Abstract Interfaces for Compiler Generating Tools

U. Kastens
University of Paderborn, FRG

1. Introduction

Compiler construction is supported by tools which generate modules for the main compiler subtasks. Different tools have to cooperate in order to produce a complete compiler. They are often developed stand-alone without consideration how to integrate the generated products. A considerable amount of engineering work is needed to configurate the whole compiler. Adaption of the generated modules often reduces the overall performance of the compiler.

A compiler tool generates a module for a compiler subtask from a formal specification. Problems of tool integration result from dependencies between the specifications for different subtasks and from integration of the modules. There are different strategies to simplify compiler development with tools: (1) A set of tools is developed together such that they fit together smoothly. (2) Independantly develpoed tools are embedded in a compiler construction environment which solves the integration problem for the user. Such a system is described in [WHK 88]. (3) Each single tool is designed with open interfaces which allow for easy integration and adaption of the generated products.

Strategy (1) is rather inflexible against exchange of single tools and adaption to implementation requirements. A compiler construction environment (2) can be composed for different tool sets. It embodies the know-how of tool integration. It can adapt the interfaces determined by a single tool but it cannot change them. The notion of the specification and the implementation of the modules are fixed. Hence we suggest in this paper to develop compiler generating tools according to (3). Such a tool takes an abstract representation of the specification as input and produces an abstract representation of the generated algorithm. Various frontends and backends can easily be derived to support suitable notions for the specifiaction and smoothly integratable implementations of the algorithm.

In Sect. 2 of this paper we dicuss the problems of compiler tool integration. Sect. 3 presents the suggested tool design concept. Two examples of tool design on that base are given in Sect. 4.

The ideas presented in this paper arose from the work on the development of compiler construction environments performed in cooperation with the group of W.M. Waite, University of Colorado, Boulder.

2. Integration of compiler tools

A task oriented compiler design refines the compiler task into a sequence of subtasks: lexical analysis, syntactic analysis, semantic analysis, code selection, and assembly. (Global optimization and peeephole optimization may be inserted before and after code selection.) The tasks are implemented by central compiler modules (scanner, parser, attribute evaluator, code generator, and assembler) which act as filters transforming one representation of the source program into another (character sequence, token sequence, structure tree, intermediate language, instruction sequence, object code). These filters use functions of additional modules which act as abstract data types, store properties of certain objects and provide access to them (e.g. identifiers, literal values, declared objects, ranges of declarations, registers). Such a compiler structure is well accepted, systematic methods for the solution of the subtasks are well known and can be found in most compiler texts [ASU 86], [WG 84].

Different compiler tools are in use to generate each of the central filter modules. (The ADT-Modules are usually not generated but adapted from precoined implementations.) Such a tool takes a specification of the subtask and generates the compiler module from it. The input to the tool specifies the translation step in some notion of a certain formal concept specific for that subtask, e.g. regular expressions for the scanner, a context-free grammar for the parser, an attribute grammar for the attribute evaluator, and tree patterns with associated instructions for the code generator. The compiler module generated by the tool is an algorithm of a certain class (e.g. finite state machine, stack automaton, etc.) implemented by a systematic technique (e.g. table driven) and expressed in a certain implementation language. Furthermore the module implementation includes interfaces to surrounding modules. Fig. 1 describes this situation for a set of tools graphically.

With such a set of generating tools a compiler is constructed mainly by developing the specifications. Additionally two classes of more technical problems have to be solved - often without help from the tools: The single specifications are interrelated and have to be ensured to be consistent. The generated modules must be integrated into the whole compiler.

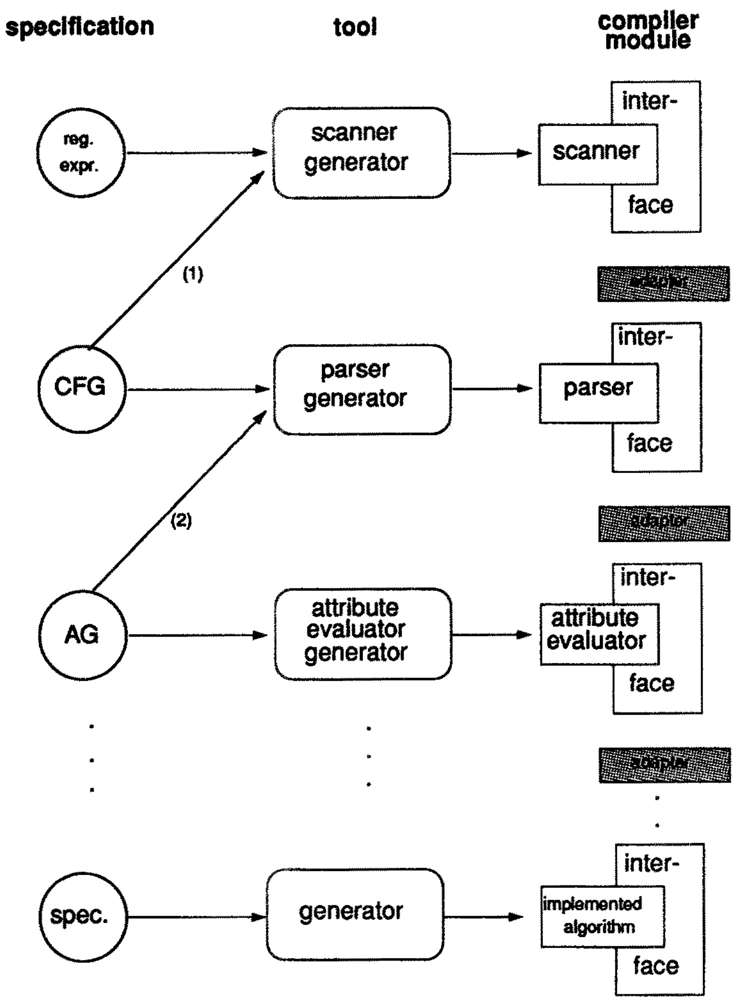

Fig.1: Tools generating compiler modules

As an example for relationships between specifications consider the scanner and parser generation steps: The symbols specified for the scanner are the terminal symbols of the context-free grammar for the parser. Usually the parser specification includes the notion of keywords and special symbols. Instead of specifying them twice (for the scanner and parser generator) they could be extracted from the CFG and directed to the scanner generator (arc (1) in Fig. 1). That leaves the scanner specification with regular expressions for identifiers and literals. A more subtle relationship exists between the specifications for the parser and the attribute evaluator: The attribute grammar is based on a CFG which describes the structure of the tree to be attributed. It is an abstraction of the derivation tree computed by the parser. Hence the concrete grammar for the parser specification has to be augmented by actions which construct that strucure tree (arc (2) in Fig. 1).

On the side of the generated products often a considerable amount of technical work is required for module integration: The modules may be generated in different implementation languages, e.g. a parser generated in C by YACC [Jo 75] and an attribute evaluator in Pascal from GAG [KHZ 82]. The interfaces of the modules may be based on different assumptions how information is passed from one module to the other (e.g. encoding and passing of tokens from the scanner to the parser, or execution of actions initiated by the parser). For both cases it might be necessary to adapt the interfaces. Since generated modules usually are not prepared for being modified additional procedures transforming the data structures or linking different languages have to be inserted. Such an adaption can reduce the performance of the compiler. It may even be impossible to adapt the modules reasonably. In the latter case a certain combination of tools must be discarded.

If each of the tools is used stand-alone the compiler developer has to adapt specifications and generated products for each compiler generated. The know-how for that work is not compiler specific but determined only by the combination of the tools. There are different strategies to reduce the amount of that work:

(1) Fixed tool sets: A certain set of tools is developed such that specifications and products easily fit together, e.g. LEX [Le 75] and YACC [Jo 75] or GAG [KHZ 82] and PGS [De 77]. If interfaces of the tools are hidden for the user such a toolset may even appear as a monolithic generator producing a readily integrated compiler (or part of it) from an integrated specification.

(2) Open compiler construction environments: The desired set of tools is combined under control of a tool mangement system. It takes care for relations between specifications, directs them to the tools and integrates the generated products. Hence it solves the tool integration problems for that specific tool combination. Such an environment is Eli [WHK 88] constructed with the tool mangement system ODIN [CL 88].

The integration problems discussed above will arise again in solution (1) if one of the tools in the set should be replaced by a generator with different characteristics, or if the set is completed by additonal tools. Such an exchange of single tools is usually impossible in monolithic systems. Environments like (2) support modifications of the tool set best of all: The know-how for the specific tool integration is embedded initially in the environment and can be modified when tools are changed. The management system applies that knowledge automatically for the construction of each compiler. We constructed several environments solving the integration problems for different tool sets. From that experience we learnt that tools should be designed with interfaces which simplify their integration.

3. Abstract tool interfaces

A main reason for the difficulties with the integration of compiler tools discussed in the previous section lies in the design of the tools themselves: Neither the notion of the input specification nor the implementation of the generated compiler module can easily be modified without rewriting large parts or all of the tool. Hence we suggest to develop tools according to the following design principle: The central task of a tool is the computation of an algorithm from a specification. The central part of the tool should be restricted to exactly that task. Its input and output interfaces both are abstract data structures instead of notions of the specification and implementations of the algorithm. Frontends and backends transform the notion and implement the algorithm. They can easily be modified without touching the central tool part in order to integrate the tool smoothly. Fig. 2 shows such a tool structure with the *abstract specification* and the *abstract algorithm* as its central interfaces.

The *abstract specification* is a data structure representing structure and information contents of a specification. Its data format is reasonably simple for any compiler tool due to the underlying formal specification concept, e.g. CFGs in the case of a parser generator. A frontend for the tool producing that format from some notion can easily be constructed or modified, e.g. in order to let a parser generator be used with the same input notion as that of YACC or PGS. Furthermore specification information can be exchanged between tools (as in Fig. 1) directly on the base of the abstract specification. Even interactive syntax directed editors can be adapted to the tool on that interface. Such a frontend should be restricted to transform the notion of the specification into the abstract representation. Any check for consistency or further transformation which is independant of the notion should be performed either in the central part of the tool or by additional modules with the abstract representation as both input and output interface.

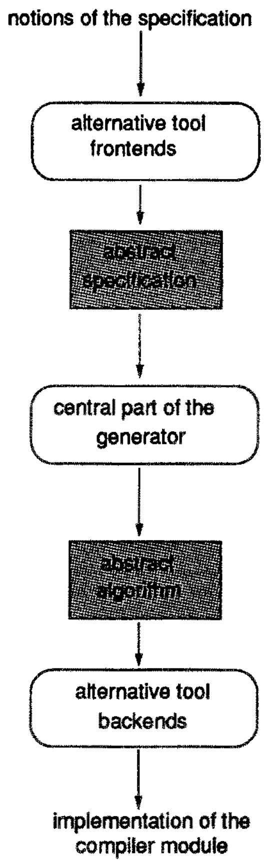
notions of the specification
alternative tool frontends
abstract specification
central part of the generator
abstract algorithm
alternative tool backends
implementation of the compiler module

Fig. 2: Abstract tool interfaces

The central part of the tool generates an *abstract algorithm.* It is described by a data struture based on the underlying formal concept of the algorithm, e.g. the transition function of an automaton. It does not contain any implementation decision for the algorithm. Various backends can be developed for that interface implementing the algorithm using different techniques, e.g. either table driven or directly programmed, and using different implementation languages. Furthermore such a tool backend can easily be modified in order to produce compiler modules which fit to other modules without the need for explicit and costly adapters.

This general principle is applicable throughout all of the compiler generating tools because they are based on formal concepts for both the specification and the generated algorithms.

(It is of course applicable outside of compiler construction, too, where generation concepts have these characteristics. This generalization aspect is beyond the scope of this paper.) A scanner generator transforms representations of regular expressions (derived from character oriented or graphical notions) into a finite state machine described by its transition function. A parser generator takes a context-free grammar (with actions inserted), checks it for the required grammar class and produces an abstract parsing algorithm. It is represented by the grammar with inserted director sets in the case of an LL-parser, or by the transition function in case of an LR-parser. An attribute evaluator generator transforms a representation of an attribute grammar into a tree walking automaton. In the next section it will be shown how that algorithm can be represented easily. The generator of a code selector may take the description of tree patterns with associated actions for code generation and augments them by strategic information for pattern application.

4. Two Examples for Tool Design

In this section we briefly describe the design concepts of two projects for compiler tools where we applied the principles of abstract interfaces described in the previous section. In the first example a set of tools based on context-free grammars is presented. The second example refers to an attribute evaluator generator developed as a successor of the GAG system.

COMAR

COMAR is a data format for the representation of context-free grammars. It was developed for interfacing various tools which analyse and transform CFGs [HKPW 87]. It plays the role of abstract interfaces as introduced in Sect. 3, and is the base of an extensible set of CFG-tools.

The information structure of a CFG consists of objects which are terminals, nonterminals, productions and other objects which may augment the productions (e.g. actions, director sets, etc.). The abstract data structure of a CFG is a list of definitions of such objects. The productions in extended BNF are described by trees for their right hand sides. Additional application specific properties are associated to the objects: e.g. actions or first- and follow-sets associated to productions.

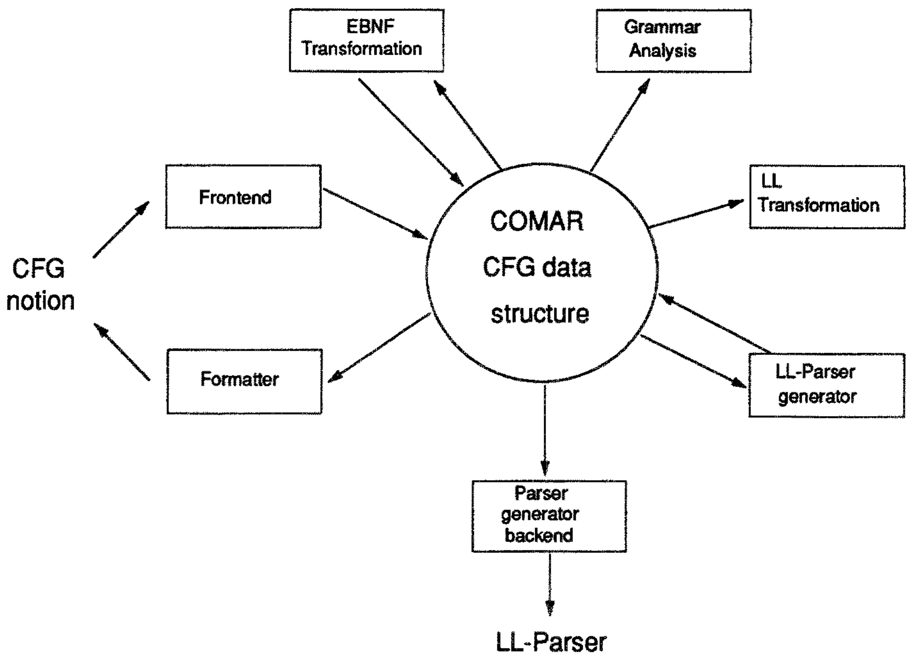

Fig. 3: COMAR Tool Set

The COMAR data format is defined in terms of the interface description language IDL [La 87]. From that definition the IDL-tool [SS 86] automatically generates an implementation of the data structure in C with access functions, and routines for reading and writing a standardized file representation of a COMAR structure. The development of a CFG tool can immediately concentrate on its specific task using such a complete implementation of the CFG representation. On this base we developed an initial, extensible set of CFG tools which cooperate via the COMAR structure, Fig. 3. Upto now it contains a frontend transforming a CFG notion into COMAR, a formatter for the opposite direction, tools for EBNF-transformation LL(1)-check, LL-transformations, and an LL(1)-parser generator [Bu 88].

The design of that LL(1)-parser generator follows the principle of abstract interfaces presented in this paper: Its input interface is a CFG in COMAR. The generator augments it by LL(1)-director sets. The result - again a COMAR structure - is an abstract representation of the LL(1)-parser. A backend of the generator implements it in C using a directly programmed non-recursive technique which yields fast parsers.

Another parser generator based on the LALR(1)-technique is being developed for COMAR. In this case the generated abstract algorithm of a LR-parser is not described by an augmented CFG as in the LL-case. It is a data structure specifying the transition function instead. Again suitable backends will implement it using a certain technique and implementation language.

LIGA

An attribute evaluator is the central module of the semantic analysis phase a compiler. The language independant generator for attribute evaluators LIGA is developed as a successor system for the well established generator GAG [KHZ 82]. Whereas the GAG system is fixed to a certain input language for AG specification (ALADIN) and the target implementation language Pascal, the design concept for LIGA is based on abstract tool interfaces, Fig. 4.

The information structure of an AG consists of objects which are symbols of a CFG with associated attributes, and CFG productions each with a set of attribute rules. An attribute rule specifies the context dependent computation of an attribute by a functional expression over other attributes. Similarly as for COMAR a data structure is defined for the abstract respresention of an AG. It consists of a list of the object definitions with the productions and the attribution expressions being represented by trees. (The meaning of operators and functions in the attribution is specificied outside the AG in terms of the implementation language.) Again that data structure is defined in IDL in order to take advantage of its automatic implementation.

In the AG the attribution of a production is a list of attribute rules in an arbitrary order. The LIGA system generates a tree walking attribute evaluator controlled by a visit-sequence for each production [Ka 80]. A visit-sequence is an ordered sequence of attribute rules with operations for tree movement (ancestor and descendant visits) inserted. Hence the abstract algorithm for such an attribute evaluator can be easily described by a slight extension of the AG data structure: For each production the LIGA system sorts the attribute rules according to the attribute dependencies and inserts visits at the appropriate positions. (How to compute the visit sequences and how to optimize attribute implementation is beyond the scope of this paper and can be found in [Ka 80], [Ka 87], and [Ha 87]. The central part of the LIGA system produces the abstract attribute evaluator as an augmented AG representation. Various backends may implement this evaluator using different trechniques (table driven, recursive procedures, directly programmed without recursion) and different implementation languages. The first existing backend applies the latter technique in C, which yields in general the fastest implementation.

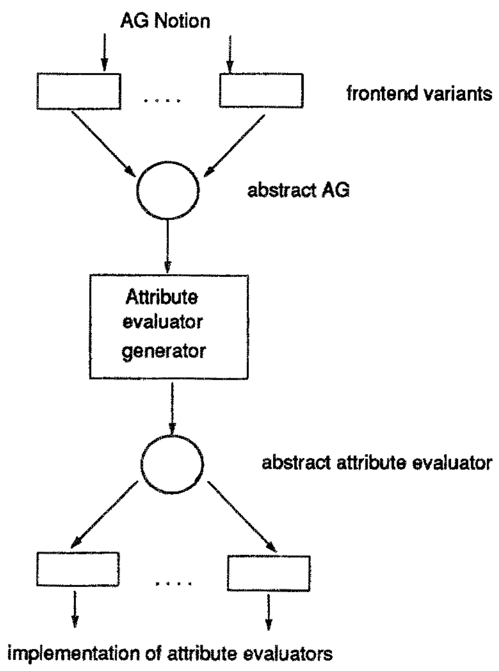
AG Notion
....
frontend variants
abstract AG
Attribute evaluator generator
abstract attribute evaluator
....
implementation of attribute evaluators

Fig. 4: LIGA system with abstract interfaces

5. Conclusion

The main tasks in compiler construction are solved by systematic algorithms. Tools are available which generate compiler modules from specifications. They are often developed as closed stand-alone systems difficult to combine with other tools. Integration of their products requires considerable engineering work and may loose efficiency because the tools fix all implementation decisions.

In order to improve the applicability of generating tools we suggest in this paper a modular tool structure: The central part of the generator translates an abstract representation of a specification into an abstract algorithm. These interfaces are open to meet the requirements of the tools environment by variations of the frontends and backends. Especially the implementation decisions for the generated products are separated from the generation of the abstract algorithm. We demonstrated this design principle applied to the development of a CFG toolset and an attribute evaluator generator.

Further work should be devoted to the development of other compiler tools on that base and to increase the variants of the existing tools. Furthermore we believe that the design principle is as well applicable to tools of areas different from compiler construction.

References

ASU 86	Aho, A.V., R. Sethi and J.D. Ullman, "Compilers", Addison Wesley, Reading MA, 1986
Bu 88	Budde, V., "Werkzeuge zur Grammatik-Transformation und LL-Parsergenerierung auf der COMAR-Datenbasis, Diplomarbeit, FB 17, Universität-GH Paderborn, 1988
Cl 88	Clemm, G.M., "The Odin Specification Language", in International Workshop on Software Version and Configuration Control '88, Teubner, Stuttgart, 1988, 144-158
De 77	Dencker, P. "Ein neues LALR-System", Institut für Informatik, Universität Karlsruhe, Diplomarbeit, 1977
Ha 87	Hall, M.L., "The Optimization of Automatically Generated Compilers", Ph.D. Thesis, Dept. of Computer Science, University of Colorado, Boulder, CO. 1987
HKPW 87	Heuring, V.P., Kastens, U., Plummer, R.G., Waite, W.M., "COMAR: A Data Format for Integration of CFG-Tools", University of Colorado, Boulder, ECE-Report 87/124, 1987, to appear in The Computer Journal
Jo 75	Johnson, S.C., "YACC - Yet Another Compiler-Compiler", Computer Science Technical Report 32, Bell Telephone Laboratories, Murray Hill, NJ, July 1975
Ka 80	Kastens, U. "Ordered Attributed Grammars", Acta Informatica 13, 229-256, 1980
Ka 87	Kastens, U. "Lifetime Analysis for Attributes", Acta Informatica 24, 633-651, 1987
KHZ 82	Kastens, U., Hutt, B., Zimmermann, E. "GAG - A Practical Compiler Generator", Lecture Notes in Computer Science 141, Springer, 1982
La 87	Lamb, D.A., "IDL: Sharing Intermediate Representations", TOPLAS, vol. 9, no. 3, 1987, 297-318

Le 75	Lesk, M.E., "LEX - A Lexical Analyzer Generator", Computer Science Technical Report 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975
SS 86	Snodgrass, R., Shannon, K., "Supporting flexible and efficient tool integration", SoftLab Document No. 25, Computer Science Department, Universtiy of North Carolina, Chapel Hill, N.C., May 1986
WG 84	Waite, W.M., G. Goos, "Compiler Construction", Springer Verlag, New York, NY, 1984
WHK 88	Waite, W.M., Heuring, V.P., Kastens, U. "Configuration Control in Compiler Construction", Int'l Workshop on Software Version and Configuration Controll SVCC, Grassau, Januar 88