Computer-aided building of a compiler: an example

Carine Fédèle and Olivier Lecarme
I 3 S
Université de Nice-Sophia Antipolis and CNRS
rue Albert Einstein, F - 06560 Sophia Antipolis

Introduction: A software product automatically built from a formal specification presents useful characteristics: conformity to the specification, any change automatically reflected in the product, generated from one source text only. Thus, reliability, portability, compatibility, and adaptability are enforced. This is especially interesting when the software product is a compiler, because of the fundamental importance of such a tool.
Most software systems must rely on some imperative language like Ada, Modula-2, or C. Their reliability and performance are strongly related to those of this underlying language. However, software shops too often produce compilers entirely by hand, ignoring even parser generators. Current commercial compilers are generally far from international standard definitions, their reliability is doubtful, and they are often not maintained after delivery. All our software products thus rely on a frail basis.
The automatic production of a full compiler, with compile-time and run-time performances similar to those built by hand, seems to be a goal far from reach. Building a compiler has most of the characteristics of a craft. However, this situation forbids having good implementations of most languages on most machines.
We think that building a compiler should take more characteristics of an industrial process. We aim at making progress towards complete tools for automatizing the production of all components of compilers, and for validating them. Our approach is the following: fully automatize wherever existing theory provides a sufficiently powerful framework; provide more and more complete aids in the other areas; and never sacrifice run-time performances of compilers and object programs to automatization.
Our paper describes the processing, using our system Cigale, of a completely new multi-paradigm programming language, Leda (designed by T. A. Budd). In the framework of strong typing and true compilation, it tries to combine aspects of the imperative (value-oriented) style of programming, the logical (relation-oriented) one, the functional one, and the object-oriented one. Thus one finds in the language the concepts of procedures, relations, functions, and classes, together with the controlled backtracking mechanism of generators. Cigale is a compiler writing system, written in Pascal, and producing compilers written in Pascal. It is the last avatar of the system initially designed in 1974, at the University of Montreal, by one of the authors.
Building the semantic analyzer: Building the scanner and parser for Leda is straightforward. The next step is to obtain a semantic analyzer, i.e. a program that checks all semantic constraints that must be satisfied by source programs, and solves semantic ambiguities of overloaded operators. It uses visibility rules and type checking rules. It understands the meaning of declarations, and links all utilizations of names in the text to the proper declarations.
For describing the semantic aspects of the language, Cigale provides the basic concept of an *attribute grammar*. The execution of semantic actions is triggered by the parser, just before reducing the right-hand side of the production to the left-hand side symbol. Since the parser is a bottom-up one, and attributes are stored in the parsing stack, with no

derivation tree being built, only synthesized attributes are normally available: semantic actions can use the values of right-hand side attributes, and assign values to left-hand side attributes. An attribute that occurs on both sides is automatically copied from right to left at the end of the semantic action. Moreover, it is possible to mention a 'stack context' in grammar rules, i. e. symbols found in the stack. Thus we can access their synthesized attributes. This provides exactly the power of inherited attributes evaluated from left to right.

Three basic concepts are represented in the symbol table: *names*, which are generally identifiers; *types*, which are descriptors for every characteristic that can be attached to a name; and *values*.

Beginning with the simplest declarations, we attach attributes to non-terminals, and add semantic actions to productions. In most cases, the semantic action generally reduces to a simple procedure call. In many cases, however, the semantic action is unduly complicated by error handling. Similarly, it seems clear that some situations occur in most programming languages, and could be built in the system, instead of having to be programmed by users. This is exactly the case, and we are presently developing a higher-level notation for semantic actions. This will be processed by Cigale, and translated into calls to existing procedures, with most parameters automatically generated. One of the main advantages of such an idea is that it is language-independent. Thus, the notation can be translated in any language we want.

The symbol table: The symbol table of Cigale is a collection of name descriptors, which contain general-purpose information (name, scope level, etc), as well as more specific information related to the class of the name. For example, a constant identifier descriptor contains a pointer to a type descriptor and another to a value descriptor; a parameter identifier descriptor contains its passing mode, a pointer to its type descriptor, etc.

Names must be stored in the symbol table in such a way that their declaration order (in the source text) is not lost. Moreover, one must be able to collect all identifiers pertaining to the same scope or declaration level. The organization provides for fast access and easy erasing of all names of a given level. Modularity and overloading are handled in the actual implementation of this organization.

Users can extend the symbol table at will, using extensions. They declare the extension descriptor, and link it to the predefined descriptor by using predefined procedures. They also program the management procedures related to this extension. Whatever generality we intend, we cannot pretend to be capable of implementing every language. We consider only conventional imperative languages, and even with this important limitation, the price of generality cannot be ignored.

What we described illustrates an operational approach to source language semantics. By contrast, the higher-level notation will be more declarative. Our approach will be to isolate frequent paradigms, present in most language definitions.

Conclusion: Users will add dynamic semantics and code generation to their attribute grammar in the same way as they did for static semantics. Cigale provides another set of declarations of types, functions, and procedures for this purpose, and an extension to the declarative notation in order to handle these aspects too. Code generation is aimed at the universal intermediate language EM, for which excellent back-ends already exist for most present computers and micro-chips.

While lexical and syntactic aspects have been successfully given almost perfect descrip-

tion means, all the semantic aspects are much more reluctant. Numerous formalisms have been designed, but all attempts to build compiler generators using them have been disappointing: formalisms themselves are generally much too complicated to be used as readable descriptions, and above all, systems that use them fail to be true compiler generators. This results from the very ambitious goal generally aimed at by most current research teams: to provide only one full formal definition of the source language, with no operational parts and no reference to any object machine. Of course, this cannot result in something comparable with an industrial compiler, both in flexibility and efficiency. Our approach is much less ambitious than the one we just summarized. We believe it to be also much more realistic. Moreover, we think it is presently the only one that can make progress towards the automatic construction of validated and efficient compilers.