A Tool Box for Compiler Construction

J. Grosch
H. Emmelmann

GESELLSCHAFT FÜR MATHEMATIK
UND DATENVERARBEITUNG MBH

FORSCHUNGSSTELLE FÜR
PROGRAMMSTRUKTUREN
AN DER UNIVERSITÄT KARLSRUHE

Project

Compiler Generation

A Tool Box for Compiler Construction

Josef Grosch
Helmut Emmelmann

Jan. 21, 1990

Report No. 20

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

A Tool Box for Compiler Construction

J. Grosch, H. Emmelmann
GMD Forschungsstelle an der Universität Karlsruhe
Vincenz-Prießnitz-Str. 1, D-7500 Karlsruhe, Germany
E-Mail: grosch@karlsruhe.gmd.de, emmel@karlsruhe.gmd.de

Abstract

This paper presents a set of tools supporting the construction of nearly every compiler phase. Design goals of this tool box have been practical usability, significantly reduced effort for compiler construction, and high quality of the generated compilers. Especially efficiency should be competitive to hand crafting.

Currently modules in the target languages C and Modula-2 can be generated. First realistic applications demonstrate the excellent performance of the tools and show that the tools allow the construction of production quality compilers.

1. Introduction

Within this paper we present a compiler generation tool box. It contains tools for nearly every compiler phase. We believe the tools are applicable for realistic compiler projects.

The tools generally accept as input a specification written in a language specific to the tool and produce modules in a target language (C or Modula-2). Therefore a tool can be seen as a generic solution of a compilation subproblem, which is instantiated by the specification.

Using tools instead of hand crafting a compiler has several advantages: The effort necessary to build a compiler is substantially reduced. Instead of writing a program a much shorter specification has to be developed. The tools can perform many consistency checks on the specifications. Writing automatically checked specifications drastically reduces the amount of possible errors and hence increases the reliability of the resulting compiler.

The tool box consists of the following tools:

Rex	scanner generator
Lalr	LALR(1) parser generator
Ell	LL(1) parser generator
Ast	generator for abstract syntax trees
Ag	attribute evaluator generator
Estra	transformation of attributed syntax trees
Beg	back end generator
Reuse	library of reusable modules

All the tools were originally programmed in Modula-2 and run under UNIX. Using the Modula-2 to C translator called *Mtc* [Mar90] (see section 6.1), the sources also exist in C. Currently most of the tools generate modules in the target languages C and Modula-2.

The next two sections present the design goals and the common features of the tools. Section 4 describes the compiler model we prefer. In section 5 all the tools are sketched briefly. Section 6 reports about the experiences of two realistic applications of the tools. Section 7 gives a summary and describes future work.

2. Design Goals

The major design goals of the tool box were:

- practical usability for realistic languages
- automatic generation of production quality compilers
- substantial increase in compiler construction productivity and compiler reliability
- quality of the resulting compiler competitive to hand crafting

By defining these goals we wanted to produce a tool box which will be used in practical compiler construction work. Therefore we considered competitiveness to hand crafting important, because we feel that tools promising a high productivity and reliability but which produce compilers whose code quality or efficiency is lower than hand crafted compilers, will hardly be used.

3. Common Implementation Decisions

Our design goals lead to several design decisions common to all of our tools. Nearly every tool needs a programming language in which the user can specify certain actions, conditions, or calculations. That is obviously true for attribute grammars, but also the back end generator needs to evaluate several attributes and conditions. Even the parser generators need such a language for the specification of semantic actions.

We decided to select the target language (namely C or Modula-2). Specifications therefore may contain pieces of target language code. Besides some pattern replacement the text is copied unchanged to the generated modules. The disadvantage of that method is that the target language code can not be checked completely by the tool. For example the attribute grammar tool can not check if attribute evaluations do not have side-effects. On the other hand it gives a great deal of flexibility, as the whole power of the target language is available. It also drastically increases the practical usability, as interfacing to other components (perhaps hand-written ones) is easily possible. It finally keeps the tools and the specification languages simple. The user is not forced to learn a new language to express conditions or actions.

Our experience with prior tools is that during realistic compiler design a lot of small special problems occur, which the tool can not handle. Therefore loopholes, possibilities how the user of the tools can easily plug in hand-written parts, are necessary. Loopholes also allow to keep tools simple, as one is not forced to provide a solution for every special case, immediately. It is possible to use the loophole until a really good solution is found to be build in a tool.

The tools are largely independent of each other. This is achieved by the property that none of the generated modules has a fixed kind of output. Instead this output is specified using statements from the target language and thus can be chosen arbitrarily. The independence of the tools allows for a large degree of freedom in the compiler design. An exception are the tools *Ag* and *Estra* which operate on syntax trees specified using *Ast*. Therefore they depend on *Ast* and all three tools require the compiler to use an attributed abstract syntax tree.

4. Compiler Model

Although the tools do not directly enforce any specific compiler model, we want to present the model we prefer and which we believe is supported best by the tools. We still consider semantic analysis to be the hard part of a compiler. Therefore we base semantic analysis and the

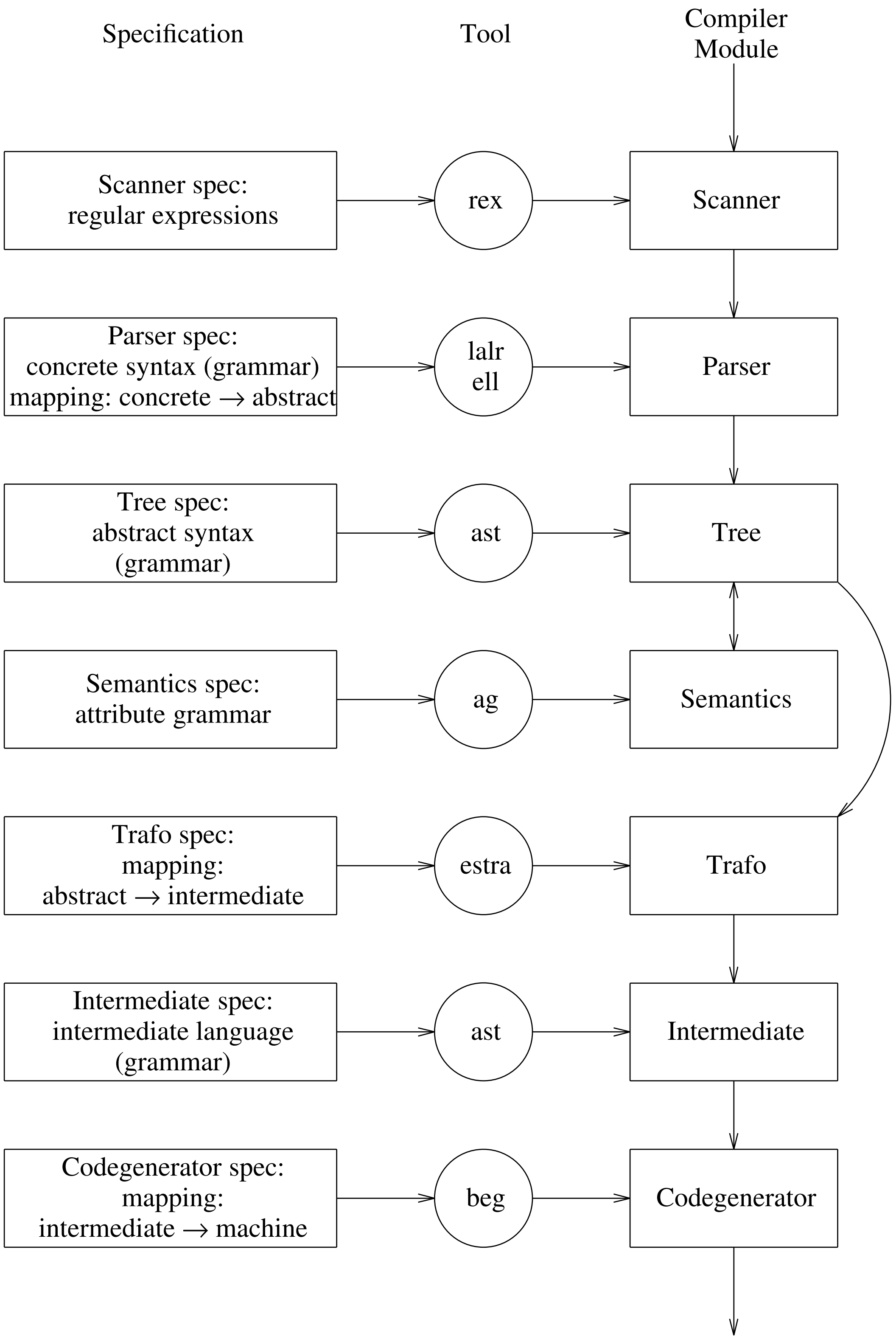
Specification
Tool
Compiler Module
Scanner spec: regular expressions
rex
Scanner
Parser spec: concrete syntax (grammar) mapping: concrete → abstract
lalr ell
Parser
Tree spec: abstract syntax (grammar)
ast
Tree
Semantics spec: attribute grammar
ag
Semantics
Trafo spec: mapping: abstract → intermediate
estra
Trafo
Intermediate spec: intermediate language (grammar)
ast
Intermediate
Codegenerator spec: mapping: intermediate → machine
beg
Codegenerator

Fig. 1: Compiler Model

generation of an intermediate language on the abstract syntax. We explicitly construct the abstract syntax tree which might be decorated with attributes during semantic analysis. Besides the abstract syntax, which can be regarded as a first (high-level) intermediate representation, we prefer to use a second (low-level) intermediate representation as interface to the code generator. This has advantages for optimizations and for pattern directed code selection.

Figure 1 shows our preferred compiler model. 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.

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. 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.

5. The Tools

The following sections briefly sketch the tools that make up the tool box. We only describe the features of the tools - for details, for the specification techniques, or for examples there exist separate documents.

5.1. Rex

The scanner generator *Rex* has been developed with the aim to combine the powerful specification method of regular expressions with the generation of highly efficient scanners [Gro87b, Gro88, Gro89a]. The name *Rex* stands for *regular expression tool,* reflecting the specification method.

A scanner specification consists in principle of a set of regular expressions each associated with a semantic action. Whenever a string constructed according to a regular expression is recognized in the input of the scanner its semantic action which is a sequence of arbitrary statements written in the target language is executed. To be able to recognize tokens depending on their context, *Rex* provides start states to handle left context and the right context can be specified by an additional regular expression. If several regular expressions match the input characters, the longest match is preferred. If there are still several possibilities, the regular expression given first in the specification is chosen.

Rex generated scanners automatically provide the line and column position of each token. For languages like Pascal and Ada where the case of letters is insignificant tokens can be normalized to lower or upper case. There are predefined rules to skip white space like blanks, tabs, or newlines and there is a mechanism to handle include files. The generated scanners are table-driven deterministic finite automatons. The tables are compressed using the so-called comb-vector technique [ASU86].

The most outstanding feature of *Rex* is its speed. The generated scanners process nearly 200,000 lines per minute without hashing of identifiers and up to 150,000 lines per minute if hashing is applied. (Keywords do not require hashing if they are recognized directly by the generated automaton.) This is 4 times the speed of *Lex* [Les75] generated scanners. In typical cases *Rex* generated scanners are 4 times smaller then *Lex* generated ones. Usually *Rex* takes only 1/10 of the time of *Lex* to generate a scanner.

5.2. Lalr

The parser generator *Lalr* has been developed with the aim to combine a powerful specification technique for context-free languages with the generation of highly efficient parsers

[Gro88, GrV88]. As it processes the class of LALR(1) grammars we chose the name *Lalr* to express the power of the specification technique.

The grammars may be written using EBNF constructs. Each grammar rule may be associated with a semantic action consisting of arbitrary statements written in the target language. Whenever a grammar rule is recognized by the generated parser the associated semantic action is executed. A mechanism for S-attribution (only synthesized attributes) is provided to allow communication between the semantic actions.

In case of LR-conflicts a derivation tree is printed to ease the location of the problem. The conflict can be resolved by specifying precedence and associativity for terminals and rules. Syntactic errors are handled fully automatically by the generated parsers including error reporting, recovery, and repair. The generated parsers are table-driven. Like in the case of *Rex*, comb-vector technique is used to compress the parse tables. The generator uses the algorithm described by [DeP82] to compute the look-ahead sets.

Parsers generated by *Lalr* are two to three times faster than *Yacc* [Joh75] generated ones. They reach a speed of 580,000 lines per minute on a MC 68020 processor excluding the time for scanning. The size of the parsers is only slightly increased in comparison to *Yacc*, because there is a small price to be paid for the speed.

5.3. Ell

The parser generator *Ell* processes LL(1) grammars which may contain EBNF constructs and semantic actions. It generates recursive descent parsers [Gro88, GrV88, Gro89b]. A mechanism for L-attribution (inherited and synthesized attributes evaluable during one preorder traversal) is provided. Like *Lalr*, syntax errors are handled fully automatic including error reporting from a prototype error module, error recovery, and error repair. Those satisfied with the restricted power of LL(1) grammars may profit from the high speed of the generated parsers which lies around 900,000 lines per minute.

5.4. Ast

Ast is a generator for *a*bstract *s*yntax *t*rees [Gro91a, Gro91b]. It generates program modules or abstract data types to handle attributed trees. Besides trees attributed graphs can be handled as well. The nodes of these data structures may be associated with arbitrary many attributes of arbitrary type. The specifications for this tool are based on extended tree grammars. They can be regarded as a common notation for concrete and abstract syntax as well as for attributed trees and graphs. An extension mechanism provides single inheritance. Internally the trees are stored as linked records. *Ast* can be requested to generate many operations for trees: node constructors combine aggregate notation and storage management. Reader and writer procedures transfer graphs from/to files in ASCII or binary format. The order of subtrees in a list can be reversed. Procedures for commonly used traversal strategies like top down (depth first) or bottom up (reverse depth first) are provided. An interactive graph browser allows the inspection of graphs in a readable way and thus supports debugging.

5.5. Ag

Ag is an attribute evaluator generator [Gro89d, Gro90]. It processes ordered attribute grammars (OAGs) [Kas80] as well as higher order attribute grammars (HAGs) [VSK89]. It is based on the abstract syntax or to be more specific on tree modules generated by *Ast*. Therefore the tree structure is fully known. The terminals and nonterminals may have arbitrary many attributes which can have any target language type. This includes tree-valued attributes. *Ag* allows attributes local to rules and offers an extension mechanism which provides (single) inheritance for attributes and for attribute computations. It also allows the elimination of chain rules. The attribute computations are expressed in the target language and should be written in a functional style. It is possible to call external functions of separately compiled modules. Non-functional statements and side-effects are possible but require careful consideration. The syntax of the specification language is designed to support compact, modular, and readable documents. An attribute grammar can consist of several modules where the context-free grammar is specified only once. There are shorthand notations for copy rules and threaded attributes which allow the user to omit many trivial attribute computations. The generated evaluators are very efficient because they are directly coded using recursive procedures. Attribute storage is optimized by implementing attributes as local variables and procedure parameters whenever their lifetime is contained within one visit.

5.6. Estra

Estra is a generator for *e*fficient *s*yntax *tr*ee *tra*nsformers [Vie89]. The generated transformation modules take an attributed tree as input and map it to an arbitrary output. The output can be a new tree, a linear code such as e. g. P-Code, source code like e. g. Pascal, or a sequence of procedure calls. In any case the input tree remains unchanged in order to avoid to reevaluate the attributes for reasons of consistency. However, subtrees of the input tree may be reused to construct an output tree. The intended application areas for the transformations are the generation of intermediate representations out of abstract syntax trees and optimizers operating on internal tree representations of any level. *Estra* cooperates with *Ast*: the input trees are constructed by modules generated with the latter tool.

The specification of a transformer is rule based. A rule consists of a pattern describing a tree fragment and an action. Actions are composed of target language statements. It is possible to specify several transformations. The subtrees of a pattern can be transformed in any order. They can be transformed several times by the same or by different transformations. The actions have read access to the attributes of the input tree. Additional synthesized and inherited attributes may be evaluated during the transformation. The application of rules can be restricted by conditions. Ambiguities may be resolved using costs.

Two implementations of the pattern-matcher can be selected: a directly coded dynamic programming algorithm or a table-driven tree pattern-matcher. In both cases the transformation has two phases. While the first one determines the patterns that match with minimal costs the second one executes the associated actions.

5.7. Beg

Beg (a *b*ack *e*nd *g*enerator) produces code selectors and register allocators [Emm89a, Emm89b]. Code selection is performed using tree pattern matching. The target instructions are described using rules containing tree patterns. The resulting code generator accepts a tree oriented intermediate language. An input tree is translated by covering the tree by the patterns and afterwards emitting the corresponding instructions. Rules are annotated with cost values which allows the code generator to select a cover of minimal cost, that means the sum of the costs of all rules in the cover is minimal.

Therefore the user only describes ambiguously how certain intermediate language constructs can be translated. He need not to program the algorithm to select the best way to translate a specific input tree. A good way to develop a code generator description is to first describe only a subset of the machine's instructions, big enough to compile the whole language. This results in a running compiler, which may produce inefficient code. Afterwards gradually more and more rules can be added which finally leads to a compiler producing good code.

Beg implements the determination of the minimal cover using a directly coded version of the dynamic programming algorithm [Emm89a, ESL89].

The generation of register allocators is of specific importance, because hand crafting is a rather difficult and tedious job and because errors in the register allocator are sometimes very difficult to find. Within rules, the characteristics with respect to register allocation of an instruction can be specified: the allowed registers for each operand, the registers changed by side-effects, and whether the instruction is a two address instruction or not. Additionally the register set of the target machine has to be described. Even the double register problem (e. g. IBM 370) can be handled.

Two kinds of local register allocators can be requested: the on the fly register allocator handles simple register sets. However, it provides satisfying results for many machines and is very efficient. In some cases the general register allocator is necessary which performs some kind of look-ahead. Therefore it requires an extra pass.

5.8. Reuse

Reuse is a library of reusable modules oriented towards compiler construction [Gro87a]. It contains modules or abstract data types which are needed in almost every compiler:

- a dynamic storage handler
- a module that provides dynamic and flexible arrays
- a facility to store strings of arbitrary length
- a module for string handling
- an identifier table which maps strings to unique integers using hashing
- modules for commonly used data structures such as sets of integers or binary relations between integers with no limitation of the domain

6. Application Experiences

This section reports the experience of applying the tool box for realistic problems.

6.1. Modula to C Translator

The largest application for the tool box so far was the generation of a Modula-2 to C translator [Mar90]. The program called *Mtc* translates Modula-2 programs into readable C code without any restrictions (even nested procedures and modules). It is largely generated and follows the compiler model proposed in section 4. Instead of generating an intermediate language, *Mtc* produces C code and therefore there is no need for a machine code generator. It incorporates as much of a semantic analysis as is needed for this task. The semantic analysis is rather complete and contains scope handling, name analysis, and type determination. However it does not check for semantic errors, as we assume that only correct programs will be translated. Table 1 gives the sizes of the specifications and the generated source modules. The design and implementation of *Mtc* was completed within a master thesis and took approximately 6 person months. The program is stable and has already converted more than 100,000 lines from Modula-2 to C.

The binary program comprises 301,240 bytes. It runs at a speed of 810 tokens per second or 167 lines per second on a SUN workstation (MC 68020 processor). This figures are computed by taking only the size of the translated modules into account. If we include the definition modules which are imported transitively and which are scanned, parsed, and analyzed as well, we arrive at 1320 tokens per second or 418 lines per second. In comparison, the compilers supplied by the manufacturer run at a speed of 97 lines per second (excluding header files) or 163 lines per second (including header files) in the case of C and 48 lines per second in the case of Modula-2. The run time of *Mtc* is distributed as follows:

part	specification			source module			tool	
	formal	code	total	def.	impl.	total	name	references
scanner	392	133	525	56	1320	1376	Rex	[Gro87b, Gro88]
parser	951	88	1039	81	3007	3088	Ell	[Gro88, GrV88]
tree	189	51	240	579	2992	3571	Ast	[Gro89c]
symbol table	115	938	1053	413	1475	1888	Ast	[Gro89c]
semantics	1886	151	2037	9	3288	3297	Ag	[Gro89d]
code generator	2793	969	3762	47	7309	7356	Estra	[Vie89]
reusable parts	-	-	-	819	2722	3541	Reuse	[Gro87a]
miscellaneous	-	-	-	698	3153	3851		
total	6326	2330	8656	2702	25266	27968		

Table 1: Sizes of the specifications and source modules of *Mtc*

scanning + parsing + tree construction	42 %
semantic analysis	33 %
code generation	25 %

The semantic analysis spends 95 % in attribute computations using user supplied code and 5 % in tree traversal or visit actions respectively. By the way, there are up to five visits to 11 node types.

Mtc uses approximately 360 K Bytes dynamic memory per 1000 source lines to store the abstract syntax tree, the attributes, and the symbol table without optimization of attribute storage. Another 600 K Bytes are used by the transformer generated with *Estra*. This is bearable with today's memory capacities. Contrary to the literature this shows that it is possible to store all attributes in the tree. We even do this for the environment attribute. This becomes possible by implementing the symbol table as an abstract data type in the target language. The implementation is time and space efficient by taking advantage of pointer semantics and structure sharing.

6.2. Code Generator for Modula-2 Compiler

In another application we replaced the hand-written code generator of the GMD Modula-2 compiler *Mocka* by two versions produced by *Beg*. Target machine was the MC 68020 processor. The specification of the code generator comprises 1625 lines. It contains 187 rules and 99 intermediate language operators. To compare the quality of the generated code, we measured the execution time for a collection of benchmark programs (see Table 2). *Mocka* denotes the GMD Modula-2 compiler with the hand-written code generator, *Begra* and *Begfly* refer to the code generators produced by *Beg* with the general register allocator and the on the fly register allocator respectively, and *Sun* is the SUN Modula-2 compiler version 1.0. On the average code generators produced by *Beg* generate code that is as fast as the one from the hand-written code generator.

	Perm	Towers	Queens	Intmm	Mm	Puzzle	Quick	Tree	Bubble	FFT	Σ
Mocka	40	58	37	53	103	285	32	72	56	152	888
Begra	42	57	35	54	104	297	32	58	56	153	888
Begfly	42	57	34	54	102	299	33	56	56	151	884
Sun	67	90	28	83	101	263	41	81	63	150	967

Table 2: Comparison of code quality (run times in *ticks* = 1/60 seconds)

Mocka	2.9
Begfly	3.2
Begra	3.9
Sun	25.4

Table 3: Comparison of compilation times (times in sec.)

Table 3 compares the run times of the compilers for processing a program with 1116 lines. All measurements were carried out on a diskless SUN 3/60, all measured times are user times. The size of the code generator increased from 5140 lines (17,117 tokens) for the hand-written version to 9574 lines (27,044 tokens).

7. Summary and Future Work

We presented a complete tool box of compiler construction tools which supports the construction of all phases of a compiler. The tools are very powerful and flexible and largely independent of each other. They support a wide range of compiler designs and allow the construction of production quality compilers for many programming languages. First realistic applications demonstrate the excellent performance of the tools.

The optimization of attribute storage of *Ag* will be improved allowing attributes to be treated as global variables and global stacks. The transformation of non-OAG grammars into OAG ones should be automized.

A redesign is planned for the tool *Estra*. The specification language will become simpler and clearer and the tool will be integrated better with *Ast*. The efficiency of the generated transformation modules can be improved both in terms of run time and storage consumption.

The optimization phase of a compiler clearly needs to be supported. This could be either done by a reusable, language independent optimizer, by an optimizer which can be parameterized, or by some means of an optimizer generator.

The tool *Beg* will be extended in the directions of the generation of a global register allocator, support for instruction scheduling, and a facility for the direct generation of binary object code.

Acknowledgement

We thank all that have contributed to the development of this toolbox either by active participation or with their ideas: Michael Besser, Carsten Gerlhof, Bob Gray, Rudolf Landwehr, Matthias Martin, Thomas Müller, F. W. Schröer, Dirk Schwartz-Hertzner, Doris Vielsack, Bertram Vielsack und William M. Waite.

References

[ASU86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading, MA, 1986.

[DeP82] F. DeRemer and T. Pennello, Efficient Computation of LALR(1) Look-Ahead Sets, *ACM Trans. Prog. Lang. and Systems 4*, 4 (Oct. 1982), 615-649.

[Emm89a] H. Emmelmann, Automatische Erzeugung effizienter Codegeneratoren, GMD-Studie Nr. 158, GMD Forschungsstelle an der Universität Karlsruhe, 1989.

[Emm89b] H. Emmelmann, BEG - A Back End Generator - User Manual, Arbeitspapier Nr. 420, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1989.

[ESL89] H. Emmelmann, F. W. Schröer and R. Landwehr, BEG - a Generator for Efficient Back Ends, *SIGPLAN Notices 24*, 7 (July 1989), 227-237.

[Gro87a] J. Grosch, Reusable Software - A Collection of Modula-Modules, Compiler Generation Report No. 4, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1987.

[Gro87b] J. Grosch, Rex - A Scanner Generator, Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1987.

[Gro88] J. Grosch, Generators for High-Speed Front-Ends, *LNCS 371*, (Oct. 1988), 81-92, Springer Verlag.

[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.

[Gro89a] J. Grosch, Efficient Generation of Lexical Analysers, *Software—Practice & Experience 19*, 11 (Nov. 1989), 1089-1103.

[Gro89b] J. Grosch, Efficient and Comfortable Error Recovery in Recursive Descent Parsers, Compiler Generation Report No. 19, GMD Forschungsstelle an der Universität Karlsruhe, Dec. 1989.

[Gro89c] J. Grosch, Ast - A Generator for Abstract Syntax Trees (Revised Version), Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Gro89d] J. Grosch, Ag - An Attribute Evaluator Generator, Compiler Generation Report No. 16, GMD Forschungsstelle an der Universität Karlsruhe, Aug. 1989.

[Gro90] J. Grosch, Object-Oriented Attribute Grammars, in *Proceedings of the Fifth International Symposium on Computer and Information Sciences (ISCIS V)*, A. E. Harmanci and E. Gelenbe (ed.), Cappadocia, Nevsehir, Turkey, Oct. 1990, 807-816.

[Gro91a] J. Grosch, Ast - A Generator for Abstract Syntax Trees, Compiler Generation Report No. 15, GMD Forschungsstelle an der Universität Karlsruhe, Sep. 1991.

[Gro91b] J. Grosch, Tool Support for Data Structures, *Structured Programming 12*, (1991), 31-38.

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

[Kas80] U. Kastens, Ordered Attribute Grammars, *Acta Inf. 13*, 3 (1980), 229-256.

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

[Mar90] M. Martin, Entwurf und Implementierung eines Übersetzers von Modula-2 nach C, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1990.

[Vie89] B. Vielsack, Spezifikation und Implementierung der Transformation attributierter Bäume, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, June 1989.

[VSK89] H. H. Vogt, S. D. Swierstra and M. F. Kuiper, Higher Order Attribute Grammars, *SIGPLAN Notices 24*, 7 (July 1989), 131-145.

Contents