ISSN 0105-8517

INTERPRETATION AND CODE GENERATION BASED ON INTERMEDIATE LANGUAGES *

by

Peter Kornerup
Bent Bruun Kristensen
Ole Lehrmann Madsen

DAIMI PB-88
May 1978

* This work has been supported by the Danish Natural Science Research Council, Grants No. 511-1546 and No. 511-5524.

Institute of Mathematics University of Aarhus
DEPARTMENT OF COMPUTER SCIENCE
Ny Munkegade - 8000 Aarhus C - Denmark
Phone 06-12 83 55

Interpretation and Code Generation Based on Intermediate Languages*

Peter Kornerup **,
Bent Bruun Kristensen,
Ole Lehrmann Madsen,

Computer Science Department
Aarhus University
Denmark

Abstract

The possibility of supporting high level languages through intermediate languages to be used for direct interpretation **and** as intermediate forms in compilers is investigated. An accomplished project in the construction of an interpreter and a code generator using one common intermediate form is evaluated. The subject is analyzed in general, and a proposal for an improved design scheme is given.

* This work has been supported by the Danish Natural Science Research Council, Grants No. 511–1546 and No. 511–5524.
** Part of this work was performed while this author was on leave at the University of Southwestern Louisiana, Lafayette, USA.

1. Introduction

The purpose of this paper is to investigate the possibility of designing suitable languages to be used for interpretation and as intermediate forms in compilers, partly by informing about the experiences from a project involving the design of a stack machine, the construction of an emulator on a microprogrammable minicomputer system, and the construction of a codegenerator for a traditional minicomputer, partly by analyzing the subject in general.

Intermediate languages have mostly been defined in connection with selfcompiling compilers as a means of aid in bootstrapping such compilers onto other host machines (e.g. with BCPL [16] and PASCAL [22]). Such intermediate languages are then primarily designed for a straightforward interpretation. By writing an interpreter in some language already available on the new host system (usually a high level language), an interpretive (and slow) language processor can be available while rewriting the code generation parts in the compiler. Usually the code generation does not take the same intermediate language as its input, but uses other intermediate forms, often because a one pass scheme is wanted where the code generation is integrated in the analytic part. However, for the implementation of such language processors on minicomputer systems of a traditional architecture, some advantages may be gained by using multipass schemes to reduce storage requirements.

Hence it would be advantageous to be able to use such intermediate languages directly to give efficient code generation on minicomputers. This requires that the intermediate language (the hypothetical machine) is being designed not only for interpretation, but also such that sufficient information for code generation is available.

For microprogrammable processors an intermediate language may be used in a production language processor system by more or less direct interpretation of the language. By a suitable assembly process a compiled program in the intermediate (symbolic) form may be brought into a compact internal representation to be efficiently executed by a microprogrammed interpreter. The construction of such an assembler and interpreter will most often turn out to be much easier than that of a code generator for a conventional computer.

It is worth noticing, that the process of constructing that part of a compiler which can produce such an intermediate language, also is the part which can be constructed by an automated process based on a formal description of the source language, given a suitable translator writing system. Hence it is most likely, not only that a high degree of portability is achieved, but also that more correct language processors can be constructed, because the source language as well as the intermediate language can be precisely defined.

The question of concern is not the old UNCOL problem, but to define a suitable level for such intermediate languages such that various implementation strategies can use the same interface to the particular source language processor.

The question will be analyzed on the background of some practical projects, described below:

During the last few years some microprogrammable processors, named RIKKE [18] and MATHILDA [4], have been designed and built at the Computer Science Department, Aarhus University, and it was decided to implement PASCAL [22] on a configuration of these.

The first step was to design a pseudo–machine (P–code [11]) to support PASCAL. The main goal of the design was that it should be possible to implement a microprogrammed interpreter for P–code on the RIKKE/MATHILDA system. Furthermore it should be easy to implement a compiler from PASCAL to P–code. A final requirement was that it should be possible to use P–code as an intermediate form in the process of implementing PASCAL on traditional computers.

All three steps were realized, and this paper will try to draw some conclusions on the experience gained, before analyzing the subject in general.

Finally some criteria for the design of such intermediate languages are being extracted, and their application to PASCAL is exemplified in an improved intermediate form given in an appendix.

2. The P–code design and the compiler

The P–code machine is oganized with three stacks, a runtime stack for the allocation of data segments of procedures (activation records), an address stack for the evaluation of addresses and integer arithmetic, and an evaluation stack for expressions in reals and sets, and the manipulation of packed records. Variables in the data segments are addressed by a block level number and an ordinal number. Furthermore, there is an area for the code of the procedures, organized in segments, one for each procedure.

Straightforward stack code consists of instructions loading operands on the stack, and operators working on the stack, i.e. a postfix Polish form. To avoid some explicit loads of operands on the stacks some of the operators in P–code also exist in a version with one of the operands being an argument of the instruction. This is the case for all arithmetic and relational operators which may have an integer constant as an argument. Store operations have the address to be stored into as an argument of the instruction, avoiding an explicit load of the address. Also the jump instructions have the jump address as an argument.

P–code also contains operators for maintaining the runtime stack, i.e. mark, enter and exit instructions. But it does not contain special instructions reflecting control structures like repetition or conditionals, nor does it have instructions for accessing components of structured data, except for instructions to pack and unpack fields of a word and bit testing and insertion (for packed records and powersets).

A compiler from PASCAL to P–code has been implemented by means of an LALR(1) parser generator [12]. Basically a simple translation scheme is used, however for several reasons the compiler became more complicated than anticipated. To optimize in space and time, a quite complicated pseudo–evaluation is performed, which includes handling of the variants of the instructions, subexpressions in literals, and certain optimizations of boolean expressions. During the design of P–code, and the construction of the compiler, an interpreter of P–code written in PASCAL was used for testing purposes.

Except for files other than input and output, the compiler and P–code supports what may be known as PASCAL 1 [22].

3. The interpreter and the code generator

The next step in implementing PASCAL on the RIKKE/MATHILDA system was to design and implement a microcoded interpreter for the P–code machine [4]. RIKKE and MATHILDA are two independent processors, with 16 bit respectively 64 bit word sizes, coupled together both directly and by means of an additional 64 bit wide shared memory called WIDE STORE. Furthermore RIKKE has its own 16 bit wide memory. The whole system constitutes the RIKKE/MATHILDA system. The P–code machine is realized on this system in the following way: The evaluation stack is kept in MATHILDA, the address stack in RIKKE, the runtime stack in WIDE STORE, and the code segments in the memory of RIKKE. Decoding of instructions and operations on the address stack are performed by RIKKE, which also controls MATHILDA and WIDE STORE. MATHILDA is a slave unit which performs complicated operations on the evaluation stack, directed by RIKKE. It was the idea to experiment with non–standard arithmetic in MATHILDA, by substituting the arithmetic code modules in MATHILDA with more experimental ones (e.g. [10]).

Finally a code generator was constructed to evaluate P–code as an intermediate form in compilation for a standard architecture [13]. As host, a Data General NOVA was chosen which is a 16 bit mini. This is a fairly typical minicomputer with a small instruction set, restricted memory size (up to 32 K), four registers, two of which may be used as index registers. Load, store and jump instructions have an insufficient addressing scheme, which complicates storage allocation and code generation.

A pseudo–evaluation of the P–code program is performed in order to generate efficient code. This makes it possible to postpone some address calculation and loading of operands to some later stage at runtime, and thereby to avoid some unnecessary loads and stores. No registers are allocated permanently for special purposes, all are allocated dynamically for varying purposes. To support extensive reuse of existing contents of registers, these are carefully described and checked during allocation.

It has been the idea to generate an inline code sequence from the P–code instruction stream, and to abstain from any sort of interpretive scheme. However as certain P–code instructions turned out to result in fairly long code sequences, some of these were translated into runtime system calls (i.e. enter, exit, vector operations etc.), whereas other still are translated into inline code, although similarly they could be transformed into system calls (e.g. powerset testing, masking of record fields etc.). The code generator was written in PASCAL and the translation process is carried out as a cross compilation (using a CDC 6400). Except for unimportant parts with respect to the evaluation of P–code (e.g. reals, procedures/functions as parameters), the code generator supports the P–code machine.

4. Evaluation of P–code for interpretation

The main goal of the project was to implement PASCAL on RIKKE/MATHILDA and for that purpose P–code seems to be a reasonable success. It turned out to be an easy job to write the interpreter for P–code. For the programmers the hardest problem was to become familiar with RIKKE but after that it was a straightforward process to write the interpreter (less than 500 microinstructions without any attempts to optimize), using available assembler/simulator tools.

By studying the P–code generated by the compiler, several ways of compressing the code by expanding the instruction set are apparent. We shall not discuss these in detail, just mention a few cases. In general one finds that some instructions always appear in connection with special sequences of other instructions and such sequences may be replaced by single instructions. As mentioned, variables are addressed by a block number and an ordinal number. In practice most variables referenced are either global or local (due to Wirth), and it may be worthwhile to have special load/store operations for such variables saving the space for an explicit block number. If one of the operands of an arithmetic or relational operation is a constant then this operand has been included as an argument of the instruction. It seems natural to extend this, and also have instructions which take operands from memory, e.g. to use an "accumulator–stack" architecture.

When designing the stack machine it may be difficult to see which complicated instructions are worthwhile including. Some statistics are needed in order to decide this, and such information can be obtained from our test interpreter (written in PASCAL). It might then be possible to change and experiment with the instruction set.

A practical problem may arise when implementing a complicated instruction. I/O interrupts have to be programmed at the lowest level, i.e. the micro code must at certain intervals test whether an interrupt at the P–code level has to occur. It seems most natural to do so between the individual P–code instruction fetches. However, the execution time of some of the P–code instructions may then be too long to satisfy a reasonable fast service of some devices. It may therefore be necessary to organize the realization of such instructions with suitable break points where to test for interrupts.

5. Evaluation of P–code as an intermediate form for code generation

As an intermediate form in a compiler for PASCAL in general, P–code is less usable. First of all P–code has some deficiencies which means that the coding of some instructions become absurdly complicated. In fact some information is lost and has to be reestablished by a prescan of the P–code program. These drawbacks may however easily be changed in the design, in the compiler and in the interpreter without major troubles. The reason for this is that the primary purpose of the P–code was to interpret it on RIKKE, and the code generation phase was realized after the design was frozen.

A more serious problem arises if really efficient code is wanted. It is very important that the original constructions from the source language can be recognized. For instance, to make efficient use of registers, one must know whether a label in the intermediate form is a part of an if–statement or can be branched to by a goto–statement, etc. (e.g. see [5] sec. 17.4). Although it is possible to recognize most of the PASCAL operations, the control structures cannot be recognized, because such constructs have been compiled into sequences of primitive instructions.

Another serious problem with P–code is that storage allocation has been made by the compiler. It clearly depends on the target machine how variables are to be allocated in core. Suppose that a variable in the object code is addressed by an index register (pointing to the start of the data segment (activation record)) and a constant offset (inside the data segment). In the NOVA however only a small constant offset can be used as an instruction argument. If the offset is too large, indirect addressing has to be used. It may then be desirable to use one word per variable in the low end of the data segment, and if the variable is of a structured type then use the word as a pointer (dope vector) to the actual storage area of the variable. In a computer like the CDC 6400, it is possible directly to address all locations, i.e. a consecutive allocation without the use of dope vectors may do.

We conclude that storage allocation should not have been performed. The intermediate form should contain all the declarative information of the source program.

6. Evaluation of the compile phase

We evaluate P–code to support PASCAL as well as O–code supports BCPL [16], as O–code has the same deficiencies with respect to storage allocation, declarative information and recognition of constructs in the source language.

One of the purposes by choosing a stack design was that the compiler should be easy to implement. The compressed stack code turned out to be a disadvantage, as the compiler has become rather complicated in order to utilize the P–code machine efficiently. The compiler is only simple if the stack code is basically a postfix Polish form. The compressed stack code is not advantageous when using P–code as an intermediate form, since the code generator will have to cope with more versions of the same operation. Hence the process of compressing the code for interpretation should be moved to a later (assembly) phase, thus keeping a more clean P–code as the interface between the compiler and interpreter.

Similar attempts on compressing the object code for a stack machine have been made when implementing BCPL on RIKKE by interpretation of O–code [19]. But here the compression took place in the assembly phase, taking the symbolic O–code from the BCPL compiler into the load modules.

7. Compromizing on the intermediate form

As discussed above the attempt to design one intermediate form suitable for both direct interpretation and for code generation seems very attractive, but fails because of several conflicting requirements.

However if we relax on the requirement that the intermediate form has to be immediately interpretable, a solution is possible. Relaxing on this requirement does not seem unreasonable, since some processing before execution of such a form will always be necessary (e.g. assembly, linking and possibly loading). Hence postponing some more binding to this phase will only add some pure processing to an already existing overhead of scanning the code.

Relaxing on the requirement of direct interpretation implies that the intermediate form in all cases has to be transformed by some sort of code generation before execution. The intermediate form then has to satisfy the following major criteria:

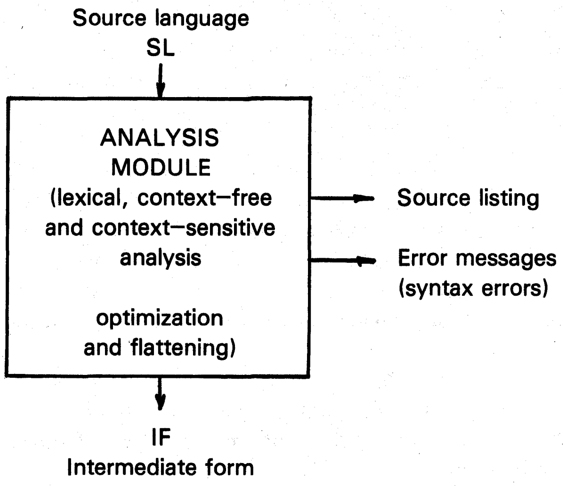
- No binding must take place which will discard information useful in generating efficient code for classical machine architecture of a wide variety.
- With a minimum of additional binding and transformation of representation, the code should be executable with a "reasonable" space and run time efficiency on an interpretive host.
- The intermediate form should contain as little explicit structure as possible to allow an external representation suitable to transmission between machines.

8. A compiler model

Before discussing the design criteria in more detail, let us specify the tasks of the phases, respectively leading to the intermediate form, and from this form into executable forms.

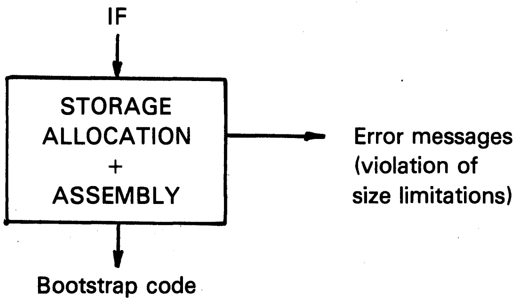
The first phase, responsible for bringing the source language into the intermediate form, consists of lexical, context–free and context–sensitive analysis and flattening (linearisation of the syntax tree):

Phase 1:



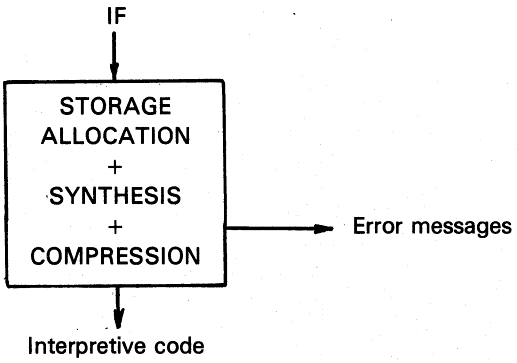
Depending on the application, the intermediate form may then be transformed in a second phase into various executable forms. This involves storage allocation and synthesis of executable code, which however in its simplest form may reduce to an assembler task.

Phase 2a:
(Bootstrap–code for interpretation.)


IF
STORAGE ALLOCATION + ASSEMBLY
Error messages (violation of size limitations)
Bootstrap code

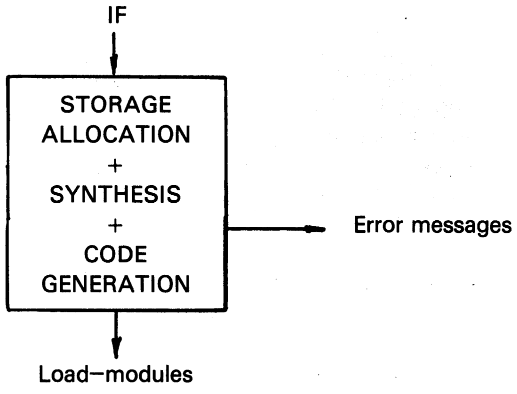
For a more efficient, interpretive language processor system however, a more compressed and complex code may be produced. The synthesis of executable code may then involve some sort of pseudo–evaluation and other local optimizations to reduce the space and time requirements of the produced code, when executed on a suitably defined interpreter.

Phase 2b:
(Efficient code for interpretation.)



Finally, and probably the most complicated situation, the second phase may have to produce executable code for some "real–life" standard computer, in the form of load modules for the particular system in question. This version of phase 2 may have to cope with particular idiosyncrasies of the addressing scheme, utilization of special purpose registers in limited amounts, and interfacing to operating system utilities and limitations.

Phase 2c:
(Code generation for standard host.)



9. Design considerations

The major criteria stated in Section 7 imply that the intermediate form resulting from the first phase has to be a truly language dependent, but completely machine independent, representation of the program. With respect to storage economy and data referencing on a wide variety of hosts it is essential that storage allocation is left to the machine dependent part. Also it is necessary that data structures and control structures are identifiable such that the best host facilities may be utilized in code generation. Hence in the intermediate form all declarative information on data has to be accessible to allow storage allocation to take place later, and all referencing of data must have the form of a reference to the appropriate declarative information.

The necessary binding to permit an interpretation (a phase 2a or 2b) will then consist of a data storage allocation, and the substitution of final addresses into the code wherever data or the code itself is being referenced. Possibly also some transformations on the code itself have to take place, deleting some information, and expanding some constructs into several more primitive ones. However it should be possible to realize a spectrum of such codes for interpretation, depending on the time and effort spent on the storage allocation and transformation of the code. Transformations in phase 2 (b or c) may include pseudo–evaluation, or in general tree transformations, to achieve the optimal boundary between code and interpretive machine, for the given language and the host machine.

The intermediate form has to represent the syntactic tree of the program, or rather the directed graph, taking the context dependent aspects into consideration. However, the requirement that the representation of the program can be easily transmitted, implies that it must contain as little explicit structure as possible because such structure is representation dependent. A tree structure can however easily be represented in linear form, such that only additional cross–node links have to be made explicit. Such links are necessary to associate declaration and usage of symbols, and these can be established in the linearized representation of the structure by using explicit pointers into other linear structures (e.g., symbol tables) which can be more homogeneous. A more detailed discussion of these issues is postponed until the next section.

The intermediate form may consist of a Polish (parenthesis free: prefix, infix, or postfix), a parenthesized representation, or possibly a mixture of these. Also triples [5] or other tuples representing nodes might be used although these use explicit references to another, enforcing restrictions upon the format of the tuples, since the links have to use enumeration of the tuples for referencing.

Another problem with tuples is that it is necessary to store them in a random access store (i.e., they cannot be processed in sequence) to emit reverse Polish (for a stack machine), unless the tuples themselves appear in the linear stream in an order corresponding to say a preorder scan of the tree (in which case a preorder Polish does the same job, but without links). Quadruples [5] are to be "executed" in the order in which they appear, i.e., they are bound to a particular architecture, implying that we will rule these out as a possible representation of a general purpose intermediate form.

For expressions a postfix Polish representation is very suitable, allowing a very simple code generation for ordinary stack machines (e.g., for a simple interpreter), and very easy to generate from the source by means of a simple translation scheme. Also it is quite straightforward to generate code for an accumulator machine (single or multiple accumulator, or "push–down accumulator"), using a pseudo–evaluation. However a pseudo–evaluation cannot generate the most optimal code for a machine with a limited set of fast–access registers (accumulators), since it is not possible to take full advantage of algebraic rules to minimize the number of intermediate results which has to be saved temporarily. Such optimizations, and also the elimination of common subexpressions, require an analysis of, and transformations on, more global information of the program (i.e., a partial or complete program tree).

Before looking into the possibilities of optimizations, let us briefly discuss the representation of control structures in the intermediate form [20]. Although a postfix Polish representation could be used, it is however very inconvenient. Some prefix and infix information is needed, i.e., a parenthesized notation is preferable. To obtain such a translation by a postfix simple syntax directed translation scheme, additional transformations to the input grammar are required (e.g., quite trivial but ugly productions have to be included: e.g. <while> ::= **while**, <while clause> ::= <while> <expression> **do**).

As to the external representation of the intermediate form, it might be convenient with two such representations, one highly encoded form for communication between machines, and one symbolic representation on the level of an assembly language, for human interpretation. However, the encoded form should preferably still be in some standard character code (and even a printable subset) because such codes are more easily portable.

Let us return to the possibilities of performing machine independent optimizations, and optimizations which might be advantageous to a varying degree on various hosts. In the first class one might include elimination of redundant computations (common subexpressions) and loop optimizations, in the second one might think of reducing the number of temporaries needed during expression evaluation to reduce storage referencing. Obviously this last optimization is not needed on a machine with an abundance of fast registers, but may be very advantageous on a one–accumulator architecture.

Such optimizations should not utilize machine dependent features, e.g., say associativity in floating point operations, but might use commutativity of certain boolean and integer operations. Including these optimizations in the phase 1 translation to the intermediate form has the advantage that these might be performed in the same way on a range of implementations, reducing the risk of violating the language definition in particular implementations. Also it is possible to perform these optimizations in a separate module (which might optionally be called between phase 1 and a phase 2) which takes the intermediate form as input and delivers a modified but equivalent form as its output.

Some of the modifications may actually change the intermediate form, e.g., perform some reordering, whereas others may only add further tokens to the representation of the program, e.g., add redundant information which is the result of a detailed flow analysis of the program, and include information on common subexpressions, identification of basic blocks etc.

To summarize, we may think of distributing the optimization work in the compiler scheme as follows:

- **Optimization Analysis:**
  (in phase 1, or separate optimization module).
  Adding redundant information to the intermediate form as the result of an analysis (e.g. data flow analysis) of the source, e.g. adding information identifying common subexpressions and constant expressions in loops, or how a variable is referenced before it is assigned a new value.

- **Machine independent transformations:**
  (in separate optimization module).
  Reordering and transformations on the intermediate form utilizing the information from the optimization analysis.

- **Machine dependent transformations:** (in phase 2c).
  Utilization of the analysis information for efficient code generation (e.g. register allocation and utilization).

10. Handling of declarative information

As stated earlier there is no problem in linearizing the purely context–free syntax of the program, where however for later convenience a combination of say postorder and infix representation might be advantageous. Since we want all context–dependent syntax to be checked and resolved during the first phase, the intermediate form even in a linear representation has to include references from any application of a symbol to its declaration (which in turn again may reference other declarations). Furthermore redundant references may be included, e.g. references to type definitions may be added in various places to indicate say the type of the arguments or the result of an operation.

Considering the intermediate form as a linear stream of pseudo–instructions, there is hence a need for a labeling of some of these, in such a way that particular instructions may be referenced from other instructions. Each pseudo–instruction in the intermediate form corresponds to a node in the graph representing the source program. However, since the graph is basically a tree, with some additional cross–node links, only such links have to be made explicit, using a simple labeling of particular nodes.

For some languages the symbol table is so simple that pseudo–instructions describing it may be emitted from phase 1 when parsing declarative statements, at the same time as the phase 1 symbol table is being built. Other languages permit more complex, possibly circular (recursive) declarations, in which case the complete symbol table may have to be built before it can be emitted (flattened), i.e., phase 1 may involve a multi–pass scheme. Although PASCAL allows some simple circularities, it is however possible to emit the symbol table in intermediate form during parsing of the declarations.

Since storage allocation is to take place during phase 2, the symbol table (–structure) will have to be at least partially reconstructed. Some information from phase 1 may not be needed (e.g. external representation of identifiers, in the case where the implementor does not want to give the user extensive run–time diagnostics) whereas new information will be added as the result of the storage allocation.

The flattening and linearization of the symbol table, and its subsequent reconstruction will add some overhead to the translation, being the price of this scheme. However, if production efficiency of the compiler becomes crucial, a merging of phases 1 and 2, employing a sharing of the symbol table structure, can easily be realized. The cost will be less modularity in the compiler, and hence decreased maintainability.

11. Portability

If a phase 1 and a phase 2a are implemented in the language they are supposed to compile, such a system will allow the compiler to be bootstrapped onto new hosts at a reasonable cost. Phase 1 will be completely machine independent, and a phase 2a bootstrap version would only have to be modified for a new host in its storage allocation part, which may even be parameterizable. A simple interpreter for the phase 2a output code is then the only part which has to be handcoded on the new host if a donor machine is accessible for translating the phase 1 and 2a into the bootstrap code.

If a donor system is not accessible, phase 2a and the bootstrap interpreter have to be hand–translated into some language(s) on the new host.

Major parts of any phase 2c type system may be used when constructing a phase 2c for a new host. The code for scanning the intermediate form, the reconstruction of the symbol table, and major parts of pseudo–evaluation may not only serve as a model for the new implementation, but may be directly used.

Compared to standard one–pass compilers, this system has the advantage that syntax analysis and code generation are implemented completely independently, with a well–defined interface in between. Since the syntax of the intermediate form is much simpler than that of the source language, the task of modifying a phase 2 is much simpler than that of making the modifications in a one–pass compiler.

12. Historical remarks and conclusions

The original efforts of implementing PASCAL via the P–code machine [4] grew out of a project of implementing the BOBS parser generator system [12] during the period 1971–1973. The implementation of the PASCAL P–code compiler, the P–code interpreter realization and the P–code to the NOVA compiler project were realized during 1974–75. A draft manuscript [8] on the experience with the P–code was written in the fall of 1975, but was never completed. Sections 1–6 of this report are however a revised version of part of that manuscript. The basic ideas and design criteria of the remaining part of this paper were also present in [8], and have been used in the work on an intermediate form for Platon [14] and in the Beta implementation work [6].

The design criteria discussed in this report materialized during 1976 in a preliminary design of an intermediate form for PASCAL, described in the appendix, together with a brief discussion of the principles applied. A phase 1 translator [2] from a slightly enhanced PASCAL into such an intermediate form was implemented during 1977 together with a phase 2a bootstrap system [7]. At present phase 2c type compilers for the RC4000/8000 [3] and DEC10 systems are being implemented, and work on some interpretive systems is being initiated.

The goals of the proposed design scheme seem so far to have been achieved. A phase 1–type translation of PASCAL into such an intermediate form serving multiple purposes was quite easily defined and implemented. Work on several versions of phase 2 type translations has indicated that major parts of the code may be identical in these versions, i.e. there is a skeleton of a phase 2 translator which may be used as a basis for a variety of implementations.

References

[1] Aho, A.V. and J.D. Ullman: The Theory of Parsing, Translation and Compiling. Volume 1: Parsing (1972). Volume 2: Compiling (1973). Prentice–Hall, Englewood Cliffs, N.J.

[2] Bardino, J., O. Jacobsen and E. Knudsen: Pascal Compiler Project. DAIMI (internal report), 1977.

[3] Bardino, J. and E. Knudsen: A Pascal Code Generator for the RC4000/8000 Computers. DAIMI (internal report), 1978.

[4] Eriksen, S.H., B.B. Kristensen, O.L. Madsen and B.B. Jensen: An Implementation of P–code on RIKKE/MATHILDA. Daimi, January 1975.

[5] Gries, D.: Compiler Construction for Digital Computers. John Wiley & Sons, Inc., 1971.

[6] Hammerskov, J., B.B. Kristensen and O.L. Madsen: A Compiler Model and its Application to Beta. Presented at Workshop on Simula and Compiler Writing, Oslo 1977.

[7] Jacobsen, H.J. and P.C. Nørgård: Notes on the Pascal Bootstrap Interpreter. Daimi (internal report), 1978.

[8] Kornerup, P., B.B. Kristensen and O.L. Madsen: Interpretation and Code Generation based on Hypothetical Machines. DAIMI (internal report), 1975.

[9] Kornerup, P. and B.D. Shriver: An Overview of the MATHILDA System. Sigmicro Newsletter, January 1975.

[10] Kornerup, P. and B.D. Shriver: A Unified Numeric Representation Arithmetic Unit and its Language Support. IEEE Transactions on Computers, Vol. C–26, No. 7, July 1977.

[11] Kristensen, B.B., O.L. Madsen and B.B. Jensen: A Pascal Environment Machine (P–code). DAIMI PB–28, 1974.

[12] Kristensen B.B., O.L. Madsen, B.B. Jensen and S.H. Eriksen: A Short Description of a Translator Writing System (BOBS–System). DAIMI PB–41, 1974.

[13] Kristensen, B.B.: Code Generation from P–code to NOVA Machine Code. DAIMI (internal report), 1975.

[14] Madsen, O.L., B.B. Kristensen, J. Staunstrup: Use of Design Criteria for Intermediate Languages. DAIMI PB–59, 1976.

[15] Miller, P.L.: Automatic Creation of a Code Generator from a Machine Description. MAC–TR–85, May 1969, Project MAC, M.I.T. Cambridge, Ma.

[16] Richards, M.: The Portability of the BCPL Compiler. SOFTWARE–PRACTICE and EXPERIENCE, Vol. 1, 1971.

[17] Shriver, B.D.: A Small Group of Research Projects in Machine Design for Scientific Computation. DAIMI PB–14, June 1973.

[18] Staunstrup, J. and E. Kressel: RIKKE–1 Reference Manual. DAIMI MD–7, April 1974.

[19] Sørensen, O.: The Emulated Ocode Machine for the Support of BCPL. DAIMI PB–45, April 1975.

[20] Wilcox, T.R.: Generating Machine Code for High–Level Programming Languages. Ph.D. Thesis, Cornell University, 1971.

[21] Wilner, W.T.: B1700 Memory Utilization. Fall Joint Computer Conference, AFIPS 1972, pp. 579–586.

[22] Wirth, N.: The Programing Language Pascal. Acta Informatica 1, 35–63 (1973).

[23] Wirth, N.: The Design of a Pascal Compiler. SOFTWARE–PRACTICE and EXPERIENCE, Vol. 1, 1971.

Appendix A.

In this appendix we will illustrate the design criteria from the previous sections by applying them to Pascal. First we will describe a general intermediate form designed this way, then we scetch a simple stack machine for bootstrapping and finally an efficient stack machine for use in "practice".

A1. The Pascal Intermediate Form.

In the following PIF means Pascal Intermediate Form.

The analysis part of the compiler is assumed to include:

- lexical analysis,
- context free syntax analysis,
- context sensitive syntax analysis, and
- generation of a PIF version of the input program.

Each Pascal construct (language element) is translated uniquely into an equivalent PIF instruction. This is done in such a way that the original source program can almost uniquely be recognised. However some reorganisation has taken place and some explicit information has been added in order to ease the succeeding phases:

- assignments, expressions, and variable denotations are converted into postfix polish,
- control structures are in a parenthesised form, marking the beginning, breakpoints between clauses and the end. All these markings has as a parameter the level of nesting inside the current procedure,
- declarative PIF instructions may be explictly referenced from other places in the PIF. This is done by giving these instructions a unique identifying number (label). In this way the PIF instructions for the

declaration parts correspond to a linearisation of the symbol table as discussed in the previous sections.

- all applications of a name are explicitly referencing the PIF instruction declaring that name,
- all operators (including selectors in structured variables) have been added type information of the operands and the result where this is not obvious. This type information is in form of an explicit reference to the PIF instruction declaring the type, and
- some of the information connected with a PIF instruction may be a reference to a list of names and their denotations. For instance a PIF instruction for declaring a record type has associated such a list of names, corresponding to the fields of the record. Similarily a procedure has a parameter list and a list of locally declared names. Such lists are represented by having a PIF instruction indicating the start of a new list, by PIF instructions for declaring the names of the current list (the list specified by order of occurence) and a PIF instruction for ending the current list.

Comments on the PIF:

- The PIF still contains nested structures in the declarations and the statements, but the nesting depth is explicit,
- if a type is declared with a name, all references to the type is to the PIF instruction declaring the type and not the one declaring the name. This is done in order to treat types without names and types with names in the same way. This is an arbitrary choice and could be made different. Constants given a name in a CONST part are treated in the same way,
- some information from the lexical level has been discarded. In practice PIF instructions indicating line numbers should be included, such that it is possible to generate useful runtime diagnostics.

In the following the PIF is described by means of a modified BNF:

- { } is used to group clauses on the right side of a production,

- * means that the preeceding clause should be repeated zero or more times,
- **name** denotes a Pascal identifier,
- **string** denotes a Pascal string,
- **empty** denotes the empty string.

Context sensitive parts of the PIF are described by the following kind of pseudo syntax:

- *label* denotes a unique integer label identifying the particular (immediately preceeding) instruction when referenced from other places in the PIF. No two *label*s are identical.
- ↑N, where N is a nonterminal or a terminal , is an integer field which is a reference (*label*) to a PIF instruction which can be generated from N,
- *nd* is a number being the current level of nesting. If more than one *nd* appears on a right side they are identical.

A1.1. Syntax of Pascal Intermediate Form (PIF).

The following is a grammar for a Pascal intermediate form. However the grammar will allow sequences of instructions which will never be generated by a compiler. E.g. the grammar allows arbitrary declarations to appear within a scalar list.

```
<PIF-program> ::=
        PIFPROGRAM label name ↑PARAMLIST  ↑NAMELIST
            <standard-environment>
            <PIF-block>*
        ENDPIF
```

{ The program is treated as a procedure with a parameter list and a list of all local declarations; **name** is the name of the program }

```
<standard-environment> ::=
      { The PIF for all predefined labels, constants, types, variables
      ,procedures and functions }

<PIF-block> ::= <declaration-block> | <statement-block>

<declaration-block> ::=
       NAMELIST label
          <PIF-block>*
       ENDNAMELIST ↑NAMEDEF
      { encloses the PIF for all labels, constants, types, variables, procedures,
      and functions declared local to a procedure/function (not including the
      parameter list) }

|      PARAMLIST label
          <declaration-PIF>*
       ENDPARAMLIST ↑NAMEDEF
      { encloses all declarations in a parameterlist of a procedure/function }

|      SCALARLIST label
          <declaration-PIF>*
       ENDSCALARLIST ↑SCALAR
      { encloses all declarations of a scalarlist }

|      <fieldlist>

{ all of the above lists ends with a reference to the instruction, to which the list
is associated }

<declaration-PIF> ::=
       NAMEDEF label name <kind>
      { declares a name of a certain kind }
|      LITERAL label string ↑<type>
      { declares a literal constant. The external representation of the constant
      is kept in string. ↑<type> refers to the type of the constant }
|      LABEL label string {declares a label}
|      <type>
|      <declaration-block>

<fieldlist> ::=
       FIELDLIST label
          <declaration-PIF>*
```

```
            <taglist>
        ENDFIELDLIST ↑<record or tag>

<taglist> ::= empty
|       TAGFIELD ↑NAMEDEF ↑<type>
        { <taglabel> <taglabel>* <fieldlist> }*
        { ↑NAMEDEF refers to the field being the tagfield variable (if it exists).
        ↑<type> is the type of the tag selector }

<taglabel> ::= TAGLABEL label ↑LITERAL ↑FIELDLIST
        { a taglabel refers to a literal defining the label and to the fieldlist of that
        variant }

<record or tag> ::= RECORD | TAGLABEL
        { a fieldlist can be referenced from either a RECORD or a TAGLABEL }

<kind> ::=
        CONST ↑LITERAL
|       TYPE ↑<type>
|       VAR ↑<type>
|       FIELD ↑<type>
|       VARPARAM ↑<type>
|       VALUEPARAM ↑<type>
|       PROC ↑PARAMLIST ↑NAMELIST
|       FORMALPROC
|       FUNCTION ↑PARAMLIST ↑NAMELIST ↑<type>
|       FORMALFUNC ↑<type>

<type> ::=
        SUBRANGE label ↑<type> <min> <max>
|       SCALAR label ↑SCALARLIST
|       POINTER label ↑<type>
|       FILE label ↑<type>
|       SET label ↑<type>
|       RECORD label ↑FIELDLIST
|       ARRAY label <index-type> <element-type>

<min> ::= ↑LITERAL
<max> ::= ↑LITERAL

<index-type> ::= ↑<type>
<element-type> ::= ↑<type>
```

```
<statement-block> ::=
        BEGIN ↑NAMEDEF
            <statement-PIF>*
        END ↑NAMEDEF
       { delimits the PIF for the statement part of a procedure/function,
       ↑NAMEDEF refers in both cases to the NAMEDEF instruction which
       declares the procedure/function }

<statement-PIF> ::=
        LABELDEF ↑LABEL
|       GOTO ↑LABEL
|       <variable-denotation> <expression>
        STORE ↑<var-type> ↑<exp-type>
|       <function-name> <expression>
        FUNCRESULT ↑NAMEDEF ↑<exp-type>
|       PROCCALL nd ↑NAMEDEF
            <actual-parameter>*
        ENDCALL nd ↑NAMEDEF
|       IF nd
            <expression>
        THEN nd
            <statement-PIF>*
        ELSE nd
            <statement-PIF>*
        ENDIF nd
|       IF nd
            <expression>
        THEN nd
            <statement-PIF>*
        ENDIF nd
|       WHILE nd
            <expression>
        DO nd
            <statement-pif>*
        ENDWHILE nd
|       REPEAT nd
            <statement-PIF>*
        UNTIL nd
            <expression>
        ENDREPEAT nd
```

```
|       FOR nd ↑NAMEDEF
            <expression>
        FORINIT nd
            <expression>
        FORTODO nd
            <statement–PIF>*
        FORTOEND nd
|       FOR nd ↑NAMEDEF
            <expression>
        FORINIT nd
            <expression>
        FORDOWNTO nd
            <statement–PIF>*
        FORDOWNEND nd
|       WITH nd
            <variable–denotation>
        WITHDO nd
            <statement–PIF>*
        ENDWITH nd
|       CASE nd
            <expression>
        OF nd
            {CASELABEL ↑LITERAL {CASELABEL ↑LITERAL}*
                <statement–PIF>* ENDOFCASE nd}*
        ENDCASE nd

<actual–parameter> ::=
        <expression> PARAMETER ↑NAMEDEF ↑<expression–type>

<variable–denotation> ::=
        NAME ↑NAMEDEF
|       <variable–denotation>  <expression> INDEX ↑ARRAY
|       <variable–denotation> REFERENCE ↑POINTER
|       <variable–denotation> FIELD ↑NAMEDEF ↑RECORD
|       WITHFIELD ↑NAMEDEF ↑RECORD withnd
```

{ WITHFIELD is a field of a record specified in an enclosing WITH statement. *withnd* is the nesting depth of that WITH statement. WITH statements referring to more than one record variable is assumed to be unfolded into nested WITH statements referring to only one record variable }

```
<expression> ::=
        <expression> <monadic–operator>
|       <expression> <expression> <dyadic–operator>
|       CONSTANT string ↑<type>
|       <variable–denotation> LOAD ↑<type>
|       CALLFUNC nd ↑NAMEDEF
            <actual–parameter>*
        ENDCALL nd ↑NAMEDEF
|       SETCONSTRUCTOR ↑SET
            { <expression> SETELEMENT ↑SET
            | <expression> <expression> SETRANGE ↑SET }*
        ENDSET ↑SET

<monadic–operator> ::= NOT | MONADICMINUS | FLOAT

<dyadic–operator> ::= INTDIV | MOD | AND |  OR
              |<typed–operator> ↑<operand–type>

<typed–operator> ::=  EQ | NE | LT | GE | GT
        | IN | SETINTERSECTION | SETUNION | SETDIFFERENCE
        | PLUS | MINUS | MULT | REALDIV
```

A1.2. Example .

An example of a Pascal program and its associated PIF:

```
PROGRAM pifex;
LABEL 10;
CONST sonmax=3;
TYPE  sex=(male,female)
        tree=↑node;
        node=record
                    info: integer;
                    son: ARRAY[1..sonmax] OF tree;
                    CASE s: sex OF
                        male: (youngest: integer);
                        female: ()
                END;
VAR t: tree; s: sex;
PROCEDURE q(a: integer; VAR sx: sex);
 .
 .
 .
END; (*q*)
BEGIN (* pifex *)
        .
        .
        .
        WHILE t <> NIL DO
            WITH t↑ DO
            BEGIN  q(info,s);
                CASE s OF
                male: t:=son[youngest];
                female: GOTO 10;
                END
            END;
 10:
        .
        .
        .
END .
```

In order to ease reading, all *label*s are placed in front of their respective instruction as a label, allthough it is actually a parameter of the instruction.

```
1        PIFPROGRAM ″pifex″ 0   101
         .
         .
↑int     the type integer
         .
↑1       the literal constant 1
         .
↑nil     the type specification for NIL
         .
         .
         .
101      NAMELIST
102      LABEL      ″10″
103      LITERAL    ″3″ ↑int
104      NAMEDEF    ″sonmax″ CONST 103
105      SCALAR     106
106      SCALARLIST
107      LITERAL    ″male″ 105
108      NAMEDEF    ″male″ CONST 107
109      LITERAL    ″female″ 105
110      NAMEDEF    ″female″ CONST  109
         ENDSCALARLIST 105
111      NAMEDEF    ″sex″ TYPE 105
113      NAMEDEF    ″tree″ TYPE 112
114      FIELDLIST
115      NAMEDEF    ″info″ FIELD ↑int
116      SUBRANGE   ↑int ↑1 103
117      ARRAY      116 112
118      NAMEDEF    ″son″ FIELD 117
119      NAMEDEF    ″s″ FIELD 105
         TAGFIELD   119 105
120      TAGLABEL   107 121
121      FIELDLIST
122      NAMEDEF    ″youngest″ FIELD ↑int
         ENDFIELDLIST
123      TAGLABEL   109 124
124      FIELDLIST
         ENDFIELDLIST 123
         ENDFIELDLIST 125
```

```
125       RECORD       114
126       NAMEDEF      "node" TYPE 125
112       POINTER      125
127       NAMEDEF      "t" VAR 112
128       NAMEDEF      "s" VAR 105
129       NAMEDEF      "q" PROC 130 131
130       PARAMLIST
132       NAMEDEF      "a" VALUEPARAM ↑int
          ENDPARAMLIST 129
131       NAMELIST
          .
          .
          .
          ENDNAMELIST 129
          BEGIN        129
          .
          .
          .
          END          129
          .
          .
          .
          ENDNAMELIST 1
          BEGIN        1
          .
          .
          .
          WHILE        1
          NAME         127                        (t)
          LOAD         112
          CONSTANT     "nil" ↑nil
          NE           112
          WHILEDO      1
          WITH         2
          NAME         127                        (t)
          WITHDO       2
          PROCCALL     3 129                      (q)
          WITHFIELD    115 125 1                  (info)
          LOAD         ↑int
          PARAM        132                        (a)
          NAME         128                        (s)
          PARAM        133                        (sx)
```

```
ENDCALL     3  129               (q)
CASE        3
NAME        128                  (s)
LOAD        105
OF          3
CASELABEL   107                  (male)
NAME        127                  (t)
WITHFIELD   118   125  2         (son)
WITHFIELD   122   125  2         (youngest)
LOAD        ↑int
INDEX       117
LOAD        112
STORE       112  112
ENDOFCASE 3
CASELABEL   109                  (female)
GOTO        102                  (10)
ENDOFCASE 3
ENDCASE     3
ENDWITH     2
ENDWHILE    1
LABELDEF    109                  (10:)
.
.
.

END         1
ENDPIF
```

A2. A Simple Stack Machine.

To illustrate how close the PIF is to an executable code we wil describe a simple stack machine based on the PIF and intended for bootstrapping. The major changes that have to take place when translating into this code are the following:

- declarative instructions and type information have been removed,
- variables have been assigned addresses consisting of a block number and an ordinal number inside that block,
- instructions for the control structures are converted into explicit jumps.

We do not describe the complete architecture of the stack machine as the example should be self explanatory. It should be obvious that a translation from the PIF to the simple stack code is simple and almost one to one.

The variables have been assigned the following addresses:
t: 1,1; s: 1,2; Relative in a node, info: 0; son: 1; s: 4; youngest: 5;
q: 0,1 (level 0 is assumed to contain descriptors for all procedures and functions).

Storage locations being referenced by jump instructions are labelled with their respective absolute locations.

Simple stack code for the example:

```
111   loadadr    1  1        (t)
      load
      literal    0
      ne
      jumpfalse  149         (WHILEDO)
      loadadr    1  1        (t)
      load
      store      1  3        (temporary location)
      mark
      loadadr    1  1        (t)
      load
      literal    0
      plus
      load
      loadadr    1  2        (s)
      call       0  1        (q)
      loadadr    1  2        (s)
      load
      case       146
130   loadadr    1  1        (t)
      loadadr    1  3        (temporary)
      load
      literal    1           (son)
      plus
      loadadr    1  3
      load
      literal    5           (youngest)
      plus
      load
      index      1  1        (lower bound, element size)
      load
      store
      jump       148         (ENDCASE)
144   jump       149         (GOTO 10)
      jump       148         (ENDCASE)
146   130                    (jumptable for
      144                    (case statement)
148   jump       111         (ENDWHILE)
149   .                      (10:)
      .
```

A3. An Efficient Stack Machine.

The architecture of the efficient stack machine differs in the following ways:

- the instruction set has been extended in order to reduce the size of the code and to increase execution speed. E.g. all stack top operations can now have a constant or an address as an argument,
- the representation of the instructions in memory should be considered carefully. An analysis of generated code may allow significant compression of the code [21], by using "short" opcodes (few bits) for frequently occurring instructions, and similarly arguments in many instructions may be represented in shorter fields by extending the opcodes with bits specifying the format of the arguments.

Furthermore the translation from the PIF is no longer one to one but several optimisations are performed in order to use the extended instruction set. Load of operands on the stack are postponed until they are used in an operation. In fact this translation is close to codegeneration for an ordinary register machine, except for efficient register allocation.

The proposed "efficient stack machine" is just a sketch of how such a machine might look like. If one really wants to build a Pascal machine a much more careful study must be undertaken.

As before we let the example "define" the machine. We will just explain the format of an address. An address consists of a block number (BN) and an ordinal number (ON) and then optionally an indirect marking (@) followed by a post ordinal displacement (PO). The effective address is:

```
        A:=display[BN] + ON; IF @mark THEN A:=memory[A] + PO;
```

Some instructions use the address (e.g. loadadr) other use the contents (value) of the memory cell addressed by A (e.g. loadvalue). PO displacement is useful in connection with pointer variables for records.

The code is

```
111     loadvalue  1  1           (t)
        jumpeq 0   to 129         (WHILEDO)
        mark
        loadvalue  1  1 @0        (t↑.info)
        loadadr    1  2           (s)
        call       0  1           (q)
        loadvalue  1  2           (s)
        case       126
119     loadadr    1  1  @1       (t↑.son)
        loadvalue  1  1 @5        (t↑.youngest)
        index      1  1
        load
        store      1  1           (t)
        jump to    128            (GOTO 10)
125     jump to    129
126     119
127     125
128     jump to    111            (ENDWHILE)
129     .                         (10:)
        .
        .
```

Kornerup, Peter, 1939-
Interpretation and code generation based on intermediate languages / Peter Kornerup Bent Bruun Kristensen, and Ole Lehrmann Madsen.-- Aarhus, Denmark: Department of Computer Science, Institute of Mathematics, University of Aarhus, 1978.
(DAIMI; PB-88)

I.-II. Joint authors. III. Title.