ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

N. Wirth

A Fast and Compact Compiler for Modula-2

J. Gutknecht

Compilation of Data Structures: An New Approach to Efficient Modula-2 Symbol Files

July 1985

64

Address of the authors:

Institut für Informatik
ETH-Zentrum
CH-8092 Zürich / Switzerland

A Fast and Compact Compiler for Modula-2

N. Wirth

Abstract

In the past decades, languages of ever growing complexity have emerged, and with them compiler that are increasingly bulky, slow, and often generate code that is much less than optimal. One of the new languages is Modula-2. It offers many advanced features (in fact most of Ada's), but through its regular structure permits the design of a relatively compact and fast compiler. We present the structure of this compiler that is specified by less than 5000 lines of program and recompiles itself in less than 2 minutes. It is based on straight-forward methods of parsing, searching, and code generation.

Introduction

The first compiler for the language Modula-2 [1, 2] became operational in 1979. It had been developped at ETH Zürich on a PDP-11 computer with a 56K byte store and emerged through a succession of bootstrapping steps starting with a compiler for the much simpler, experimental language Modula [3]. The small size of the store required the use of a multipass scheme with its unavoidable overhead due to the heavy use of disk store and of generating and reparsing the output data of each pass.

The compiler was thereafter ported to the Lilith computer, which then was equipped with a 256K byte store. At several other places work proceeded on either the design of new, but similarly structured compilers, or on porting the ETH 4-pass compiler onto commercially widespread systems. It then appeared to me that with the emergence of increasingly powerful microprocessors combined with relatively large stores, most of these compilers would neither take advantage of the powerful hardware nor of the relative simplicity of the language. I decided to construct a new compiler afresh. It is the subject of this paper and has the following highlights:

- The compiler is based on the single-pass strategy. As a result it is *fast.* The gain in speed compared to the previous compiler ranges from 4 to 8, depending on the compiled program.
- The compiler refrains from the use of any overly sophisticated algorithm or technique and relies on well-known principles. It is *compact,* consisting of five modules with a total size of less than 5000 lines of source text or 30'000 bytes of object code. These figures are in marked contrast to those of most high-level language compilers.
- Instead of using special routines or post-passes for code optimization, care is taken to generate *reasonably effective code* in the first place. The characteristics of speed, compactness, and code quality are multiplicative factors of the time required for recompilation, i.e. the time the compiler takes to compile itself. Recompilation time is a good indicator for a compiler's design quality. The new compiler requires *less than 2 minutes* to recompile itself on Lilith (comparable in power to a VAX 750).
- Modula-2 is an engineering language; hence, the facility of *separate compilation* of modules is quintessential. The compiler must provide and use information to guarantee type consistency checks also across module boundaries, and it must do so without undue overhead. A new scheme was devised which precompiles declarations into densly coded symbol files [4].
- The partitioning of the compiler into modules was chosen such that references to the target computer's architecture are concentrated in a few modules. When *retargeting* the compiler, essentially a single module has to be designed afresh, whereas the other modules undergo minor adjustments only.

The single-pass strategy inherently imposes the restriction on source programs that objects (constants, variables, procedures, etc.) must be declared textually *before* they are referenced. This is, however, mostly just a minor inconvenience. In order to accommodate also the cases where the restriction is a genuine impediment, namely when procedures are mutually recursive, the compiler accepts a so-called forward declaration, in which a procedure's heading including the result type and parameters is specified. (This is analogous to a procedure declaration in a definition module; hence this facility is implemented by use of an already existing mechanism).

The Overall Structure

The multipass compiler's structure closely mirrors the typical tasks of a compilation process; each major task is accomplished by a separate program, a pass, that scans the preceding pass' output sequentially. All passes access a common data base, namely the so-called *symbol table* representing declared objects. This data structure remains in main store at all times. Fig. 1 shows that the number of files invloved is considerable, and it explains the high percentage of compilation time spent on reading and writing on backing store. Also, the effort spent on serializing information and on subsequently parsing it is not negligible. Error diagnostics are collected from each pass and ultimately merged with the source text by an additional lister pass.

The structure of the single-pass compiler mirrors the compilation tasks by its module structure rather than by a sequence of execution steps (see Fig. 2). The savings gained by eliminating the serialization and reparsing operations are reflected by the total size of the compiler. The sum of the lengths of the modules is considerably less than the sum of the lengths of the passes. The tasks of type consistency checking and of code generation are merged into a single module. This is because for both operations practically the same information has to be either retrieved from the symbol table or computed afresh. By tying type checking and code generation together, repeated access can be avoided.

The compiler refrains from generating a program listing, i.e. a copy of the source text with inserted line numbers and/or program counter values. This again reduces the number of files involved. For the benefit of the symbolic debugger, information relating positions in the object code to positions in the source text is inserted in the so-called reference file. This file primarily represents the symbol table in serialized form. It enables the debugger to translate the state of the computer back into the state of the computation expressed in terms of the source language. The positioning information enables the debugger to highlight the offending statement (see Figs. 3, 4).

Further savings in compiler size and complexity were made possible by using the same format for symbol files (generated when compiling a definition module, read when compiling an importing module) and for reference files (generated when compiling a program module, read by the debugger). By using the same routines, the generation of

symbol files becomes practically for free. The technique of serializing the symbol table information is described in detail in [4], and the interface to the module generating and reading symbol and reference files is shown below.

```
DEFINITION MODULE RefFiles;
 FROM DataDefs IMPORT ObjPtr;
 FROM FileSystem IMPORT File;

 VAR ModNo: CARDINAL; (*current module number*)
   ModList: ObjPtr; (*list of loaded modules*)
   RefFile: File;

 PROCEDURE InitRef;

 PROCEDURE InRef (VAR filename: ARRAY OF CHAR; VAR mod: ObjPtr);
  (*insert objects from named symbol file into symbol table; assign module
   object to mod*)

 PROCEDURE OpenRef; (*create new symbol/reference file*)
 PROCEDURE CloseRef(adr: INTEGER; pno: CARDINAL);

 PROCEDURE OutUnit(unit: ObjPtr);
  (*output local objects of unit, i.e. module or procedure*)
 PROCEDURE OutPos(sourcepos, pc: CARDINAL);
END RefFiles.
```

The Parser

The parser acts as the main program from which routines residing in other modules are called when a certain language construct has been recognized. The parsing scheme employed here is the straight-forward technique of top-down analysis using recursive procedures (recursive descent). This scheme, although less powerful than more modern and more sophisticated bottom-up strategies, is applicable thanks to the simple and systematic LL(1) syntax of Modula-2. A primary advantage is that the entire parsing process and its interaction with code generation is explicitly visible in the source text; no use is made of encoded information precompiled by a parser generator.

The scanner forms a separate module. It reads a sequence of characters and yields a sequence of Modula-2 symbols, i.e. of identifiers, numbers, and special symbols. It converts numbers into the target computer's binary representation, and to this small degree is machine dependent. It also maintains a table of strings representing identifiers which, in all other parts of the compiler, appear as indices to this table. The scanner also distinguishes between identifiers and keywords (such as IF, END) by looking up each encountered potential identifier in a keyword table. This is necessary because keywords are not lexically distinguishable from identifiers. The scanner interface is the following:

```
DEFINITION MODULE Scanner;
 FROM FileSystem IMPORT File;
 CONST IdBufLeng = 8000;
```

```
TYPE Symbol = (null,
    times, slash, div, rem, mod, and,
    plus, minus, or,
    eql, neq, lss, leq, gtr, geq, in,
    arrow, period, comma, colon, ellipsis, rparen, rbrak, rbrace,
    of, then, do, to, by,
    lparen, lbrak, lbrace, not, becomes, number, string, ident,
    semicolon, bar, end, else, elsif, until,
    if, while, repeat, loop, with, exit, return, case, for,
    array, pointer, record, set,
    begin, code, const, type, var, forward, procedure, module,
    definition, implementation, export, qualified, from, import, eof);
(*sym, id, numtyp, intval, dblval, realval are implicit results of GetSym*)
VAR sym: Symbol;
 id:      CARDINAL; (*valid if sym = ident*)
 numtyp:  CARDINAL; (*valid if sym = number*)
 intval:  CARDINAL; (*valid if sym = number and numtyp = 1*)
 dblval:  LONGINT;  (*valid if sym = number and numtyp = 2*)
 realval: REAL;     (*valid if sym = number and numtyp = 4*)
 scanerr: BOOLEAN;
 source:  File;
 IdBuf:   ARRAY [0 .. IdBufLeng-1] OF CHAR; (*identifier buffer*)
PROCEDURE InitScanner;
PROCEDURE Diff(i, j: CARDINAL): INTEGER;
 (*alphabetic order of IdBuf[i] and IdBuf[j]*)
PROCEDURE Enter(id: ARRAY OF CHAR): CARDINAL;
PROCEDURE KeepId;  (*called from declarations*)
PROCEDURE GetSym;
PROCEDURE Mark(n: CARDINAL);  (*mark error position*)
PROCEDURE CloseScanner;
END Scanner.
```

The Symbol Table Generator

The symbol table is the data structure obtained from processing declarations. It thus reflects the context-sensitive aspects of the language. A good symbol table organization does not only use storage economically, but makes frequently needed information quickly accessible. In this compiler, the data are contained in a linked structure with essentially three types of nodes (records). The primary type is called *Object* and represents a declared, named object. In view of the requirement of fast retrieval, objects are inserted in ordered binary trees with the object's identifier as key. Each open scope is represented by its own tree. The roots are headers linked together in the sequence of their opening. Any search proceeds top-down through this link and through the individual trees. Fig. 5 displays the data structure after processing the following declarations:

```
MODULE Main;
 VAR i, s: INTEGER; f, g: REAL;
  a, t: ARRAY [0 .. 99] OF REAL;
```

```
PROCEDURE p(x, y: REAL);
  VAR k: CARDINAL; ...
```

The nodes of the data structure are variant records. The tag field *class* discriminates between constants, variables, types, procedures, etc. Common to all variants is, apart from the name, the object's type. This reflects the premise of a strongly-typed language that all objects be of a certain type which embodies the object's invariant characteristics. Another common property is whether an object is imported or exported (field *impexp*). Imports are achieved by inserting a copy of the imported object record in the importing scope. An export from an inner, nested scope is considered as an import in the outer scope. The following type declarations specify the record structure in detail.

```
ObjClass =  (Header, Const, Typ, Var, Field, Proc, Module);

ObjPtr =    POINTER TO Object;
StrPtr =    POINTER TO Structure;
ParPtr =    POINTER TO Parameter;
PDPtr  =    POINTER TO PDesc;

Object =
 RECORD
  name: CARDINAL; (*index to name buffer*)
  typ: StrPtr;
  left, right: ObjPtr;
  impexp: BITSET;
  CASE class: ObjClass OF
   Header:   kind: ObjClass; (*Proc, Module or Typ*)
             heap: ObjPtr |
   Const:    conval: ConstValue; nextConst: ObjPtr |
   Typ:      mod: ObjPtr |
   Var:      varpar: BOOLEAN;
             vmod, vlev: CARDINAL; vadr: INTEGER |
   Field:    offset: INTEGER |
   Proc:     pd: PDPtr;
             firstParam: ParPtr;
             firstLocal: ObjPtr;
             pmod: CARDINAL |
   Module:   key: KeyPtr;
             firstObj: ObjPtr;
             modno: CARDINAL
 END
END;

PDesc =   RECORD (*Object extension for procedures*)
            num, lev: CARDINAL; adr, size: INTEGER
          END
```

The type of an object is represented by a record type called *Structure*, which again appears in several variants. The discriminator is a field called *form* which distinguished between standard, enumeration, array, record, set, and other types. Common to all forms is the attribute *size* indicating the amount of storage needed for variables of this type.

```
StrForm = (Undef, Bool, Char, Card, Int, Double, Real, String,
      Enum, Range, Pointer, Set, Array, Record, ProcTyp, Opaque);

Structure =
 RECORD
  strobj: ObjPtr; (*object (type) naming structure*)
  size: INTEGER;
  CASE form: StrForm OF
   Undef, Bool, Char, Int, Card, Double, Real, String: |
   Enum:     firstConst: ObjPtr; NofConst: CARDINAL |
   Range:    RBaseTyp: StrPtr;
             min, max: INTEGER |
   Pointer:  PBaseTyp: StrPtr |
   Set:      SBaseTyp: StrPtr |
   Array:    ElemTyp, IndexTyp: StrPtr;
             dyn: BOOLEAN |
   Record:   firstFld: ObjPtr |
   ProcTyp:  firstPar: ParPtr;
             resTyp: StrPtr |
   Opaque:
  END
 END
```

As types can be nested (e.g. an array type specifies an index and an element type, and a record type specifies the types of its fields), the resulting data structure is usually recursive. The data structures resulting from the following simple declarations are shown in Figs. 6 and 7.

Example 1:

```
CONST N = 100;
VAR   i, j: CARDINAL;
      [illegible]: ARRAY [0 .. N-1] OF CARDINAL
```

Example 2:

```
TYPE Color = (red, blue, green);
     TreePtr =   POINTER TO TreeNode;
     TreeNode =  RECORD key: INTEGER; kind: Color;
                       left, right: TreePtr
                 END
```

The third type of node that occurs in the symbol table besides objects and structures represents parameters. Procedure parameters actually play a double role, and this circumstance is mirrored by their double representation. On the one hand, parameter specifications must be accessible when compiling a procedure call. Here the parameters' sequence and their types are relevant, and this information belongs, strictly speaking, to the type specification of the called procedure. On the other hand, the formal parameter specifications must be accessible when compiling the procedure body. Here the parameters' names, types, and addresses are relevant. Parameters assume the same role as local variables, and the fact that their names are to be searched suggests that they be included in the search tree as local objects. Hence, parameters are indeed represented twice, once as records of type *Object*, once as records of type *Parameter*. (The field *name* is used temporarily only when processing the formal parameter list; the formal parameter names are irrelevant when compiling procedure calls).

```
Parameter = RECORD name: CARDINAL; varpar: BOOLEAN;
              typ: StrPtr; next: ParPtr
            END;
```

The structure resulting from the following example is shown in Fig. 8.

Example 3:

```
PROCEDURE P(k: INTEGER; VAR m: INTEGER);
  VAR i, j: INTEGER;
BEGIN ...
END P
```

Elements of the data structure are allocated dynamically. This, however, does not imply that an automatic storage allocator is necessary, nor even that it would be advantageous. The fact that scopes associated with procedures are properly nested suggests the sequential allocation of records in an area operated as a stack. Upon closing a procedure's scope the allocation point is simply reset to its position when the scope was opened (ResetHeap). Notably, this does not apply to scopes associated with modules or records, because their local objects must be accessible also after the declaration was processed, either through qualified identifiers or record selectors.

An additional benefit of the double representation of procedure parameters now becomes apparent: their representation as local objects is discarded when the procedure body has been completed, whereas the *Parameter* records remain allocated and attached to the procedure object. This implies that the latter are generated before opening the local scope.

This simple scheme of resetting an allocation pointer (the same is done for the identifier buffer) provides an economical and efficient solution to storage management and contrasts favourably with systems relying heavily on so-called garbage collection. Scavenging is indeed superfluous, if one takes care not to produce garbage in the first place. The

definition part of the module which constructs the symbol table is listed below.

```
DEFINITION MODULE TableHandler;
 FROM DataDefs IMPORT ObjPtr, ObjClass, StrPtr, StrForm, ParPtr, PDPtr;

 VAR topScope, Scope: ObjPtr; (*header of scope located by Find*)

 PROCEDURE FindInScope(id: CARDINAL; root: ObjPtr): ObjPtr;
 PROCEDURE Find(id: CARDINAL): ObjPtr;

 PROCEDURE NewObj(id: CARDINAL; class: ObjClass): ObjPtr;
 PROCEDURE NewStr(form: StrForm): StrPtr;
 PROCEDURE NewPar(ident: CARDINAL; isvar: BOOLEAN; last: ParPtr): ParPtr;
 PROCEDURE NewImp(scope, obj: ObjPtr);

 PROCEDURE NewScope(kind: ObjClass);
 PROCEDURE CloseScope;

 PROCEDURE CheckUDP(obj, node: ObjPtr);
  (*check for pointer types that are still undefined at end of declaration part*)
 PROCEDURE MarkHeap;
 PROCEDURE ReleaseHeap;
 PROCEDURE InitTableHandler;
END TableHandler.
```

Code Generation

The quality of generated code is a crucial characteristic of a compiler. If the architecture of the target computer is suitably designed, generating good code should be relatively straightforward. But unfortunately the widely used processors have their shortcomings in this respect. A compiler designer must therefore compromise between code quality and compiler simplicity. The simpler the compiler, the faster it can be expected to be loaded and to execute and, of course, the smaller is the effort needed for its construction. The more complex the compiler, the better should be the quality of generated code and therefore the speed of execution. A characteristic figure for benchmarking compilers is the time T_{sc} needed for self-compilation, because in a rough approximation it reflects the product of compiler complexity (size) and efficiency (density) of generated code, and therefore is independent on the point of compromise the designer chose regarding compilation versus execution speed.

For the present compiler, we chose a reasonably simple and systematic strategy which permits to generate good, although not optimal code without the need for specific optimization passes. No effort is made towards code improvements that might as well be obtained by improving the source program. The employed code generation scheme is well known and explained as follows:

Each syntactic rule denotes a language construct; its meaning is defined by an associated

evaluation rule. In a purely context-free language the result (i.e. the meaning of the construct) depends on the values of the constituents (of the right part) only. In declarative programming languages, their restricted context sensitivity is represented by the symbol table that was generated when processing declarations, and which therefore must be inspected when referencing declared objects. If we disregard, for the sake of clarity, this restricted and well understood dependence on context, we may postulate that every syntactic rule of the form

$$P_i:\ S_0 \leftarrow S_1\, S_2 \ldots S_n$$

is accompanied by an evaluation function F_i of the form

$$A(S_0) = F_i(A(S_1), A(S_2), \ldots, A(S_n))$$

and a code sequence

$$Q_i(A(S_1), A(S_2), \ldots, A(S_n))$$

A(S) denotes a set of attributes associated with the symbol S. For example, the attributes of a constant are its type and its value, and the attributes of a variable are its type and its storage address. The attributes are defined by a type called *Item* with a variant discriminator called *mode*.

```
TYPE ItemMode =
  (conMd, typMd, varMd, fldMd, procMd, stkMd, adrMd, inxMd);

TYPE Item =
  RECORD typ: StrPtr;
   CASE mode: ItemMode OF
    conMd:     val: ConstValue |
    typMd:     |
    varMd:     adr: INTEGER; level: CARDINAL |
    fldMd:     offset: INTEGER |
    procMd:    proc: ObjPtr |
    stkMd, adrMd, inxMd:
   END
  END
```

This systematic and quite general scheme of code generation, intimately coupled with the syntactic analysis, was described in [5]. It has been adopted here with the difference that the functions F and the code sequences Q are distributed as program statements throughout the recursive descent procedures representing syntax analysis. Recursive descent parsing appears as particularly convenient in this connection: the resulting attribute $A(S_0)$ is represented as parameter of the corresponding parsing procedure. This is shown by the following simplified example for expressions.

```
expression = term {addop term}.
```

The corresponding function F is embodied by statements computing the attribute values of the result x, and the corresponding Q is embodied by the procedures *load* and *GenOp* which append elements to the generated code sequence. If both operands happen to be constants, the compiler computes their sum (first insuring that no arithmetic overflow will occur); otherwise code is issued to load the operands (onto a stack or into registers) followed by the addition operator. Evidently, this scheme easily allows to evaluate constant expressions at compile time, to represent multiplications and divisions by powers of 2 by shift operations, and similar code optimizations.

```
PROCEDURE expression(VAR x: Item);
  VAR y: Item;
BEGIN term(x);
  WHILE sym = addop DO
    GetNextSym; term(y);
    IF (x.mode = conMd) & (y.mode = conMd) THEN
      x.val := x.val + y.val
    ELSE
      load(x); load(y); GenOp(addop)
    END
  END
END expression
```

The construction of an explicit tree structure representing the parsed part of the text is carefully avoided. This contrasts with most modern and more sophisticated techniques of compilation which crucially depend on the presence of such a tree for finding matches of applicable code sequences and for selecting the best candidate [6]. The careful avoidance of matching techniques is one reason for the compactness and speed of this compiler. Adherence to a systematic underlying principle is as important for code generation as it is for syntax analysis. Even so, code generation is a field where details and particulars dominate. They are dictated by the architecture of the target architecture, and hence shall not be discussed here.

Compilers are inherently non-portable programs, because at least the code generators are genuinely machine-specific. A special effort has therefore been undertaken to separate the parts which depend on the language rather than the target (scanner, parser, table generator) from those parts that reflect the machine architecture (code generator). The module structure of Modula has been an indispensible asset in this respect. Nevertheless, we have chosen not to be dogmatic about a totally clean separation. If the compiler is retargeted, most modules undergo at least a very slight adaptation (e.g. change of constants), whereas only the code generator is mostly, but not totally redesigned. The compactness of the compiler is a most welcome benefit in such a project.

Conclusions

The author believes that a compiler should be a reasonably compact and fast system, and that a programming language should be designed to make such compilers feasible. The compiler described here proved that this postulate is realistic and that there is no justification for tolerating bulky and slow compilers. The consequences of having a fast compiler available are considerable:

- The need for so-called incremental compilation vanishes. This is fortunate, because incremental compilers for structured, strongly typed languages are particularly complicated. It appears that the module is an appropriate unit for separate (incremental) compilation.
- It becomes much easier to retarget a compiler and to custom-tailor it to a machine's architecture, if the compiler is compact and fast.
- Any program, and a compiler in particular, must be trustworthy. The absence of errors can be achieved only if the designer has a full and clear understanding of the entire program and of the invariants governing the individual parts. The bulkier a program, the smaller is the chance for total comprehension.

This latter point is beyond doubt the most essential, and it should be a constant reminder to avoid unnecessary complexity and sophistication. The limits of programmed systems used to be set by the computers' memory size and processing speed. This is no longer so; now the limits are determined by the intellectual limitations of the human designer.

This project demonstrates that a compact, efficient, and intellectually manageable compiler is possible even for a language as advanced as Modula-2, and that it can be constructed with very limited manpower. We issue the performance figures of this compiler as a benchmark for future compilers for Modula and other high-level languages, and also as a challenge for their designers and promoters.

Acknowledgement

The author gratefully acknowledges the valuable contribution of J. Gutknecht. Frequent discussions and his scrutiny have led to many improvements and simplifications. He designed and implemented the module for generating and reading symbol files, a sophisticated and crucial part of the compiler.

References

1. N. Wirth. *Programming in Modula-2.* Springer-Verlag, Berlin, Heidelberg, New York, 1982.

2. J. Gutknecht. Tutorial on Modula-2. *BYTE* 9, 8 (Aug. 1984), 157-176.

3. --. Modula: A language for modular multiprogramming. *Software, Practice and Experience,* 7, 3-35 (1977).

4. J. Gutknecht. Compilation of data structures: A new approach to efficient Modula-2 symbol files. (1985) To be published.

5. H. Weber and N. Wirth. EULER: A generalization of ALGOL, and its formal definition. *Comm. ACM, 9* (Jan. 1966) 11-23, (Febr. 1966) 89-99.

6. R.S. Glanville and S.L. Graham. A new method for compiler code generation. *Proc. 5th Annual ACM Symp. on Principles of Programming Languages.* (1978) 231-240.

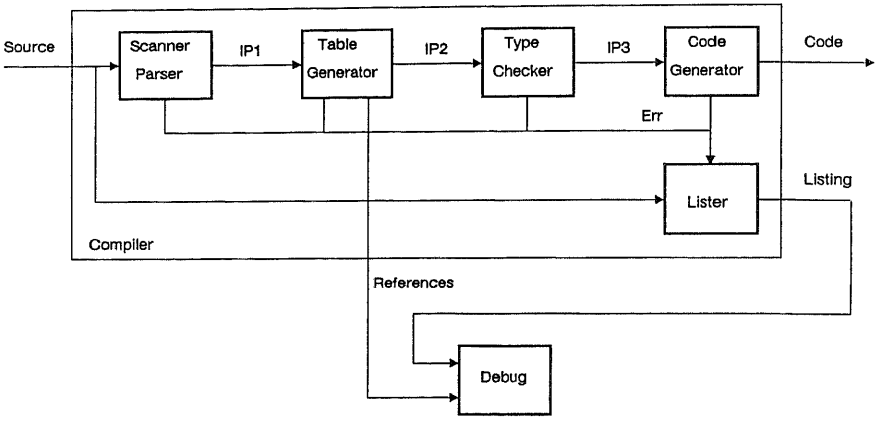

Fig. 1. Structure of multipass compiler

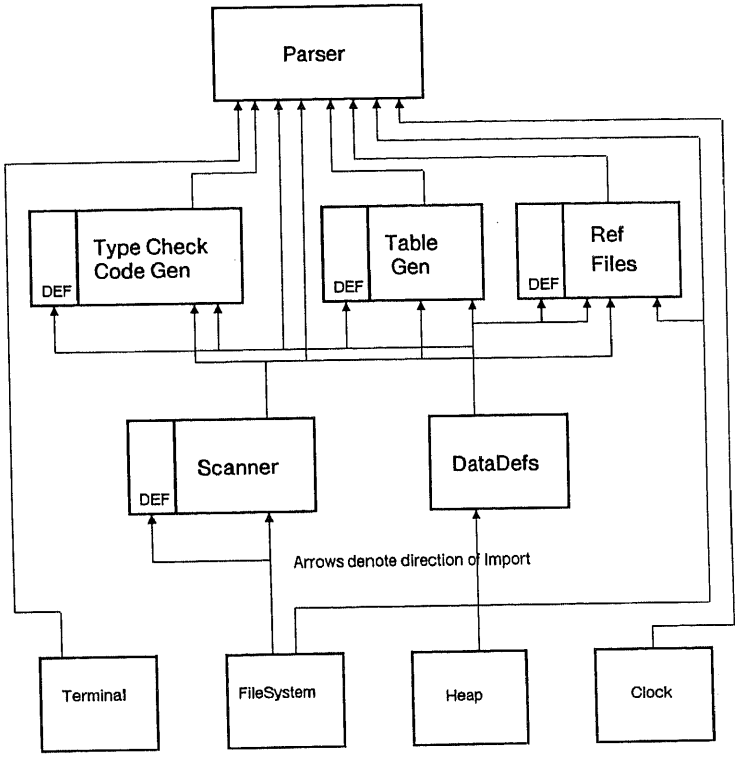

Fig. 2. Module structure of single-pass compiler

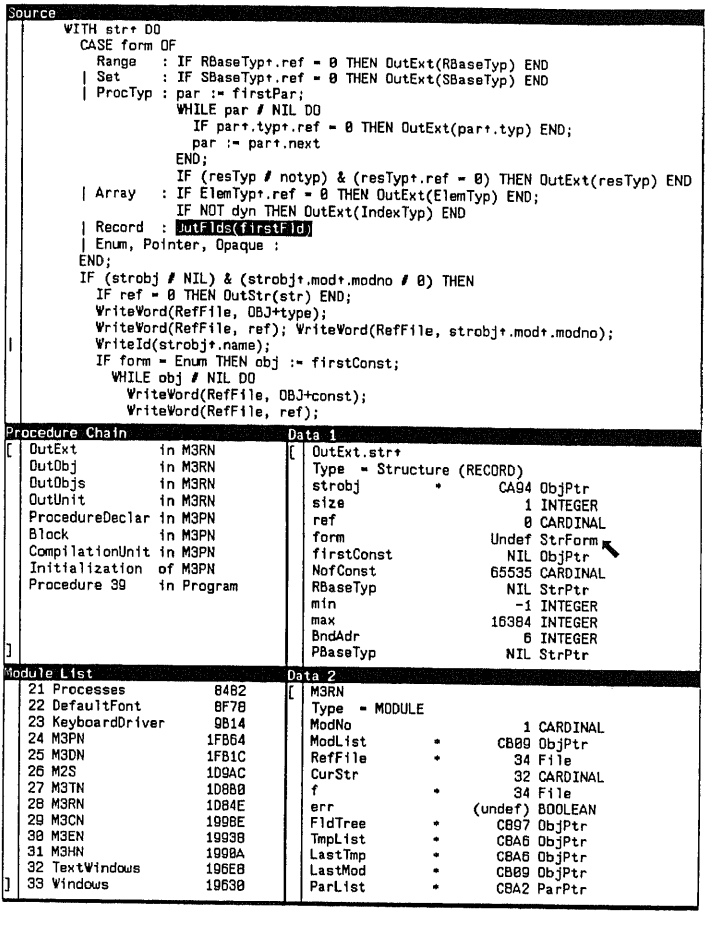

Fig. 3. Debug output: range error in case

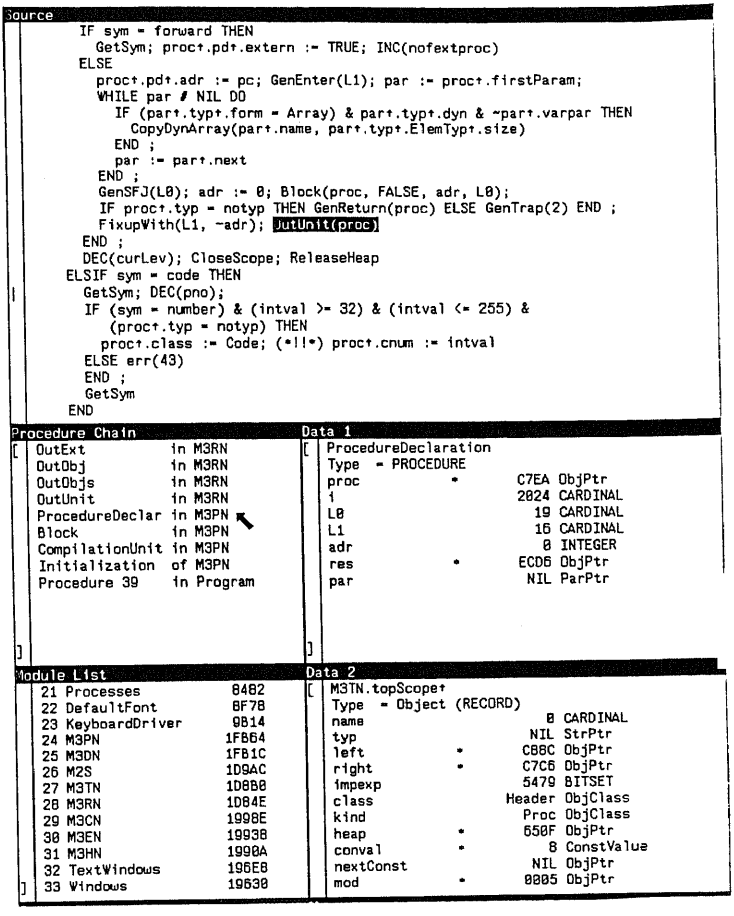

Fig. 4. Debug output: call of OutUnit

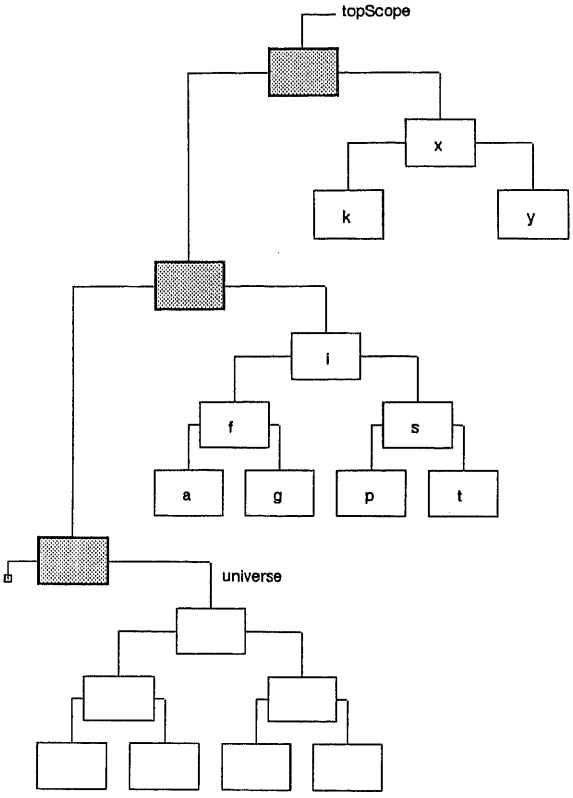

Fig. 5. Search tree structure

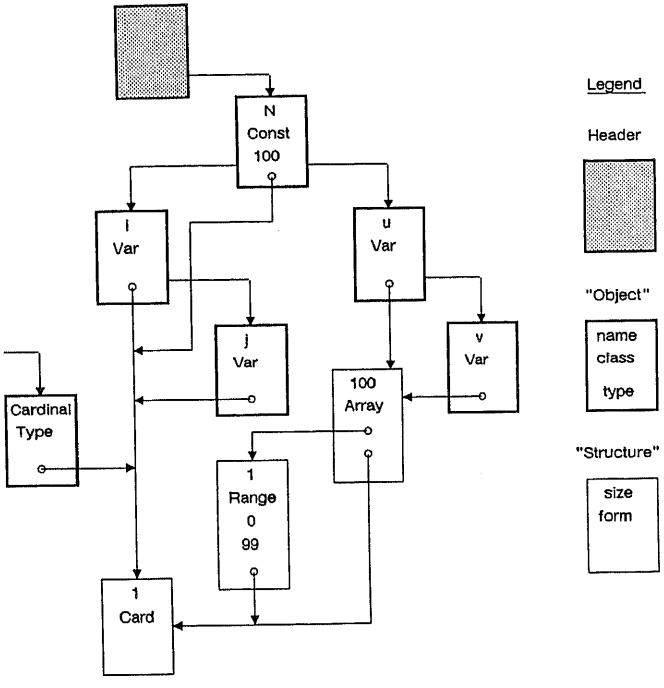

Fig. 6. Data structure of example 1

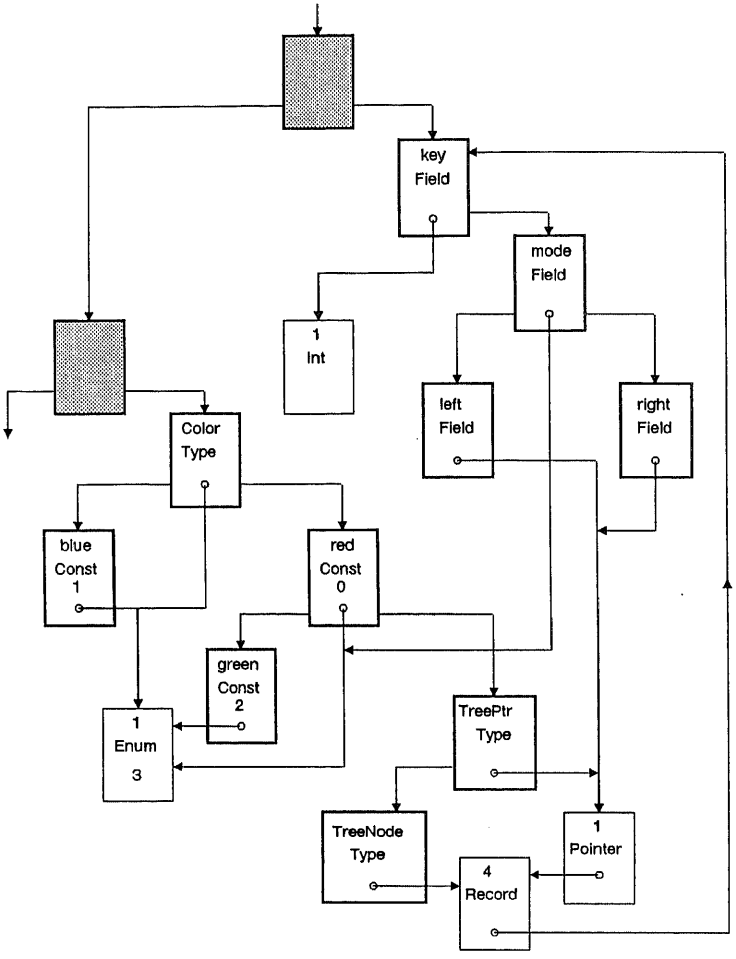

Fig. 7. Data structure of Example 2

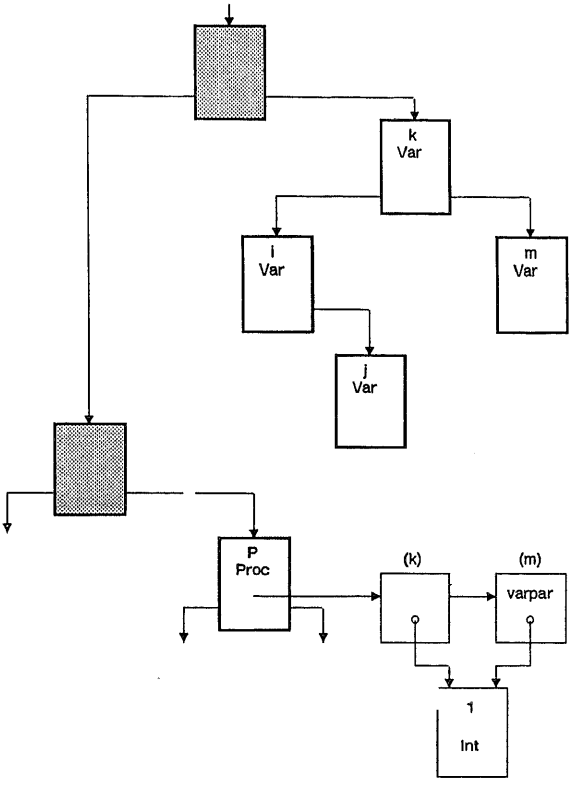

Fig. 8. Data structure of Example 3

Compilation of Data Structures: An New Approach to Efficient Modula-2 Symbol Files

J. Gutknecht
Institut für Informatik, ETH Zürich

Abstract

Modular programming languages have introduced a new class of compilation units. In Modula-2, they are called definition modules. A definition module may specify the interface to a logically connected program part or it may define a global data structure. In most cases, it serves both purposes simultaneously.

The product of compilation of a definition module is a sequential file, the so-called symbol file. It consistently describes all data objects and their structures that are defined or referenced by the module.

Hereafter, we shall present a general method to map a module's data structure onto a linear file and we shall apply it to the creation of genuine and generalized symbol files. Generalized symbol files serve as a basis for a program debugger that enables a programmer to debug Modula-2 programs on an appropriate abstraction level.

This method has been worked out in the course of a recent project with the goal of a fast Modula-2 one pass compiler family [6]. In contrast to previous approaches that can be characterized as being based on a reconstructing parse process, our method emphasizes the data objects themselves as elementary units. Thus, it leads to particularly efficient symbol files both in terms of processing speed and of length.

Introduction

In computer science, the notion of compilation is conventionally associated with the translation of a sequence P of program statements, being formulated in a certain programming language, into elementary machine instruction codes c(P). However, the development of programming languages, in particular their metamorphosis from pure programming notations to systems engineering tools has implied a generalized conception of compilation.

In fact, modern and modular programming languages [1, 2, 3, 4] have introduced a new class of compilation units. In the case of Modula-2 [3], they are called *definition modules*. They can appear in two logically different basic forms or in any combination thereof.

A definition module of the first kind merely defines the interface to a certain program part. More precisely, it specifies a program part from a functional, but static point of view. Each procedure is represented by its heading, i.e. by its name and formal parameter list. The actual algorithms required to meet the specifications and the appropriate local data structure are elaborated in the corresponding *implementation module*.

The second form of appearance of definition modules can be summarized as declaration of global data structures. No specific implementation module is assigned to a definition module of this type (Actually, the corresponding implementation module is empty).

The significance of definition modules is tightly connected with the *import mechanism*. Modules of any kind can import objects which are declared within a definition module. We shall say that the latter *exports* these objects.

Let M be a compilation unit that imports objects from a definition module D. Then, at the time of compilation of M, D must be available to the compiler in a suitable form s(D) that is called the *symbol file* of D. Obviously, we can regard the translation from D to s(D) as a compilation of D.

We shall now work out three basic requirements on symbol files. Let N be a further module that belongs to the same program and that also imports from D. Then, it must be guaranteed that the compilations of M and N, although done at different times, refer to the same version of D. Hence our first requirement: s(D) must include a *key* that uniquely characterizes a particular version of D.

Second, the processing of imports from D during the compilations of M and N should be as efficient as possible. This requirement is of particular importance, if D itself imports objects from further definition modules. In fact, s(D) should subsume all information about objects that are defined in D or are directly or indirectly imported by D.

Consider, for example, the collection of modules as presented in Appendix B1. This example illustrates our explanations and underlies all figures of the current text. Fig. 1a shows the corresponding module constellation. Its compilation leads to the symbol file configuration as displayed in Fig. 1b. Notice that all resulting symbol files are autonomous and on the same "level". Thus, the compilation has levelled off the modular hierarchy.

Further notice that the symbol file of F is not explicitly involved in the compilation of implementation module M.

Let us resume the analogy of the translations from P to c(P) and from D to s(D). Of course, both translations must keep semantics invariant. However, while the final product c(P) usually depends on the compiler and on a specific machine, such dependencies are not desired with s(D). In fact, in modern programming systems, symbol files normall constitute the interface to the operating system.

As a consequence, and this is our third requirement, the data contained on a symbol file for each object should describe exactly the intrinsic properties of the object.

In the following sections we shall introduce a symbol file format and its implementation that complies with our requirements. It is strictly *data object* oriented. Although the very idea of compilation suggests techniques that represent the relevant data entities in a systematically and efficiently coded form, this aspect has been neglected in previous approaches.

As a matter of fact, most of the earlier methods can be characterized as being based on a *syntax* driven reconstruction process. In most cases, the syntax is a compressed form of the original language syntax (for example [5]). These techniques necessitate reconstructing of suitable logical data units by a parse process and are therefore not only in contradiction with our second requirement, but lead also to unnecessary large symbol files. Compared to [5], our compiler generates symbol files that are in average 30 % shorter. Notice that doing without compilation of definition modules at all (for example [1]) can be viewed as an extreme variant of this method.

Abstracting from the specific context in which our method is applied, we can regard it as a method to compile a data structure into a sequential file. As such, it may obviously serve different purposes. A particularly attractive application concerns *program debugging.*

It is a legitimate request that a program P can be debugged on the same level of abstraction as it had previously been created. This request implies that the debugger has access to the data structure of P. An elegant solution is to create a *generalized symbol file* S(P) at the compilation of P's implementation.

The specifications for generalized symbol files essentially coincide with those for genuine symbol files. Therefore, we can in fact use the same technique for their creation. However, an important novel aspect is the hierarchy of nested procedures and submodules that is typical for implementation modules.

Section 1. Structure of Symbol Files

In its most general form, a definition module specifies a data structure and a collection of operations. More precisely, it defines a set of *objects* together with their *structures.* There exist five kinds of Modula-2 objects: types, constants, variables, procedures and modules. The classification of structures is much more difficult. In fact, there is an infinite number of essentially different structures. However, each of them is constructed according to universal

rules. A structure is either *elementary* or based on further structures.

The class of elementary structures comprises all standard structures, for example BOOLEAN, CHAR, INTEGER, WORD, ADDRESS, BITSET, PROC, but also enumerations. On the other hand, subranges, pointers, sets, arrays, procedure-types and records are non-elementary structures. The two last-named are *compound structures*. Their *components* are parameters and fields respectively.

Objects and structures are recorded on the symbol file in the form of *blocks*. Roughly speaking, each object, component and structural unit is described by one block. A so-called *reference number* is assigned to every structural unit. This is the ordinal number of the structural unit within the sequence of all structures on the symbol file. Standard structures are implicitly connected with the first few numbers. They need not be recorded on the symbol file. Both objects and structural units can refer to further structural units by their reference number.

We have previously postulated that a symbol file should be self-consistent, i.e. it should include the complete description of all objects that are imported from other definition modules. Thus, in general, objects on a symbol file stem from different modules.

Each involved module is represented by its *module anchor*. This is a block containing the module's name and a key. The key specifies a particular version of the module. A sequence number is assigned to each module anchor. Object-blocks, typically type-blocks, point to a specific module via this number.

A file-type word initiates the symbol file and a tag-block terminates the list of described objects. The tag-block characterizes the list as describing the objects of a main module. Thus, for the time being, we get the following syntactical structure:

```
SymFile = FileType MainSection.
MainSection = ObjList MainTag.
ObjList = {ModAnchor | Structure | Component | Object}.
```

Genuine symbol files give a concise and self-consistent description of the corresponding definition modules. Their counterparts are generalized symbol files. They describe the data-structures of implementation modules. While the existence of genuine symbol files is of paramount importance for separate compilation, generalized symbol files are indispensable as a base for program debugging on an appropriate abstraction level.

With generalized symbol files, a new complication arises: the concept of nested scopes. In fact, an implementation module typically appears as a hierarchy of nested submodules and procedures. Accordingly, a generalized symbol file consists normally of different sections, where each *section* describes one modular or procedural unit:

```
SymFile = FileType {Section} MainSection.
Section = ObjList (ModTag | ProcTag).
MainSection = ObjList MainTag.
```

Each ModTag contains the identification number of the respective submodule and each ProcTag contains a procedure number. Notice that the above syntax indicates that the set of

sections is arranged on the generalized symbol file as a linear sequence. As a matter of fact, the compilation process itself implicitely linearizes the nested structure (see Section 3).

The previous genuine symbol files appearing as a special case of generalized symbol files, we shall henceforth leave out the qualification, if a statement applies to both kinds.

Let us now look more closely at the objects, components and structures recorded on symbol files:

```
Structure = Enum | Range | Pointer | Set | ProcTyp | FuncTyp | Array | DynArray |
  Record | Opaque.
Component = ParRef | Par | Field.
Object = VarRef | Var | Constant | String | Type | Procedure | Function | Module |
Code.
```

The object-kind VarRef needs further explication. We notice that formal parameters play a kind of double role. On the one hand, they serve to specify a procedure's structure and on the other hand they represent variables. Consistently, each formal parameter appears twice on the generalized symbol file: as a variable in the connected scope and as a component of a procedure (structure).

A Var variable and a Par component are created in the case of a value-parameter, a VarRef variable and a ParRef component in the case of a var-parameter.

Section 2. Loading Symbol Files

In the previous section we discussed the structure of symbol files from a purely syntactical point of view. Now, we shall formulate additional *postulates* that govern the sequence of blocks on the symbol file and thus guarantee a simple and efficient processing of existing symbol files both in terms of time and of memory space.

As we have seen, genuine symbol files are processed by the compiler and generalized symbol files by the debugger. Although the methods used by the two programs to process symbol files are conceptually similar, we shall subsequently concentrate on the compiler, i.e. on genuine symbol files.

The symbol file loader is activated by the main part of the compiler whenever an import statement demands loading of a symbol file. Its global data base is a *module list* (see Fig. 2), i.e. a dynamic chain of module descriptor records. Each time a module anchor is read from a symbol file, the module list is searched for that module. If no appropriate entry is present, the symbol file loader extends the module list by the new module and assigns a module number (the next in natural order) to it. Otherwise, it compares the old key with that of the anchor. Disagreement of the two keys signals a version mismatch.

Each module descriptor is connected with an *object tree* [6], i.e. a dynamic and alphabetically ordered binary tree structure of object descriptors. Upon reading an object block, the symbol file loader creates an object descriptor record and links it to the appropriate tree. Analogously, it creates a structure record for each structure block.

These records become constituents of the compiler's *symbol table*. Establishing correct linkages between the various descriptor records is a crucial activity of the symbol file loader. We shall introduce the following postulates to simplify this task.

Postulate 1. If an object refers to a module-anchor, the descripton of the anchor preceeds the descripton of the object.

Postulate 2. If an object, a component or a structural unit refers to a further structural unit, the descripton of the latter preceeds the descripton of the former.

Postulate 3. Compound structures are represented by the sequence of their components, followed by the object descriptor or structure descriptor.

Postulate 4. If a structure is defined by a type-object, the descriptor of this object immediately follows the description of the structure.

As a further simplification we decide that symbol files are always *loaded as a whole*. Hence, although an import statement may refer to objects selectively, the symbol file loader loads the complete set of objects (and structures) that are stored on the appropriate symbol file.

However, the symbol file loader should avoid multiple loading of an object or structure. In fact, remember that operating system modules typically export large record types that are imported by several modules of a specific program. Multiple loading of such structures would imply a substantial waste of memory space and worse, if not detected, could lead to incorrect incompatibilities at type checking.

Therefore, if an object is read from a symbol file that is already present in the symbol table, the symbol file loader releases the memory space used by the new instance and its structure description. Each object and structure is represented by its very first loaded instance that we shall call *primary instance*.

As an example, assume that in the scenario of Fig. 1 the implementation module of M is being compiled and that its own symbol file s(M) has already been loaded. Then, if an "import from D" statement demands loading of s(D), the symbol file loader detects that object D1 is already present in the symbol table and thus avoids allocating memory for the same object and its structure a second time.

To facilitate memory management, we shall formulate an additional postulate:

Postulate 5. The stream of a structure description must not be broken by the description of a different structure or of an object.

Considering that space for object descriptors and structure records is continuously allocated in a heap, it is obvious that this postulate guarantees for correctness of the following surprisingly simple algorithm: if an object (and its structure) is to be kept, update the heap pointer to the current top, otherwise leave it in its previous state, i.e. pointing to the previous object.

We can now describe the symbol file loader in more detail. Two tables constitute the

main part of its local data structure: a *module table* and a *structure table* (see Fig. 2). They are indexed by reference numbers and serve as translation tables from reference numbers to descriptor records.

The symbol file loader processes symbol files sequentially. It maps each block into a descriptor record. In the case of a module anchor or a structure block the loader connects this block's reference number with its descriptor record by initializing the corresponding pointer entry in the translation table. Remember that successive structure blocks (and module blocks respectively) correspond to successive translation table entries.

If a structure, component or object refers to a module or to a further structure via a reference number, Postulates 1 and 2 assert that the referenced item has previously been loaded. Hence, the appropriate linkage in the symbol table can be established via the corresponding translation table entry.

Suppose now that the symbol file loader processes a type-object that has already been loaded with an earlier symbol file. Then, as we mentioned above, the new instance will be discarded. However, it is connected with a structure that might be referenced later. In concordance with the type compatibility specifications, the corresponding entry in the translation table is initialized to point to the structure descriptor of the very first loaded instance of this type. Notice that Postulate 4 asserts that no references to the structure of the second instance can be made before detecting the double-existence of the structure.

For each sequence of component blocks, the symbol file loader builds up an appropriate data structure. On their occurrence, components are inserted into a *parameter chain* (in the case of procedure parameters) or into a *field tree* (in the case of record fields). This component data structure is getting coupled with the actual object or structure as soon as the associated block is encountered on the symbol file. Postulate 3 guarantees correctness of this method.

To summarize this section, we may say that compiling the import list of a compilation unit amounts on loading all involved symbol files and thus creating the *external part* of the symbol table. Two translation tables are used to map reference numbers into record pointers, one for modules and one for structures. Entries in these tables always point to the primary instance of the corresponding item.

Sequence-postulates were introduced that make the loading process as simple and efficient as possible. Obviously, these are postulates on the symbol file generator.

Section 3. Generating Symbol Files

As we have seen, a genuine or a generalized symbol file is generated at each compilation. In opposition to Section 2, we shall deal in this Section mainly with generalized symbol files.

Having processed all import lists of a compilation unit, the main part of the compiler makes an initial call to the symbol file generator. Its first action is opening the symbol file and generating an anchor block for each module in the global module list (see Fig. 2). Thus, Postulate 1 is satisfied.

Whenever the main part of the compiler has compiled a modular or a procedural unit, it subsequently calls the symbol file generator to generate a section describing the data structure of that unit. It follows that all sections of inner procedures preceed (in the sequence of their declarations) the section of an outer procedure. Notice that each submodule and procedure appears twice on the symbol file: first as a unit and second as an object of an outer unit. See Appendix B3 as an example.

The generating of a unit-section is controlled by the appropriate object tree in the compiler's symbol table. The symbol file generator works off this tree, object by object. According to Postulate 2, it completes the description of an object's structure before it generates the actual object descriptor.

Upon completing a structure's description, the symbol file generator assigns a reference number to the structure (the next one in natural order) and stores this number in the symbol table record associated with the structure. When the same structure is later referenced again, the reference number entry indicates that this structure has already been handled, i.e. written to the symbol file.

Suppose now that a structure is introduced via a type. Then, to satisfy Postulate 4, it must be guaranteed that the corresponding type-object is the first within the set of objects referring to that structure. Therefore, the symbol file generator works off the object tree in two passes: constants and types are processed in the first pass while variables, procedures, and modules are handled in the second. Both passes are preorder traversals. This prevents from constructing degenerate trees when later loading the symbol file.

Let us now investigate in more detail the algorithm to generate a structure description. It turns out to be recursive. In fact, if a structure refers to further structures (e.g. an array referring to index type and element structure, or a record with field-structures), it follows from Postulate 2 that their description must preceed the description of the basic structure.

Two circumstances slightly complicate the algorithm. The first difficulty arises from *recursive pointer-structures* and the second concerns *externally declared structures*.

Let us assume the following declarations:

```
TYPE P = POINTER TO R;
 R = RECORD
    p: P; ...
   END;
```

Without precaution, our recursive algorithm would obviously enter an infinite loop when trying to generate the structure description of P. To circumvent this situation, the symbol file generator splits up the description of a pointer structure into two parts: a handle and a linkage.

More precisely, if the symbol file generator encounters a pointer structure, it generates a *pointer handle block* and creates at the same time an anonymous *synthetic object* representing the pointer's base type. Thereby, the handling of the base structure is postponed. It is resumed (at the latest) when the synthetic object is processed. Then, instead of an object

descriptor block, a so-called *linkage block* is generated. It connects the pointer handle with the corresponding base structure. Notice that the handling of one synthetical object may lead to the creation of another.

Further notice, as a fine point, that at time of loading a symbol file, a linkage request can be obsolete. This is the case, if the pointer structure referred to by the linkage block is not the primary instance and has therefore been discarded. Consider, for example, the compilation of implementation module M. At the time of loading symbol file D, pointer D1 is already in from M's own symbol file. However, by inspecting the (primary) instance of the pointer structure referred to by the translation table, the loader can easily recognize if a linkage has already been established.

We have earlier explained that the symbol file generator proceeds by scannin the object trees of (internal) units. Therefore, *imported objects* are not directly reached. If, however, an *externally defined type* is used to declare a structure, it must be detected by the structure handling algorithm and recorded on the symbol file.

Remember that Postulate 5 does not permit an object block to break the stream of a structure description. Hence, before the description of a given structure can be generated, externally defined types that are connected with this structure must be tracked down and handled in a *preliminary pass*. Notice that the handling of an external enumeration type must implicitly include the handling of each of its values.

So far, we have concentrated on mapping data structures to the generalized symbol file. Considering, however, that it should provide a debugger with information about all aspects of a module, it must include connections between the program counter and source statements. To that aim, the main compiler regularly calls the symbol file generator to write so-called *source position blocks* onto the symbol file.

We conclude by summarizing this Section. After processing the import list, the main compiler invokes the symbol file generator for the first time. Later calls occur whenever a modular or procedural unit has been completed. At each such call, the symbol file generator works off the object tree that is connected with that unit. Thereby, it respects certain sequence postulates that simplify a future loading of the symbol file.

Generally speaking, the symbol file generator makes extensive use of the recursive method to map a module's data structure into *post-fix notation*. Appendix B2 may serve as a comprehensible example illustrating the foregoing.

Acknowledgment

I would like to thank N. Wirth for actively heading this compiler project, for numerous discussions, including conceptual aspects as well as details, and his willingness to redesign an approach, if it turned out not to be completely satisfying.

References

1. UCSD Pascal.

2. J.G. Mitchell, W. Maybury, R. Sweet. *Mesa Language Manual,* Version 5. Xerox PARC, CSL-79-3, Palo Alto, 1979.

3. N. Wirth. *Programming in Modula-2.* Springer-Verlag, Berlin, Heidelberg, New York, 1982.

4. *The Programming Language Ada, Reference Manual,* Lecture Notes in Computer Science, Volume 106, Springer-Verlag, Berlin, 1981.

5. L. Geissmann. Separate Compilation in Modula-2 and the Structure of the Modula-2 Compiler on the Personal Computer Lilith. Dissertation ETH No. 7286, 1983.

6. N. Wirth. A fast and compact compiler for Modula-2. To be published.

Appendix A: Format of Symbol Files

Symbol files are word files. Their syntactical structure is as follows.

```
SymFile    =FileType {Section} MainSection.
FileType   =333B.
Section    =ObjList (ModTag | ProcTag).
MainSection=ObjList MainTag.

ObjList    ={PC-Block | ModAnchor | Linkage | Structure | Component | Object .

Structure  =Enum | Range | Pointer | Set | ProcTyp | FuncTyp | Array | DynArray |
              Record | Opaque.
Component=ParRef | Par | Field.
Object     =VarRef | Var | Constant | String | Type | Procedure | Function | Module | Code.

PC-Block   =000000B sourcepos.
              ...
           =167777B sourcepos.

ModAnchor=170000B key1 key2 key3 name.
ModTag     =170001B ModNo.
ProcTag    =170002B ProcNo.
MainTag    =170003B adr pno.
Linkage    =170004B StrRef BaseRef.

Enum       =171000B size NoConst.
Range      =171001B size BaseRef min max.
Pointer    =171002B size.
Set        =171003B size BaseRef.
ProcTyp    =171004B size.
FuncTyp    =171005B size ResRef.
Array      =171006B size ElemRef IndxRef.
DynArray   =171007B size ElemRef.
Record     =171010B size.
Opaque     =171011B size.

ParRef     =172000B StrRef.
Par        =172001B StrRef.
Field      =172002B StrRef offset name.

VarRef     =173000B StrRef level address name.
Var        =173001B StrRef level address name.
Constant   =173002B StrRef ModRef value name.
String     =173003B StrRef string name.
Type       =173004B StrRef ModRef name.
Procedure  =173005B ProcNo level address size name.
Function   =173006B ResRef ProcNo level address size name.
Module     =173007B ModNo name.
Code       =173010B cnum name.

value      =word word.
name       =string.
```

string = len char {char char}.
len, char = 0C | 1C | .. | 377C.

All symbols whose structure is not explicitly indicated are words.

Predefined reference numbers for standard structures:

1 UNDEFINED 2BOOLEAN 3 CHAR 4 INTEGER
5 CARDINAL 6 LONGINT 7 REAL 8 LONGREAL
9 STRING 10 WORD 11 ADDRESS 12 BITSET
13 PROCEDURE14 .. 31 reserved for future standard structures

The reference number of the first non-standard structure is 32.

Appendix B: An Example

B1: Module Hierarchy

```
DEFINITION MODULE G;
 CONST G1 = 1024;
 TYPE G2;
    G3 = RECORD x,y,w,h: CARDINAL END;
    G4 = PROCEDURE(G2);
 PROCEDURE G5(VAR vwr: G2; blk: G3; act: G4);
 PROCEDURE G6(vwr: G2);
 PROCEDURE G7(x, y: CARDINAL): G2;
END G.

DEFINITION MODULE F;
 FROM G IMPORT G3;
 TYPE F1 = [0 .. 31];
 F2 = (f, g);
 F3 = (h, k, l);
 F4 = POINTER TO F5;
 F5 = RECORD
      nxt: F4; x, y: CARDINAL
    END;
 F6 = RECORD
      pat: F1; x, y: CARDINAL;
      t: F3; p: F4
    END;
 PROCEDURE F7(fig: F6; mod: F2; blk: G3);
END F.

DEFINITION MODULE E;
 FROM F IMPORT F2;
 FROM G IMPORT G3;
 TYPE E1 = [0..7];
    E2 = F2;
    E3 = RECORD
         fnt: E1; pos, len, x, y: CARDINAL
       END;
 VAR E4: SET OF E1;
    E5: ARRAY [0..4095] OF CHAR;
    E6: [0..4095];
 PROCEDURE E7(mod: E2; txt: E3; blk: G3);
END E.

DEFINITION MODULE D;
 FROM E IMPORT E3;
 FROM F IMPORT F6;
 TYPE D1 = POINTER TO D2;
    D2 = RECORD nxt: D1;
         CASE t: BOOLEAN OF
          TRUE: txt: E3| FALSE: fig: F6
         END
       END;
 PROCEDURE D3(obj: D2; x, y: CARDINAL): D1;
END D.

DEFINITION MODULE M;
```

```
 FROM D IMPORT D1;
 FROM G IMPORT G2, G3;
 TYPE M1 = ["A".."Z"];
 VAR M2: ARRAY M1 OF RECORD
           obj: D1; vwr: G2; blk: G3
         END;
END M.

IMPLEMENTATION MODULE M;
 FROM D IMPORT D2;
 FROM E IMPORT E2, E3, E5, E6, E7;
 FROM G IMPORT G1, G2, G3, G5;
 VAR v: G2; b: G3; t: D2;
 MODULE T;
  IMPORT E3, E5, E6;
  EXPORT C;
  VAR pos: CARDINAL;
  PROCEDURE C(str: ARRAY OF CHAR; VAR txt: E3);
   VAR lim: CARDINAL;
   PROCEDURE P(VAR x, y: CARDINAL);
    VAR eol: BOOLEAN;
   BEGIN eol := FALSE
   END P;
  BEGIN lim := HIGH(str)
  END C;
 BEGIN pos := E6
 END T;
 PROCEDURE R(vwr: G2);
 BEGIN E7(f, t.txt, b)
 END R;
BEGIN
 WITH b DO
  x := 0; y := 0; w := G1-1; h := G1-1
 END;
 G5(v, b, R); C("example", t.txt)
END M.
```

B2: Symbol File corresponding to Definition Module E

```
0   ModAnchorkey1 43746  key2 126   key3 57300   name E
1   ModAnchorkey1 43746  key2 126   key3 52580   name F
2   ModAnchorkey1 43746  key2 126   key3 48160   name G
32  Range     size 1     BaseRef 5  min 0        max 7
    Type      StrRef 32  ModRef 0   name E1
33  Enum      size 1     NoConst 2
    Type      StrRef 33  ModRef 1   name F2
    Constant  StrRef 33  ModRef 1   value 1      name g
    Constant  StrRef 33  ModRef 1   value 0      name f
    Type      StrRef 33  ModRef 0   name E2
    Field     StrRef 32  offset 0    name fnt
    Field     StrRef 5   offset 3    name x
    Field     StrRef 5   offset 4    name y
    Field     StrRef 5   offset 1    name pos
    Field     StrRef 5   offset 2    name len
34  Record    size 5
    Type      StrRef 34  ModRef 0   name E3
35  Set       size 1     BaseRef 32
    Var       StrRef 35  level 0    address 3    name E4
36  Range     size 1     BaseRef 5  min 0        max 4095
37  Array     size 2048  ElemRef 3  IndxRef 36
    Var       StrRef 37  level 0    address 4    name E5
38  Range     size 1     BaseRef 5  min 0        max 4095
    Var       StrRef 38  level 0    address 5    name E6
    Field     StrRef 5   offset 0    name x
    Field     StrRef 5   offset 2    name w
    Field     StrRef 5   offset 3    name h
    Field     StrRef 5   offset 1    name y
39  Record    size 4
    Type      StrRef 39  ModRef 2   name G3
    Par       StrRef 33
    Par       StrRef 34
    Par       StrRef 39
>   Procedure ProcNo 1   level 0    address 0    size 0       name E7
    ModTag    ModNo 0
    RefTag    adr 6      pno 1
```

B3: Generalized Symbol File corresponding to Implementation Module M

```
0   ModAnchorkey1 43746  key2 127   key3 7520    name M
1   ModAnchorkey1 43746  key2 127   key3 2360    name D
2   ModAnchorkey1 43746  key2 126   key3 57300   name E
3   ModAnchorkey1 43746  key2 126   key3 52580   name F
4   ModAnchorkey1 43746  key2 126   key3 48160   name G
    pc 000020  pos 368
    pc 000022  pos 387
    VarRef     StrRef 5  level 2    address 4    name x
    VarRef     StrRef 5  level 2    address 5    name y
    Var        StrRef 2  level 2    address 6    name eol
    ProcTag    ProcNo 3
    pc 000041  pos 404
    pc 000043  pos 425
32  DynArray   size 2    ElemRef 3
    Var        StrRef 32 level 1    address 4    name str
```

	Var	StrRef 5	level 1	address 7	name lim	
	ParRef	StrRef 5				
	ParRef	StrRef 5				
	Procedure	ProcNo 3	level 1	address 11	size 2030	name P
33	Range	size 1	BaseRef 5	min 0	max 7	
	Type	StrRef 33	ModRef 2	name E1		
	Field	StrRef 33	offset 0	name fnt		
	Field	StrRef 5	offset 3	name x		
	Field	StrRef 5	offset 4	name y		
	Field	StrRef 5	offset 1	name pos		
	Field	StrRef 5	offset 2	name len		
34	Record	size 5				
	Type	StrRef 34	ModRef 2	name E3		
	VarRef	StrRef 34	level 1	address 6	name txt	
	ProcTag	ProcNo 2				
	pc 000044	pos 440				
	pc 000050	pos 452				
	Par	StrRef 32				
	ParRef	StrRef 34				
	Procedure	ProcNo 2	level 0	address 11	size 15	name C
	Var	StrRef 5	level 0	address 7	name pos	
	ModTag	ModNo 1				
	pc 000055	pos 490				
	pc 000065	pos 509				
35	Opaque	size 1				
	Type	StrRef 35	ModRef 4	name G2		
	Var	StrRef 35	level 1	address 4	name vwr	
	ProcTag	ProcNo 4				
	pc 000122	pos 536				
	pc 000125	pos 544				
	pc 000130	pos 552				
	pc 000137	pos 563				
	pc 000146	pos 583				
	pc 000157	pos 595				
	pc 000166	pos 617				
36	Range	size 1	BaseRef 3	min 65	max 90	
	Type	StrRef 36	ModRef 0	name M1		
	Field	StrRef 5	offset 0	name x		
	Field	StrRef 5	offset 2	name w		
	Field	StrRef 5	offset 3	name h		
	Field	StrRef 5	offset 1	name y		
37	Record	size 4				
	Type	StrRef 37	ModRef 4	name G3		
	Var	StrRef 37	level 0	address 5	name b	
	Module	ModNo 1	name T			
	Par	StrRef 35				
	Procedure	ProcNo 4	level 0	address 41	size 4	name R
	Var	StrRef 35	level 0	address 4	name v	
38	Pointer	size 1				
	Type	StrRef 38	ModRef 1	name D1		
39	Range	size 1	BaseRef 5	min 0	max 31	
	Type	StrRef 39	ModRef 3	name F1		
40	Enum	size 1	NoConst 3			
	Type	StrRef 40	ModRef 3	name F3		
	Constant	StrRef 40	ModRef 3	value 2	name l	
	Constant	StrRef 40	ModRef 3	value 1	name k	
	Constant	StrRef 40	ModRef 3	value 0	name h	

```
41  Pointer   size 1
    Type      StrRef 41    ModRef 3  name F4
    Field     StrRef 39    offset 0  name pat
    Field     StrRef 5     offset 1  name x
    Field     StrRef 40    offset 3  name t
    Field     StrRef 41    offset 4  name p
    Field     StrRef 5     offset 2  name y
42  Record    size 5
    Type      StrRef 42    ModRef 3  name F6
    Field     StrRef 38    offset 0  name nxt
    Field     StrRef 2     offset 1  name t
    Field     StrRef 42    offset 2  name fig
    Field     StrRef 34    offset 2  name txt
43  Record    size 7
    Type      StrRef 43    ModRef 1  name D2
    Var       StrRef 43    level 0   address 6   name t
    Field     StrRef 38    offset 0  name obj
    Field     StrRef 37    offset 2  name blk
    Field     StrRef 35    offset 1  name vwr
44  Record    size 6
45  Array     size 156     ElemRef 44IndxRef 36
    Var       StrRef 45    level 0   address 3   name M2
    Linkage   StrRef 43    BaseRef 38
    Field     StrRef 41    offset 0  name nxt
    Field     StrRef 5     offset 1  name x
    Field     StrRef 5     offset 2  name y
46  Record    size 3
    Type      StrRef 46    ModRef 3  name F5
    Linkage   StrRef 46    BaseRef 41
    ModTag    ModNo 0
    RefTag    adr 9        pno 4
```

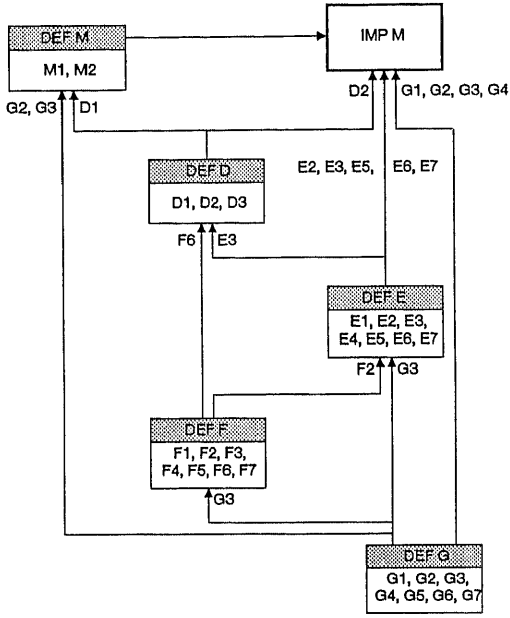

Figure 1a. Dependency diagram for module M before compilation
(see Appendix B1)

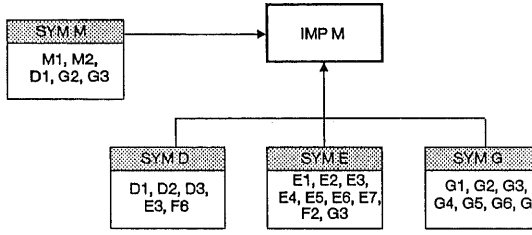

Figure 1b. Dependency diagram for module M after compilation

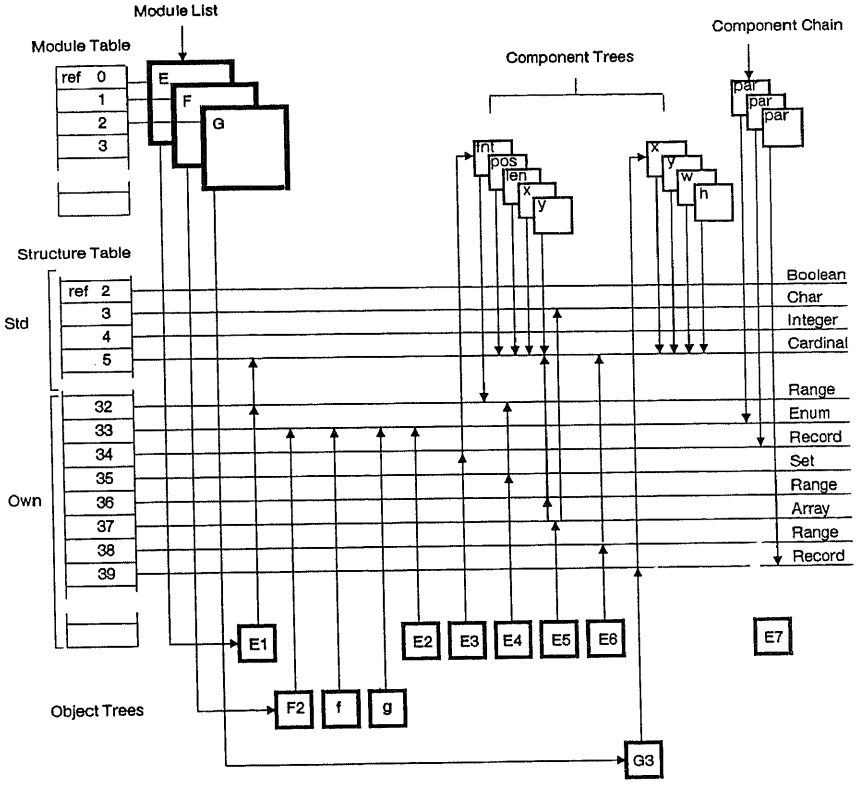

Figure 2. Data structure of the symbol file loader
(see Appendix B2)