Toolbox Introduction

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Toolbox Introduction

Josef Grosch

Sept. 25, 2002

Document No. 25

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Breslauer Str. 64c
76139 Karlsruhe
Germany

Phone: +49-721-91537544
Fax: +49-721-91537543
Email: grosch@cocolab.com

Toolbox Introduction

Abstract

This document introduces into the usage of the Karlsruhe Toolbox for Compiler Construction. It should be read by those who effectively want to use the toolbox as first document. Those who want to learn about the toolbox and its contents in general are referred to the document "A Toolbox for Compiler Construction".

This document gives an overview about the documentation of the toolbox. It describes how the individual tools interact in order to generate a complete compiler. The general structure of a makefile to control the tools is discussed.

Table 1: Document Set

Filename	Title	Pages
intro	Toolbox Introduction	14
toolbox	A Tool Box for Compiler Construction	11
werkzeug	Werkzeuge für den Übersetzerbau	11
reuseC	Reusable Software - A Collection of C-Modules	31
reuseJava	Reusable Software - A Java Package	3
reuse	Reusable Software - A Collection of Modula-2-Modules	27
prepro	Preprocessors	17
rex	Rex - A Scanner Generator	58
scanex	Selected Examples of Scanner Specifications	21
scangen	Efficient Generation of Table-Driven Scanners	12
lark	Lark - An LALR(2) Parser Generator With Backtracking	90
ell-bnf	The Parser Generator Ell	28
lalr	Lalr - A Generator for Efficient Parsers	18
ell	Efficient and Comfortable Error Recovery in Recursive Descent Parsers	15
highspee	Generators for High-Speed Front-Ends	11
autogen	Automatische Generierung effizienter Compiler	10
ast	Ast - A Generator for Abstract Syntax Trees	74
toolsupp	Tool Support for Data Structures	12
ag	Ag - An Attribute Evaluator Generator	38
ooags	Object-Oriented Attribute Grammars	10
multiple	Multiple Inheritance in Object-Oriented Attribute Grammars	11
objects	Object-Orientation in the Cocktail Toolbox	6
puma	Puma - A Generator for the Transformation of Attributed Trees	39
trafo	Transformation of Attributed Trees Using Pattern Matching	15
wag	Efficient Evaluation of Well-Formed Attribute Grammars And Beyond	11
minilax	Specification of a MiniLAX-Interpreter	37
cookbook	Semantic Analysis Cookbook - Part 1: Declarations	93
faq	Cocktail FAQ - Frequently Asked Questions	9

1. Document Overview

The documentation of the Karlsruhe Toolbox for Compiler Construction consists of separate user's manuals for the individual tools and additional papers describing further aspects such as implementation details, examples, and applications. The documents are written in English. For a few of them there are German versions as well.

1.1. Format

The documents exist in several formats: Postscript, PDF, troff, and ASCII text. The documentation of the Java version of the library reuse is in the format HTML. The formats are distinguished by the file suffixes .ps, .pdf, .me, .txt, and The Postscript files are formatted for two paper sizes: letter format and A4 format. The troff files need processing with the program *pic* and the device independent version of troff called *ditroff* using me-macros or even better the GNU version of troff using commands such as

```
pic | tbl | eqn | ditroff -me
groff -pte -me | lpr
```

Depending on the format, the documents are located in the directories doc.ps, doca4.ps, doc.pdf, doc.me, doc.txt, or doc.html. The pictures are missing in the ASCII text format because of obvious limitations.

1.2. Documents

Table 1 lists the titles of the documents, the corresponding filenames (without suffix), and the number of pages.

1.3. Outlines

In the following the contents of every document is outlined shortly:

Toolbox Introduction
: An introduction for effective users of the toolbox which should be consulted first. It gives an overview about the document set and describes how the tools interact.

A Tool Box for Compiler Construction
: Explains the contents of the toolbox and the underlying design. The individual tools are sketched shortly and some application experiences are reported.

Werkzeuge für den Übersetzerbau
: A German version of the previous document.

Reusable Software - A Collection of Modula-2-Modules
: Describes a library of useful routines written in Modula-2 which are oriented towards compiler construction. The output of some tools has to be linked with this library.

Reusable Software - A Collection of C-Modules
: Describes a library of useful routines written in C which are oriented towards compiler construction. The output of some tools has to be linked with this library.

Reusable Software - A Java Package
: A brief description of a useful package of reusable classes written in Java is given. The package is oriented towards compiler construction.

Preprocessors
: Describes several preprocessors for the extraction of scanner specifications out of parser specifications and for the conversion of *lex*/*yacc* input to *rex*/*lark* input or vice versa. There are eight

preprocessors:

lpp -z	converts an attribute grammar to *lark* input
rpp	combines a grammar and a scanner specification to *rex* input
l2r	converts *lex* input to *rex* input
y2l	converts *yacc* input to *lark* input
r2l	converts *rex* input to *lex* input
lpp -u	converts an attribute grammar to *yacc* input
bnf	converts a grammar from EBNF to BNF
l2cg	converts *lark* input to an attribute grammar

Rex - A Scanner Generator
: The user's manual for the scanner generator *rex*.

Selected Examples of Scanner Specifications
: A collection of scanner specifications for *rex* dealing mostly with pathological cases.

Efficient Generation of Table-Driven Scanners
: Describes internals of the scanner generator *rex* like the so-called tunnel automaton and the linear time algorithm for constant regular expressions.

Lark - A LALR(2) Parser Generator With Backtracking
: The user's manual for the LR(1) and LALR(2) parser generator *lark*. This tool is able to parse non-LR(k) languages with its backtracking capability.

The Parser Generator Ell
: The user's manual for the LL(1) parser generator *ell* (recursive descent) and the grammar transformation tool *bnf*. It describes the input language common to both tools.

Lalr - A Generator for Efficient Parsers
: Describes details of the implementation of the parsers generated by *lark* and *lalr* (a predecessor of *lark*) and further outstanding features of these tools.

Efficient and Comfortable Error Recovery in Recursive Descent Parsers
: Describes the implementation of the parsers generated by *ell* especially with respect to automatic error recovery.

Generators for High-Speed Front-Ends
: A summary of the highlights of the scanner and parser generators and a comparison to *lex*/*yacc* and *flex*/*bison*.

Automatische Generierung effizienter Compiler
: A German version of the previous document.

Ast - A Generator for Abstract Syntax Trees
: The user's manual of *ast*, a tool supporting the definition and manipulation of attributed trees and graphs.

Tool Support for Data Structures
: Also describes *ast*, but less precise than the previous document.

Ag - An Attribute Evaluator Generator
: The user's manual of *ag*, a generator for evaluators of ordered attribute grammars (OAG).

Object-Oriented Attribute Grammars
: Also describes *ag*, like the previous document, with emphasis on the object-oriented features.

Multiple Inheritance in Object-Oriented Attribute Grammars
: Extends the object-oriented attribute grammars described in the previous document to multiple inheritance.

Object-Orientation in the Cocktail Toolbox
: Surveyes the aspects of object-orientation that appear at the specification level of several tools.

Puma - A Generator for the Transformation of Attributed Trees
: The user's manual of *puma*, a tool for the transformation of attributed trees which is based on pattern matching and unification.

Transformation of Attributed Trees Using Pattern Matching
: Also describes *puma* using a more introductory style and compares it to similar tools.

Specification of a MiniLAX-Interpreter
: The annotated input to generate a compiler for the example language MiniLAX.

Semantic Analysis Cookbook - Part 1: Declarations
: A collection of recipes for name analysis and symbol tables using attribute grammars.

Cocktail FAQ - Frequently Asked Questions
: This list answers some frequently asked questions primarily with respect to efficiency.

For readers intending to use the tools the following documents are of primary interest:

intro	Toolbox Introduction
toolbox	A Tool Box for Compiler Construction
prepro	Preprocessors
rex	Rex - A Scanner Generator
lark	Lark - A LALR(2) Parser Generator With Backtracking
ell-bnf	The Parser Generator Ell
ast	Ast - A Generator for Abstract Syntax Trees
ag	Ag - An Attribute Evaluator Generator
puma	Puma - A Generator for the Transformation of Attributed Trees

Of secondary interest might be:

reuseC	Reusable Software - A Collection of C-Modules
reuseJava	Reusable Software - A Java Package
reuse	Reusable Software - A Collection of Modula-2-Modules
scanex	Selected Examples of Scanner Specifications
faq	Cocktail FAQ - Frequently Asked Questions

The other documents either describe internals of the tools or are excerpts of the above.

2. Generating a Compiler

A compiler usually consists of several modules where every module handles a certain task. The toolbox gives very much freedom for the design of a compiler and supports various structures.

Figure 1 presents our preferred compiler structure. In the right column are the main modules that constitute a compiler. The left column contains the necessary specifications. In between there are the tools which are controlled by the specifications and which produce the modules. The arrows represent the data flow in part during generation time and in part during run time of the compiler.

In principle the compiler model works as follows: A scanner and a parser read the source, check the concrete syntax, and construct an abstract syntax tree. They may perform several normalizations, simplifications, or transformations in order to keep the abstract syntax relatively simple.

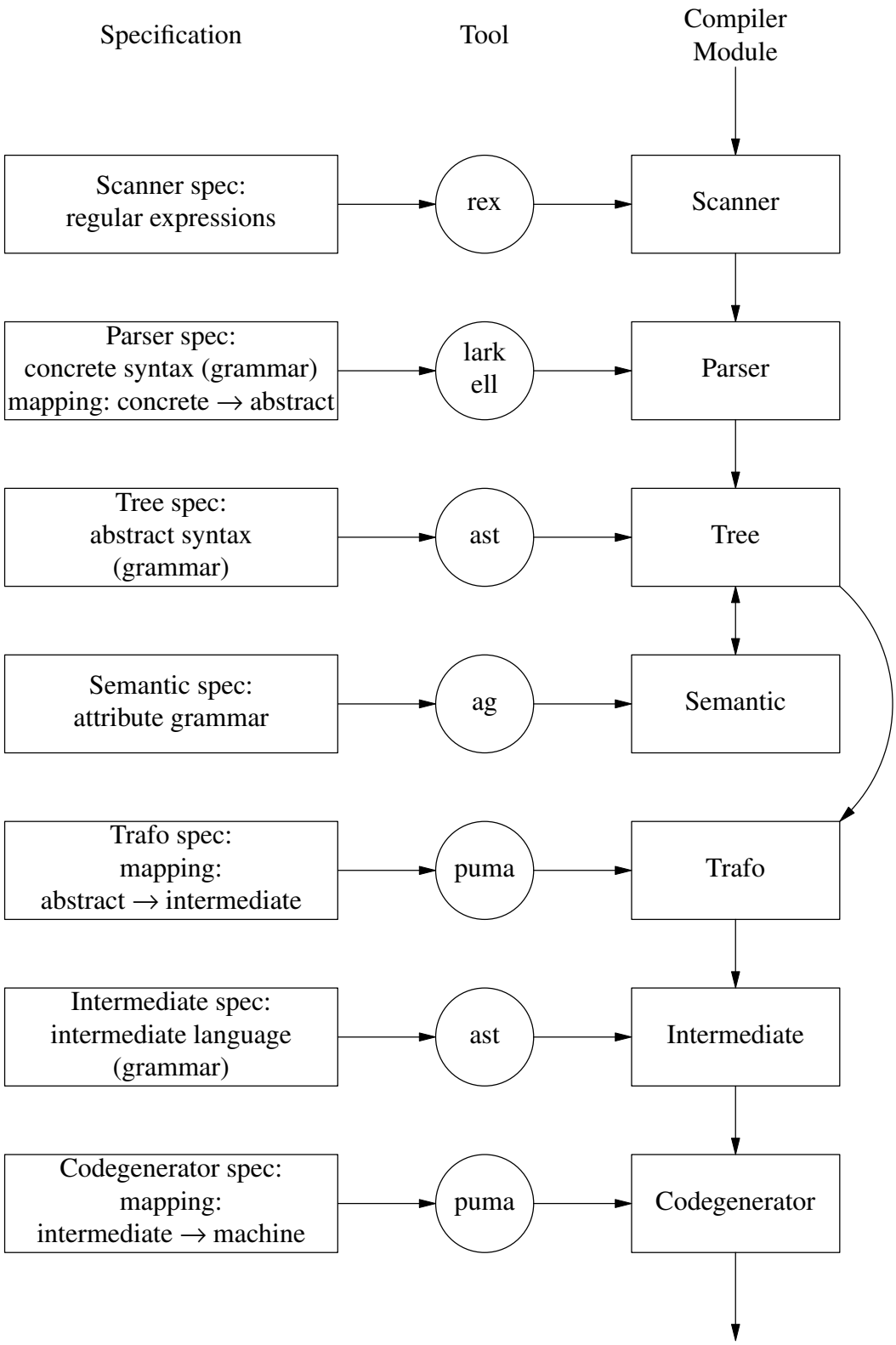

Fig. 1: Compiler Structure

Semantic analysis is performed on the abstract syntax tree. Optionally attributes for code generation

may be computed. Afterwards the abstract syntax tree is transformed into an intermediate representation. The latter is the input of the code generator which finally produces the machine code.

The picture in Figure 1 is relatively abstract by just listing the main tasks of a compiler. Every task is generated by a tool out of a separate specification which is oriented towards the problem at hand. The generation processes seem to be independent of each other.

For a real user a more closer look than the one of Figure 1 is necessary. Figure 2 describes the actual interaction among the tools. It describes the data flow starting from specifications and ending in an executable compiler. Boxes represent files, circles represent tools, and arrows show the data flow. The input specifications are contained in the files at the left-hand side. The tools generate modules containing source code in the implementation languages C or Modula-2. This modules are shown at the right-hand side. Every module conists of two files with the following suffixes:

implementation language C:

.h	header or interface file
.c	implementation part

implementation language Modula-2:

.md	definition module
.mi	implementation module

Files outside the left- and right-hand side columns contain intermediate data. The various kinds of information in the files are distinguished by different file types as explained in Table 2. The few dependencies between tools are shown by the data flow via intermediate files. These dependencies are explained in more detail in the next sections.

Table 2: File Types

Suffix	Meaning
.prs	scanner and parser specification (including S-attribution)
.scn	rest of scanner specification
.rpp	intermediate data: scanner description extracted from .prs
.rex	scanner specification understood by *rex*
.lrk	parser specification understood by *lark*
.ell	input for *ell* (= input for *lark* with EBNF constructs)
.cg	input for *ast* and *ag*
.pum	input for *puma*
.ST	intermediate data: storage assignment for attributes
.TS	intermediate data: description of attributed tree
.h	C source: header or interface file
.c	C source: implementation part
.md	Modula-2 source: definition module
.mi	Modula-2 source: implementation module
.exe	compiled and linked executable compiler

2.1. Scanning and Parsing

Three parser generators are contained in the toolbox. First, the user has to decide which one to use. I will not start arguing here in favour of one or the other grammar class. If I am asked, I

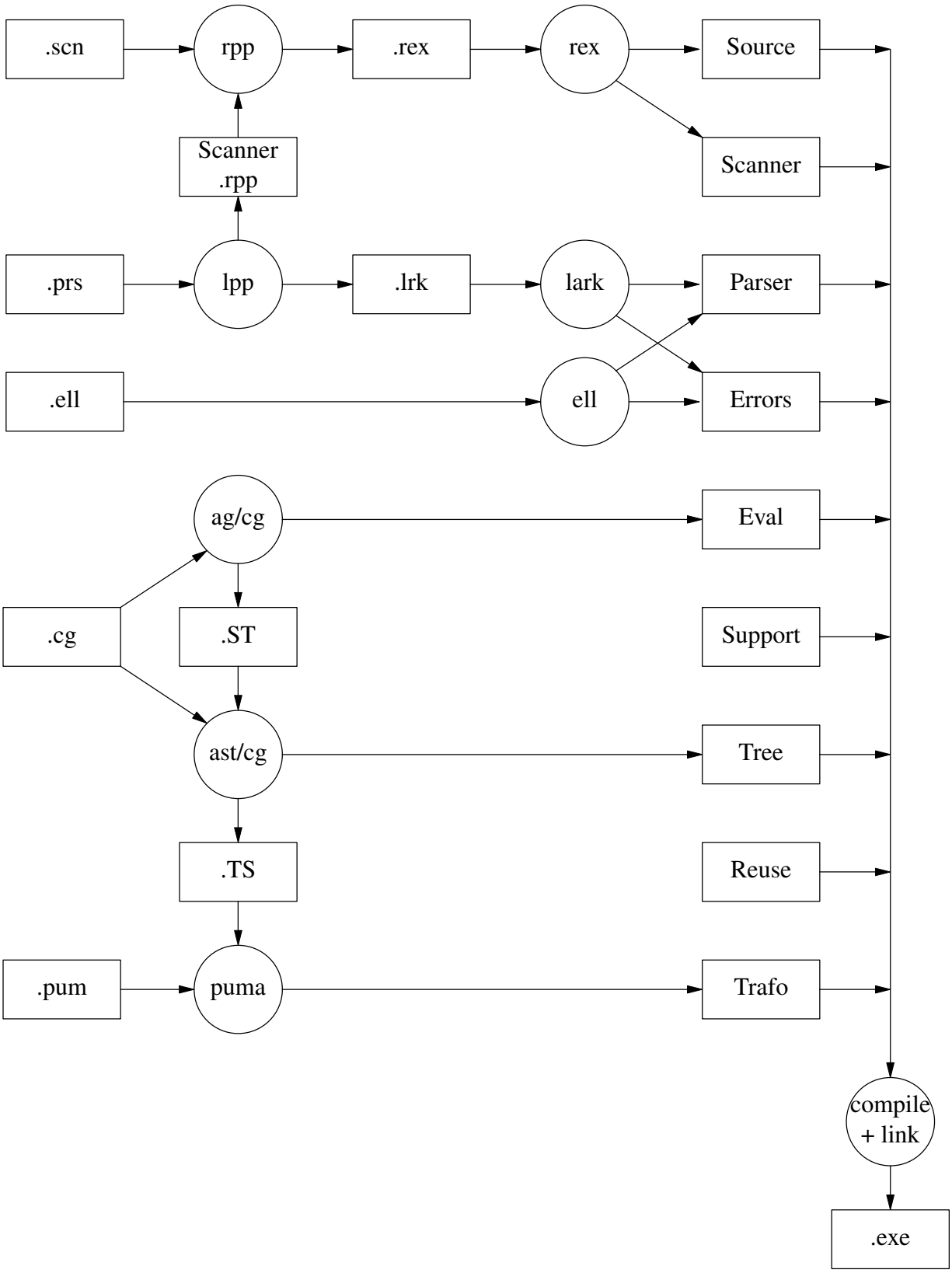

Fig. 2: Interaction and data flow among the tools

recommend to use LALR(1) grammars and the tool *lark*. The parser generators and their common input language (types .lrk and .ell) are documented in "The Parser Generator Ell" and "Lark - A LALR(2) Parser Generator With Backtracking". From the syntactic point of view all three tools understand almost the same input language. The only incompatibility concerns the different notation to access attributes. From the semantic point of view there are of course differences with respect to the grammar class and the kind of attribution evaluable during parsing. Whereas *lark* accepts LALR(2) grammars and is able to evaluate an S-attribution (SAG) during parsing, *ell* accepts LL(1) grammars and is able to evaluate an L-attribution (LAG). All, *lark* and *ell* generate a module with the default name *Parser* which serves as basename for the file name, too. This module name can be chosen freely using an appropriate directive in the input of the parser generators. Both parser generators can also supply a module called *Errors*. This is a simple prototype for handling error messages that just prints the messages. However, it is recommended to use the more comfortable module *Errors* from the library *reuse*. In simple cases, this module is just linked to the user's program. If modifications are necessary this module should be copied from the library along with its companion module *Position* into the user's directory. The module *Position* defines a data structure to describe source positions.

The scanner generator *rex* and its input language (type .rex) are documented in "Rex - A Scanner Generator". *rex* generates a module with the default name *Scanner* which serves as basename for the file name, too. *rex* can also generate a module called *Source* which isolates the task of providing input for the scanner. By default it reads from a file or from standard input. Again, it is recommended to use the module *Source* from the library *reuse*. In simple cases, this module is just linked to the user's program. If modifications are necessary or the module should provide input for a scanner with a name different to *Scanner* then this module must be requested from *rex*.

It is possible to combine several scanners and parsers either generated by *lark* or *ell* into one program as long as different module names are chosen.

If the parser generator *ell* is to be used, the inputs of *rex* and *ell* have to be specified in the languages directly understood by these tools (types .rex and .ell). If the LALR(2) parser generator *lark* is to be used, a more comfortable kind of input language is available. It is possible to extract most of a scanner specification out of a parser grammar. Therefore it is recommended to specify scanner and parser by two files of types .scn and .prs. Further advantageous of this approach are that concrete syntax, abstract syntax, and attribute grammar are written in one common language (types .prs and .cg) and that the attribution to be evaluated during parsing is written using named attributes. This attribution is checked for completeness and whether it obeys the SAG property. The language to describe concrete and abstract syntax is documented in: "Ast - A Generator for Abstract Syntax Trees". The addition of attribute declarations and attribute computations are documented in: "Ag - An Attribute Evaluator Generator". The use of this language especially as input for scanner and parser generation is documented in: "Preprocessors". This document also describes the preprocessors *lpp* and *rpp*. *lpp* converts input of type .prs into input directly understood by *lark* (type .lrk) and it extracts most of the scanner specification which is written on the intermediate file named Scanner.rpp. The *rex* preprocessor *rpp* merges this extracted scanner specification with additional parts provided by the user (type .scn) and produces input directly understood by *rex*. The language in files of type .scn is an extension of the input language of *rex*. These extensions are also documented in: "Preprocessors".

Table 3: Library Units Needed by Generated Modules

Tool	Module	C	Modula-2
rex	Source	rSystem	rSystem
rex	Scanner	rSystem, General, Source, DynArray, Position	rSystem, Checks, General, Strings, IO, Pack, DynArray, Position, Source
lark	Parser	rMemory, DynArray, Sets, Position, Errors,	Pack, DynArray, Sets, Strings, IO Position, Errors
ell	Parser	Errors	rSystem, Strings, Position, Errors
ell	Errors	rSystem, Sets, Idents, Position	rSystem, Strings, Idents, Sets, IO, Position
reuse	Errors	rSystem, rMemory, Sets, Idents, Position	rSystem, rMemory, Strings, StringM, Idents, Sets, IO, Position, Sort
ast	Tree	rSystem, General, rMemory, DynArray, StringM, Idents, Sets, Position	rSystem, General, rMemory, DynArray, IO, Layout, StringM, Strings, Idents, Texts, Sets, Position
ag	Eval	-	IO
puma	Trafo	rSystem, General	rSystem, IO

2.2. Semantic Analysis and Transformation

Our preferred compiler design constructs an abstract syntax tree as underlying data structure for semantic analysis. Afterwards this tree is usually mapped to some kind of intermediate language by a phase termed transformation.

The syntax tree as central data structure is managed by the module *Tree*. This module is generated by the tool *ast* out of a specification in a file of type .cg. The tool *ast* and its input language are documented in: "Ast - A Generator for Abstract Syntax Trees". The construction of trees is usually done during parsing. It is specified within the semantic actions of the input of the parser generator.

One possibility for the specification of semantic analysis is the use of an attribute grammar. The tool *ag* generates an evaluator module (named *Eval* by default) out of an attribute grammar. As this tool also has to know the structure of the abstract syntax tree both, *ast* and *ag* usually process the same input file. The tool *ag* and the extensions to the input language of *ast* for attribute grammars are documented in: "Ag - An Attribute Evaluator Generator".

The optimizer of *ag* decides how to implement attributes. They can be either stored in the tree or in global stacks or global variables. This information is communicated from *ag* to *ast* in files of type .ST. (This feature is not implemented yet.)

The tool *puma* generates transformers (named *Trafo* by default) that map attributed trees to arbitrary output. As this tool also has to know about the structure of the tree this information is communicated from *ast* to *puma* via a file of type .TS. The tool *puma* and its input language are documented in: "Puma - A Generator for the Transformation of Attributed Trees".

The names of the modules produced by the tools *ast*, *ag*, and *puma* can be controlled by directives in the input. Figure 2 uses the default names. By chosing different names it is possible to combine several tree modules, attribute evaluators, and transformers in one program.

Table 4: Modules in the Library Reuse

Module	Task	C	Modula-2
rMemory	dynamic storage (heap) with free lists	y	y
Heap	dynamic storage (heap) without free lists	-	y
DynArray	dynamic and flexible arrays	y	y
Strings	string handling for single byte character strings	-	y
WStrings	string handling for double byte character strings	-	y
StringM	string memory	y	y
Idents	identifier table - unambiguous encoding of strings	y	y
Lists	lists of arbitrary objects	-	y
Texts	texts are lists of strings (lines)	-	y
Sets	sets of scalar values (without run time checks)	y	y
SetsC	sets of scalar values (with run time checks)	-	y
HugeSets	sets of scalar values (up to 2**32)	-	y
Relation	binary relations between scalar values	-	y
RelatsC	binary relations of scalar values (with run time checks)	-	y
IO	buffered input and output	-	y
StdIO	buffered IO for standard files	-	y
Layout	more routines for input and output	-	y
Position	handling of source positions	y	y
Errors	error handler for parsers and compilers	y	y
Source	provides input for scanners	y	y
Sort	quicksort for arrays with elements of arbitrary type	-	y
Pack	compresses (scanner and parser) tables into strings	-	y
General	miscellaneous functions	y	y
rSystem	interface to the operating system	y	y
rString	portable string handling	y	-
rGetopt	portable version of Unix getopt	y	-
rFsearch	support for searching files	y	-
rSrcMem	storage for source code	y	-

2.3. Compiling and Linking

All the source modules generated by the tools have to be compiled by a compiler appropriate for the implementation language (C or Modula-2). Additional hand-written modules can be added as necessary. In Figure 2 the module *Support* indicates this possibility. In the last step all binaries have to be linked together with a few modules of the library *reuse* to yield an executable compiler.

The use of modules from the library *reuse* depends on the implementation language and the used tools. There is a C and a Modula-2 version of this library. The Modula-2 version is documented in: "Reusable Software - A Collection of Modula-2-Modules". The C version is documented in: "Reusable Software - A Collection of C-Modules". Table 3 lists the library units needed by tool generated modules. Additionally the user may engage further modules from this library for various tasks. Table 4 lists the modules that might be of interest. The right-hand side columns describe the availability of the modules with regard to the implementation language.

3. Makefile

The tools of the toolbox are conveniently controlled by an appropriate makefile. This eases the invocation of the tools and minimizes the amount of regeneration after changes. Figure 2 can be used to derive a makefile because it describes most of the dependencies among tools and files. The following makefiles control the generation of compilers for the example language MiniLAX in the target languages C, C++, and Modula-2. The annotated specification of this language is documented in: "Specification of a MiniLAX-Interpreter". The makefiles are examples a user can start with.

The makefiles in the next sections deviate in the following from the fundamental structure presented in Figure 2:

- The attribute evaluator module is called *Semantic* instead of *Eval*.
- The transformation module is called *ICode* instead of *Trafo*.
- The hand-written modules are called *ICodeInt* and *minilax*. The latter constitutes the main program.
- There are two support modules for semantic analysis called *Defs* and *Types*. These are generated by tools (*ast/cg* and *puma*), too.

3.1. C

The macro LIB specifies the directory where the compiled library *reuse* is located. The name of this library is *libreuse.a*. The -I flag in the macro CFLAGS specifies the directory where the header files of the library modules are located.

The first command (target minilax) is for linking the compiled modules to an executable compiler. The succeeding entries describe the dependencies during the generation phase and the invocation of the tools. The dependencies before the target clean are those needed during compilation.

```
LIB     = $(HOME)/lib
INCDIR  = $(LIB)/include
CFLAGS  = -I$(INCDIR)
CC      = cc

OBJS    = minilax.o Scanner.o Parser.o Tree.o \
          Types.o Defs.o Semantic.o ICode.o ICodeInt.o

minilax:        $(OBJS)
        $(CC) $(CFLAGS) $(OBJS) $(LIB)/libreuse.a -o minilax -lm

Scanner.rpp Parser.lrk:        minilax.prs
        lpp -cxzj minilax.prs

minilax.rex:    minilax.scn Scanner.rpp
        rpp < minilax.scn > minilax.rex

Scanner.h Scanner.c:    minilax.rex
        rex -cd minilax.rex

Parser.h Parser.c:      Parser.lrk
        lark -cdi Parser.lrk

Tree.h Tree.c:  minilax.cg
        ast -cdimRDI0 minilax.cg

Semantic.h Semantic.c:        minilax.cg
        ag -cDI0 minilax.cg

Defs.h Defs.c Defs.TS:     Defs.cg
        ast -cdim4 Defs.cg

Tree.TS:        minilax.cg
        echo SELECT AbstractSyntax Output | cat - minilax.cg | ast -c4

Types.h Types.c:        Types.pum Tree.TS
        puma -cdipk Types.pum

ICode.h ICode.c:        ICode.pum Tree.TS Defs.TS
        puma -cdi ICode.pum

Parser.o:       Parser.h Scanner.h Tree.h Types.h Defs.h
Semantic.o:     Semantic.h Tree.h Defs.h Types.h
Tree.o:         Tree.h
Defs.o:         Defs.h Tree.h
Types.o:        Tree.h Types.h
ICode.o:        Tree.h Types.h ICodeInt.h
minilax.o:      Scanner.h Parser.h Tree.h Semantic.h Defs.h ICode.h \
                ICodeInt.h Types.o

clean:
        rm -f Scan*.? Parser.? Tree.? Sema*.? Defi*.? Types.? ICode.? *.TS
        rm -f core _Debug minilax mini*.rex Parser.lrk Scan*.rpp yy*.h *.o

.c.o:
        $(CC) $(CFLAGS) -c $*.c
```

3.2. C++

The macro LIB specifies the directory where the compiled library *reuse* is located. The name of this library is *librucpp.a*. The -I flag in the macro CFLAGS specifies the directory where the header files of the library modules are located.

The first command (target minilax) is for linking the compiled modules to an executable compiler. The succeeding entries describe the dependencies during the generation phase and the

invocation of the tools. The dependencies before the target clean are those needed during compilation.

```
LIB     = $(HOME)/lib
INCDIR  = $(LIB)/cplusinc
CFLAGS  = -I$(INCDIR)
CPPC    = g++

OBJS    = minilax.o Scanner.o Parser.o Tree.o \
          Defs.o Types.o Semantic.o ICode.o ICodeInt.o

minilax:        $(OBJS)
        $(CPPC) -o minilax $(OBJS) -lm $(LIB)/librucpp.a

Scanner.rpp Parser.lrk: minilax.prs
        lpp -c++xzj minilax.prs

minilax.rex:    minilax.scn Scanner.rpp
        rpp < minilax.scn > minilax.rex

Scanner.h Scanner.cxx:  minilax.rex
        rex -c++d minilax.rex

Parser.h Parser.cxx:    Parser.lrk
        lark -c++dis Parser.lrk

Tree.h Tree.cxx:        minilax.cg
        ast -c++dimRDI0 -e minilax.cg

Semantic.h Semantic.cxx:        minilax.cg
        ag -c++DI0 minilax.cg

Defs.h Defs.cxx Defs.TS:        Defs.cg
        ast -c++dim4 Defs.cg

Tree.TS:        minilax.cg
        echo SELECT AbstractSyntax Output | cat - minilax.cg | ast -c++4

Types.h Types.cxx:      Types.pum Tree.TS
        puma -c++dipku Types.pum

ICode.h ICode.cxx:      ICode.pum Tree.TS Defs.TS
        puma -c++diu ICode.pum

Defs.o:         Defs.cxx Defs.h Tree.h Types.h
ICode.o:        ICode.cxx ICode.h Tree.h Defs.h Types.h ICodeInt.h
ICodeInt.o:     ICodeInt.cxx ICodeInt.h
Parser.o:       Parser.cxx Parser.h Scanner.h Tree.h Defs.h Types.h
Scanner.o:      Scanner.cxx Scanner.h
Semantic.o:     Semantic.cxx Semantic.h Tree.h Defs.h Types.h
Tree.o:         Tree.cxx Tree.h Defs.h Types.h
Types.o:        Types.cxx Types.h Tree.h Defs.h
minilax.o:      minilax.cxx Parser.h Scanner.h Tree.h Defs.h Types.h \
                Semantic.h ICode.h ICodeInt.h

clean:
        rm -f Scanner.*[hcx] Parser.*[hcx] Tree.*[hcx] Semantic.*[hcx] \
           Defs.*[hcx] Types.*[hcx] ICode.*[hcx] *.TS \
           core* *.dbg minilax minilax.rex Parser.lrk Scanner.rpp yy*.h *.o

.cxx.o:
        $(CPPC) $(CFLAGS) -c $*.cxx
```

3.3. Modula-2

The first command (target minilax) describes compilation and linking using the GMD Modula-2 compiler MOCKA. This compiler does its own dependency analysis among the sources. Therefore the makefile does not contain any dependency descriptions between sources and binaries. The -d flag of the compiler call mc specifies the directory where the library *reuse* is located. The rest of the makefile describes the generation phase and the invocation of the tools.

```
SOURCES = Scanner.md Scanner.mi Parser.md Parser.mi \
          Tree.md Tree.mi Semantic.md Semantic.mi \
          Types.md Types.mi Defs.md Defs.mi \
          ICode.md ICode.mi ICodeInt.md ICodeInt.mi minilax.mi

minilax:        $(SOURCES)
        echo p minilax | mc -d ../../reuse/src

Scanner.rpp Parser.lrk:        minilax.prs
        lpp -xzj minilax.prs

minilax.rex:    minilax.scn Scanner.rpp
        rpp < minilax.scn > minilax.rex

Scanner.md Scanner.mi:      minilax.rex
        rex -d minilax.rex

Parser.md Parser.mi: Parser.lrk
        lark -ci Parser.lrk

Tree.md Tree.mi:        minilax.cg
        ast -dimRDI0 minilax.cg

Semantic.md Semantic.mi:      minilax.cg
        ag -DI0 minilax.cg

Defs.md Defs.mi Defs.TS:   Defs.cg
        ast -dim4 Defs.cg

Tree.TS:        minilax.cg
        echo SELECT AbstractSyntax Output | cat - minilax.cg | ast -4

Types.md Types.mi:      Types.pum Tree.TS
        puma -dipk Types.pum

ICode.md ICode.mi:      ICode.pum Tree.TS Defs.TS
        puma -di ICode.pum

clean:
        rm -f Scan*.m? Parser.m? Tree.m? Sema*.m? Defi*.m? Types.m? ICode.m?
        rm -f core *.TS *.[dimor] _Debug minilax mini*.rex Parser.lrk
        rm -f Scan*.rpp
```

Contents