The Design of CP-6 Pascal

Jim des Rivieres
Wilf R. LaLonde

SCS-TR-1
June 1982

School of Computer Science
Carleton University
Ottawa, K1S 5B6
Canada

This research was supported in part by contracts and grants from Carleton University, the Natural Sciences and Engineering Research Council of Canada, and Honeywell Canada Ltd.

The Design of CP-6 Pascal

Jim des Rivieres and Wilf R. LaLonde
School of Computer Science
Carleton University

Abstract

The general design of a production Pascal compiler for a large mainframe is discussed. This compiler was designed for both production and student use in such a way as to allow it to serve as the basis for a full-blown interactive programming environment. These design goals, which might appear to be in conflict, can, in fact, be achieved simultaneously.

1. Introduction

CP-6 Pascal is a production compiler for full ISO DP7185 Pascal developed for Honeywell L66/DPS8 hardware running the CP-6 operating system. Although it is packaged as a typical batch compiler, it was designed as the core of an interactive programming environment consisting of an open-ended collection of tools designed to support program development and maintenance.

A key aspect of a programming environment is the representation used for user programs; it is from this representation that most tools will operate. In CP-6 Pascal, we decided on a centralized database containing a tree structured representation of the source program, segmented on procedure and function boundaries. In contrast to most compilers, the tree structured program representation also contains the symbol table in a distributed form.

The compiler is organized into a loosely-coupled collection of phases (see [1] for details). All interphase communication is achieved by placing information in the centralized database. Each phase is driven by an interpreter that serves to insulate that phase from the database system. Several of the phases, those that walk the program tree, are driven by a specially designed tree walking interpreter.

2. Design Objectives

The traditional batch compiler has only one component: a translator. The user builds his source file with the system file editor; he compiles it with the compiler which translates it into an object code file, reports any errors found, and produces a listing; he then links the object code file with the system library and finally executes it.

There are several unfortunate aspects about this arrangement, the most important ones having to do with the lack of an internal program representation and with the inability to communicate with the compiler on anything but the "entire program" level. As a result, anyone wishing to implement a tool which would, say, perform static data analysis on a Pascal program, would have to start from the beginning, designing a parser and an internal representation. What a waste of time and effort! The compiler already has these capabilities. However, they are not usually available.

Having little desire to implement another traditional Pascal compiler, the first design decisions made were to require that (1) there be a comprehensive internal representation for source programs, and this internal representation would not be solely for the use of the compiler, and (2) the components of the compiler would be decoupled whenever possible, making it possible to use them in other tools.

The decision to opt for a comprehensive common internal representation to be used by the compiler and a host of other tools implies that the internal representation is really a database system that provides a full set of facilities for creating, interrogating, modifying, and destroying data objects describing Pascal programs. The design of the database system was influenced by the way LISP systems represent user programs, but evolved with some important differences. We chose as the basis a tree-structured representation of the source program. This is not the parse tree, but rather an abstract syntax tree. In addition to the basic structural fields of the nodes, we added other fields in which to record additional information. We knew early that the tree representation would require substantial space and that it would be impossible to keep entirely in primary memory. On the other hand, if the internal program representation was organized like a normal disk-based database system, the cost of access would be exorbitant. The compromise we decided on was to segment the tree. We would ensure that all necessary segments were in primary memory when they were being manipulated, but allow them to be written to disk when they we not. The exact manner in which the tree is to be segmented depends on the size of the units processed by the various modules of the compiler. The natural choice for this unit is the routine; i.e., either a procedure, function, program, or module. Hence, the tree is segmented on routine boundaries. To further reduce space requirements, each routine is partitioned into two segments: one for the declarations and one for the body of the routine. Note: if the declarations contain inner routines, only the parameter information is contained by the segment, not the internals of the routine. Thus the segments themselves are arranged into a tree mirroring the macroscopic routine structure of the source program. When processing a routine, the only segments that need be in primary memory are the routine´s declaration and body segments and the declaration segments of the routines enclosing the routine being processed. This is a direct consequence of the scope rules of the language.

In order to break down the processing into modules that could be used in other tools, we decided on the following sequence of phases. The first phase converts the normal printable representation of a Pascal program into a segmented tree, a process we call tree binding. Unfortunately, this phase has no choice but to process an entire program in one shot. However, once the tree has been built it is possible to do routine-at-a-time processing for all other phases of the compiler. The tree-based processing naturally splits into four phases: name lookup, type checking, storage assignment, and code generation. We often refer to the first three processes as name binding, type binding, and address binding. An important design decision was that the various phases confine their activities to the routine passed to it and not call any other phases. There would be a master controller that would be responsible for orchestrating the phases. This was to ensure that the possible order of activation of the various phases be as unconstrained as possible, making it easy to add new phases and allowing these components to be used in other tools not yet considered.

But if the phases confine their activities to a single routine and are not allowed to call any other phases, how are the phases to communicate? To avoid problems with memory management (remember, segments are not always in memory due to their size) and unnecessary dependencies on the order of phase activation, information is passed between phases by recording it in the segment in the form of extra fields associated with the various nodes. One further rule is needed to ensure that it is possible to re-process a segment (remember, we are planning for an interactive environment where the user may wish to make changes): each field of a node has to "belong" to a particular phase, and a phase is only allowed to change the values of fields that belong to it. This also prohibits the tree-based phases from mangling the source program since the basic structural fields belong to the tree binder.

There were two language-related design goals. The language to be implemented had to conform to the draft proposed international standard for Pascal (ISO DP7185). But what about extensions? Instead of just adding extensions, we decided that the compiler should be able to handle multiple dialects of Pascal. One of these dialects would be standard Pascal, but there might be other, perhaps even incompatible, Pascal-like languages also acceptable to the compiler. We wanted to ensure the compiler could be easily extended in the future to handle new language features as the need arose. The best way that we could see of keeping the compiler flexible was to design and implement an extended dialect of Pascal concurrent with the implementation of standard Pascal. This experimental language, called $lang, is syntactically incompatible with Standard Pascal mainly because of statement end-brackets such as ENDIF, ENDFOR, ENDPROCEDURE. $lang incorporates numerous extensions including: PL/I-style comments, numbers in a base other than ten, include files, relaxed order of declarations, non-numeric labels, constant expressions, a varying length string datatype, non-sequential files, read-only formal parameters, generic file types, modules, type checking, increment and decrement statements, otherwise cases, extended FOR statement, synonyms in WITH statements, EXIT, NEXT, and RETURN statements, various new built-in routines, and an ADA-like exception handling facility. The major extension to type checking across separately compiled modules was also anticipated and allowed for.

We also had to consider who would be using the compiler. For the next few years, our implementation of Pascal would probably be the only one available to CP-6 sites (including ours). This meant that it would have to be suitable for both student and production use. Heavy use by students would mean that the compiler would have to thoroughly check all programs for errors and report problems through well worded error messages (there are over 600 of them). It would also be essential to provide superior runtime error checking (e.g. undefined variables). Production use would require the generation of CP-6 compatible object code files that could be linked with non-Pascal generated object programs. Moreover, runtime error checking code would have to be controllable.

There were a couple of other general design principles to which we subscribe. First, the compiler should not impose any arbitrary limits on the size or complexity of the programs to be compiled. This principle was easy to satisfy since the decision to use a centralized internal program representation would require the existence of extensive dynamic memory management facilities. Second, great efforts would be made to make the code for the compiler readable because this would be the only way to satisfy our most important objective: the system should have NO BUGS.

3. The Structure of a Binder

The organization of each major module of the compiler, called a binder, is shown in Figure 1. The controllers for each of the binders are written in a very simple "charts" programming language. These chart programs are translated into tables which are interpreted by a standard table interpreter. All primitive actions are contained in the set of binder-dependent support routines that are called by the table interpreter when appropriate.

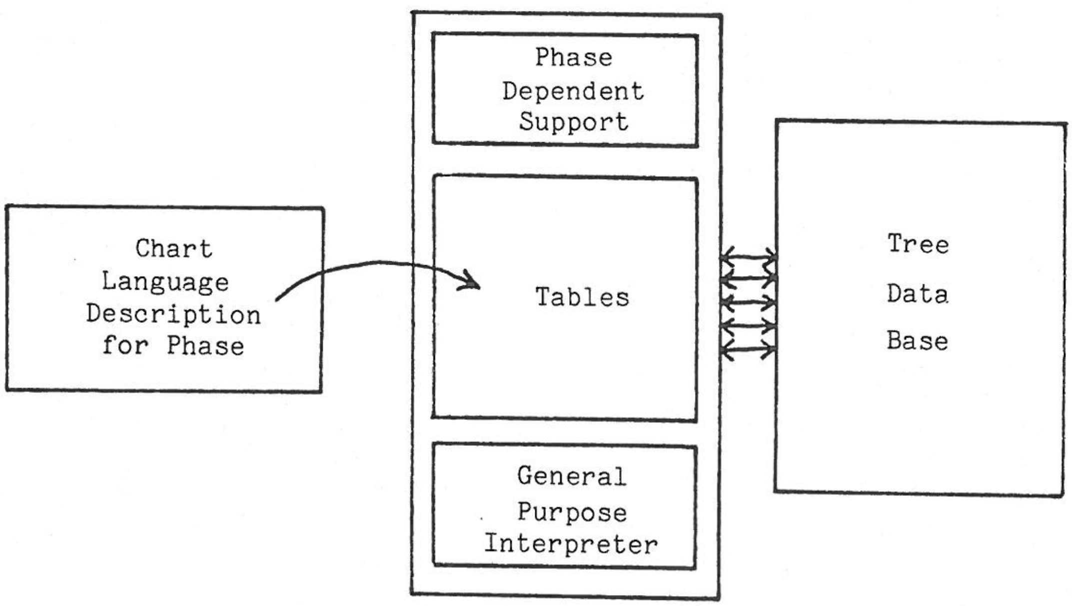

Figure 1. Binder Structure.

There are in fact two different chart languages used in the compiler: sequential charts and tree charts. The sequential chart language, which is used only in the front-end of the compiler, is based on S/SL [2]. A sequential chart program is a one-dimensional representation of a syntax chart (sometimes called "railroad" charts). Thus, sequential chart programs are nothing more than top-down deterministic parsers for arbitrary sequential streams. An example is shown in Figure 2.

```
CHART IF_STATEMENT
   ; Parse an IF statement and build an If_Node.
   If                                      ; Opening keyword.
      (Push_Last_Line_Number_Attribute)    ; Remember the line number.
      EXPRESSION                           ; Parse an expression.
   Then                                    ; Mandatory keyword.
      STATEMENT                            ; Parse a statement.
      CHOOSE
         CASE Else:                        ; Else part present?
            STATEMENT                      ; Yes, parse a statement.
         CASE OTHERWISE:
            (Push_Nothing_Node)            ; No, supply an empty statement.
      ENDCHOOSE
   (Push_Subtree
      If_Node                              ; Build an If_Node and
      Line_Number_Attribute                ; fill in the fields
      Expression_Link                      ; from items on the
      Then_Link                            ; tree stack.
      Else_Link)
; End of IF_STATEMENT CHART.
```

Figure 2. Sample sequential chart.

The tree charts language is an adaptation of the sequential charts concept for walking trees. All of the binders in the back-end of the compiler (i.e. the tree-based binders) are controlled by tree charts programs. From the viewpoint of the tree charts language, a tree (really a graph) consists of a collection of typed nodes with fields, and a tree chart program walks from node to node via the links recorded in the fields. The tree charts language provides facilities for:

- Checking the type of the current node.
- Shifting attention to the node recorded in a particular field of the current node (via an IN-OUT construct).
- Shifting attention to the node returned by a call to an external action.
- Choosing one of several paths based on the type of the current node.
- Invoking an action and passing constant parameters to it.
- Choosing one of several paths based on the result of an external action.
- Calling a named subchart. All iteration is done with recursive calls.

Note that there is no need for error recovery in the tree charts interpreter since the trees being walked are always well-formed. A portion of the charts used in the type binder is shown in Figure 3 to give the reader a feel for the flavour of the tree charts language.

```
CHART TYPE_CHECK_IF_STATEMENT
   IN Expression_Link
      TYPE_CHECK_EXPRESSION; Returns the result type in Type_Link_Register.
   OUT
   (Note_Line_Number); In case a subsequent error message is generated.
   IN (Link_Of REGISTER Type_Link_Register)
      CHOOSE
         CASE Error_Node:
            ; Previous error message given.
         CASE Boolean_Node:
            ; Perfect.
         CASE OTHERWISE:
            (Error_Via_Note If_Expression_Must_Be_Boolean)
      ENDCHOOSE
   OUT
   IN Then_Link
      TYPE_CHECK_STATEMENT
   OUT
   IN Else_Link
      TYPE_CHECK_STATEMENT
   OUT
; End of TYPE_CHECK_IF_STATEMENT CHART.
```

Figure 3. Sample tree chart.

4. The Binders

The previous section described the general organization of a binder. This section discusses the nature of the specific binders and when they are invoked. Terminology: a x-binder is a phase that either produces x´s or is responsible for computing x-type information.

The first phase of the compiler, which internalizes the source program, has four distinct components: the source reader, the word binder, the keyword binder, and the tree binder. The source reader acquires records from the source file and worries about include files. Its output, a stream of characters, is processed by the word binder. The word binder (scanner or lexical analyzer in more conventional terminology) produces a stream of words (tokens). Next, the keyword binder modifies the stream of words by replacing uses of reserved identifiers with special words. This stream of words is then processed by the tree binder (parser or syntactic analyzer in more conventional terminology). All syntax errors are diagnosed and repaired by this binder, and the result is a well-formed, segmented tree representation of the entire source program. All of these components run together; the character and word streams never really exist.

Once the database for the source program has been built, subsequent phases decorate the tree with assorted information. All tree-based phases operate on a single routine at a time. The first tree-based phase has two components: the name binder and the type binder. The name binder links identifier usages nodes with their corresponding definition nodes, links together definitions into contours according to the scope rules of the language, and diagnoses undeclared and multiply-declared identifiers. The type binder propagates type information throughout the tree, detecting and reporting type violations. These two binders, although conceptually distinct, must be tightly coupled to avoid duplicating work. In particular, to name bind a reference to a record field, the record itself must have been previously type bound. For example, consider "A[i]^[j].Field"; to determine the proper record type for Field, much type propagation needs to be performed.

The next tree-based phase is the address binder which does storage allocation for types, variables, and constants. The final phase, code generation, has three parts: the code binder, the peephole optimizer, and the object unit generator. The code binder produces relocatable L66/DPS8 machine code, calls the peephole optimizer (not implemented yet), and invokes the object unit generator to produce a CP-6 object unit.

5. Evaluation

The following lists summarize our assessment of the design and the implementation.

Design:

- The idea of having a centralized program representation is a sound one. However, the database system must be more flexible if it is to be useful for other programming environment tools. In particular, there needs to be a way of adding new node types and new fields to old node types without having to recompile the compiler and reform all previously-created databases.
- The decision to go for routine-at-a-time processing for all of the tree-base phases was a great asset. Besides forcing many things to be made explicit, the exceptionally flexible order of phase invocation [1] allowed us to drastically change the order very late in the project.
- Charts have proven to be good insulation (R35) from the vagaries of the database system and forced the actions of the individual phases to be explicitly listed and organized, resulting in a much more precise and concise design. In fact, prior to the release of the first version of the compiler, this property allowed us to make major changes to the structure of the nodes to improve performance.
- We feel that the tree charts language has potential, but more work is needed. In particular, it would have helped if the language had local variables and parameters. As it was, we had to fake these capabilities with external actions.

- We are quite satisfied with the LL(1)-type technology. This may seem a bit surprising given one of the authors´ expertise in LR(k) techniques. The charts translators are straightforward. It took us about three days to change a sequential chart translator into one for tree charts; it takes about a week to write a charts translator from scratch. In comparison, LR(k) parser generators are far too complex.

Implementation:

- The compiler is slow. This is due mainly to the interpretive overhead of the chart interpreters, to the memory management overhead, and to the slow calling sequence of the implementation language. Our current efforts are directed at making substantial improvements in these areas.
- The compiler is big. This is due to the large size of the nodes in the internal program representation and to the amount of source code in the compiler, a consequence of the highly modular nature of the design and our desire to produce extra readable code. The large size was not viewed as a major problem since it was designed for use on a large machine.
- Bugs are scarce, even trivial ones. We were most pleased with it and so are its users. The complaints are invariably about the speed of the compiler, not about its quality.

Statistics

- Length of time to implement: approximately 5 man-years over 1.5 years.
- Number of lines of code (including comments):
  Sequential Chart Code: 6100 lines.
  Tree Chart Code: 23600 lines.
  PL/6 (Implementation Language) Code: 48500 lines.
- Compiler Speed: approximately 1250 source lines per minute (half the speed of the CP-6 FORTRAN-77 compiler).

6. Concluding Remark

The authors now view the design and implementation of compilers for Pascal-like languages as a task better left to others.

7. Acknowledgments

We would like to express our sincere appreciation for the invaluable contributions made by the other members of the CP-6 Pascal project: Nick Briggs, John Stewart, and Chris Carter.

8. References

[1] LaLonde,W.R. and des Rivieres,J. A Flexible Compiler Structure That Allows Dynamic Phase Ordering, School of Computer Science report SCS-TR-3, Carleton University, Ottawa, June 1982.

[2] Holt, R.C., Cordy, J.R., and Wortman, D.B. An Introduction to S/SL: Syntax/Semantic Language, ACM TOPLAS, Vol. 4, No. 2, April 1982, pp. 149-178.

Carleton University
School of Computer Science

Bibliography of SCS reports

SCS-TR-1 THE DESIGN OF CP-6 PASCAL
Jim des Rivieres and Wilf R. Lalonde, June 1982.

SCS-TR-2 SINGLE PRODUCTION ELIMINATION IN LR(1) PARSERS: A SYNTHESIS
Wilf R. Lalonde, June 1982.

SCS-TR-3 A FLEXIBLE COMPILER STRUCTURE THAT ALLOWS DYNAMIC PHASE ORDERING
Wilf R. Lalonde and Jim des Rivieres, June 1982.

SCS-TR-4 A PRACTICAL LONGEST COMMON SUBSEQUENCE ALGORITHM FOR TEXT COLLATION
Jim des Rivieres, June 1982.