifies the module name of the scanner to be used by the parser. The second identifier specifies a name which is used to derive the names of the parsing module, the parsing routine, the parse tables, etc. If the names are missing they default to *Scanner* and *Parser*. In this document we refer to these names by <Scanner> and <Parser>. The parser name may be followed by a set of target code sections which is copied unchecked and unchanged to the input of the parser generator or the parser module, respectively:

```
SCANNER Name
PARSER  Name
IMPORT { target_code }
EXPORT { target_code }
GLOBAL { target_code }
LOCAL  { target_code }
BEGIN  { target_code }
CLOSE  { target_code }
```

The precedence and associativity for operator tokens can be specified after the keyword PREC using LEFT, RIGHT, and NONE for left-, right-, and no associativity. Each group headed by one of the latter three keywords introduces a new group of tokens with increasing precedence. The precedence and associativity information is copied unchanged to the parser generator.

Example:

```
PREC LEFT MONOP
     NONE SEQ
     LEFT '+' '-'
     LEFT '*' '/' MOD
```

The precedence and associativity information is usually propagated implicitly to the grammar rules by taking it from the right-most token in a rule. Rules without an operator token can get the precedence and associativity from an operator token by adding a PREC clause at the end of a right-hand side.

Example:

```
Expr = <
     = Expr Expr PREC SEQ .   /* explicit  prec. + assoc. of SEQ          */
     = '-'  Expr PREC MONOP . /* overwrite prec. + assoc. of '-' by MONOP */
     = Expr '+' Expr .        /* implicit  prec. + assoc. of '+'          */
     = Expr '-' Expr .        /* implicit  prec. + assoc. of '-'          */
> .
```

Tokens or terminal symbols are mapped automatically to integer numbers to be used as internal representation. This mapping can be overwritten by explicitly giving codes for terminals.

Example:

```
Ident    :   [Ident: tIdent] .   /* implicitly coded      */
IntConst : 5 [Value: int   ] .   /* explicitly coded as 5 */
```

The attribute declarations for terminals are turned into a declaration of the type *tScanAttribute*. The scanner communicates attribute values of terminals to the parser using a global variable called *Attribute* which is of this type. This type is a union type (variant record) with one member (variant) for each terminal with attributes. The names of the terminals are taken for the names of these members (variants). However, this leads to problems if the terminals are named by strings or by keywords of the implementation language. Therefore terminals may have two names. The second one is used as member name in the type *tScanAttribute*. The predefined attribute *Position* mentioned above is always included in this type. Assignments of attribute values in the scanner therefore have to use two selections to describe an attribute:

```
Attribute.<selector name>.<attribute name> = ...
```

Example:

definition of terminals including attributes and member selectors in the parser specification

```
Ident          : [Ident: tIdent] .   /* selector name: Ident   */
':=' sAssign   : [Ident: tIdent] .   /* selector name: sAssign */
TRUE sTRUE     : [Ident: tIdent] .   /* selector name: sTRUE   */
'..'           : [Ident: tIdent] .   /* selector name: yy17    */
```

access of attributes in the scanner specification

```
Attribute.Ident.Ident
Attribute.sAssign.Ident
Attribute.sTRUE.Ident
Attribute.yy17.Ident
```

access of attributes in the parser specification (at node directly)

```
Ident                     Position
```

access of attributes in the parser specification (from a child node)

```
Ident:Ident               Ident:Position
':=':Ident                ':=':Position
TRUE:Ident                TRUE:Position
'..':Ident                '..':Position
```

The preprocessor for the parser specification is part of the program *cg*. It is called by the following command:

cg -xzvjc [*Parser*.pars]

The input is read either from the file given as argument or from standard input if the argument is missing. The output is written to a file named <Parser>.lalr. The program is controlled by the following options:

x	generate scanner specification for rex
z	generate parser specification for lalr
v	omit actions in the generated parser specification
j	allow undefined node types; define implicitly as terminals
c	generate C source code (default is Modula-2)

Appendix 1 of the *ag* manual [Gro89] contains the complete formal syntax understood by the program *cg*. Appendix 2 of this manual shows a parser specification for the example language MiniLAX. It separates context-free grammar and attribute computations in two modules. The attribute computations in the module *Tree* use one attribute also called *Tree* and describe the construction of an abstract syntax tree by calling functions generated by *ast*. The implementation language is C.

2.2. Scanner Specification

The scanner specification has to contain only those parts that can not be extracted automatically from the parser specification. This is as already mentioned above the description of user-defined tokens like identifiers and numbers, comments, and the computation of attributes for the tokens. The formalism to describe this fragmentary scanner specification (.scan) is the input language of rex [Gro87]. It may contain three insertion points which instruct the preprocessor *rpp* (rex preprocessor) to include certain generated parts. Moreover, tokens in return statements can be denoted by the same strings or identifiers as in the parser specification.

```
INSERT tScanAttribute
```

in the EXPORT section is replaced by the generated declaration of the type *tScanAttribute* and the head or external declaration of the procedure *ErrorAttribute*.

```
INSERT ErrorAttribute
```

in the GLOBAL section is replaced by the body of the generated procedure *ErrorAttribute*.

The third insertion point lies in the RULE section and has the following syntax (only the brackets `[ ]` are used as meta characters and denote optional parts):

```
INSERT RULES [CASE-INSENSITIVE] [[NOT] #<start_states>#] [{ <target_code> }]
```

It is replaced by as many rules as there are tokens extracted automatically from the parser specification. Every rule has the following format:

```
NOT # <start_states> # <token> : { <target_code> return <code>; }
```

The start states including the keyword NOT and the target code are optional and are copied to the generated rule as indicated. If CASE-INSENSITIVE is specified, the regular expressions for the tokens are constructed to match lower as well as upper case letters. Note, only rules for tokens without explicitly declared attributes are constructed automatically.

Within a rule, return (or RETURN) statements are used to report the internal code of a recognized token. The expression of those statements can be any expression of the implementation language or a string or an identifier used in the parser specification. The latter are replaced by their internal representation.

Example:

```
return 5;
return Ident;
return ':=';
```

The program *rpp* is called by the following command:

rpp [*Scanner*.rpp] < *Scanner*.scan > *Scanner*.rex

The fragmentary scanner specification is read from standard input. The scanner specification extracted from the parser specification is read from a file given as argument. This argument is optional and defaults to *Scanner.rpp*. The scanner specification understood by *rex* is written on standard output. The basename *Scanner* in the command line above is usually substituted by the name of the scanner module.

Appendix 1 contains a scanner specification for the example language MiniLAX. It uses C as implementation language.

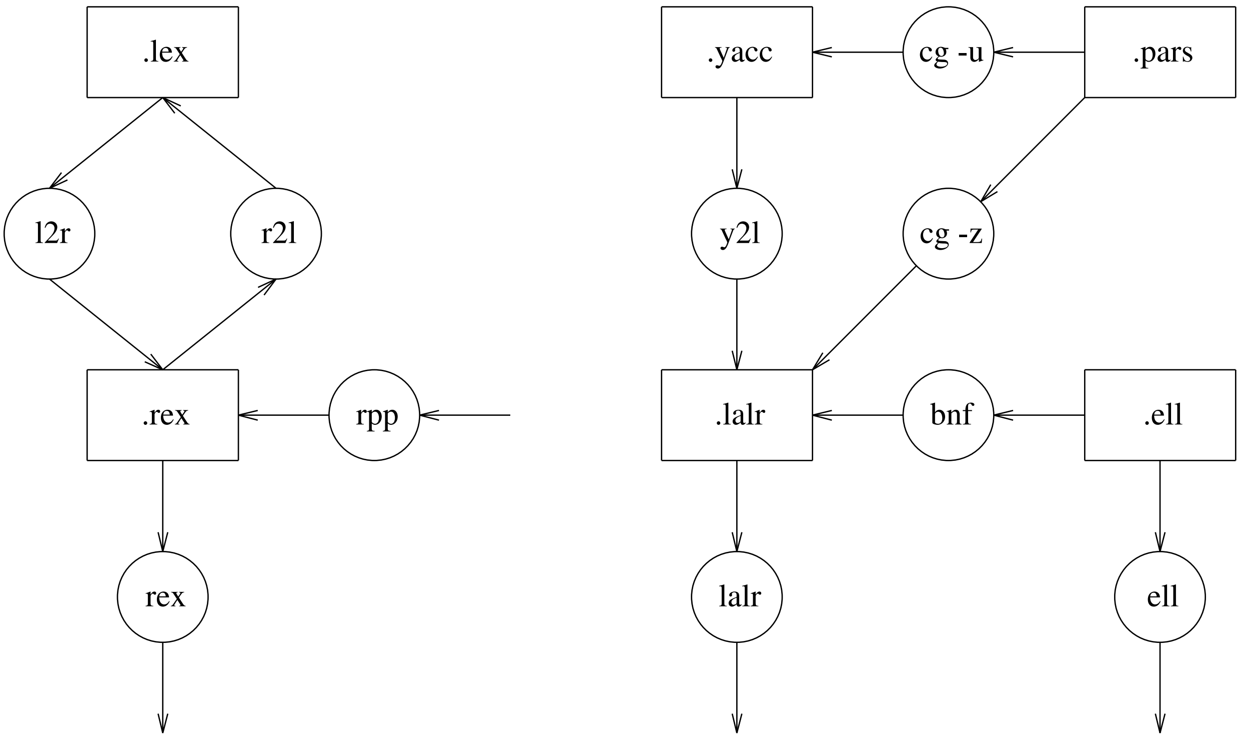

Fig. 2: Conversion programs for scanner and parser specifications

3. Conversion of Scanner and Parser Specifications

3.1. Input Languages

The input languages of *rex* and *lalr* have been designed to be as readable as possible. Although they contain the same information as inputs for *lex* [Les75] and *yacc* [Joh75] they are syntactically incompatible. Several conversion programs allow the transformation of input for *rex* and *lalr* to input for *lex* and *yacc* or vice versa. Fig.2 shows the possible conversions for scanner and parser specifications. Table 2 lists the existing filter programs and the types of their input and output files.

The option '-v' instructs *y2l* and *cg* to omit the semantic actions in the produced output. The following restrictions or problems are known to exist because they can not be mapped to the

Table 2: Filters for input conversion

filter	input	output
l2r	.lex	.rex
r2l	.rex	.lex
y2l	.yacc	.lalr
cg -u	.pars	.yacc
cg -z	.pars	.lalr
bnf	.ell	.lalr

target tool:

l2r

- yymore
- REJECT
- yywrap (redirection can be done with *rex*, but differently)
- %T (specification of the character set is not possible)

r2l

- yyPrevious
- EOF (specifies actions to be performed upon reaching end of file)

y2l

- The conversion of token definitions may not be completely automatic.
- If scanning depends on information collected by the parser then parsers generated by *yacc* and *lalr* may behave differently because they do not agree upon the order or timing, respectively, of the execution of read (shift) and reduce actions.

bnf

- The attribute computations for *ell* and *lalr* are different and are not converted.

3.2. Interfaces

The interfaces of scanners and parsers generated by *lex/yacc* and *rex/lalr* are incompatible. The differences are primarily caused by different names for the external (exported) objects. Table 3 lists the interface objects. The interfaces of the scanners and parsers generated by *rex* and *lalr* can be switched from the default as listed in Table 3 to an approximation of the *lex* and *yacc* interfaces. This is controlled by *cpp* commands:

```
# define lex_interface
```

in the EXPORT section of a *rex* specification selects a lex-style interface for the scanner.

```
# define lex_interface
```

in the EXPORT section of a *lalr* specification tells the parser to use a lex-style interface for the scanner.

```
# define yacc_interface
```

in the EXPORT section of a *lalr* specification selects a yacc-style interface for the parser.

The output of the preprocessors *l2r* and *y2l* automatically selects lex- and yacc-style interfaces. The following problems are known, currently:

Table 3: Interfaces of generated scanners and parsers

object	yacc/lex	lalr/rex
parse routine	int yyparse ();	int Parser ();
stack size	YYMAXDEPTH	yyInitStackSize
attribute type	YYSTYPE	tParsAttribute
global attribute	YYSTYPE yylval;	tScanAttribute Attribute;
position type		typedef struct { short Line, Column; ... } tPosition;
attribute type		typedef struct { tPosition Position; ... } tScanAttribute;
scanner routine	int yylex ();	int GetToken ();
error repair		void ErrorAttribute ();
line number	int yylineno;	*member* Attribute.Position.Line
token buffer	char yytext [];	
token length	int yyleng;	

- The output of *l2r* provides the matched string in the array *yytext* to be used in action statements. This is done by calling the procedure *Getword*. However, many actions do not need *yytext*. Deleting superfluous calls of *Getword* will make the scanner significantly faster.
- Access to the line counter *yylineno* has to be replaced by access to

```
Attribute.Position.Line
```

- If both, scanner and parser specification have been converted by *l2r* and *y2l* in order to be fed into *rex* and *lalr*, the two preprocessor statements which define *lex_interface* should be deleted in order to select the standard interface. This offers more comfort with respect to the information about the source position.

Acknowledgements

The preprocessor *bnf* has been implemented by Bertram Vielsack. The preprocessors *y2l*, *l2r*, *rpp*, and *cg -xz* have been implemented by Thomas Müller.

References

[Gro87] J. Grosch, Rex - A Scanner Generator, Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1987.

[GrV88] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Compiler Generation Report No. 8, GMD Forschungsstelle an der Universität Karlsruhe, Apr. 1988.

[Gro89] J. Grosch, Ag - An Attribute Evaluator Generator, Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Gro91] J. Grosch, Ast - A Generator for Abstract Syntax Trees, Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1991.

[Joh75] S. C. Johnson, Yacc — Yet Another Compiler-Compiler, Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975.

[Les75] M. E. Lesk, LEX — A Lexical Analyzer Generator, Computing Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

Appendix 1: Scanner Specification for MiniLAX

```
EXPORT  {
# include "Idents.h"
# include "Positions.h"

INSERT tScanAttribute
}

GLOBAL  {
# include <math.h>
# include "Memory.h"
# include "StringMem.h"
# include "Idents.h"
# include "Errors.h"

INSERT ErrorAttribute
}

LOCAL   { char Word [256]; }

DEFAULT {
MessageI ("illegal character", xxError, Attribute.Position, xxCharacter, TokenPtr);
}

EOF     {
   if (yyStartState == Comment)
      Message ("unclosed comment", xxError, Attribute.Position);
}

DEFINE  digit   = {0-9} .
        letter  = {a-z A-Z} .

START   Comment

RULE
          "(*"  :- {yyStart (Comment);}
#Comment# "*)"  :- {yyStart (STD);}
#Comment# "*" | - {*\t\n} + :- {}

#STD# digit +   : {(void) GetWord (Word);
                   Attribute.IntegerConst.Integer = atoi (Word);
                   return IntegerConst;}

#STD# digit * "." digit + (E {+\-} ? digit +) ?
                : {(void) GetWord (Word);
                   Attribute.RealConst.Real = atof ((char *) Word);
                   return RealConst;}

INSERT RULES #STD#

#STD# letter (letter | digit) *
                : {Attribute.Ident.Ident = MakeIdent (TokenPtr, TokenLength);
                   return Ident;}
```

Appendix 2: Parser Specification for MiniLAX

```
PARSER

GLOBAL {# include "Idents.h" }

BEGIN  { BeginScanner (); }

PREC    LEFT    '<'                             /* operator precedence  */
        LEFT    '+'
        LEFT    '*'
        LEFT    NOT

PROPERTY INPUT

RULE                                            /* concrete syntax      */

Prog            = PROGRAM Ident ';' 'DECLARE' Decls 'BEGIN' Stats 'END' '.' .
Decls           = <
   Decls1       = Decl .
   Decls2       = Decls ';' Decl .
> .
Decl            = <
   Var          = Ident ':' Type .
   Proc0        = PROCEDURE Ident ';' 'DECLARE' Decls 'BEGIN' Stats 'END' .
   Proc         = PROCEDURE Ident '(' Formals ')' ';'
                                      'DECLARE' Decls 'BEGIN' Stats 'END' .
> .
Formals         = <
   Formals1     = Formal .
   Formals2     = Formals ';' Formal .
> .
Formal          = <
   Value        = Ident ':' Type .
   Ref          = VAR Ident ':' Type .
> .
Type            = <
   Int          = INTEGER .
   Real         = REAL .
   Bool         = BOOLEAN .
   Array        = ARRAY '[' Lwb: IntegerConst '..' Upb: IntegerConst ']' OF Type .
> .
Stats           = <
   Stats1       = Stat .
   Stats2       = Stats ';' Stat .
> .
Stat            = <
   Assign       = Adr ':=' Expr .
   Call0        = Ident .
   Call         = Ident '(' Actuals ')' .
   If           = IF Expr THEN Then: Stats ELSE Else: Stats 'END' .
   While        = WHILE Expr DO Stats 'END' .
   Read         = READ '(' Adr ')' .
   Write        = WRITE '(' Expr ')' .
> .
Actuals         = <
   Expr1        = Expr .
   Expr2        = Actuals ',' Expr .
> .
```

```
Expr            = <
   Less         = Lop: Expr '<' Rop: Expr .
   Plus         = Lop: Expr '+' Rop: Expr .
   Times        = Lop: Expr '*' Rop: Expr .
   Not          = NOT Expr .
   '()'         = '(' Expr ')' .
   IConst       = IntegerConst .
   RConst       = RealConst .
   False        = FALSE .
   True         = TRUE .
   Adr          = <
      Name      = Ident .
      Index     = Adr '[' Expr ']' .
   > .
> .

                                        /* terminals (with attributes)  */

Ident           : [Ident: tIdent] { Ident       := NoIdent      ; } .
IntegerConst    : [Integer      ] { Integer     := 0            ; } .
RealConst       : [Real : float ] { Real        := 0.0          ; } .

MODULE Tree
                        /* import functions for tree construction       */
PARSER GLOBAL   {
# include "Tree.h"

tTree  nInteger, nReal, nBoolean;
}

BEGIN   {
   nInteger     = mInteger      ();
   nReal        = mReal         ();
   nBoolean     = mBoolean      ();
}
```

```
                        /* attributes for tree construction            */
DECLARE
  Decls Decl Formals Formal Type Stats Stat Actuals Expr = [Tree: tTree] .

RULE                    /* tree construction =                         */
                        /* mapping: concrete syntax -> abstract syntax  */

Prog    = { => { TreeRoot = mMiniLax (mProc (mNoDecl (), Ident:Ident,
                 Ident:Position, mNoFormal (), ReverseTree (Decls:Tree),
                 ReverseTree (Stats:Tree)));};                          } .
Decls1  = { Tree := {Decl:Tree->\Decl.Next = mNoDecl (); Tree = Decl:Tree;};    } .
Decls2  = { Tree := {Decl:Tree->\Decl.Next = Decls:Tree; Tree = Decl:Tree;};    } .
Var     = { Tree := mVar (NoTree, Ident:Ident, Ident:Position, mRef (Type:Tree));}.
Proc0   = { Tree := mProc (NoTree, Ident:Ident, Ident:Position, mNoFormal (),
                    ReverseTree (Decls:Tree), ReverseTree (Stats:Tree));        } .
Proc    = { Tree := mProc (NoTree, Ident:Ident, Ident:Position, ReverseTree
           (Formals:Tree), ReverseTree (Decls:Tree), ReverseTree (Stats:Tree)); } .
Formals1= { Tree := {Formal:Tree->\Formal.Next = mNoFormal ();
                    Tree = Formal:Tree;};                                       } .
Formals2= { Tree := {Formal:Tree->\Formal.Next = Formals:Tree;
                    Tree = Formal:Tree;};                                       } .
Value   = { Tree := mFormal (NoTree, Ident:Ident, Ident:Position,
                    mRef (Type:Tree));                                          } .
Ref     = { Tree := mFormal (NoTree, Ident:Ident, Ident:Position,
                    mRef (mRef (Type:Tree)));                                   } .
Int     = { Tree := nInteger;                                                   } .
Real    = { Tree := nReal;                                                      } .
Bool    = { Tree := nBoolean;                                                   } .
Array   = { Tree := mArray (Type:Tree, Lwb:Integer, Upb:Integer, Lwb:Position); } .
Stats1  = { Tree := {Stat:Tree->\Stat.Next = mNoStat (); Tree = Stat:Tree;};    } .
Stats2  = { Tree := {Stat:Tree->\Stat.Next = Stats:Tree; Tree = Stat:Tree;};    } .
Assign  = { Tree := mAssign (NoTree, Adr:Tree, Expr:Tree, ':=':Position);       } .
Call0   = { Tree := mCall (NoTree, mNoActual (Ident:Position), Ident:Ident,
                    Ident:Position);                                            } .
Call    = { Tree := mCall (NoTree, ReverseTree (Actuals:Tree), Ident:Ident,
                    Ident:Position);                                            } .
If      = { Tree := mIf (NoTree, Expr:Tree, ReverseTree (Then:Tree),
                    ReverseTree (Else:Tree));                                   } .
While   = { Tree := mWhile (NoTree, Expr:Tree, ReverseTree (Stats:Tree));       } .
Read    = { Tree := mRead (NoTree, Adr:Tree);                                   } .
Write   = { Tree := mWrite (NoTree, Expr:Tree);                                 } .
Expr1   = { Tree := mActual (mNoActual (Expr:Tree->\Expr.Pos), Expr:Tree);      } .
Expr2   = { Tree := mActual (Actuals:Tree, Expr:Tree);                          } .
Less    = { Tree := mBinary ('<':Position, Lop:Tree, Rop:Tree, Less);           } .
Plus    = { Tree := mBinary ('+':Position, Lop:Tree, Rop:Tree, Plus);           } .
Times   = { Tree := mBinary ('*':Position, Lop:Tree, Rop:Tree, Times);          } .
Not     = { Tree := mUnary (NOT:Position, Expr:Tree, Not);                      } .
IConst  = { Tree := mIntConst (IntegerConst:Position, IntegerConst:Integer);    } .
RConst  = { Tree := mRealConst (RealConst:Position, RealConst:Real);            } .
False   = { Tree := mBoolConst (FALSE:Position, false);                         } .
True    = { Tree := mBoolConst (TRUE:Position, true);                           } .
Name    = { Tree := mIdent (Ident:Position, Ident:Ident);                       } .
Index   = { Tree := mIndex ('[':Position, Adr:Tree, Expr:Tree);                 } .

END Tree
```

Contents