A Flexible Compiler Structure That Allows Dynamic Phase Ordering

Wilf R. LaLonde
Jim des Rivieres

SCS-TR-3
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.

* This paper was presented at the SIGPLAN'82 SYMPOSIUM ON COMPILER CONSTRUCTION, Boston, Mass., June 23-25, 1982.

A Flexible Compiler Structure That Allows Dynamic Phase Ordering

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

Abstract

Traditionally, compilers have been internally organized into separate phases that are invoked in a fixed, predetermined order. However, the optimal phase invocation order is usually different for different contexts, such as (1) large batch-oriented production environments, (2) small batch-oriented student environments, and (3) interactive programming environments.

We discuss the constraints that must be met in order to obtain a compiler with flexible phase ordering, a property that allows the compiler to be designed and implemented without consideration for the intended operating environment. Flexible phase ordering allows the compiler to be specially configured after the fact. A production compiler based on this principle is also discussed.

Introduction

In designing a compiler, two fundamental issues must be decided early: the size of the internal work units; e.g., should the grain of processing be a statement or a procedure, and the phase design; i.e., how many phases should there be and what should they do. There is in fact a third issue that is usually not explicitly identified, namely the phase ordering. That is, in what order should the work units and the phases be coupled to achieve the overall compilation task? We claim that this is a separate fundamental issue. Furthermore, the first two decisions should be made independently of the third. By following some simple guidelines when making the first two decisions, a robust compiler design can be obtained with a minimum of constraints on the ordering. With such a loosely constrained design, it is possible to configure the same compiler to perform efficiently under diverse operating conditions, enabling it to take advantage of the size of the program to be compiled, memory limitations, and operating environment (e.g., batch versus interactive/incremental compilation modes).

This kind of flexibility can be provided because the loosely constrained design imposes a very relaxed partial ordering on the activation order of the phases. The extra degrees of freedom provided allow substantial dynamic control of the activation order of the phases, enabling diverse control strategies to be used. This flexibility frees the designer from having to consider the phase activation order as integral to the design and abolishes the need to take into account the operating conditions when designing and implementing the bulk of the compiler. The operating conditions need only be taken into consideration when designing the phase activation module.

We first discuss a compiler model that permits such flexibility and then consider its advantages and disadvantages. The model is general enough to support compilers for both interactive incremental programming environments and conventional batch environments.

A Minimally Constrained Model

We view a compiler (Figure 1) as a collection of phases that manipulate a common internal program representation (IPR). The initial phase (or phases) converts the character representation of a program into the more flexible IPR. Subsequent phases perform the majority of the work and it is only these phases that we will be interested in. They might include a name binding phase that looks up names in a symbol table, a type binding phase that verifies and propagates type information, an address binding phase that associates memory locations with specific variables, and a code generation phase.

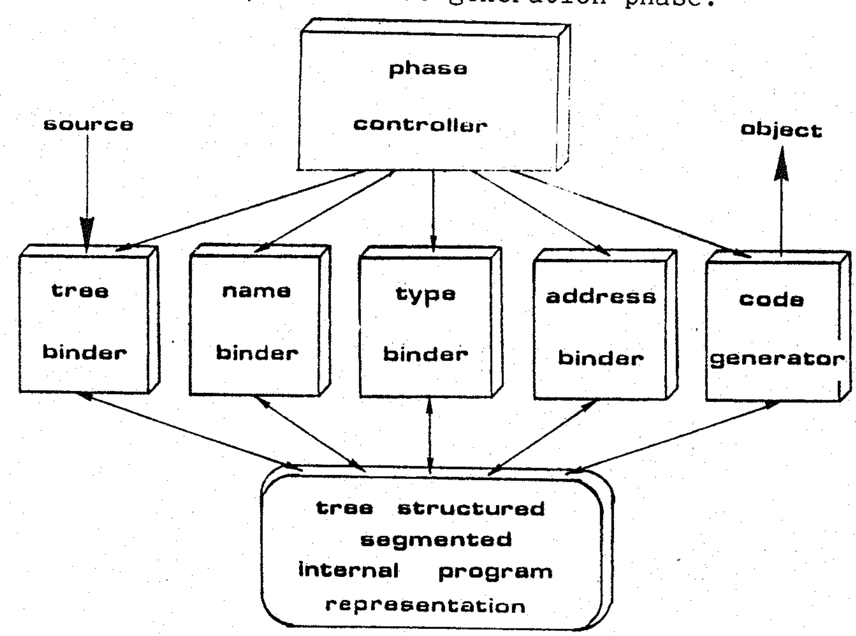


Figure 1. Compiler Model.

During compilation, information computed in one phase is used in subsequent phases. Consequently, there is a constraint on the order in which the phases can be applied. A phase dependency graph (Figure 2) can be used to describe this constrained invocation order.

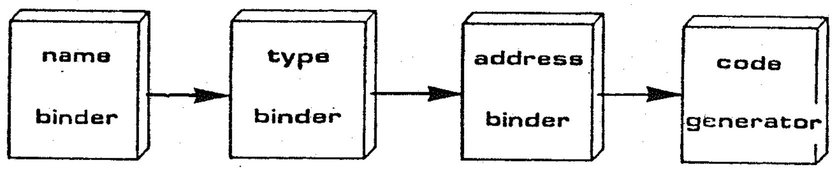


Figure 2. Phase Dependency Graph.

Having modularized the compiler into phases, it is only natural to modularize the IPR into a collection of work units that can be processed individually. In general, a work unit contains all the information about the portion of the program that it represents. The exact form of the IPR is not critical to the discussion. However, our examples will make use of an IPR that is an annotated tree with each work unit corresponding to that portion of the tree representing a procedure or module, as illustrated in Figure 3.

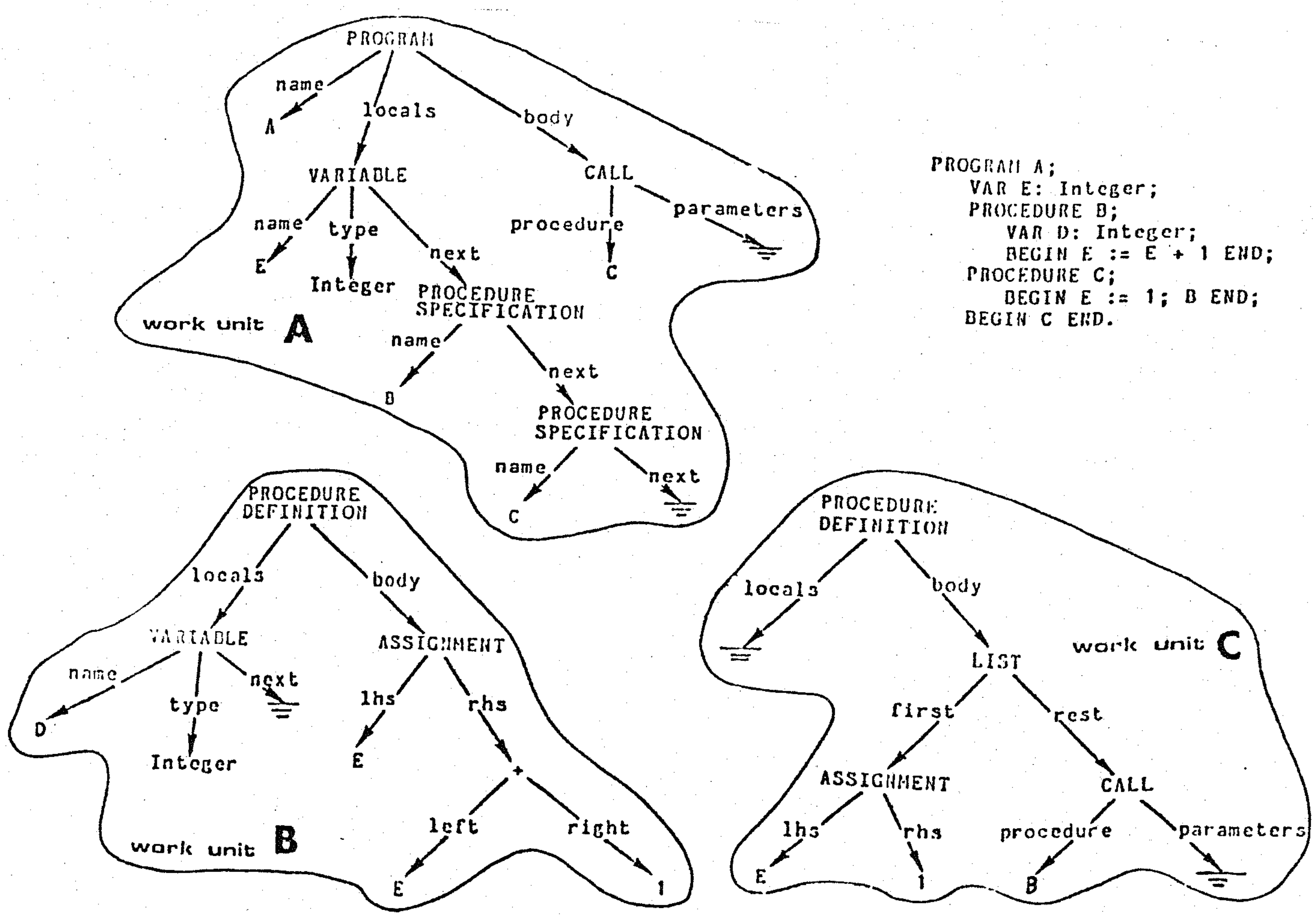


Figure 3. Internal Program Representation.

In general, there also exist constraints on the order in which a particular phase can be applied to the collection of work units. For example, many phases need to process the work units in a top down fashion; i.e., from outermost to innermost procedure. The exact ordering required depends very strongly on the design of the specific phases. A work unit dependency graph (Figure 4) can be used to describe this relationship. Note that there is one such dependency graph per phase.

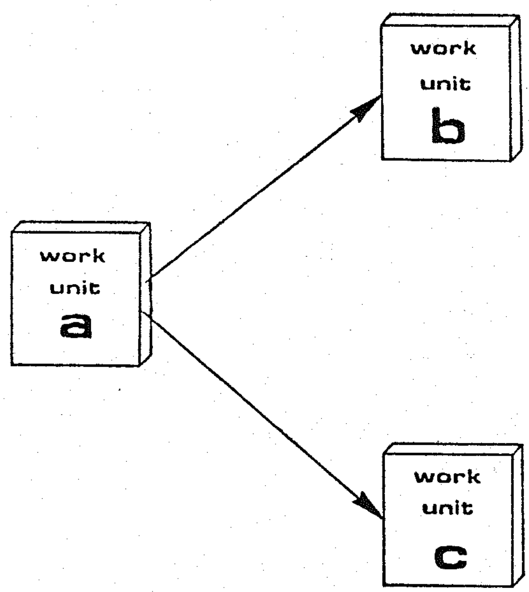
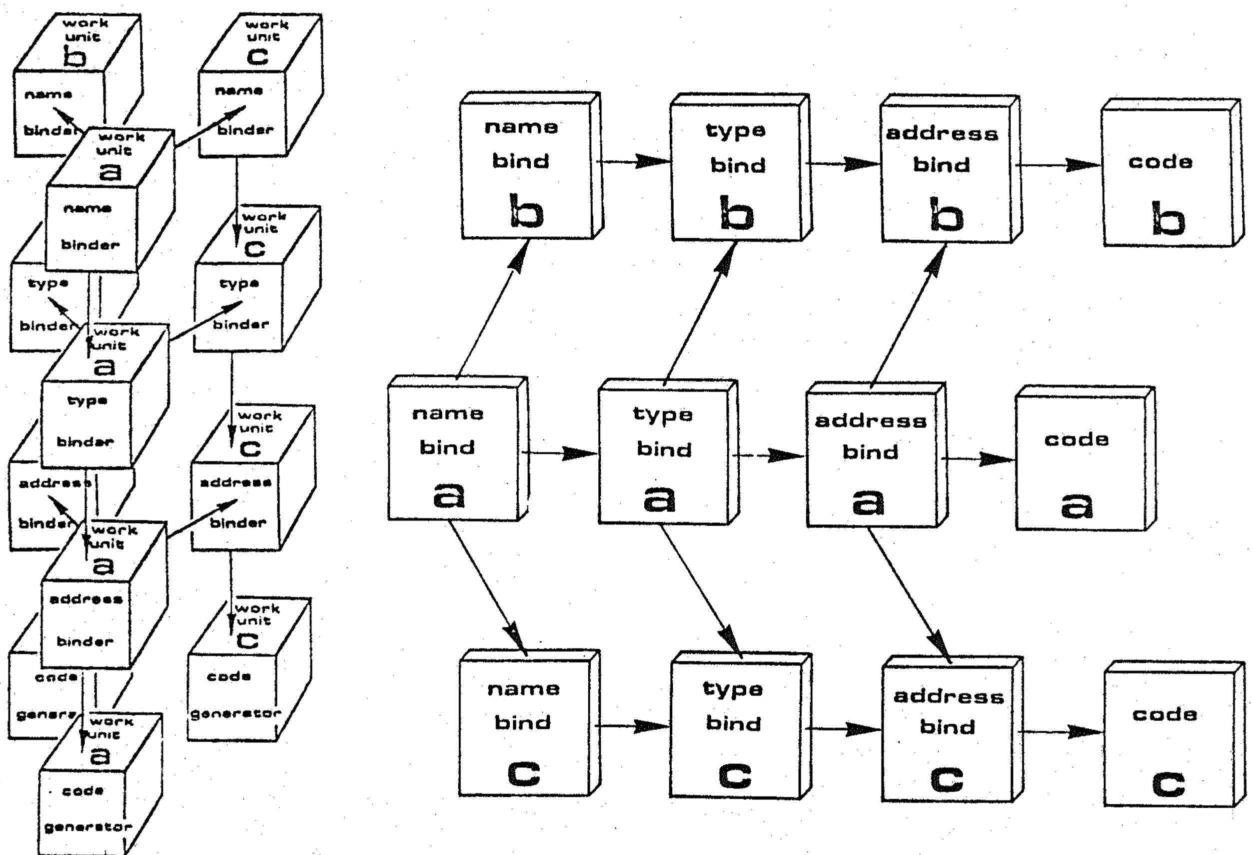


Figure 4. Work Unit Dependency Graph
Dictating Top-Down Processing.

With the compiler sectioned into phases and the IPR partitioned in work units, compilation is a matter of applying all phases to all work units in some suitable order. This ordering is constrained both by the phase dependencies which restrict how a program can be passed through the successive phases and by the work unit dependencies which restrict how the work units can be passed through individual phases. These dependencies can be combined into a single compilation dependency graph by constructing nodes labelled with both work units and phases (the number of nodes is the product of the number of work units and the number of phases). Nodes with the same work unit labels but different phase labels are connected according to the phase dependency constraints whereas nodes with the same phase labels but different work unit labels are connected according to the work unit dependency constraints. Figure 5 depicts a compilation dependency graph obtained from the phase dependency graph of Figure 2, the work unit dependency graph of Figure 4 used for the name, type, and address binding phases, and a work unit dependency graph for a code generator with no constraints.



(a) 3-dimensional.

(b) 2-dimensional.

Figure 5. Compilation Dependency Graph.

The next step is to ensure that no other constraints are added to the minimal constraints imposed by the compilation dependency graph. This will provide the compiler with maximal flexibility in applying the individual phases to the work units. We shall see in the next section that this flexibility is substantial.

In order to achieve this flexibility, it is sufficient to insist on the following.

(1) The compilation dependency graph must be acyclic.
(2) Each phase activation processes exactly one work unit at a time to completion.
(3) The phases do not invoke other phases; i.e., control of the phases is not the responsibility of the individual phases.
(4) Each information slot in the work unit must be owned by a specific phase and only the owner is allowed to modify it when it is processing that unit. Moreover, the slots must be considered uninitialized when the phase is activated.

(5) A phase activation is allowed to inspect information stored in a work unit by a previous phase activation only if that combination of phase and work unit occurs as a prerequisite of the current phase and work unit combination. By a prerequisite, we mean a predecessor in the compilation dependency graph.
(6) No other information about phase ordering may be assumed.

The above requirements effectively characterize our notion of a phase: a modular process operating on modular work units, with localized non-destructive effects, with limited access to other work units, and subordinate to a global phase controller. A design that satisfies the requirements provides the phase controller with maximal flexibility in selecting a linear sequence of phase activations consistent with the compilation dependency graph.

These requirements have several implications on the phases, the work units, and their coupling. In particular,

(7) The phases must be feedback-free; i.e., ordered such that each phase depends only on previous phases. Violation of the feedback-free property implies that the phase dependency graph and therefore the compilation dependency graph has cycles. It also means that the phases could not be processing each work unit to completion. A feedback-free organization of phases is easily achieved and quite commonly used in multi-pass compiler design.
(8) All information about a work unit that is needed after it has been processed by a specific phase must be stored explicitly in that work unit. Passing information between phases by any means other than through the work unit would introduce further constraints on the ordering of the phase activations.
(9) Prior to activating a specific phase on a specific work unit, all prerequisites must have completed. This is essential because the information computed by the prerequisites may be accessed while processing the unit. However, this is the only constraint which the phases controller must abide by since no phase can impose additional constraints on the ordering. In particular, some activations may be done in parallel.
(10) A work unit can always be reprocessed by a phase. This is a consequence of the fact that phases behave in a non-destructive manner and do not rely on their slots having initial values.
(11) If reprocessing of a work unit is required (e.g., if the program is modified via a structure editor), then the amount of reprocessing is dictated by the compilation dependency graph. Thus the design supports incremental recompilation.

The guidelines proposed above clearly separate the control of the phase activations from the design of the phases and the work units. This separation permits the same compiler framework to be controlled by a phase ordering module that implements any one of a number of different strategies consistent with the compilation dependency graph.

Phase Ordering Strategies

Most compilers are irrevocably committed to one particular strategy for sequencing the phase activations. In contrast, our model permits a compiler to be designed that can use any one of a wide variety of phase ordering strategies. The specific strategy used can be easily changed after the fact. To illustrate, we consider two specific alternatives that result in drastically different phase orderings and we discuss their merits.

We define a phase switch minimizing strategy as one that invokes a specific phase on all work units so that the phase can be subsequently discarded and replaced another. This approach is applicable whenever the cost of switching phases is greater than the cost of switching work units. It is well-suited to situations where the complete user program can be maintained in primary memory. With this strategy, it is also possible to completely avoid the costly optimization and code generation of programs containing undefined variables or type violations. It is especially useful when a large compiler is being used to compile small student programs. Many multi-pass compilers employ a phase switch minimizing strategy. However, the work unit is usually an entire program or external procedure. Hence they have no work unit dependencies. Our compiler is the only one that we are aware of that has non-trivial work unit dependency graphs and is still able to use a phase switch minimizing strategy. Figure 6 illustrates one of the 48 possible phase switch minimal orderings consistent with the compilation dependency graph of Figure 5.

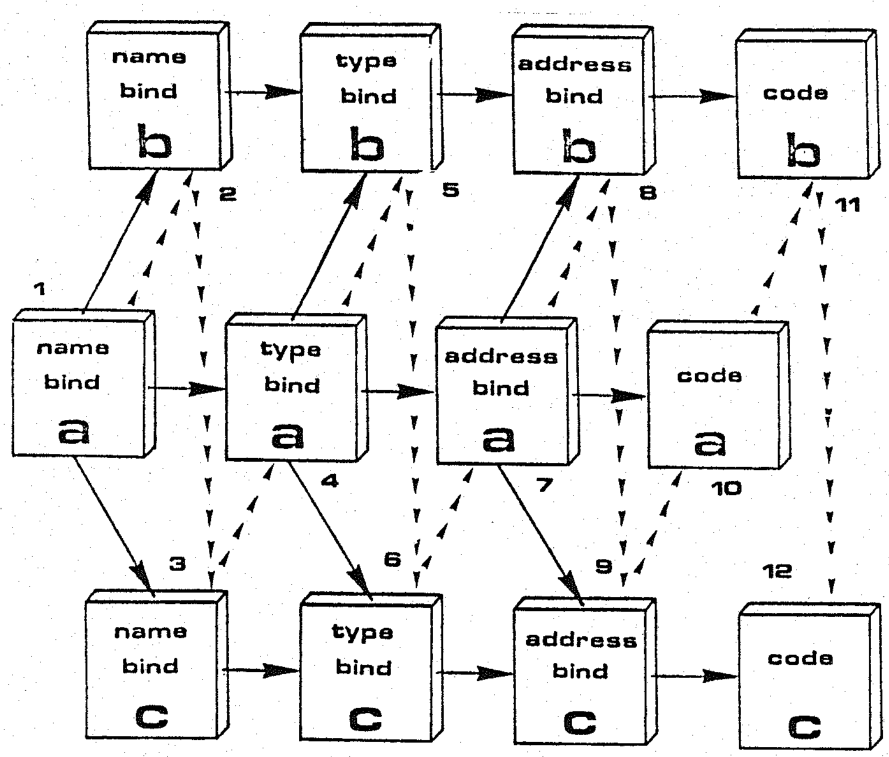


Figure 6. Phase Switch Mininal Ordering.

Alternatively, a work unit switch minimizing strategy attempts to invoke all phases to completion on a specific work unit. The approach is applicable whenever the cost of switching work units is greater than the cost of switching phases. It is ideal if the number and size of the phases is small enough to maintain them all in primary memory. The size of the user program then becomes irrelevant, making it suitable for compiling the large programs found in production environments. An example of a compiler that uses a work unit switch minimizing strategy is Bliss/11 [1]; its work unit is a subroutine. One of the two possible unit switch minimal orderings consistent with the compilation dependency graph of Figure 5 is illustrated in Figure 7.

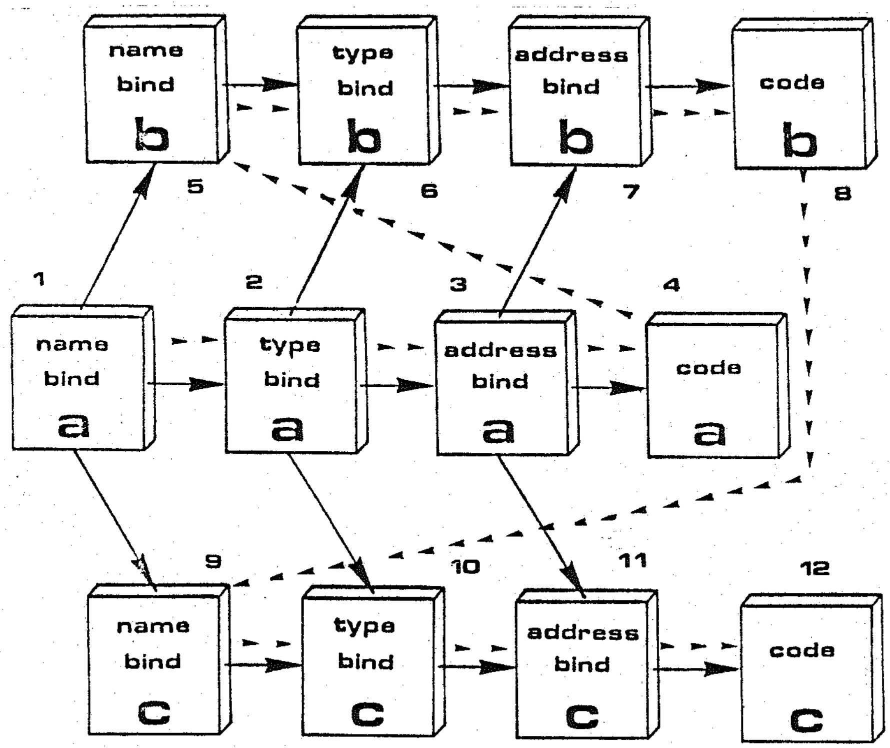


Figure 7. Unit Switch Mininal Ordering.

The most demanding of all compilation contexts is that of an interactive programming environment. In this context, compilers need to retain flexibility in the processing order so as to support incremental recompilation. When changes are made to parts of a program, the extent of the required reprocessing can be determined from the compilation dependency graph. The actual processing order is of course also guided by the same graph according to some specific strategy such as the two mentioned above. Given estimates of the costs of switching phases and work units, it would no doubt be possible to select an ordering that is optimal for that particular instance.

Implementation

We have designed and successfully implemented a production Pascal compiler [2] for the Honeywell CP-6 operating system following the above guidelines. The IPR is an attributed tree structure similar to the DIANA [3] design for ADA, the work unit is that portion of the tree corresponding to a routine, and the phases (subsequent to tree construction) consist of name and type binding, address binding, and code generation. The only difference between our compiler and the conceptual compiler structure discussed in the previous sections is that the name binding and type binding phases in our compiler have been combined to avoid feedback. In order to type bind the Pascal expression R.F, the field F must first be name bound which in turn can only be done after type binding R. This solitary feature of the language causes the name and type binding tasks to be inextricably intertwined.

All phases in the CP-6 Pascal compiler have identical work unit dependency graphs, except for the code generation phase which has a completely unconstrained one. By sharing a single stored work unit dependency graph and by keeping with each work unit an indication of those phases that have processed it, it was possible to avoid constructing an explicit compilation dependency graph.

Although our compiler is currently packaged as a standard compiler, it was designed for eventual integration into an interactive programming environment. Initially, we implemented a work unit switch minimizing strategy but a variation of the phase switch minimizing strategy was later adopted since it proved to be more efficient in our university setting where Pascal is used extensively in programming courses. The variation (Figure 8) differs from a pure phase switch minimizing strategy in that the address binding and code generation phases are always activated in sequence. Only the phase ordering module needed revision --- a one hour effort!

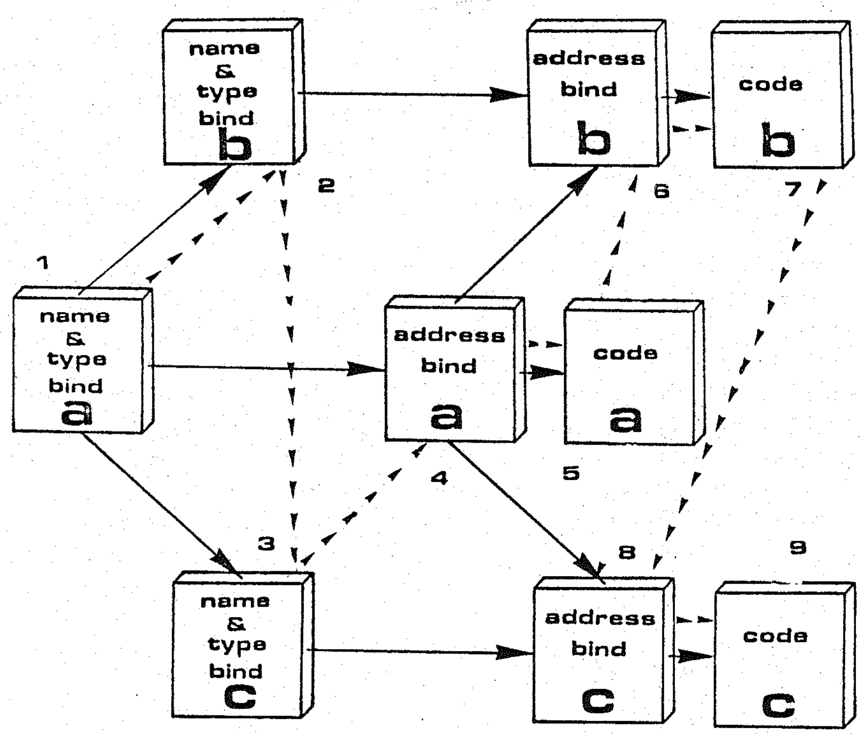


Figure 8. CP-6 Pascal's Ordering.

Conclusions

Several advantages have already been mentioned: the increased modularity, not only of the phases, but also of the work units (all information about a work unit is part of that work unit), the many control strategies that are possible because of relaxed constraints on phase ordering, and the framework that allows operating conditions to be taken into account when configuring the compiler rather than during its design and implementation.

Another definite asset is that the scheme can be easily extended to handle separate compilation by adding a new compiler module that acquires one copy of each essential work unit produced by separate compilations and constructs a composite work unit dependency graph for each phase. The primary reasons that a separate compilation facility is usually so painful to add to an existing compiler are a) the failure to make explicit all information required by subsequent phases of the compiler, b) the failure to localize the modifications made by a phase, c) the failure to design phases that operate in a non-destructive manner, and d) the lack of high-level control of the phases so that they can be used in an environment where some parts of the program exist in an already-processed state.

Additionally, we have shown that it is actually possible to design a compiler that can be effective in both traditional batch and interactive/incremental environments. In batch environments, a flexible phase ordering is advantageous for efficient memory management. In interactive environments, it is necessary to support user modifications efficiently.

On the other hand, requiring that all interphase information be made explicit by recording it in the work unit and insisting that information never be discarded from the work units means that the amount of information is large relative to a more modest implementation. Even in batch environments, a work unit memory manager becomes mandatory unless a large virtual address space is available.

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.

References

[1] Wulf,W. et al. The Design of an Optimizing Compiler, Elsevier, 1975.

[2] des Rivieres,J. and LaLonde,W.R. The Design of CP-6 Pascal. School of Computer Science report SCS-TR-1, Carleton University, Ottawa, (forthcoming).

[3] Goos,G. and Wulf,Wm.A. (eds.) DIANA Reference Manual. Department of Computer Science report CMU-CS-81-101, Carnegie-Mellon University, Pittsburgh, March 1981.

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.