Experiences with Gentle: Efficient Compiler Construction Based On Logic Programming[1]

Jürgen Vollmer
June 7, 1991
GMD Research Group at the University of Karlsruhe,
Vincenz–Prießnitz–Straße 1, D–7500 Karlsruhe 1
email: vollmer@karlsruhe.gmd.de, Phone: +/49/721/6622-14

Abstract

Gentle [Schröer 89] is a compiler description language in the tradition of two level grammars [Koster 71] and logic programming [Warren 80]. It provides a common declarative notation for high level description of analysis, transformation, and synthesis. Imperative constructs like global variables and dynamic arrays, needed for efficient compiler construction, are introduced as well. A tool has been implemented to check the wellformedness of *Gentle* descriptions, and to generate very fast (generation speed 260.000 lines per minute) very efficient compilers (compilation speed nearly 90.000 line per minute on Dec 3100 workstation). The language and a supporting tool were designed and implemented by F.W. Schröer in 1989.

Logic Programming and Compiling

Using logic programming as a compiler–writing tool has a long tradition. [Warren 80] shows how *Prolog* may be used for this purpose and writes:

> To summarize, *Prolog* has the following advantages as a compiler–writing tool:
>
> 1. Less time and effort is required.
> 2. There is less likelihood of error.
> 3. The resulting implementation is easier to 'maintain' and modify.

The practicability of *Prolog* for compiler writing "depends on how efficiently *Prolog* itself is implemented". About the way this could be done [Warren 80] states:

> It is likely that most of the improvement will be attributable to a few relatively simple but heavily used procedures (e.g. lexical analysis, dictionary lookup), and so a mixed language approach may be an attractive possibility. An alternative view (which I favour) is to look for more sophisticated ways of compiling special types of *Prolog* procedure, guided probably by extra pragmatic information provided by the *Prolog* programmer.

The language *Gentle*, presented in this paper, uses both ways to improve the execution speed of the generated compiler. *Gentle* uses dynamic programming to overcome restrictions imposed by this approach. This technique is not discussed here.

[1]published in *Proceedings of the Third International Symposium on Programming Language Implementation and Logic Programming - PLILP 1991*, Springer 1991, Lecture Notes in Computer Science Nr 528, pp 425–426

Gentle

Compilation is often viewed as a process translating the source text into a sequence of intermediate languages, until the desired output is synthesized. These intermediate languages may be viewed as terms, and *Gentle* offers a simple and efficient way to transform these terms. These transformations are described in a declarative way using predicates. Besides the specification of terms and rules transforming them, the concrete syntax of the context free source language is declared using the same declarative notation. Due to the special nature of the task Horn logic as *Gentle*'s foundation is modified in several ways:

- *Gentle* is a typed language. Term type declarations are used to specify terms. Predicates have a typed signature. The type of local variables is derived from the context in which they occur. Global variables are declared explicitly together with their types. For example:

```
-- term declaration
EXPR = const  (INTEGER), var (IDENTIFIER),
       binary (OP, EXPR, EXPR).
OP =   plus, minus, mult, div.
-- external type declarations
'TYPE' IDENTIFIER.
-- context free grammar predicate signatures
'TOKEN' Identifier (-> IDENTIFIER).
'TOKEN' PLUS.
'NONTERM' Expression (-> EXPR).
-- term transformation predicate signatures
'ACTION' CodeExpr (EXPR -> REGISTER).
'CONDITION' get_meaning (IDENTIFIER->OBJECT).
-- local variables in a clause
CodeExpr (Expr -> ResultReg)
-- global variable / table declaration
'VAR' INT Level.
'TABLE' NODE_ATTRIBUTES Graph [NODE].
```

- The data flow inside the predicates is fixed. In a clause, the parameters left form the arrow `->` are input parameters, those right of it are output parameters.
- The notion of a variable is more like that of functional languages.
- Several kinds of predicates are offered for different jobs during compilation. The context free grammar is specified using `'TOKEN'` and `'NONTERM'` predicates. A parser generator is used to generate

a parser out of the context free grammar specified by the *nonterm* clauses.

Term transformation is specified by `'ACTION'` and `'CONDITION'` predicates. *Action* and *condition* predicates transform their input terms into output terms. Side effects (like writing to a file or a global variable) may be caused by them. *Action* predicates are used as an assertion a transformation must fulfill, while a *condition* predicate is used to test terms for the given condition.

```
-- a grammar clause
Expression (-> var (Id)) :
   Identifier (-> Id).
Expression (-> binary (plus, Left, Right)) :
   Expression (-> Left)
   PLUS
   Expression (-> Right).
-- an action clause
CodeExpr (const (N) -> ResultReg):
     GetNewReg (-> ResultReg)
     EmitCode (load_constant (N, ResultReg)).
-- a condition clause
Is8bit (N) :
     LessEqual (-128, N) LessEqual (N, 127).
```

- Backtracking is restricted, such that once a tail of a clause has been proven completely, all alternatives for that clause are discarded.

The price one must pay is that the *Gentle* proof procedure is not complete, but a more efficient implementation is possible. Our experience shows that the full power of the *Prolog* backtracking mechanism is not needed for compiler writing.
Compilers have to maintain global data, like symbol tables, or must deal with graphs, like basic block graphs, used for optimization. For an efficient implementation of these kinds of data, *Gentle* offers global variables and global tables, which are something like dynamic growing arrays in imperative languages.

```
-- creation of a new table entry
new (-> Node) : 'KEY' NODE Node.
-- read / write access of a table entry
Graph [Root] -> node (Info, Succ1, Succ2)
Graph [Root] <- node (Info, Succ1, Succ2)
```

Access to procedures implemented in other languages is possible. As a result *Gentle* is a mixture of declarative (backtracking, notation), functional (variables, fixed data flow) and imperative (global data, external procedures) features, which is well suited specifying compilers at a very high level of abstraction.

Measurements

Several projects are implemented using *Gentle*. Some of the important ones are:

- A program transformation tool for object oriented languages, which translates *Trellis* programs to *C++*. The specification consists of 6000 lines of *Gentle* program and external *C++* code for the lexical analyzer.
- A compiler for a subset of a *Pascal* like language, called *MiniLax* and with target processors M68k and VAX. The *Gentle* specification consists of 896 lines for the front–end and 534 lines for the code generator.
- In the *ESPRIT* project *Rex* a specification language for time critical, distributed systems was designed and is currently implemented.
- A compiler for a functional–logic language called *guarded term ML* is currently under development. As target processor a M68k is used.
- An industrial compiler for an object oriented language.
- *Gentle* itself is implemented with *Gentle*.

The followings table shows the generation speed of the *Gentle* tool and the compilation speed of the generated compiler. The times are measured on a *Dec station 3100 (MIPS processor)* using the *UNIX time* command (user time). The average of three runs is given. The generated *C* programs are compiled using the optimization capability of the *C* compiler.
For the *Generation* measurement, the *Gentle* tool generates itself. The specification consists of 3939 lines of *Gentle* program and 405 line *Gentle* library. Output are 4861 lines of *C* program, 1455 lines input for the parser generator and 30 lines auxiliary input to the scanner and parser generator. For the *Compilation* measurement, the generated *MiniLax* compiler compiles a 8055 line input program and generates a 58.718 lines of M68k assembler text.

	time (sec)	lines per minute
Generation	1,0	260.640
Compilation	5,4	89.499

References

[Koster 71] C.H.A Koster. Affix grammars. In J.E.L Peck, editor, *ALGOL 68 Implementation*, pages 95–109. North Holland, 1971.

[Schröer 89] F.W. Schröer. Gentle. In J. Grosch, F.W. Schröer, and W.M. Waite, editors, *Three Compiler Specification*, GMD – Studien Nr. 166, pages 31–36. GMD, Forschungsstelle an der Universität Karlsruhe, August 1989.

[Vollmer 91] Jürgen Vollmer. A tutorial on gentle. Arbeitsberichte der GMD Nr 508, GMD, German National Research Center for Computer Science, Vincenz–Prießnitz–Straße 1, D–7500 Karlsruhe–1, February 1991.

[Warren 80] David H.D. Warren. Logic programming and compiler writing. *Software–Practice and Experience*, 10:97–125, 1980.