SOFTWARE—PRACTICE AND EXPERIENCE. VOL. 22(11), 911–936 (NOVEMBER 1992)

Towards a Toolkit for Building Language Implementations

CARINE FÉDÈLE AND OLIVIER LECARME
13 S, University de Nice-Sophia Antipolis and CNRS, 250 avenue Albert Einstein, F-06560 Valbonne-Sophia Antipolis, France

SUMMARY

The current work of the authors in the area of software tools for automatic construction of compilers is described. This focuses on attempts to provide for automatic production of the semantic-analysis and intermediate-code-generation parts of the Cigale compiler-writing system, developed at the University of Nice. This work relies on use of the Amsterdam Compiler Kit (ACK) to ensure a full set of optimizers and code generators based on a semi-universal intermediate language, and, therefore, emphasizes the filling of the gap between parsing and the intermediate language. It is intended as a pragmatic contribution to the automation of the production of true compilers (rather than mere program evaluators) that generate efficient machine code.

KEY WORDS Compiler generation Code generation Semantic analysis Cigale Amsterdam Compiler Kit

INTRODUCTION

During the short history of computer science, programming languages have been one of the most productive areas for designers' imaginations. From time to time, tentative universal programming languages have been proposed (the most prominent being PL/I, Algol 68 and Ada). Yet, the number of different programming languages continues to grow steadily, and there is no evidence that this will change in the near future.

In the past decade, one could imagine that only a small number of different machines would soon dominate the market: a few models of microprocessors were the heart of most personal computers and workstations; major machine constructors provided full compatibility for their complete range of machines. But many new RISC architectures have been publicized recently, microprocessor series continue to develop, and there is no evidence that the number of different computers will stop increasing.

As a consequence, language implementations must continue to be built, for new languages on existing machines, or for old languages on new machines, or both. The industrial production of good language implementations is thus a major challenge.

Moreover, it seems that current academic research has more interest in the automatic construction of program evaluators than in true compilers, i.e. implementations generating efficient machine code. On the contrary, industrial users of language implementations are much more interested in optimized code than in the fully

0038–0644/92/110911–26$18.00 *Received 31 October 1989*
 Revised 13 December 1990 and 20 July 1991

automatic generation of compilers. Software shops too often produce language implementations entirely by hand, ignoring even parser generators. As a consequence, current commercial compilers are generally very far from international standard definitions, their reliability is doubtful, and they are often not maintained after delivery. Since all software products must be written in some programming language, this can mean that they rely on a very fragile basis.

We place ourselves in a somewhat restricted (but economically very important) context: we consider only conventional procedural languages, which can be translated fully into machine language (with a small run-time support, of course). For these languages, we consider that it is more important than ever to make the construction and validation of language implementations easier. By this we mean that the construction of a full language implementation for a language of realistic size should take no more than one person-year for a competent, but not exceptional, software engineer. Of course, this holds only if the language does not contain too many new features involving run-time difficulties. It is very important that industrial compilers be built using the full power of existing theories, but this must be non-contradictory with their efficiency. Building a compiler has at present most of the characteristics of a craft, [1] which makes it a fascinating challenge for interested programmers. However, this situation makes it impossible to have very good implementations of most languages on most machines. We think that building a compiler must take on most of the characteristics of an industrial process: the aim of our work is to make progress towards complete tools for automating the production of all components of language implementations, and for validating them.

THE SUCCESSIVE PHASES OF A COMPILER

A compiler may be viewed as a black box, which accepts as input a *source text,* written in some higher-level language, and translates it into an *object text,* written

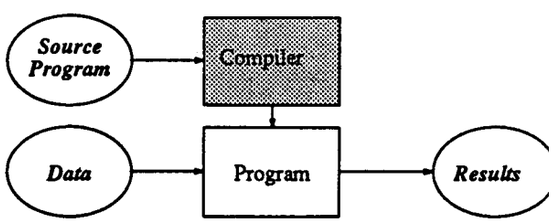


Figure 1. The compiler as a black box

in some lower-level language, understandable by some machine (Figure 1). The target machine may be an actual one, or a virtual machine, in which case the compiler translates source text into an *intermediate language.*

For the rest of the paper, and for the sake of discussion, we shall consider a language implementation as canonically divided into a number of successive phases. However, we want to emphasize that this does not mean that these phases must always constitute separate components of the language implementation. In fact, a compiler is very often syntax-directed, i.e. the parser is the main procedure, which calls the other phases as needed (they are implemented as subroutines or modules;

see Figure 2). This may be true at least for the first pass of the language implementation, if it is composed of several successive passes.

The scanner

This takes the source text as input, and transforms it into a sequence of tokens, while omitting some information, such as comments. The lexical part of programming languages may be described using regular expressions. Practical scanner generators exist; one of the most widely-known scanner generators is Lex. [2] However, it is often more expeditious to program the scanner by hand, since its work is not very complicated, is very similar from language to language, and the interface between scanning and parsing is more easily realized.

The parser

This checks the syntax of the source text, and builds or enumerates some tree representation of its syntactic structure. Parser generators have been the subject of intensive studies for a long time, and the result is currently satisfactory. The syntax of the source language is specified using a context-free grammar, for example in Yacc, [3] LLGen [4] or Syntax. [5] On the other hand, it is very cumbersome, or even impossible, to build ‘a parser manually, except when using the recursive-descent method. [6]

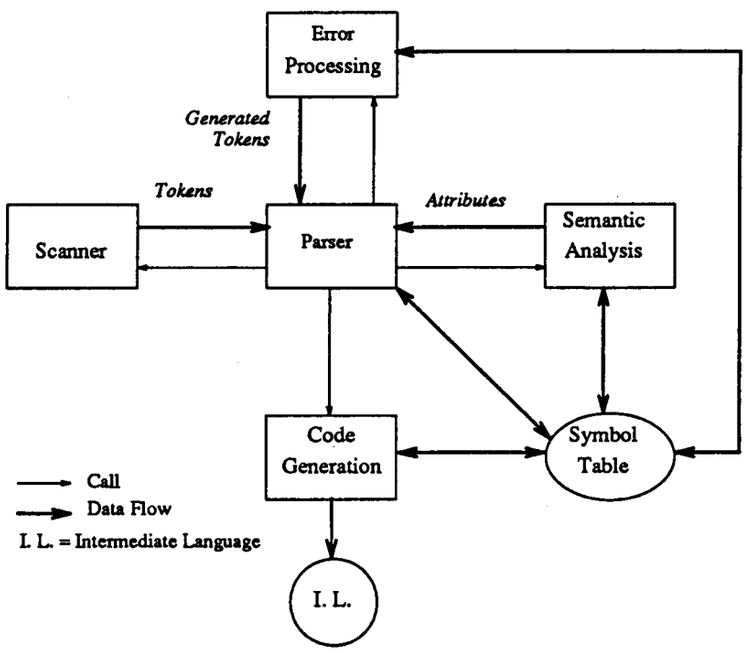


Figure 2. A syntax-directed compiler

The semantic analyser

The exact purpose of this component is subject to dispute. There are two main semantic aspects. *Static semantics* defines the set of context-dependent rules that provide for checking the legality of programs, independently of a specific set of input data. More precisely, it uses visibility rules and type-checking rules. It specifies the meaning of declarations, and of all uses of names in text. In fact, 'static semantics' is a misleading name, and this should be called 'context-sensitive syntax'. However, we cannot avoid the common usage.

Dynamic semantics provides a model for program execution. It specifies the effect of executing a statement, or the value returned by evaluating an expression, in a given environment. For example, the statement `var := exp` has the consequence of assigning to var the value returned by `exp`. The exact boundary between static semantics and dynamic semantics is fuzzy. A rule of thumb could be that the static aspects deal with declarations and identifier utilizations, and the dynamic aspects deal with evaluations and executions.

Here, we consider the semantic analyser as dealing only with static semantics. Dynamic semantics will be handled during code generation. Generators for semantic analysers are still in the domain of research. In fact, several tentatively universal formalisms, intended for describing the semantics of programming languages, have been designed,[7] but none is entirely satisfactory: denotational, operational, axiomatic, natural,[8] algebraic abstract data types,[9] W-grammars, production systems, attribute grammars, or coupled attribute grammars[10] are the most prominent formalisms. At present, the most popular formalism is one of the simplest, i.e. attribute grammars. Several semantic-analyser generators are based on them: GAG,[11] HLP/TOOLS,[12,13] MUG2,[14,15] FNC-2[16] or the Synthesizer Generator.[17]

In most cases, these systems or generators do not result in true compilers, but instead they yield program evaluators, i.e. a special case of interpreters. The advantage is that these systems are inherently portable, since they do not depend on any target machine. They actually rely on some abstract machine, whose machine language is the language in which they are programmed. The price to pay is high inefficiency. Moreover, the implementation language must have been implemented in some other way.

The code generator

This component's responsibility is to generate (hopefully) efficient object code. One solution is to produce target-machine code directly. Building a very crude code generator is feasible, but trying to generate code that uses the full power of the target machine is generally a major challenge. The other solution is to produce code in an intermediate language. The code generator is much simpler than in the first case, since the target code has been designed with this in mind. The part of the language implementation that translates source text into intermediate code is machine-independent, and can be used on several machines.

Thus we choose this second solution, because it greatly increases portability[18] and facilitates reusability of implementation components. Moreover, many very interesting optimizations can be done on the text in intermediate code,[19] and they present the very important advantage of being independent both of the source

language and the object language. We choose EM, [20] a successful attempt at a source language-independent intermediate language (see Appendix I).

Code-generator generators are not numerous: see for example References 21, 22, and 23. Current research is somewhat scarce, and much more progress is needed.

THE TOOLKIT USED IN NICE: CIGALE

Cigale is a compiler-writing system, written in Pascal, and producing compilers written in Pascal. It is the last instance of the system initially designed in 1974, at the University of Montreal, by one of the authors. [24]

Cigale accepts as input a so-called *integrated description*, which contains among other things, an attribute grammar of the language to be compiled. Its output is a compiler for this language (see Figure 3).

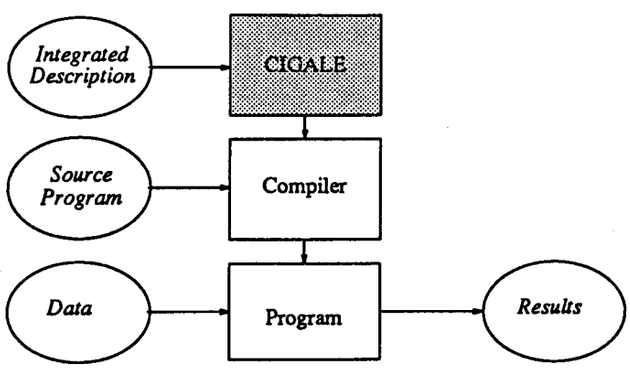


Figure 3. Cigale viewed as a black box

The integrated description

It is divided into three main parts:

1. A set of options for the various modules of Cigale.
2. Global declarations, written in Pascal, which describe the objects (constants, types, variables, procedures) needed for semantic analysis and code generation. These declarations can also be stored in several independent modules, and imported from the integrated description using `import` statements. This makes use of the separate compilation capabilities of most current Pascal compilers.
3. An LAG(1) attribute grammar, i.e. an attribute grammar where all attributes are evaluated in only one pass from left to right.* A complete example of an integrated description is given in a later section.

---

* The terminology is not fixed: LAG(1) is the acronym used in Reference 25, whereas in Reference 26 the term 'L-attributed description' is preferred. Moreover, as we show later, we are actually using a *syntax-directed translation schema*, rather than a purely declarative attribute grammar.

The components of Cigale

Cigale has five main modules, which can be executed in a relatively independent way from each other (see Figure 4). Four are described below.

1. Lexigen is the scanner generator. It chooses the parts of a skeleton scanner that must be included in the resulting compiler, according to the specific terminals used in the language to be compiled. For example, if this language does not contain real constants, the corresponding procedure is not included. It also builds the tables that describe operators, reserved identifiers or keywords. The generated scanner has the capability of recognizing some specific token categories, and an extensive set of options allows users to modify their denotation, for example by choosing the delimiters for strings, the maximum size for integers, etc.
2. Syntgen is the parser generator. It uses a bottom-up method based on weak

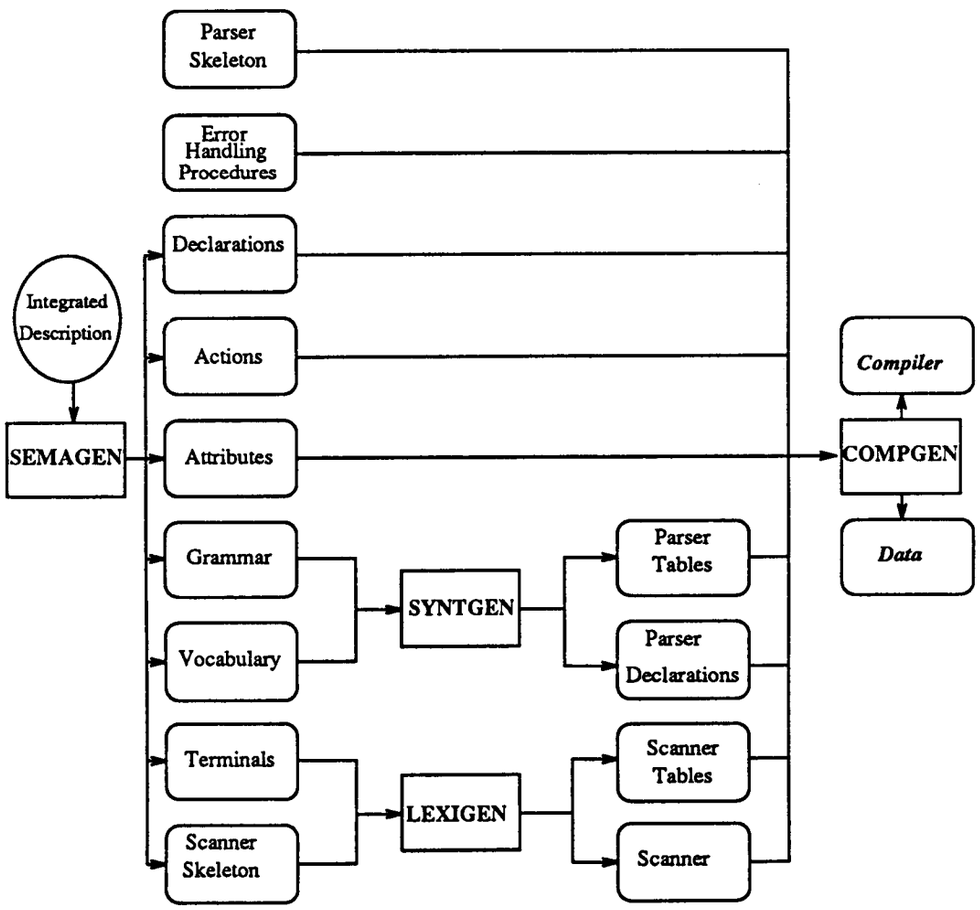


Figure 4. The main components of Cigale

precedence relations, extended with left-hand contexts (with the capability of a BC(*n* ,l) method). It takes as input the encoded grammar and its vocabulary.

The generated parser is an interpreter of Floyd–Evans productions. It includes a full syntax-error recovery mechanism, derived from the Graham–Rhodes [27] method. This mechanism proceeds in three steps: the first one locates the error, the second one condenses the left-hand and right-hand contexts at the error location, and the last one replaces the offending sentential form with a least-cost correct one.

Right-hand part conflicts in the grammar are solved using a left context, if possible. When this cannot be done, Syntgen gives hints to the users, and allows them to resolve the resulting ambiguity by using special semantic actions. Since these actions can examine the whole parsing stack, look ahead in the source text, or use available semantic information, there is no real limit to what they can do.

3. Compgen merges all parts generated by other components, and produces the Pascal source text of the generated compiler. It also merges the tables generated by other components, into a file read by the compiler at initialization time.
4. Proggen (which does not appear in Figure 4) generates test cases, and has no consequence for the compiler generation. However, it is very useful for testing the compiler, and overall for checking the apparent validity of the grammar. It produces a set of syntactically correct phrases, derived from the grammar and using all productions and all transitions between productions. These phrases contain no attempt at semantic correctness, but it would be a major challenge to try removing this weakness. Proggen also produces a linearization of the derivation tree for all generated phrases.

In order to provide for separate compilation of the generated compiler, the Unix-dependent version of Cigale provides several small components in addition to the preceding ones. They are used for:

(a) generating external procedure declarations for all first-level procedures of a Pascal module
(b) generating a ‘makefile’ for building at will the various parts of the generated compiler
(c) transforming the import directives in source texts for referring to generated files.

AIDS FOR CONTEXT-SENSITIVE SPECIFICATION

Cigale allows users to attach typed attributes to every non-terminal of the grammar, similar to procedure parameters. These attributes are synthesized, i.e. they are result parameters. However, it is also possible to refer directly to attributes of non-terminals already encountered during parsing. They are called context *attributes*, and give the full power of inherited attributes in a strictly left-to-right evaluation. Moreover, users can program semantic actions, executed just before right-hand part reductions, i.e. just after the right-hand part of a grammar production has been recognized. These semantic actions can assign values to left-hand attributes, using right-hand and context attribute values, and more generally do any useful work.

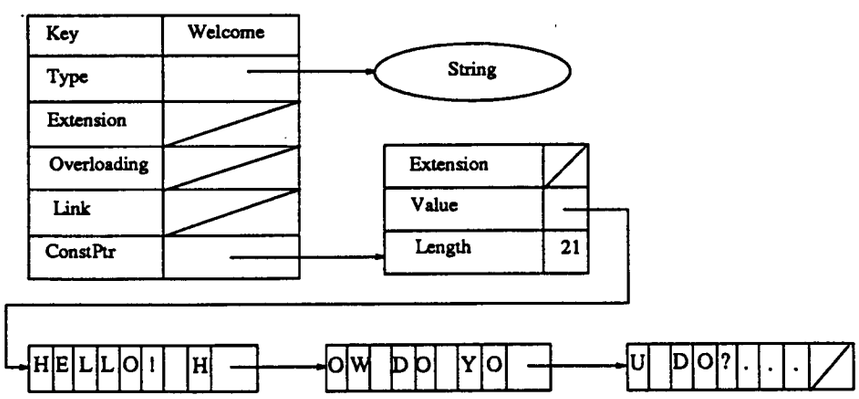
For every new language, compiler writers must (among other things) design and

describe a symbol table, along with all its access and handling procedures. Thus, we considered it useful to define for Cigale users as general a built-in symbol table as possible, i.e. a symbol table that would encompass the various capabilities of most existing programming languages. The structure of this symbol table is based upon a trio of concepts found under various names in all programming languages: we call them *names*, *types* and *values.* The symbol table is a collection of name descriptors, which contain general-purpose information (key, scope level, etc), as well as more specific information related to the name class. For example, a constant-identifier descriptor contains a pointer to a value descriptor (which gives the type and value of the constant); a parameter-identifier descriptor contains its transmission mode, its type, etc. Figures 5 and 6 show the representations of two simple examples.

Organization of the symbol table (see Figure 7)

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 names pertaining to the same scope or declaration level. In order to achieve this, we use an array Level (indexed with the level number), where every element is a list of doublets, which point to the name descriptors of this level. An additional indirection (instead of a simple list of names) is needed for languages where a single name may belong to several scopes at the same time (in Modula-2, for example). With this representation, it is easy to erase all names of a given level, when exiting a block, for example. Another level of indirection is needed for overloaded names: in the same block, two identical names may be used for two different objects, for example in Ada or in polymorphic languages.

Another array, Homonyms, indexed with the unique keys given by the scanner, serves to access all names that are homonyms. Thus, to search for a given name is very fast, since one accesses the list of overloaded name descriptors directly, and has only to check its scope level. If necessary, overloading resolution is then carried out .



const *Welcome* = 'Hello! How do you do?'

Figure 5. Representation of a constant declaration

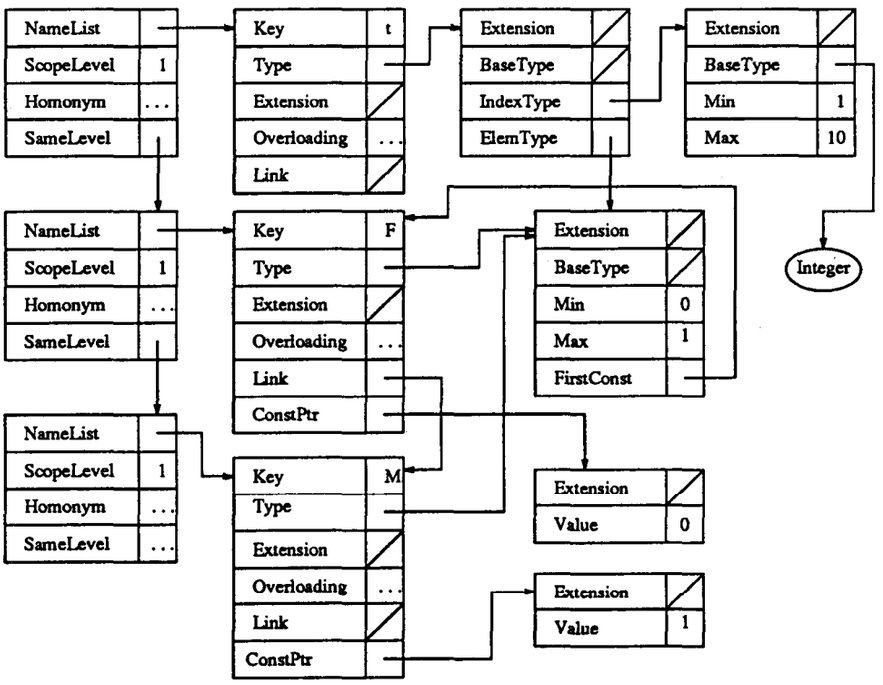


Figure 6. Representation of a type declaration

Use of the symbol table

It is important to note that the structure of the symbol table is not frozen: users can extend it at will. In fact, a name descriptor is a Pascal record, and we have defined in it a field Extension, as a pointer to a record type NameExtension, which users can fill as they like. They will also define the management procedures related to this extension. The same feature is defined for value and type descriptors. Another important remark is that, whatever generality we aim at, we cannot pretend to be capable of implementing any language. As stated at the beginning of this paper, we consider only conventional procedural languages. Even with this important limitation, the price of generality cannot be ignored.

One of the intended main aims of this method is that the internal contents of the symbol table are hidden from users. As many primitive procedures as necessary have been defined for bookkeeping the table and accessing the information it contains. Some procedures correspond to general actions, others are more specialized and handle only constants, types, variables, subroutines, or expressions. Some procedures are provided for declaring a name in the current block, or for accessing a declared name. Others serve to create integer values, or values of any other type. Figure 8 shows some examples. The result is something like a complex abstract data type.

It is interesting to remark that this approach has many similarities with the object-

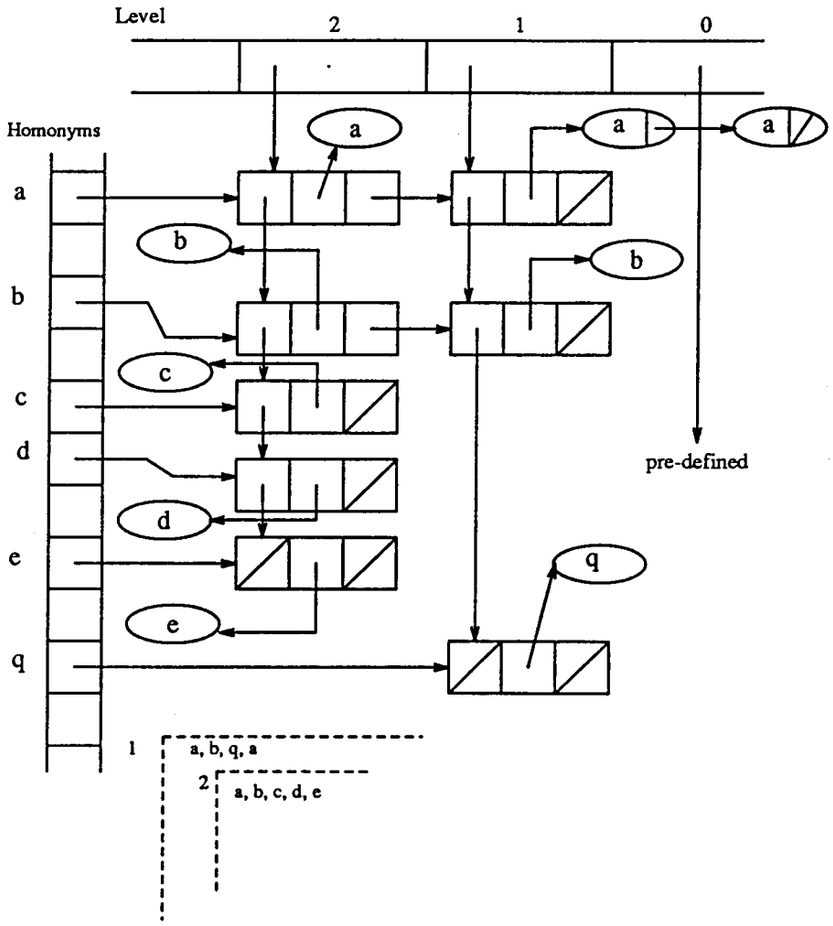


Figure 7. Organization of the symbol table

oriented one. We are aware of this, but we preferred not to include object-oriented concepts explicitly in our system. This has been already done in the TOOLS system. [13]

AIDS FOR CODE GENERATION

Most compiler-writing systems provide nothing for code generation. However, this part is of the utmost importance in industrially-used compilers. Software vendors are interested in efficient generation of very efficient code for the software products they build.

This part of a compiler is very often the most complicated one, for several reasons:

(a) no aid is available for specifying and building it
(b) the object language is complicated, full of special cases, etc.
(c) generation of efficient object code is much more complicated than generation of systematic but inefficient code.

• General procedures

DeclareName : KeyType × IdentClass × Boolean → NameDescPtr

{if the ident whose key and class are given does not exist in the symbol table at the current level, a name descriptor corresponding to this ident is created and returned. If the boolean is true (that means overloading accepted), the name descriptor is created and returned without checking.}

FindSymbol : KeyType → NameList

{returns the name descriptor list corresponding to the given name key, if it exists in the symbol table at the current level, nil otherwise.}

• Constants

IntegerValue : ValueDescPtr → Boolean

{yields true if the value described by the value descriptor is of type Integer.}

There exist similar functions for the other value types: RealValue, StringValue, etc.

CreateIntegerValue : Integer → ValueDescPtr

{creates a value descriptor with the integer value.}

PutConst : NameDescPtr × ValueDescPtr → NameDescPtr

{initializes the name descriptor with the value descriptor.}

ConstName : NameDescPtr → Boolean

{returns true if the name descriptor describes a constant.}

• Types

CreateIntegerType : TypeDescPtr × Integer × Integer → TypeDescPtr

{creates a type descriptor, whose subtype and bounds are specified.}

There exist similar procedures for the other types CreateRealType, CreateSetType, CreatePtrType, CreateRecordType, etc.

• Expressions

ComputableExp : ExpDesc → Boolean

{yields true if the expression given is computable at compile-time.}

IntegerOperation : ExpDesc × ExpDesc × OpType → Integer

{returns the integer result of the specified operation on the given left and right operands.}

CreateIntegerConst Exp : Integer → ExpDesc

{yields the expression descriptor corresponding to the given integer constant.}

CreateVarExp : → ExpDesc

{yields the expression descriptor corresponding to a variable expression.}

There exist similar functions for real expressions, etc.

Figure 8. Examples of some procedures

In the case of EM, as in the case of most intermediate languages, the second reason does not hold, since the language was designed with facility of generation as a primary aim. However, the third reason is still important, and an elaborate code generator for EM remains a major challenge: the compiler must recognize and handle all special cases, subject to optimization, in order to choose the best instruction sequence in every situation.

However, a much more interesting situation occurs here, thanks to the remaining components of ACK, and to the design objectives of EM. If code generation in the compiler is limited to the kernel of the EM instruction repertoire, the EM architecture presents more characteristics of a RISC architecture. Thus, code generation no longer has to consider special cases, and can be made according to very simple and systematic guidelines and principles. The immediate consequence is that generated code is very cumbersome and inefficient, but the results of the ACK optimizer, applied to such a code, are at least as good as those that could have been produced directly by an elaborate code generator. In fact, the ACK local optimizer searches in the code for the sequences that can be optimized, using a pattern-matching technique, and it can do this more generally and exhaustively than a code generator.

x:=1.0	.3	.3
	rom 1.0F8	rom 1.0F8
	lae .3	lde .3
	loi 8	sde 0
	sde 0	
z[1]:=x	lae 12	lae 0
	loc 1	lae 12
	lae .2	blm 12
	lae 0	
	exc 3, 1	
	aar 4	
	blm 12	

Figure 9. Optimization examples

The example in Figure 9 demonstrates this (column 1: statement in high-level language; column 2: EM code; column 3: EM code after optimization).

In the first case, an address load (lae) followed by an indirect (loi) is replaced by a direct load (lde). In the second case, an indexed address load with a constant index (lae, loc, lae, aar) is replaced by an address load (lae 12) . Moreover, two code sequences have been exchanged in order to stack operands of the array address computation (aar) and then of the block move (blm) in the required order. This demonstrates the usefulness of the optimizer.

These considerations being made, our approach to providing aids for code generation has been similar to the one adopted for context-sensitive analysis, i.e. a very pragmatic one. Instead of designing a high-level semantic specification *ab abstracto*, and then trying to use it in our existing framework, we have preferred to proceed in the following way:

1. isolate and specify the minimal instruction repertoire and set of addressing modes of EM.

2. Complete the abstract data structures with the information needed.
3. Design and build the primitive procedures necessary for handling these structures and generating instructions and addressing modes of the minimal repertoire.

Using the same approach as for context-sensitive analysis allows us to take one step more towards a simpler and more declarative notation. In the case of code generation, universal programming concepts occur mainly for statements, where most code is generated.

There exist procedures for generating labels, handling variable addresses and type sizes, as well as procedures for expression evaluation. For statements, there are procedures for generating an assignment or a branch to a label, among others (see Figure 10).

With EM, we can generate code 'on the fly', thanks to the EXC pseudo-instruction, which exchanges two instruction sequences. It is still necessary to split the rules in order to generate code in the right place. Predefined modules for expressions and statements are provided in Cigale, so users can re-use them and only modify the lexico-syntactic parts as needed.

A COMPLETE EXAMPLE

As an illustration, we now give a full example for a tiny language, which prints the result of the evaluation of an if–then–else condition. Constant integers can be declared. as well as integer variables whose value is read at run-time. The corresponding grammar in EBNF is

● **NewLabel : → Integer**
{yields a new instructions label.}
DefineLabel : Integer →
{puts in the code the given instruction label.}
There exist similar procedures for producing instructions.

● **ProdAssign : ExpDesc × ExpDesc →**
{generates code for assigning the second given expression into the first one:}
There exist similar procedures for case statement, for statement, if statement, loop statement, etc.

● **PushValue : Exp Desc →**
{stacks the given expression value, according to its class.}
PushAddress : NameDescPtr × LevelRange →
{generates code for stacking the address of the given object, declared at the given level.}
SizeAndAddrFields : NameDescPtr × Integer → NameDescPtr × Integer
{computes the size of the given field list and updates the address of every field; returns the updated list and the computed size.}
There exist similar procedures for parameters and variants.

Figure 10. Examples of some procedures

```
Axiom : definition { ','definition} 'IN' condition '.'.
definition : ident ':=' number | '?' ident.
condition : ' IF' relation 'THEN' expression 'ELSE' expression 'FI'.
relation : expression reloper expression.
expression : term {addoper term}.
term : factor { muloper factor }.
factor : number | ident | '('expression')'.
reloper : ' =' |'<>'|'<' | '<=' |'>' |'>='
addoper : '+' | '–'.
muloper : '*' | '/'
```

An example phrase in this language is

```
?a, b:=30 IN IF a(b THEN b ELSE a FI.
```

which yields 40 when the value read for a is 40. Thus, one can define constants, then use them in an expression. If the expression can be evaluated (all names are declared, no overflow), its result is output.

Appendix II gives the full integrated description for this tiny language. First (lines 1–4), we state options for the different modules of the system. After a delimiter ($), the description contains the declarations provided by the user. There is one constant declaration (line 7) and four variable declarations (line 10).

Then, several external declaration files are mentioned by import commands (line 12). This facility makes use of the separate compilation capability of the Berkeley Pascal compiler we use on the system Ultrix (DEC Vax), but similar features exist in most Pascal compilers. Files declarations, static and dynamic are provided by the system, and contain general-purpose declarations similar to those shown above. They are in a built-in Cigale directory.

In this example, the user has needed some additional procedures, which are given directly in the description (lines 14–36), but could equally well have been put in an external file (and called by an import command).

Keyword grammar (line 38) introduces the attribute grammar. Its syntactic aspects are similar to EBNF. The general form of a rule is

```
left-part = right-part : semantic action $
```

Non-terminals must appear at least once in the left part of some rule. Terminals are enclosed in special delimiters (generally simple quotes), or are predefined identifiers.

Non-terminals and predefine terminals may have attributes, which appear like formal parameters in left-parts, and actual parameters in right-parts. Attribute names are local to the rule in which they occur. They are used in semantic actions, and translated into references to the semantic stack, parallel to the parsing stack.

Semantic actions may begin with local Pascal declarations: they take the form of a procedure body where the body keyword begins the body. If there are no local declarations, this keyword may be omitted.

Non-terminal Line (lines 40–41): we call procedure EndCode for generating the program epilogue.

Non-terminal LineHead (lines 43–45): after analysis of the declarations (in this case, only a context-sensitive analysis is needed), a call to procedure MainProg is needed for generating the allocation of the global area and the opening of standard files. Before generating any instruction, we must produce those that read integer values for every declared variable. This is done by procedure ProdReadVar, called with the list of declared variables (built line 56) as argument.

In the handling of the conditional statement, the three actions needed for generating evaluations and branches will be described in the next section. A call is made to a procedure which outputs the value at the top of the stack.

Non-terminal Factor (lines 102–118): if it is an integer, an integer constant expression is created. Otherwise, we must check whether it exists in the symbol table (FindGoodSymbol). If it is a constant, the corresponding integer constant expression is created, with no code generated. If it is a variable, we must stack its address and create a variable expression. If the name does not exist, a non-declaration error is detected; we repair it by declaring the name as a zero constant. If the factor contains a unary operation, we call the procedure that checks its type and generates the code needed.

In lines 120–121, ident and integer are declared as predefine terminals. In lines 123–128, the pseudo-rule Initialize serves to provide statements to be executed just before beginning parsing. This can be used for initializing tables, for example. Here, it serves to define stop and restart symbols for *panic mode* error handling (used when the more elaborate error-repair mechanism fails). Note that references to grammar symbols may occur in semantic actions, enclosed in delimiters '(/' and '/)'. Then, we call the procedure that initializes the code file (Begin Code). The initialization part serves also to declare predefine types, here just integer and boolean.

In summary, when users want to develop a compiler for a new language, generally they do not begin from scratch. Several modules exist in Cigale for handling the main parts of programming languages. They include all general paradigms, and the predefine tools make their processing very easy. The following basic ideas can be isolated:

1. Separate lexical aspects from syntactic aspects, and specify parameters to Lexigen.
2. Build a grammar that can be handled entirely by Syntgen; this may need several attempts if the language is a complex one, and if its grammar has been written without paying intention to parsing constraints.
3. In the case of reluctant residual ambiguities in the grammar, program semantic actions to solve the problem.
4. Associate semantic attributes to most non-terminals, using predefine types of the predefine symbol table.
5. Split some grammar rules in order to place semantic actions at proper locations, and write these actions so that most should reduce to a call to some predefine procedure.

Because the predefine modules can be re-used, most of these steps have to be done in full only for new constructs of the language that are not general enough to have been included in the predefine modules. For most other constructs, users have only to modify lexical or syntactic details in the modules in order to adapt to the

specific programming language they want to compile. Thus they can pay more attention to the new, original, and probably more difficult aspects of this language.

After all these steps, a full compiler has been built.

A DECLARATIVE NOTATION

As one can understand from the preceding sections, the current set of procedures, when used in semantic actions of the integrated description, provides an operational specification of the source-language semantics (context-sensitive syntax and dynamic semantics). After our experiments with this set of primitives, it became evident that some semantic actions amount to a single procedure call, some others are more complicated, but always the same in most language descriptions.

We decided to design a more declarative notation, from which the first module of Cigale could deduce what specific primitives should be called. The first step was to isolate frequent paradigms, present in most language definitions. For example, the concept of constant declarations is a very stable paradigm, which can be described in an operational way by calling specific primitives with specific attributes, but which can also be built into a new declarative notation, as a primitive concept. The second step was to design this notation, so as to allow users to program most implementation details, and to concentrate only on the important concepts of the language to be compiled.

Using this notation, the implementation language is hidden from the Cigale user, which allows us to change it at will. At present, the notation is completely specified, but not yet implemented. We now present some examples, in order to give a flavour of its characteristics.

In the case where a ‘factor’ is a procedure, the corresponding rule in the integrated description will become:

```
factor ( e: ExpDesc)
        = ′ procedure ′ formalParameters(pnl, ptl) procBody (dp) :
        e ← constantExp ofType subroutineType ptl
          ofValue subroutineValue pnl, dp $
```

The synthesized attribute e must be a constant-expression descriptor with some type and some value. The type must be ‘subroutine’, and is specified by the list of formal parameter types. The value is a subroutine value, specified by the list of formal parameter names, together with a procedure description: reference to the body code, size of the local region, declaration level.

Another example is the conditional statement. We note that it is specified by three different actions. The first one must be performed when the condition expression is parsed; it is used for creating the else label and generating a *branch when false* to this label. The second action must be carried out when the then part is parsed; it creates the end label and generates a branch to it, then defines the else label. The last one, which must occur after the whole statement is parsed, defines the end label.

Thus we have three specific actions, BeginIf, MiddleIf and EndIf, which are used in the following:

```
statement
```

```
        = if Begin(endl) if End : Endlf endl $
if Begin(endl: integer)
        = exprThen(elsel) statementList : endl ← Middlelf elsel $
exprThen(elsel: integer)
        = ′ if ′ expression(e) ′ then ′ : elsel ← Beginlf e $
```

A complete specification of the notation would not be useful in the present paper. We anticipate no special problem for the implementation: we know what procedure calls must be generated for every statement, thanks to our bottom-up approach. Moreover, the current version of Cigale will be used for this implementation.

The notation will provide for a more declarative specification of programming-language implementations. It will be the basis for a validation tool for the integrated description. Attribute references and attribute assignments are clearly separated; thus it will be easy to check whether the attributed grammar is LAG(1): for a given production, a value must be assigned to every left-hand attribute, using only the values of right-hand or context attributes (or possibly already-assigned left-hand attributes).

CONCLUSION

Assessment

The small example given above cannot be enough for evaluating the extent to which we have attained the goals set out at the beginning of this paper. In fact, we have made two full-size experiments with actual programming languages. The first one is Oberon, [28] for which compilers were built by undergraduate students using Cigale, in the framework of a part-time project. The second experiment is more significant, since it deals with a language outside the Pascal family. Moreover, this experimental language, called Leda, was not known to us before the exercise. The detailed results are described elsewhere; [29] we give a summary of them here.

In order to obtain an operational compiler, generating operational code, for all parts of the language except those that produced difficulties at run-time, we needed about three person-months. This included discussions with the author of the language, since its design was not frozen (and is still fluid).

For making significant comparisons, we wrote programs in the subset of Leda that is equivalent to Pascal, and have compiled and executed them with our Leda compiler, the ACK Pascal compiler, and the Berkeley Pascal compiler. Despite the fact that this third compiler is written in C, we found compilation timings to be equivalent for Leda and Berkeley Pascal, ACK Pascal being 30 per cent faster. These timings include the whole translation from source text to executable text. Execution timings were equivalent for the three implementations.

In the next version of Cigale, we intend to abandon Pascal as an implementation language, and probably shift to C^{++}. We expect further improvements in compile-time performance, which will make Cigale highly competitive, while placing very small demands on the host computer.

Compared to more ambitious systems, Cigale has the fundamental advantage, in our opinion, that it disappears entirely after generating the compiler. Thus, the generated product can exist without support from the tools used to build it, and this

is true also for the programs compiled by the generated compiler. This makes Cigale suitable for building compilers on personal computers, or more generally in any situation where simplicity and frugality are important.

Summary

For many years, the idea of an automatic compiler-generator has been an ideal that many teams have tried to achieve. Many years ago, this was imagined as a sort of magic box, in which one would input formal descriptions of the three languages involved in a language implementation: source language, target language and implementation language. The result, produced completely automatically, would be a full compiler from source language to target language, written in the implementation language (see Figure 11).

One of the main obstacles encountered when trying to achieve such a tool is the true idea of a formal description of a programming language. While lexical and syntactic aspects have been given successfully some almost perfect means of description, the semantic aspects are much less tractable. Numerous formalisms have been designed, but all attempts to build compiler-generators around 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. To avoid machine-dependence and code-generation problems, they rely on the abstract machine provided by some higher-level language such as LISP or Prolog. At best, they constitute interpreter-generators, i.e. we obtain the production scheme of Figure 12. This is acceptable only for experimental languages, when run-time performance is not important. At worst, they constitute parametrized program-evaluators, along the lines of Figure 13. The supporting system is present when executing programs, and run-time performance is tolerable only for small demonstrations or experiments, not for daily use.

This results from the very ambitious goal generally aimed at by most current research teams: to provide only a full formal definition of the source language, with no operational parts and no reference to any object machine. Of course, this cannot

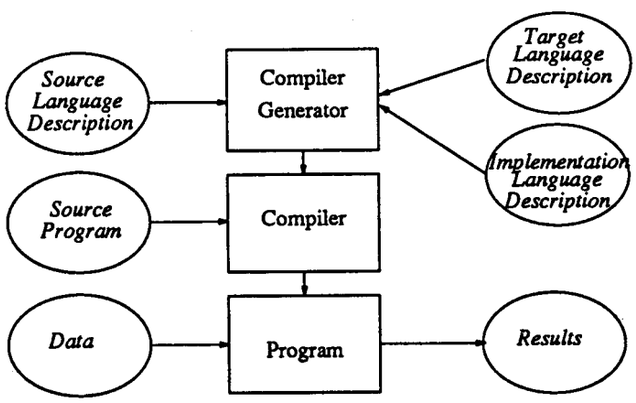


Figure 11. An ideal compiler-generator

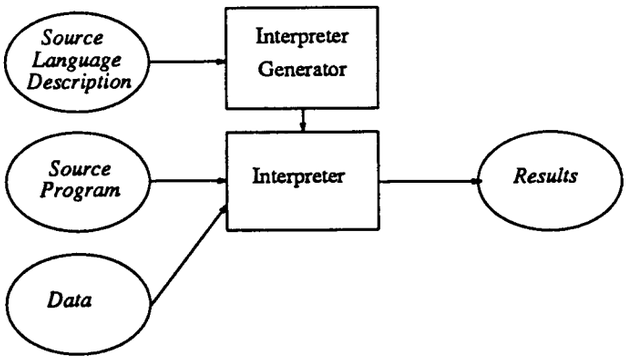


Figure 12. An interpreter-generator

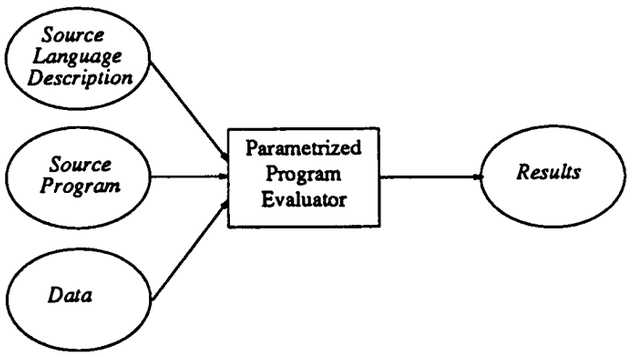


Figure 13. A parametrized program evaluator

result in something comparable with any industrial compiler, either in flexibility or in efficiency.

Our approach is much less ambitious than the one just summarized, but we believe it to be also much more realistic. Moreover, we think it is at present the only one that can help to make progress towards the automatic construction of validated and efficient implementations of languages. The idea of the magic box is forgotten, since the implementation language is built into the system. Moreover, we provide a full set of specialized components, instead of a single general-purpose program. When these components already exist, we use them if they are satisfactory, even if they were made by other people. Some components are the same in any language implementation. Other components are generated automatically from formal or semi-formal descriptions.

The same idea is used when Cigale users are building an integrated description: instead of trying to write it entirely from scratch, they can make full use of re-usable existing modules. Thus we are not trying to a give a fully automatic compiler-generator. Rather, our final aim is what we could call a *compiler building and assembly workshop.* Adding components and tools to such a workshop, we think,

will be more productive and realistic than periodically designing a completely new integrated and monolithic compiler-generator.

APPENDIX I: ACK, THE AMSTERDAM COMPILER KIT

ACK is a set of tools, [20] designed and built at the University of Amsterdam, which provides for the construction of very efficient and portable back-ends for compilers. Its design is centred around EM, an intermediate language which can be microprogrammed, or interpreted, or translated into machine code (the most frequent situation). A compiler built using ACK has the structure that appears in Figure 14.

The *front-end* can typically be built using available scanner and parser generators

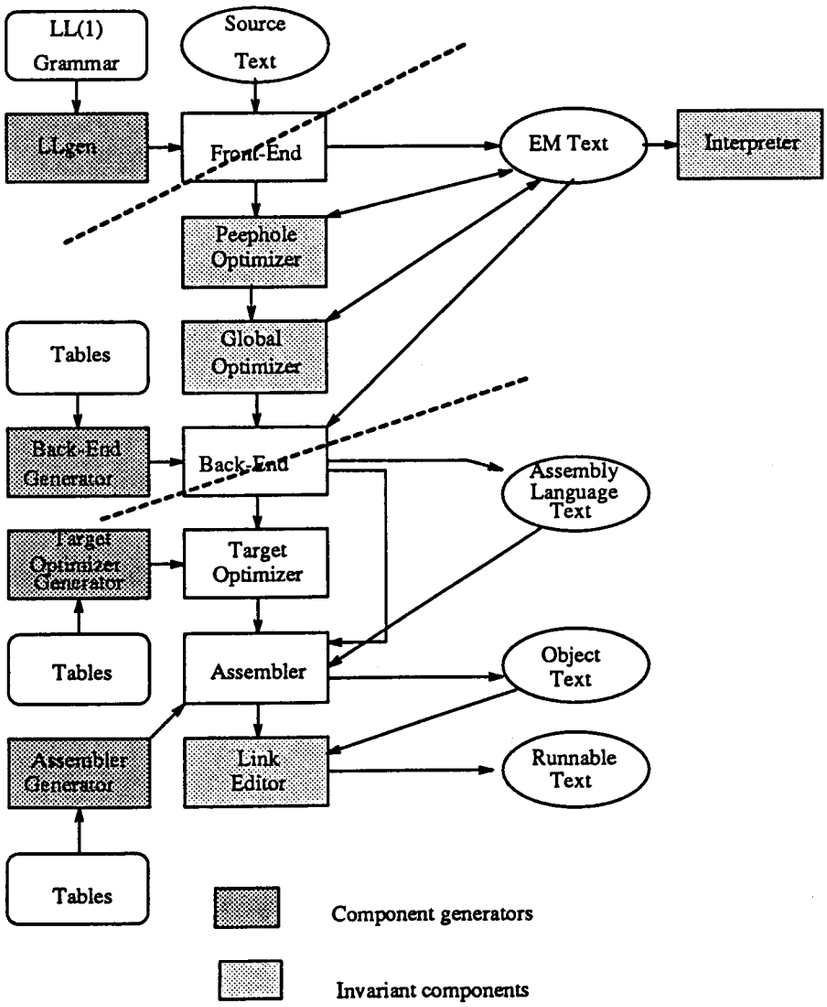


Figure 14. ACK components

(for example, the Cigale toolkit). It translates source text into EM intermediate language, and is the only part of the compiler that depends on the source language.

The *peephole* and *global optimizers* depend only on EM, and consequently are identical in every compiler. Moreover, the global optimizer can be bypassed when maximum efficiency of the target code is not necessary.

The *back-end* translates EM code into assembly code. It is generated using a code-generator generator, from a symbolic description of the target machine and its assembly code.

The *target optimizer* depends only on the target machine, and is generated in the same way as the back-end, from another description. Of course, this component is not necessary, and often does not exist at all.

The *assembler* and *link editor* can be those of the existing machine (which is the case, for example, on the DEC Vax), but a universal assembler and a universal link editor also exist, parametrized with a description of the target machine, and used in the case of cross-compilation.

The EM language is that of a stack, byte-addressed machine, with separate store memories for instructions and data. Data store is structured in words: a word is made of one, two, four, or eight consecutive bytes. Instruction and data stores are divided into fragments, whose relative location is undefined. The instruction set is very rich, but based on a small minimal set of about 30 instructions. Data-storage organization includes the notions of evaluation stack, procedure stack frames, and allocation heap. There are specific instructions for access to array components, integer and real arithmetic, local and global-variable access, procedure call and return, etc. The abstract-machine design even includes the concepts of traps and exceptions. On the other hand, there are no built-in input–output operations, but it is assumed that monitor calls similar to those of Unix can be executed.

The main advantages of ACK are the following:

1. The dependencies of the compiler components from source, intermediate, and target languages are very neatly separated, as shown in Figure 14 by dashed lines.
2. The three optimizers provided can produce object code with extremely good performance.
3. The design of the EM abstract machine is suited to most current machine architectures.
4. Numerous back-ends have already been built, for most current microprocessors.
5. EM is by far the most serious and usable attempt at achieving the Uncol ideal.
6. Back-ends, target optimizers, assemblers, and link editors can be generated automatically, or parametrized, for new target machines (this is useful for cross-compilers for microprocessors).
7. It is also possible to use native assemblers and link editors.

The current drawbacks of ACK are the following:

1. It is proprietary software.
2. The architecture of the abstract machine is already more than ten years old. This explains why some details now seem to be disputable, and probably would be designed in a different way in an abstract machine defined today.
3. Some uncommon language constructs cannot be translated easily into EM code. During our experiments, we encountered difficulties, for example, in

implementing the goal-directed evaluation in an experimental language similar to Icon [30] This needs to copy explicitly parts of the evaluation stack, although EM provides no instructions for doing this in any simple way.

4. The cost of the many successive phases of the compilers produced makes them somewhat slow: at least five complete passes are needed, when the global and target optimizers are bypassed. However, recent work [31] allows us to use a variant of this scheme, which generates object code directly along the interface provided by EM. This yields fast compilation when run-time performance is not needed.
5. Although the aids provided for building back-ends and other target-dependent modules are very useful, they do not provide for validation of the generated modules, and building a new back-end still remains an important and difficult job.
6. The only aid for generating front-ends that ACK provides is LLGen, which is somewhat limited.
7. To change even a detail in the design of the abstract machine (for example, adding an instruction) would be a major challenge, since almost all components of ACK more or less depend on the intermediate language EM.

APPENDIX II. INTEGRATED DESCRIPTION FOR THE TINY LANGUAGE

```
1   task*lexigen(groupedblanks);
2   syntgen(conflicts, ambiguities, unsolvable);
3   proggen(sentences)
4   $

6   const
7     stopsymbolnumber = 2;
9   var
10    i : NameDescPtr; niv : LevelRange; e : ExpDesc; fv : NameQueue;
12  import declarations, static, dynamic;

14  procedure Error(number:integer);
15  begin
16    case number of
17      1: writeln('Expression not good');
18      2: writeln('Variable not declared')
19    end
20  end; {Error}

22  procedure WriteExpr(e:ExpDesc);
23  begin
24    if not ExpError(e)
25    then begin ProdWrite(e); ProdWriteln end
26    else begin Error(2); Prod EndCode(76) end
27  end; {WriteExpr}

29  procedure Prod ReadVar(v:NameDescPtr);
```

```
30  begin
31    while v () nil
32    do begin
33       e : = BldVarExp(PTypEntComp, v); e.Line : = CurrentLine;
34       PushAddress(v, GlobalLevel); ProdRead(e); ProdReadIn; v : = v ^.Link
35       end
36  end; { Prod ReadVar}
38  grammar
40  Line
41  = LineHead ' IN ' Condition ' . ' : EndCode $
43  LineHead
44  = DefPart :
45  MainProg; ProdReadVar(QNtoLN(fv)) $
47  DefPart
48  = Definition $
49  = DefPart ' , ' Definition $
51  Definition
52  = ident(n) ' := ' integer(v) :
53  ConstantDecl(i, n, IntegerType, BldIntCstExp(v), false) $
54  = ' ? ' ident(n) :
55  VariableDecl(i, n, IntegerType, false);
56  if not IdentError(i) then AddFN(fv, i) $
58  Condition
59  = Beginlf(endl) ' ELSE ' Expr(e) ' F1 ' : WriteExpr(e); ProdEndlf(endl) $
61  Beginlf(endl: integer)
62  = ExprThen(elsel) Expr(e) : WriteExpr(e); ProdMiddlelf(endl, elsel) $
64  ExprThen(elsel: integer)
65  = ' IF ' Relation(e) ' THEN ' : PushValue(e); ProdBeginlf(elsel) $
67  Reiation(e: ExpDesc)
68  = ExpRel(e1, op) Expr(e2) : ComputeBinaryExpr(e, e1, e2, op)$
70  ExpRel(e:ExpDesc; op:TypOp)
71  = Expr(e) RelOp(op) : PushValue(e) $
73  RelOp(op:TypOp)
74  = ' = ' : op : = oequal $
75  = ' <> ' : op := odiff $
76  = ' < ' : op := oless $
77  = ' <= ' : op := olesseq $
78  = ' > ' : op := ogreat $
79  = ' >= ' : op := ogreateq $
81  Expr(e:ExpDesc)
82  = Term(e) $
83  = ExpAdd(e1, op) Term(e2) : ComputeBinaryExpr(e, e1, e2, op) $
```

```
85   ExpAdd(e: ExpDesc; op:TypOp)
86   = Expr(e) AddOp(op) : PushValue(e) $

88   AddOp(op:TypOp)
89   = '+' : op := oplus $
90   = '–' : op := ominus $

92   Term(e:ExpDesc)
93   = Factor(e) $
94   = TerMul(e1, op) Factor(e2) : ComputeBinaryExpr(e, e1, e2, op) $

96   TerMul(e:ExpDesc; op:TypOp)
97   = Term(e) MulOp(op) : PushValue(e) $

99   MulOp(op:TypOp)
100  = '*' : op := ostar $
101  = '/' : op := oslash $

102  Factor(e:ExpDesc)
103  = entier(v) : e := BldIntCstExp(v) $
104  = ident(n) :
105  i := FindGoodSymbol( niv, n);
106  if i <> nil
107  then if IdentConstant(i)
108       then e := BldIntCstExp(GetIntCstVal(GetCst( i)))
109       else begin
110       e := BldVarExp(IntegerType, i);
111       PushAddress(i, GlobalLevel)
112    end
113  else begin
114       Error(3); e := BldIntCstExp(0);
115       ConstantDecl(i, n, IntegerType, e, false)
116       end $
117       = '(' Expr(e) ')' $
118  = AddOp(op) Factor(e1) : ComputeUnaryExpr(e, e1, op) $

120  ident(n:typident) = terminal $
121  integer(n: integer) = terminal $

123  initialize :

124  stopsymb[1] := (/ ',' /); restartsymb[1] := (/Depart/);

125  stopsymb[2] := (/ 'IN' /); restartsymb[2] := (/Condition/);

126  TablesInit; BeginCode;

127  DeclPredefInteger( –Maxint, +Maxint); DeclPredefBoolean;

128  InitQN(fv, nil)

129  $$
```

REFERENCES

1. C. N. Fischer and R. J. LeBlanc Jr., *Crafting a Compiler*. The Benjamin/Cummings Publishing Company, Inc., California, 1988.
2. M. E. Lesk and E. Schmidt, 'LEX—a lexical analyzer generator', in *UNIX Programmer's Manual, 2,* AT&T Bell Laboratories, Murray Hill, NJ, 1975:
3. S. C. Johnson, 'YACC: yet another compiler-compiler', in *UNIX Programmer's Manual,* **2,** AT&T Bell Laboratories, Murray Hill, NJ, 1975.
4. C. J. H. Jacobs, 'LLGen—an extended LL(l) parser generator', Department of Mathematics and Computer Science, University of Amsterdam, February 1987.
5. P. Boullier and P. Deschamp, 'Le système Syntax—manuel d'utilisation et de mise en oeuvre', INRIA, Rocquencourt, France, September 1988.
6. A. J. T. Davie and R. Morrison, *Recursive Descent Compiling,* Wiley, New York, 1981.
7. M. Marcotty, H. F. Ledgard and G. V. Bochmann, 'A sampler of formal definitions', *Computing Surveys,* **8,** 191–276 (1976).
8. *G.* Kahn, 'Natural semantics', *Research Report no. 601,* INRIA, Sophia Antipolis, France, February 1987.
9. J. A. Goguen, J. W. Thatcher, E. G. Wagner and J. B. Wright, 'Initial algebra semantics and continuous algebras', *Journal of the ACM,* **24,** 68–95 (1977).
10. R. Giegerich, 'Composition and evaluation of attribute coupled grammars', *Acts Informatica,* **25,** 355–423 (1988).
11. U. Kastens, B. Hutt and E. Zimmerman, 'GAG—a practical compiler generator', *Lecture Notes in Computer* Science, 141, 1982.
12. K. J. Raiha, 'Experiences with the compiler writing system HLP', *Lecture Notes in Computer Science,* **94,** 1980, pp. 350–362.
13. K. Koskimies, T. Elomaa, T. Lehtonen and T. Paakki, 'TOOLS/HLP84 report and user manual', *Report A-1988-2,* University of Helsinki, 1988.
14. R. Giegench, 'Introduction to the compiler generating system MUG2', *TUM-Info 7923,* Technical University of Munich, May 1979.
15. G. Bartmuß and R. Giegerich, 'Compiler development with MUG2—and introductory example', *TUM-Info 18102,* Technical University of Munich, April 1981.
16. M. Jourdan and D. Parigot, 'The FNC-2 system: user's guide and reference manual', *Release 0.4,* INRIA Rocquencourt, February 1989.
17. T. Teitelbaum and T. Reps, 'The Cornell program synthesizer: a syntax-directed programming environment', *Communications of the ACM,* **24,** (9), 563–573 (1981).
18. O. Lecarme, M. Pellissier and M. C. Thomas, 'Computer-aided implementation of language implementation systems: a review and classification', *Software—Practice and Experience,* **12,** 785–824 (1982).
19. A. S. Tanenbaum, H. Van Staveren and J. W. Stevenson, 'Using peephole optimization on intermediate code', *ACM Transactions on Programming Languages and Systems,* **4,** (1), 21–36 (1982).
20. A. S. Tanenbaum, H. Van Staveren, E. G. Keizer and J. W. Stevenson, 'A practical tool kit for making portable compilers', *Communications of the ACM,* **26,** (9), 654–660 (1983).
21. R. G. G. Cattell, 'Automatic derivation of code generators from machine description', *ACM Transactions on Programming Languages and Systems,* **2,** (2), 173–199 (1980).
22. B. W. Leverett, R. G. G. Cattell, S. O. Hobbs, J. M. Newcomer, A. H. Reiner, B. R. Schantz and W. A. Wulf, 'An overview of the production-quality compiler-compiler', *IEEE Computer,* **13,** 38–49 (1980).
23. A. Despland, M. Mazaud and R. Rakotozafy, 'Code generator generation based on template-driven target term rewriting', *Lecture Notes in Computer Science, 256,* Proceedings of the Conference on Rewriting Techniques and Applications, Bordeaux, France, May 1987, pp. 105–120.
24. O. Lecarme and G. V. Bochmann, 'A (truly) usable and portable compiler writing system', in J. L. Rosenfeld (ed.), *Information Processing '74,* North-Holland, Amsterdam, 1974 pp. 218–221.
25. W. M. Waite and G. Goos, *Compiler Construction,* Springer Verlag, New York, 1984.
26. A. V. Aho, R. Sethi and J. D. Unman, *Compilers,* Addison-Wesley, Reading, Massachusetts, 1986.
27. S. L. Graham and S. P. Rhodes, 'Practical syntactic error recovery', *Communications of the ACM,* **18,** (11), 639–650 (1975).

28. N. Wirth, 'The programming language Oberon', *Software—Practice and Experience*, **18,** (7), 671–690 (1988).
29. *C.* Fédèle and O. Lecarme, 'Computer-aided building of a compiler: an example', *Compiler Compilers '90, Proceedings of the Third Workshop*, Schwerin, October 1990, Akademie der Wissenschaften der DDR, Berlin.
30. V. Granet, 'Contribution à l'accroissement de la transportabilité du logiciel: les langages intermediaires semi-universels', *Doctorate Dissertation*, University of Nice, November 1988.
31. A. S. Tanenbaum, M. F. Kaashoek, K. G. Langendoen and C. J. H. Jacobs, 'The design of very fast portable compilers', *Sigplan Notices*, **24,** (11), 125–131 (1989).