N87-28304

P.56

PCG: A PROTOTYPE INCREMENTAL COMPILATION FACILITY FOR THE SAGA ENVIRONMENT

Joseph John Kimball

Department of Computer Science

University of Illinois at Urbana-Champaign

Urbana, Illinois

July, 1985

PCG: A PROTOTYPE INCREMENTAL COMPILATION FACILITY FOR THE SAGA ENVIRONMENT

BY

JOSEPH JOHN KIMBALL

B.A., Creighton University, 1980

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois

ORIGINAL PAGE IS
OF POOR QUALITY

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

A programming environment supports the activity of developing and maintaining software. New environments provide language-oriented tools such as syntax-directed editors, whose usefulness is enhanced because they embody language-specific knowledge. When syntactic and semantic analysis occur early in the cycle of program production, that is, during editing, the use of a standard compiler is inefficient, for it must re-analyze the program before generating code. Likewise, it is inefficient to recompile an entire file, when the editor can determine that only portions of it need updating.

The pcg, or Pascal code generation, facility described here generates code directly from the syntax trees produced by the SAGA syntax-directed Pascal editor. By preserving the intermediate code used in the previous compilation, it can limit recompilation to the routines actually modified by editing.

1.1 Compilation in Software Development Environments

Within the formalisms developed to aid the software lifecycle, the actual process of writing code is itself a cycle: think, edit, compile (and link-edit if needed), test, and think again.

A software development environment provides tools for program creation and maintenance. In the traditional software development environment, the most visible tools are the editor and the compiler. The division of labor between the two is as follows. The editor is used for entering and modifying code; it is text-oriented, suitable for the entry of any type of text. As a general-purpose tool, the editor cannot provide assistance for any particular language. The compiler, on the other hand, is specific to one programming language, and does two jobs: 1) it must check the source code's syntax and semantics, to ensure that the code constitutes a legal program in the language, and 2) it must then translate that legal program into executable form. Therefore, if the compiler discovers static errors in the source file, it aborts, and the programmer must return to the editing phase to make corrections. The compiler must be run repeatedly merely to obtain error diagnoses, making checking for errors very costly [Campbell and Kirslis] [Medina-Mora and Feiler].

The more helpful of traditional environments provide an automatic facility to drive the compilation and link-editing phase, for separately-compiled programs. The 'make' program [Feldman] under Unix[1] is an example. Its knowledge is embodied in 1) a user-supplied description of the dependencies among the various files, and 2) the file system's timestamp which records when a file was last modified. Given these, Unix make can determine which files must be updated after a modification to one occurs. If a file has not been reconstructed since the files from which it is built were modified,

----

1. Unix is a trademark of Bell Laboratories.

make will recompile and re-link as needed to update the program.

In the traditional environment, the knowledge-rich tools are applied late in the coding cycle: the compiler provides feedback about the legality of the source only after the entire file has been produced, and the make facility uses dependency information only to manage compilation between files.

The earlier a problem is detected, the easier it is to correct. Newer software development environments often try to move the language-specific knowledge earlier into the coding cycle, and to use the information collected by such tools throughout the cycle in an integrated fashion. The environment then has knowledge about the objects it manipulates and their current state; it can respond interactively to errors and anomalies, and it can respond to queries about the objects' state [Medina-Mora and Feiler]. Lisp programming has long benefited from such language-specific environments as Interlisp [Teitelman and Masinter]. The development of language-oriented tools is an active area of research [Campbell and Kirslis], [Donzeau-Gouge, Huet, Kahn, and Lang], [Habermann], [Reiss], [Teitelbaum and Reps]; the programming environment to be provided for a language is now often a consideration in language design [Goldberg] [Teitelman].

The syntax-directed editor is an example of the application of language-specific knowledge early in the coding cycle. Such an editor is knowledgeable about the syntax of a particular language or languages. It ensures that the code entered is correct while the programmer enters it, providing immediate feedback about syntactic (and possibly semantic) errors and misuses. The editor may also provide the programmer with access to its knowledge about the language and about the source being edited--for instance, allowing the programmer to query about the followset of a particular token

[Campbell and Kirslis], or about the attributes of a defined identifier or scope [Reiss], [Teitelman].

A language-oriented editor must perform syntactic analysis, the first phase of traditional compilation. Usually the editor maintains the source file in structured form, as a syntax tree, rather than as linear text [Donzeau-Gouge, Huet, Kahn, and Lang], [Medina-Mora and Feiler], [Teitelbaum and Reps]. When a structured editor is used for program creation, the use of a standard compiler entails the unparsing of the source file, followed by redundant syntactic analysis.

Further, just as the programmer can benefit from the editor's feedback, the compiler can benefit from knowledge of which sections of a source file have been modified through editing. Such information can enable the compiler to recompile only the affected routines within a file, providing a separate-compilation-like facility for languages which do not support separate compilation (or support it only grudgingly).

1.2 Motivation

The SAGA project is investigating formal and practical aspects of computer support for the software lifecycle [Campbell and Kirslis]. Within the SAGA environment, epos is the language-oriented editor. The programmer enters code as with a standard text editor, but can manipulate syntactic entities as well as textual entities; epos incrementally parses and error-checks the code as it is entered.

Epos up to now has not had a semantic-evaluation component; it has only checked syntactic constraints. Also, the editor maintains SAGA files as parse trees, rather than as text. Thus, compiling a SAGA file with a standard compiler entails unparsing followed by redundant syntax analysis.

SAGA Make [Badger] was originally designed for Pascal 6000 on the Cyber; that system supports the compilation of nested routines without compiling the routines which enclose them. Since Berkeley Unix's Pascal compiler pc does not support this, much of SAGA Make's functionality was lost when the SAGA system became Unix-oriented.

In environments which include syntax-directed editors, it is thus most efficient for compilers to leave the task of syntax analysis to the editor; such a compiler would generate code from the parse trees with which the editor works [Medina-Mora and Feiler]. SAGA Make demonstrates that the editor can be recording a modifications-trace as the programmer is modifying a pre-existing file; when such information is available, compilation is most efficient if it only involves the routines which were affected by the re-edit.

The system described here, pcg, is such a compilation facility for Pascal under SAGA. Pcg's symbol table component is a semantic-evaluation component added to epos; its code-generation phase is driven by the SAGA Make facility, and generates intermediate code directly from a traversal of the parse trees used by the SAGA editor. Use of Make enables it to recompile intermediate code incrementally upon re-edits of the Pascal source; this allows the programmer to keep a Pascal program in one unit, as Pascal encourages, but still have the efficiency of separate compilation.

A goal for tools in the SAGA system is that they form standard components which can be composed to form new tools. Pcg demonstrates the composition of SAGA tools to produce a new facility. Besides making use of information generated by epos and Make, pcg uses the SAGA symbol table manager to store and organize the semantic attributes it collects, and to pass this information between phases.

Because it makes full use of the parse information collected by epos, Pcg eliminates the redundant syntactic analysis in compilations generated by the SAGA Make facility. Pcg is also a step towards making full use of the semantic information in the SAGA environment. Its symbol table component serves as a prototype interactive semantic component for the editor. It provides interactive response to semantic errors, and an ability to query the symbol table about the attributes of identifiers.

1.3 Previous Work

Above we noted the traditional role of compilation in programming environments; other divisions of labor between editor and translator are possible.

The classic alternative is the interpreter-based system which is standard for Lisp. Source code is maintained internally in linked-list form, which can be directly executed by the interpreter. The system routine which parses user input thus produces a representation which is simultaneously the internal representation of the source and its executable representation. Runtime access to the source's representation supports sophisticated debugging facilities. Use of a run-time symbol table enables the programmer to replace routines at will. Because compiled routines are likewise managed by the interpreter, and communicate with other routines via the symbol table, they too may be replaced freely; but they lose most of the benefits of the debugging facilities. Interlisp [Teitelman and Masinter] is an advanced example of such a system.

MENTOR [Donzeau-Gouge, Huet, Kahn, and Lang], [Donzeau-Gouge, Lang, and Melese] also maintains program source in structured form; code for various languages is maintained as abstract syntax trees. General tree-manipulation tools are provided, and may be composed into procedures for manipulating particular languages. Editing is tree-oriented. MENTOR provides a variety of sophisticated interpreters to evaluate and transform the abstract syntax trees which represent programs. Some perform semantic checking. Compilation is performed with standard compilers, after unparsing the source into text form.

In the Cornell Program Synthesizer [Teitelbaum and Reps], source files are maintained as abstract syntax trees with associated symbol-tables, and an interpreter is provided which can directly execute these trees. Thus, although a compiler-oriented languages is used, compilation does not occur. The interpreter returns to the editor upon encountering a discontinuity in an incomplete tree, so a partial program can be run up to that point; this allows editing and testing to be highly interleaved. The standard Synthesizer is an educational rather than a development tool, and does not support compilation to machine code, nor separate compilation.

PECAN [Reiss] attempts to provide the user with multiple views of a program, including its syntax, semantics, and run-time behavior. Its compiler is oriented toward giving the user access to the semantics of programs. The user may query about the symbol table associated with a particular scope, including identifiers and their attributes; the compiler also supports the display of the expression tree representation of a given expression. PECAN's design includes an interpreter, which will execute the internal form of programs.

Cedar [Teitelman] is a compiler-oriented language whose environment attempts to be interactive and experimental like interpretive environments. To this end, it provides both a compiler and an interpreter, which can interpret the full range of expressions of the language. Cedar's interpreter allows its user to query about the type of expressions, and evaluate type-valued expressions. The system keeps track of which files need to be recompiled, though dependency-analysis is not performed.

The Incremental Programming Environment [Medina-Mora and Feiler], under the Gandalf project [Habermann], is the system which most closely resembles pcg. It tries to provide the facilities and flexibility of interpreter-based systems entirely via compilation technology, and is oriented toward the production of long-lived programs. IPE generates machine code from the syntax trees which its syntax-directed editor produces, and performs incremental recompilation on the procedural level. Rather than generating a new executable object via a standard link-editor, as pcg does, IPE provides an incremental linker which can replace the machine-code version of a changed procedure within the executable object; it recompiles procedures in the background, rather than upon user request, as pcg does. Unlike pcg, it includes a debugger which is integrated with the rest of the system.

1.4 Overview

The design and implementation of pcg is described here. Chapter 2 details the overall structure of the major components of the system, and their design goals. Chapter 3 describes the implementation of pcg's first phase, which maintains the symbol table. In chapter 4 we look at the implementation of the second phase, which performs incremental recompilation. Chapter 5 summarizes what was accomplished, and points up shortcomings and directions for further research. Appendix A details the differences between pcg's Pascal and ANSI Standard Pascal, and between pcg's Pascal and Berkeley Pascal. Appendix B is a Unix manual page for the pcg incremental recompiler.

Chapter 2

DESIGN

Here we look at the design goals which pcg addresses, with particular attention to how it is designed to interact with the other tools in the SAGA system.

2.1 Overall Structure

Pcg decomposes into two phases which must be applied in sequence. In the semantic processing phase, pcg's symbol table component generates or updates the symbol table, given the program source in the form of a SAGA parse tree. In the compilation phase, the incremental recompilation driver of pcg takes the parse tree and symbol table, and compiles the program. The incremental recompilation driver relies on the code generator for the actual generation of intermediate code, which is transformed into machine code by the latter phases of the Berkeley Pascal compiler.

The symbol table component has two configurations. The editor-resident configuration constructs a symbol table concurrently with the editing of program source, and so can provide interactive feedback to the editor's user. Normally, the editor-resident symbol table component is invisible to the user. If the user makes a semantic error, the symbol table component opens a window to emit an error message; also, the user can request information about the objects in the symbol table. The symbol table component can also be configured as a standalone program, which traverses a static parse tree to construct or update the symbol table for that program. This configuration is meant to be called by other SAGA tools.

When a syntactically-correct parse tree and semantically-correct symbol table are available, compilation can occur. This phase of pcg is invoked just as a standard compiler would be. The incremental recompilation driver controls the compilation process, using the modifications-trace generated by SAGA Make to determine which routines must be recompiled. For the routines which have been modified, or newly created, the driver calls the code generator, to generate intermediate code. The driver merges the new code with the unchanged code from previous compilations, and invokes the latter phases of the Berkeley Pascal compiler to complete compilation.

Figure 1 shows the interaction between the SAGA Pascal editor and pcg; the editor-resident symbol table component is displayed. The pcg system is separated into self-contained modules with well-defined interfaces, so that the modification or replacement of one component will not disrupt the functionality of the others.

Although SAGA syntax-directed editors have been generated for several languages, the Pascal editor is the base editor. Thus, pcg compiles Pascal. The language it accepts is currently a Pascal subset, which soon will be extended to full Pascal; see Appendix A. The particular Pascal dialect is Berkeley Pascal [Joy, Graham, and Haley].

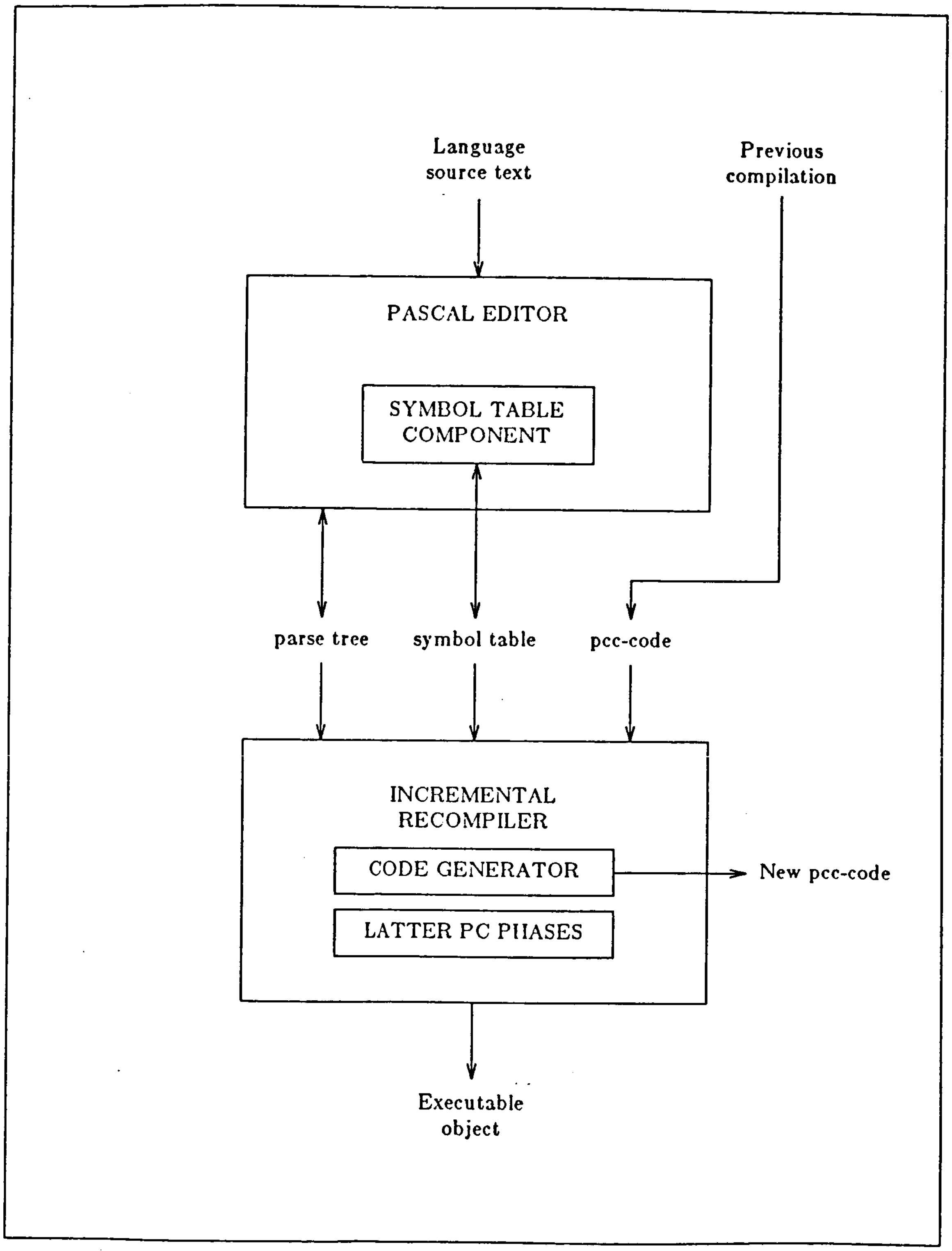

Figure 1. The SAGA Pascal editor and pcg.

Below we look at the design goals for the semantic phase's symbol table component, and the compilation phase's incremental recompilation driver and code generator.

2.2 Semantic Processing Phase

The symbol table component has two configurations, and serves as a prototype semantic component for the SAGA system. It had several design goals.

First, a goal for SAGA tools in general is that they form standardized, reusable modules which interact through well-defined interfaces [Campbell and Kirslis]. Therefore, the symbol table component tries to make as few assumptions as possible about epos and the internals of the parse tree files. To this end, pcg's symbol table component uses only the standard node-access interface to obtain parse-tree information; to communicate with the editor proper and with the programmer, it uses only the standard semantic-evaluation interface. Though dependence on the structure of the Pascal grammar is unavoidable, the symbol table component only assumes that the parse tree is well-structured with respect to that grammar, and that the tree's abstract internal relationships will not change without explicit editing actions. The symbol table component does not, for instance, store the internal node-indices of identifiers whose attributes it records, and thus the SAGA tree compactor could be run on a parse tree without invalidating the associated symbol table.

A second goal for the symbol table component is an ability to be configured as a semantic evaluator, resident in the editor, or as a separate non-interactive process which performs semantic checking and symbol-table construction. Semantic evaluation can degrade the performance of syntax-directed editors [Medina-Mora and Feiler], and so the availability of a standalone configuration adds flexibility which may be needed when system resources are strained. In this configuration, the symbol table component resembles the semantic processing of a more traditional batch compiler.

A third quality sought in the symbol table component is the ability to collect information for two related but different tasks. 1) As the symbol-table constructor for the pcg compilation system, the symbol table component must collect the information needed for compilation. 2) Like a standard compiler, it also must be able to provide diagnostics about semantic errors and anomalies; additionally, to make use of the unique interactive capabilities of an editor-resident evaluator, the in-editor version can respond to user queries about the attributes of identifiers.

Fourth, in its role as a prototype semantic evaluator for epos, the symbol table component of pcg provides some support to incremental modification of the source program. Thus, when the user modifies the parse tree by re-editing, the in-editor symbol table component responds with consistent updates to (or deletions of) symbol table entries.

Finally, the symbol table component uses the SAGA Symbol Table Manager [Richards] for storing, organizing, and retrieving the attributes it detected. This is the first major exercising of the symbol table manager, which was designed as a general facility for software development environments in which multiple tools would have to exchange semantic information.

2.3 Compilation Phase

The compilation phase is managed by the incremental recompilation driver; the actual generation of intermediate code is performed by the code generator. We first look at the design decisions for the incremental recompiler as a whole.

The first design decision for the incremental recompiler was that the replacement unit for incremental updating would be the procedure. Interpreter-based incremental systems can update their executable code on the expression level [Teitelbaum and Reps]; interpreted code need not deal with the peculiarities of hardware, and can be designed to reflect the structure of the source language. By contrast, compiled code often bears only implicit structural similarity to the original source. Because the procedure or function is a self-contained unit with a well-defined interface, recompilation on the procedural level is a reasonable implementation for incremental recompilation [Medina-Mora and Feiler].

The next design decision which determined the structure of the incremental recompiler was that it would generate intermediate code, and use a standard machine-code generating second pass to complete compilation. A code generator was developed, which generates intermediate code from a parse tree and symbol table. The code produced is binary "portable C compiler" intermediate code [Kessler], hereafter called (with some inaccuracy) 'pcc-code'. This intermediate representation is a binary, packed version of the original portable C compiler intermediate code. It is largely machine-independent. The Berkeley Pascal Compiler pc, and FORTRAN compiler f77, use pcc-code as their interface to the common machine-specific backend, which generates machine code. Use of this interface enhances the portability of pcg among Unix systems, with the other SAGA tools.

2.3.1 Incremental Recompilation Driver

When a user invokes pcg, the component of the pcg system that responds is the incremental recompilation driver. This component is the top level for the compilation phase of pcg. Given a SAGA Pascal file, it does what is necessary to ensure that its executable object is up-to-date with respect to its source.

The first design decision for the driver was that it would use SAGA Make's modifications-trace as a guide to generating new intermediate code. SAGA Make [Badger] was designed to be a largely language-independent facility in two phases. Its first phase, resident in the editor, keeps track of which routines are modified, or have their environments modified, such that they must be recompiled. Make's second phase used this modifications-trace to build a shell script which would recompile the program, and then it executed that script; this phase suffered from Berkeley Pascal's lack of facilities for compiling nested routines.

A goal met by virtue of using Make is that the code generator need only recompile the minimal number of routines necessary for updating the pcc-code and regenerating the object [Badger]. Pcg therefore preserves the pcc-code file which resulted from the most recent compilation, so that unmodified routines can be reused. Alternately, the incremental recompiler can be ordered to discard the old pcc-code and regenerate the entire program from scratch.

A third property sought in the design of the incremental recompiler is that its user interface appear as similar to that of a standard compiler as possible. Pcg can, therefore, be invoked from within Unix make scripts just as can pc. Similarly, pcg can easily interface with configuration management schemes which make use of standard compilers [Kirslis, Terwilliger, and Campbell], [Estublier, Ghoul, and Krakowiak].

This decision implies that pcg does not perform compilations in the background during editing, as does IPE [Medina-Mora and Feiler]. However, if such a system were desired, epos's capability of spawning filter processes could straightforwardly implement it.

The incremental recompilation driver tries to behave reasonably if given a SAGA file for which no symbol table, or no modifications-trace, exists, invoking the standalone symbol table component to build a symbol table if necessary.

2.3.2 Code Generation

The code-generator is modeled on the first pass of the Berkeley Pascal compiler. It produces pcc-code, which the driver then provides to the later phases of pc.

As a simplifying assumption, the code generator follows pc's internal logic and algorithms wherever possible. The Berkeley Pascal compiler has proven itself as a tool for software development; most of the SAGA system, including most of pcg itself, is compiled with pc. Pc provides a reasonable separate compilation facility, and the ability to call routines written in other portable C compiler - based languages, including Unix system calls [Joy, Graham, and Haley].

This design decision allows pcg to use pc's latter phases unchanged. Basing the code generator on the Berkeley compiler also enables a simple test of its output: if the pcc-code that it generates differs in structure from that generated by pc for a given Pascal program, then something untoward is going on.

For the sake of modularity, the code-generator's job was limited to producing pcc-code, given a parse tree, a symbol table, and a node which is the root of a subtree for a routine. Managing the further phases of compilation is left to the incremental recompiler.

The next chapters provides an overview of the implementation which tries to meet these criteria.

Chapter 3

SEMANTIC PHASE IMPLEMENTATION

These two chapters examine significant implementation details of pcg. In looking at the issues in its implementation, we pay special attention to pcg's interaction with the other tools in the SAGA system, and with the Berkeley Pascal compiler.

In this chapter we will look at the implementation of the symbol table component of pcg, which performs the semantic phase of pcg's processing; in the next, we will look at the compilation phase.

3.1 Semantic Processing

The problem of semantic analysis of programs is nontrivial. The syntactic task of parsing has been simplified by the development of the context free grammar formalism [Aho and Ullman], to the extent that automated tools such as YACC [Johnson2] and Mystro [Noonan and Collins] can construct parsers from a formal description of a grammar. But no fully satisfactory formalism for semantics has been developed, although attribute-grammar based systems for automated semantic analysis and compilation are an active research area [Paulson], [Ganapathi and Fischer], [Reps].

[Reps] distinguishes between imperative and declarative semantic evaluators. The former is procedurally specified, the latter uses a formal specification to enable the automatic generation of an evaluator. Imperative evaluators must specify both semantic actions, which are to be performed upon the insertion of program text, and semantic retractions, which update the symbol table when a deletion occurs.

The declarative method attempts to avoid the need for retractions, by eschewing the use of a global symbol table whose state must be kept consistent with the state of the syntax tree. Rather, it stores semantic information locally, throughout an attributed tree. It is unclear whether such localized context is sufficient in general [Johnson and Fischer]. An attribute-grammar based evaluator, combining both declarative and imperative aspects, is under development for the SAGA environment [Beshers and Campbell]. In the meantime, the symbol table component of pcg provides the pcg system with an ad-hoc, imperative mechanism for collecting semantic attributes and error-checking SAGA Pascal source.

3.2 The User Interface

To the user of epos, the symbol table component of pcg is merely another feature in the editor. The editor proper reports when the user enters syntactically-incorrect text, and highlights the unparseable portion of the program. Similarly, the symbol table component opens a window and emits an error message when the user enters a semantically-incorrect declaration or statement; the offending string within the program is highlighted. If the user modifies a declaration in such a way that previously-entered text which depends on that declaration is now incorrect, the error is reported and the now-incorrect strings highlighted. The attempt to re-declare an identifier, within the same block as a previous declaration of that identifier, causes the generation of an error message, the highlighting of the offending identifier, and the disregarding of the new declaration. If a new identifier is entered with a semantically-malformed declaration, the identifier is entered into the symbol table, but its attributes note that it is misdeclared. Upon correction of an error, the corrected code is displayed in the normal font again.

The editor-resident symbol table component also provides the user with the ability to query the symbol table about the attributes of currently-defined identifiers, including both standard and user-defined types, variables, and routines. A similar facility is provided in PECAN [Reiss] and Cedar [Teitelman]. This facility is particularly useful in a separate compilation environment; for instance, one can check the number and types of the parameters of an imported routine, before entering a call to that routine. It is also useful when the symbol table component informs the user that a symbol has been misused; the symbol's attributes can be inspected, to determine how to correct the mistake. Normally, the search for an identifier starts in the current block and proceeds outwards until a definition is found. It is also possible to enquire about symbols defined within contexts which are nested within the user's current context; one prefixes the identifier with a path of context names separated by dots. For instance, to enquire about the field 'i' within the record 'rec', declared within the nested function 'ftn', one enquires about 'ftn.rec.i'.

The standalone symbol table component is oriented toward use as a tool by other tools, unlike the editor-resident configuration of the symbol table component. The standalone configuration is a self-contained program that takes one argument, a SAGA file name. It loads an existing symbol table, if present, and then traverses the parse tree, to produce an updated symbol table. Nodes generating semantic errors are marked in the parse tree, and the error messages written to standard output.

3.3 Overall Structure

The task of semantic analysis is significantly complicated by a need to support incremental modifications of the program source. Existing declarations can be modified or deleted, necessitating the change or removal of symbol table entries; such changes can correct or invalidate other entries which reference the objects declared. Existing executable statements are also subject to modification or deletion, and must be re-checked for legality. The user of a syntax-directed editor can enter syntactically-incorrect or incomplete code, but the symbol table must not thereby be left in an inconsistent state.

Pcg's symbol table component is an imperative evaluator, since the semantic analysis is specified procedurally; it binds action and retraction procedures to grammar productions. When the editor reduces by such a production, or when the standalone symbol table component encounters such a production during tree traversal, then the associated procedure is invoked. The procedure traverses the affected subtree to gather needed information, and then it updates the symbol table.

Below we explore the linkages between the grammar of Pascal and the symbol table component; this provides background for understanding the use of actions and retractions. Next we look at the symbol table component's use of the SAGA symbol table manager; in this context, the use of action and retraction routines is described.

3.4 The Symbol Table Component and the Pascal Grammar

To support incremental evaluation, pcg's symbol table component must respond appropriately to modifications in program text. To this end, it distinguishes three special subsets of the production rules in the LALR(1) Pascal grammar used by the current Mystro-based SAGA editor [Aho and Ullman], [Noonan and Collins]. These subsets are the action productions, the checkable productions, and the user productions.

3.4.1 Action productions

Certain productions are distinguished as being 'action productions'. When an action production is encountered, an entry is installed into the symbol table. In general, an action production roots a subtree of least height such that the subtree contains all the information needed to determine the attributes of an identifier. By delaying until all information needed is present, the symbol table component does not need to maintain external data structures containing partial attributes, which would have to be handled specially if user input were interrupted or discovered to be syntactically malformed. On the other hand, by not delaying until a reduction to a higher-level nonterminal is performed, the symbol table component can respond most immediately to erroneous input.

3.4.2 Checkable productions

A single production lies in the set of 'checkable productions'. This is the production whose left hand side is <statement>. Within a statement, expressions must be typechecked, the use of expressions must be checked for legality, and references to declared entities must be recorded. Such actions are performed when the symbol table component encounters a reduction to <statement>.

3.4.3 User productions

The third subset of Pascal grammar rules is the set of 'user productions'. These are the productions which contain user-supplied terminals; reduction by such a grammar rule, during the non-reparsing first phase of the parse, indicates that a tree modification has occurred, which should be analyzed. The user productions are significant in the editor-resident semantic phase. The epos parser is incremental, and attempts to reparse the minimal amount needed to fit changes into the parse tree [Ghezzi and Mandrioli]; where possible, it shifts entire subtrees, rather than their frontiers. It is thus possible that a user modification can be accommodated into the tree, without the reparse propagating up to the action or checkable production which is its ancestor. If a reduction by a user production was not eventually followed with a reduction by an action or checkable production, the symbol table component detects the need to climb to that ancestor and re-evaluate the subtree it roots.

3.5 The Symbol Table Component and the Symbol Table Manager

Much of the work of the symbol table component is simplified by its use of the SAGA symbol table manager.

3.5.1 Attributes

Use of the symbol table manager is organized around the attributes which one sets up for the given application. Symbol table manager primitives are used to record symbol definitions and symbol references; (attribute, value) pairs can be attached to such entries. The symbol table manager's user must specify what type the value of an attribute may take on.

Attributes are identified by strings stored in the symbol table's strings section; referring to a particular attribute is accomplished by a reference to that string's internal identifying tag. Thus, for every attribute one defines, one must maintain a variable containing that tag, to enable one to refer to the attribute. This is an impetus toward defining record-valued attributes; such an attribute-complex can hold all the values associated with a given class of symbol.

By making the user-defined attribute type a variant record, it can be used for several attributes. The symbol table component uses the user-defined attribute type for four such 'compound attributes'; the two most important are called NameAttributes and TypeDefAttributes.

The symbol table component's use of these attributes is straightforward.

Consider an example of an action routine. When the symbol table component encounters the production

<var_decl_list> ::= <variable_list> : <type>

it invokes an action routine to inspect the <type> subtree. If it is an actual type definition, then the subtree is traversed and the attributes of the type collected. For example, if the subtree defines a subrange type, the host type and endpoints are recorded. The routine returns an anonymous type-definition symbol, which has one attribute containing the description of that type. Alternately, the <type> subtree may not be a new type definition, but an identifier: a reference to a previously-declared type. The symbol table entry bound to that identifier is retrieved, and its NameAttributes inspected to determine which anonymous type-definition symbol it names.

In either case, once the <type> subtree has been handled, another action routine traverses the list of variables. For each, it inserts a non-anonymous symbol, to be known by the identifier indicated; the new symbol's NameAttributes specify that it names a variable, whose type is that previously-obtained type-definition symbol.

The incremental parser within epos must sometimes reparse previously-analyzed code, to analyze new material inserted into that code. The possibility arises that an action routine would be called a second time, causing a spurious "identifier previously declared" error. An addition has been made to epos's semantic interface which notifies the symbol table component when a reparse moves into previously-parsed text; action routines are not called for such reductions.

3.5.2 Contexts

With the symbol table manager, when one inserts a symbol definition or symbol reference, one indicates the 'context' in which to place it. The main program, each procedure, each function, and each record type, has an associated context. Identifiers declared within blocks or records are stored within their contexts. To retrieve a symbol, given an identifier, one specifies a context in which to search; contexts can be nested, and searches proceed from an inner context outward. This makes the implementation of Pascal's block-structured scoping rules trivial.

More complex context interactions are generated by the use of grafted contexts. Thus, for example, when pcg enters the scope of a Pascal 'with' statement, it grafts a temporary context onto the current block's context. When a variable is encountered, the search for its definition is first performed in the context of the indicated record, seeking the identifier as a field; then in the current block, seeking it as a variable; and then outwards in any outer blocks. Variables and fields can therefore be handled by the same code; use of the symbol table manager promotes the orthogonal manipulation of symbols.

3.5.3 Symbol References

Besides symbol definitions, the symbol table manager also supports the recording of symbol references. If a symbol is referred to in a given context, a reference entry can be made, and attributes given to it; the symbol definition can be recovered from the symbol reference, and any recorded references can be recovered from the definition. Further, if a definition is deleted, but there exists a definition of that identifier in an outer block, any references made to the deleted symbol becomes attached to the now-visible outer definition.

This automatic action of the symbol table manager is very useful in dealing with deletions in an incremental environment. In the standard Cornell Program Synthesizer, for instance, the deletion of a declaration invalidates the entire symbol table, and necessitates re-traversing the entire parse tree to build a new one [Teitelbaum and Reps].[2] In pcg's symbol table component, outer blocks are not invalidated, since the deleted symbol was invisible there, and any nested blocks which do not refer to the deleted symbol need not be re-evaluated.

3.5.4 Retractions, Attributes, and References

We saw above that the action routines are grammar-driven. In contrast, the retraction routines are driven more by the structure of the attribute-records. The top-level retraction routine traverses the subtree given to it, seeking definitions of identifiers. On encountering such a definition, the identifier's attributes are retrieved from the symbol table, and further actions are based on those attributes. Consider the variable declaration described above. The variable's symbol table entry must be deleted. The type recorded for it is also inspected. If its attributes indicate that no identifier was bound to it, then the type definition entry is

----

2. This is handled more economically in Synthesizer-Generator based systems, which use attributed trees rather than a standard symbol table [Reps].

deleted. Otherwise, the entry which records that the variable referenced the type is deleted.

If references to the deleted entry existed, then the contexts which made those references are noted. Upon completion of the retraction, those contexts are re-evaluated, to ensure their validity. Re-evaluation consists simply of the retraction of entries defined in the routine's subtree, followed by a new tree traversal to re-install these entries and re-inspect the routine's executable statements.

3.5.5 Other Features and Limitations

Each symbol table primitive returns an error code. This provides considerable consistency-checking to the symbol table component; if an internal error occurs, then at some point a symbol-table primitive will be unable to complete its task, and an error will be reported.

Limitations of the prototype symbol table manager also affect the symbol table component. No provision is made for anonymous symbols, nor for lists of symbols; the symbol table component must simulate these features.

The symbol table manager is oriented toward the support of separate compilation, by allowing multiple symbol tables to be open simultaneously; however, the support presently provided is limited by the requirement that each such table have a unique permanent identifier. This prevents the re-use of standard modules, if the permanent-ids assigned to them clash with the identifiers of other modules already in use. A new version of the symbol table manager has been proposed; this new version will provide a virtual naming scheme for multiple open tables. Because this version is not currently available, pcg does not yet support separate compilation.

In the Berkeley Pascal model of separate compilation, included header files contain declarations of external entities; these are considered to be global, that is, declared at the level of the main program context. Although the incremental recompiler can compile separate code modules, the limitation mentioned above makes it is currently impossible for references to be made across modules. Enabling separate compilation in pcg should not be difficult when the new facility becomes available.

The next chapter is an overview of the implementation of pcg's next phase, the incremental recompilation phase.

Chapter 4

COMPILATION PHASE IMPLEMENTATION

Here we examine some of the issues involved in the implementation of the compilation phase of pcg. The major work of compilation is performed by the code generator, which generates pcc-code from a SAGA parse tree and a symbol table. Incremental recompilation is achieved by the incremental recompilation driver, which calls the code generator as needed to generate new code for modified routines. First, we look at the code generator.

4.1 Code Generation

The code-generator is very similar to the pc0 phase of the Berkeley Pascal compiler. It is essentially a translation into Pascal of the relevant parts of that program; instead of pc's namelist and parse tree, the SAGA symbol table and parse tree are its input. As output, it produces the same sort of Portable C compiler intermediate code as pc0 produces.

To examine the code generation component of pcg, we will first look at pcc-code itself, and its use in representing Pascal programs. Next we view the overall structure of the Berkeley Pascal compiler, and the system it implements. Then we will be ready to examine the general structure of the pcg code generator.

4.1.1 Pcc-code

The structure and content of pcc-code is described in [Kessler]; the philosophy and organization of the Portable C compiler is detailed in [Johnson1].

Pcc-code is a postorder linearization of the binary expression trees, and flow-of-control operators, produced by the Portable C compiler to represent C code. It makes explicit the content of the original C program, and decomposes it into simpler structures. For instance, in pcc-code, all operators and operands are explicitly typed, and needed conversion operators inserted. Also, C's structured statements are converted into simple tests and jumps.

Much of pcc-code is machine independent. The first pass is required to handle certain machine dependent constructs, such as routine prologues and epilogues, the code for switch (that is, case) statements, and initializations. This is done by emitting assembly code which will be passed unchanged through the next pass, which generates assembler from pcc-code.

Since pcc-code was designed to represent C, there is some mismatch to be dealt with in representing Pascal code. To represent Pascal expressions, C's wealth of operators are more than sufficient; many pcc-code operators are never used by pc0. On the other hand, Pascal's rich type structure sometimes requires simulation; several Pascal types (for instance, sets) are by default represented as C structures, and operations on these types are implemented by library functions. (The overhead thus incurred is obviated somewhat by the pc2 phase of the Berkeley compiler, described below.) C's structure type is convenient for such use because it is a structured type which may be the target of assignment, may be passed to functions, and may be returned by functions. (But C's support for these operations on structures causes some complication in pcc-code; pcc-code must assume, for instance, that the value of a structure-valued expression is actually a pointer to a structure, rather than the structure itself.)

4.1.2 Structure of the Berkeley Pascal Compiler

The Berkeley Pascal compiler is a five-pass compiler. The first pass, pc0, does syntax analysis, semantic checking, and generation of pcc-code. The second pass, pc1, is actually the f1 pass of the f77 FORTRAN compiler; this is the pass derived from the second pass of the Portable C compiler, which takes binary pcc-code as input and produces assembler as output. The resulting assembly language is the input to pc2, the inline expander. This filter passes most of the assembler unchanged; calls on frequently-used system functions are expanded in place into the assembly code which implements them.

Pc2's output is given to the Unix assembler as, which produces unlinked binary. The pc3 phase examines the symbol tables of binaries produced in this way, prior to linking; it does several checks on the use of globally-visible routines and variables, to enforce the rules of separate compilation in Berkeley Pascal. Finally, the binary is link-edited via Unix's ld, to produce an executable object.

Because the pcg code generator produces pcc-code such as the pc0 phase would produce, pcg can run the latter four phases of pc unchanged. Thus, pc0 is the pass most of interest here.

Pc0 is driven by its YACC-based parser [Johnson2]. The parser constructs the parse tree such that the structure of a subtree can be determined by examining its first node. As the parser recognizes declarations, routines are invoked to make entries in pc0's namelist (symbol table). The structure of namelist entries is a bit baroque, consisting of many overloaded fields, rather than a variant record structure such as is encouraged by the SAGA symbol table manager. Whenever the parser recognizes a complete procedure, function, or program, a function is invoked which traverses the resulting subtree simultaneously to check semantics and to generate pcc-code.

The runtime system created by the Berkeley compiler is essentially that of the Berkeley Pascal interpreter px, as described in [Joy and McKusick]. Px defines many system functions to implement both Pascal operators and built-in routines, such as the input and output procedures. This simplifies the use of this run-time system with C-oriented pcc-code; where pcc-code is deficient, the appropriate library function can be used. The px runtime system is almost purely stack oriented. The objects operated on are assumed to be on the stack, or else in the heap area, and are operated on by the interpreter's Pascal-oriented operators. In contrast, the pc system's use of pcc-code enables it to make use of the abilities of the f1 code generator, which generates assembly code targeted for the actual hardware, and attempts to place operands in registers as much as possible. Pc uses the stack for activation records, structured objects, parameter-passing, and extra temporaries. A display is maintained for referencing nonlocal variables from nested routines.

4.1.3 Structure of the Pcg Code Generator

The interface to pcg's code generator is simple. It takes a node, a context, and a job-specification; the node must be the root of a procedure, function, or program subtree, and the context must be the symbol-table context associated with that routine. Based on the job-specification, the code generator either generates code for the indicated routine, or else performs semantic checks on the statements within the routine.

For ease of interfacing with the other SAGA tools, particularly the symbol table manager, the code generator is implemented in Pascal. The low-level routines which actually produce the binary pcc-code are written in C, as are a set of routine which are used for bit-level operations which are occasionally necessary.

[Medina-Mora and Feiler] note that an advantage of compiler-based environments over those which are interpreter-based is the ability to produce code for a target machine which is different from the host on which the environment is running. The current implementation of the pcg code generator is targeted for the VAX[3]. The pc sources can be configured to generate code for the VAX or for the MC68000, and this capability has been provided in pcg, although the 68000-oriented code-generator has not been tested.

The pcg code generator routines can be partitioned into four sets: those which interface with the symbol table; those which actually walk the parse tree and generate intermediate code; those support routines which implement the machine-dependent aspects of code generation; and those support routines which implement the aspects of code generation dependent on the Pascal runtime system.

----

3. Vax is a trademark of Digital Equipment Corporation.

4.1.3.1 Symbol table interface.

To prevent too tight a coupling between the symbol table component and the code generator component, all symbol table accesses are isolated into a set of routines which are invoked to query the symbol table, and to change the context. Thus, for example, predicates are provided to indicate the attributes of types and variables; the isintegral predicate returns true if its argument is type integer, or a user-defined type which is a subrange of integer. Similarly, graftrecordcontext grafts a temporary context onto the current context, to implement the scoping effect of a Pascal 'with' statement or field selector. This modularity should ease the transition to the attribute-grammar based evaluator planned for the SAGA system.

4.1.3.2 Code-producing routines.

The pcc-code producing routines walk the parse tree to emit C code. Because they must walk the tree, they are very dependent on the structure of the Pascal grammar; for instance, the structure expected in a subtree is determined by checking its production number. This tight coupling is slightly alleviated by the usage of symbollic names (Pascal constants) for the rule-numbers in the grammar; however, there is no way to eliminate the dependence on the internal structure of the productions.

The code-producing routines mirror the Algol-family structure of Pascal and C. The top-level routine generates code for a procedure, function, or program; it handles program unit prologues and epilogues, and the emitting of symbol table directives which provide information to pc3 and the Unix debugger. It invokes other routines to deal with the executable statements in the program unit's body.

For each Pascal statement, there is a procedure to traverse its subtree and emit code; these emit the flow-of-control operators. At the bottom level are the routines to generate code for l-values (locations) and r-values (expressions); these emit the pcc-code expression trees.

4.1.3.3 Machine-dependent aspects.

The third class of routines in the code generator are those which implement machine-dependent aspects of code generation. An example is the alignment module, which is used by by the symbol table component to allocate offsets for variables; another is the temporaries module, which handles the allocation of temporary variables for the current block (placing them in registers when possible).

4.1.3.4 Runtime system routines.

The fourth group of routines are those which support the use of the Berkeley Pascal run-time system. A good example of this group is the sets module. Routines from this module have diverse duties relating to the Pascal set type, such as determining whether a set expression is a constant set, determining the type of a constant set, or emitting the proper Pascal-system function call to perform the indicated set operation.

4.2 Incremental Recompilation

The code generator component of pcg is controlled by the incremental recompilation driver. The driver for the pcg incremental recompiler is straightforward. When pcg is invoked to compile a SAGA file, the driver first checks that a symbol table file exists within the SAGA directory which implements the SAGA file; if no symbol table exists, the standalone symbol table component is invoked to generate one. Next, the incremental recompiler checks that a modifications-trace is available. If not, then it assumes that the entire file is to be recompiled. Alternately, the user may demand that the incremental recompiler ignore the modifications-trace, and recompile the entire file.

When a SAGA source file has been previously compiled with pcg, its SAGA directory will contain two additional file. One is the pcc-code which resulted from the last compilation. The other is a list of the routines present in that file; for each routine, the location of its last word of code, within the pcc-code file, is recorded.

The process of recompilation is a simultaneous post-order traversal of three tree of routines: the tree of routines represented by the parse tree, and the linearizations of that tree present in the two files described above. For each routine in the parse tree, if the modifications-trace indicates that the routine must be recompiled (or if the modifications-trace is unavailable), then the code-generator is invoked to generate new pcc-code from the routine's subtree and its context in the symbol table. The pointer into the file of old pcc-code is advanced past the routine. If the modifications-trace indicates that the routine need not be recompiled, then its pcc-code from the old compilation is copied verabatim into the new file, advancing the pointer.

Pcg then invokes the later phases of the Berkeley Pascal compiler, with the new pcc-code as input, to complete the compilation. If the -c (separate compilation) option was specified, then the last two phases of pc are not run, and the result of the compilation is an unlinked object, just as with pc. If the separate compilation option was not invoked, then an executable object is produced. In either case, the process produces three other files: a new pcc-code file, a new routine-locations file, and a modifications trace which now indicates that all routines are up-to-date.

Chapter 5

CONCLUSION

Pcg demonstrates that a compiler in a language-oriented environment can make use of the information gathered by other tools to improve the efficiency of compilation. The parse trees produced by the epos syntax-directed editor are sufficient for compilation; an interactive semantic evaluator, implemented with the symbol table manager, can build a symbol table to enable compilation; and the modifications-trace collected by SAGA Make can be used to eliminate redundant compilations. The final result shows the usefulness of tools which share information to avoid duplication of effort.

Pcg demonstrates the composition of tools in the SAGA environment. The SAGA tools pcg uses had not previously all been required to cooperate simultaneously. Occasionally a tool did not correctly implement its interface, or the interfaces of two tools clashed so that they could not communicate with each other without difficulty. Though such real-world difficulties occurred, the tools were composed to generate a complex application.

5.1 Statistics for Example Programs

Pcg moves the task of symbol table construction, along with the task of syntactic analysis, from the translation phase of the coding cycle into the editing phase. Further, it attempts to improve the efficiency of compilation by incrementally compiling within files. We consider figures on time and space costs collected for two sample programs.

	declarations.p (147 lines)	pxref.p (389 lines)
epos without semantic evaluation	11.0 user seconds 3.0 system seconds	43.8 user seconds 6.7 system seconds
epos with semantic evaluation	23.2 user seconds 4.8 system seconds	61.3 user seconds 9.6 system seconds

Table 1. Times required for the editor to read and analyze two files.

Table 1 shows the time required for epos to read in and analyze two files: the first consists entirely of complex declarations; the second, Wirth's cross-reference program, is a more realistic mix of declarations and code. In the first case, the symbol table component makes the editor run approximately twice as slow. These worst-case figures may be misleading. In actual interactive editing, the time cost of semantic evaluation is spread out over many interactions; subjectively, the response time of the editor does not deteriorate significantly when semantics evaluation is included.

	declarations.p	pxref.p
standard text	3682	7955
executable object	15360	27648
pcc-code	4308	38592
parse tree	69644	262156

Table 2. Size in bytes of four representations of two files.

Table 2 shows that, although the pcc-code representation can be significantly larger than the straight text representation of a given program, it is not expensive compared to the current SAGA parse tree representation. Thus, preserving pcc-code files between compilations is a reasonable course.

	declarations.p	pxref.p
pc0	0.8 user seconds	5.7 user seconds
	0.8 system seconds	1.1 system seconds
pc	4.0 user seconds	31.6 user seconds
	3.7 system seconds	6.8 system seconds
code generator		
pcg		

Table 3. Compilation times for two files, in which one 20-line procedure was modified.

Does pcg improve the efficiency of compilations? Table 3 shows that pcg performs code generation faster than pc0, but, unfortunately, the first phase consumes only about a fifth of the time of a compilation. The latter four phases of compilation are shared by pc and pcg; pcg's incremental recompilation efforts are aimed at efficiently producing an intermediate code version of a file, which must then be given to the non-incremental pc backend to complete compilation.

Certain implementation problems remain. As a prototype, pcg is insufficient for a true development environment. It will soon be extended to support full Pascal, but it must also support separate compilation if it is to be useful; this requires the resolution of the limitation in the symbol table manager previously mentioned.

5.2 Future Directions

Pcg suggests several directions for future work.

The most fundamental limitation of the pcg system is its dependence on a non-incremental machine-code generator. Any efficiency gained from incremental recompilation in the early phase is overshadowed by the time required to recompile the resulting code non-incrementally. A straightforward extension to pcg would be a facility for merging assembly-language rather than intermediate-code files; preserving assembly-language between compilations would make the first machine-dependent phase of compilation incremental. But recompilation should be incremental throughout all phases. A facility for merging binaries, such as existed in the original Cyber-based SAGA Make, or an incremental loader, such as in IPE, is required. Once such a facility is provided, pcg-style code generators can be developed for a variety of languages, and use the common backend.

If one is willing to sacrifice language-independence, then SAGA Make can be made more efficient, by using the symbol table manager's ability to record symbol references. Nested routines which do not reference a modified declaration in the outer environment need not be recompiled in response to that modification. Further, when the ability to use multiple symbol tables is realized, it will be possible to record inter-file dependencies on the procedural level; this would make it possible, for instance, to avoid recompiling a file which references an unchanged interface even though the interface resides in a file where other interfaces were modified.

Pcg only deals with Pascal. The SAGA environment is meant to support several programming languages [Campbell and Kirslis]; SAGA editors exist for Pascal, C, Ada, and Backus' FP. Since pcc-code is also used to implement FORTRAN and C, the development of pcg-type compilers for these languages would be straightforward. As we have seen, the use of standard compilers which expect text input is inappropriate for an environment such as SAGA. But the hand-coded production of pcg-style code generators could be prohibitively costly in human time, for an environment which supports many languages. The addition of the attribute-grammar based semantic evaluator to SAGA will make the production of symbol table components far less ad-hoc. Since the symbol table component is a major part of a code generating system, producing such systems will become much less costly. The attribute-grammar specification for one language, which details the attributes needed to generate a given intermediate code from that language, could serve as the basis for developing specifications for other languages which will use that same intermediate code. Also promising is research on the automatic generation of compilers from attribute-grammar specifications of a language and an architecture [Ganapathi and Fischer], [Paulson]. It may be that such a formal specification can be used to generate an entire language-based environment, including editor, compiler, and debugger.

Pcg incrementally recompiles on the procedural level. Just as in IPE, a natural development would be the integration of a source-level debugger into the editor/recompiler system, giving the ability to immediately run the actual machine code routines on sample input. This would enable rapid interleaving of program creation with program testing, as is possible in an interpreter-based environment. But by incrementally recompiling rather than interpreting, the true machine-code implementation would be the object of debugging, and the faster execution characteristic of non-interpreted code would be available.

Appendix A

PCG PASCAL, STANDARD PASCAL, AND BERKELEY PASCAL

A.1 Compliance with ANSI/IEEE 770 X3.97-1983 Standard Pascal

The SAGA pcg system complies with the requirements of ANSI/IEEE 770 X3.97-1983 with the following exceptions:

6.1.1. The case of letters making up identifiers and reserved words is significant. This follows the Unix convention.

6.1.3. Identifiers cannot be longer than 127 characters in length.

6.1.4. The directive #include may occur outside procedure-declarations and function-declarations.

6.1.5. Integers occupy the range minint..maxint, where minint = -2147483648, and maxint = 2147483647.

6.1.6. Labels may be longer than four digits in length; a warning is issued if such a label is declared.

6.1.8. If a comment begins with one type of delimiter and ends with another, a warning is issued. Nested comments are allcwed.

6.2.2.10. The required identifiers 'write' and 'writeln' have special significance within the grammar, and should not be redeclared.

6.4.3.1. (The keyword packed has no effect.)

6.4.3.2. To be a string, an array of characters need not be packed, and the lower limit of its subscript need not be 1.

6.4.3.5. The predefined type 'text' is equivalent to 'file of char'.

6.8.3.5. The case statement is currently not implemented.

6.8.3.9. The for statement is currently not implemented.

A.2 Differences between Pcg Pascal and Berkeley Pascal

This section constitutes an addendum to Appendix A of the Berkeley Pascal User's Manual. See that manual for a full description of Berkeley Pascal.

A.1. Extensions to the language Pascal.

String Padding. Pcg Pascal pads constant strings with blanks as necessary, just as Berkeley Pascal does.

Octal constants, octal and hexadecimal write. Pcg does not support these Berkeley extensions.

Assert statement. The assert statement is not supported.

Enumerated type input-output. Pcg Pascal performs the nonstandard extension of enumerated type input-output just as does Berkeley Pascal.

Structure returning functions. Pcg Pascal allow functions to return records, sets, and arrays, just as Berkeley Pascal does.

A.1. Resolution of the undefined specifications.

File name - file variable associations. Pcg Pascal associates Pascal file variables with named Unix files following the Berkeley conventions.

The files input and output. These are handled as in Berkeley Pascal.

Buffering. The buffering of 'output' is controlled by the b option, just as with Berkeley.

The character set. Just as in Berkeley, upper and lower case are distinct, and all keywords and required identifiers are expected to be all lower case. Use of ~, &, |, and # as synonyms for not, and, or, and ', are not supported.

Comments. Comments that start with one style of delimiter and end with another cause a warning message, as in Berkeley.

Option control. Options may be set in the pcg command line, in the standard Unix convention. Pcg Pascal does not support the control of options via flags in comments. See Appendix B for the options available.

Listings. No listings are produced. When errors are detected, their locations are indicted by setting a flag in the token causing the error; the token is thereby highlighted in epos's screen mode.

A.3. Restrictions and limitations.

Statements. Pcg Pascal does not currently support the following statements: goto, case, and for.

Files. The restriction that files cannot contain files is now part of the standard. As in Berkeley Pascal, files are also restricted from being members of dynamically-allocated structures.

Arrays, sets, and strings. The Berkeley restriction applies: arrays--including strings--and sets may have no more than 655355 elements; array and string subscripts are limited to the range -32768..32767.

Line and symbol length. Symbols are limited to 127 characters in length.

Procedure and function nesting and program size. The arbitrary restriction of a maximum nesting depth of 20 is maintained in pcg. There is an unknown maximum program size; it is comfortable large.

Overflow. As Berkeley notes, the Vax does overflow checking in hardware.

A.4. Added types, operators, procedures, and functions

Additional predefined types. Alfa is predefined (and may be redeclared, of course). Intset is predefined to be set of 0..127.

Additional predefined operators. '<' and '>' may be used on sets to test for proper set inclusion, as in Berkeley Pascal.

Non-standard procedures. The following Berkeley non-standard procedures are supported by pcg: argv, flush, halt, remove, and the extended two-argument reset and rewrite. These are not supported: date, linelimit, message, null, stlimit, and time.

Non-standard functions. The following Berkeley non-standard functions are supported: argc, card, and expo. These are not supported: clock, random, seed, sysclock, and wallclock.

Appendix B

MANUAL PAGE FOR PCG

NAME

pcg - Pascal code generator

SYNOPSIS

pcg [option] filename...

DESCRIPTION

Pcg functions as a Pascal compiler in the SAGA Pascal environment. If given an argument SAGA file ending with .p, it will compile the file and load it into an executable file, called, by default, a.out.

Pcg currently does not support the following Pascal statements: case, goto, for.

Pcg compiles directly from the parse tree representation of the source file used by epos. Pcg expects the SAGA file (directory) to include a symbol table, generated by the epos-resident symbol table component of pcg; but in the absence of a symbol table, pcg will generate one. If the file was compiled previously with pcg, then a subsequent recompilation will reuse unchanged procedures from the previous compilation, for efficiency's sake.

Currently, pcg does not support separate compilation. When such support becomes available, it will be modeled on the example of Berkeley pc; see pc(1).

Pcg does not support profiling with pxp(1).

The following options have the same meaning as in pc (1), cc(1), and f77(1). See ld(1) for link-edit time options.

-c	Suppress link-editing and produce '.o' file(s) from source file(s).

-g Generate additional symbol table information for sdb (which is obsolete).

-w Suppress warning messages.

-p Prepare object files for profiling; see prof(1).

-O Invoke an object-code optimizer.

-S Generate assembler code only; do not generate '.o' files.

-o name
  Name the final output file 'name' instead of 'a.out'.

The following options are the same as in pc(1).

-C Compile code to perform runtime checks, and initialize all variables to 0.

-b Block buffer the file output.

The following options are peculiar to pcg.

-F Force the generation of new intermediate code, ignoring code maintained from previous compilations.

-d Generate debugging output.

FILES

file.p	Pascal source files
~saga/bin/epospcg	editor with resident symbol table component
~saga/bin/pcg	incremental recompilation driver
~saga/bin/pcgcodegen	portable C compiler intermediate code generator
/lib/f1	assembler generator
/usr/lib/pc2	inline expander
/usr/lib/pc3	separate compilation consistency checker
/lib/c2	peephole optimizer
/usr/lib/libpc.a	intrinsic functions and I/O library
/usr/lib/libm.a	math library
/lib/libc.a	standard library, see intro(3)
~saga/src/pcg/semantic	semantic phase sources
~saga/src/pcg/codegen	code generator sources
~saga/src/pcg/increm	incremental recompilation driver sources

SEE ALSO

"Pcg: A Prototype Incremental Compilation Facility for the SAGA Environment".
Berkeley Pascal User's Manual.

AUTHOR

John Kimball

BUGS

Pcg is a prototype system, and bug reports should be sent to the author.

BIBLIOGRAPHY

[Aho and Ullman] Aho, Alfred V. and Jeffrey D. Ullman. Principles of Compiler Design. Addison-Wesley, Reading, MA (1977).

[ANSI] Joint ANSI/X3J9-IEEE Pascal Standards Committee. An American National Standard: IEEE Standard Pascal Computer Programming Language. New York: Institute of Electrical and Electronics Engineers, Inc. (1983).

[Badger] Badger, Wayne H. "Make: A Separate Compilation Facility for the SAGA Environment," Master's Thesis, University of Illinois at Urbana-Champaign (1984).

[Beshers and Campbell] Beshers, George M, and Roy H. Campbell. "Maintained and Constructor Attributes." ACM SIGPLAN Symposium on Language Issues in Programming Environments (June 1985).

[Campbell and Kirslis] Campbell, Roy H. and Peter A. Kirslis. "The SAGA Project, a System for Software Development," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA (April 1984).

[Cooper] Cooper, Doug. Standard Pascal User Reference Manual. W. W. Norton and Co., New York, NY (1983).

[Donzeau-Gouge, Huet, Kahn, and Lang] Donzeau-Gogue, Veronique, Gerard Huet, Gilles Kahn, and Bernard Lang. "Programming Environments Based on Structured Editors: The MENTOR Experience," INRIA Research Report No. 26, Rocquencourt, France (May 1980).

[Donzeau-Gouge, Lang, and Melese] Donzeau-Gogue, V., B. Lang, and B. Melese. "Practical Applications of a Syntax-Directed Program Manipulation Environment," Proceedings of the Seventh International Conference on Software Engineering, IEEE; Orlando, FA (March 1984).

[Estublier, Ghoul, and Krakowiak] Estublier, J., S. Ghoul, and S. Krakowiak. "Preliminary Experience with a Configuration Control System for Modular Programs," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburg, PA (April 1984).

[Feldman] Feldman, S. I. "Make--A Program for Maintaining Computer Programs," Unix Programmer's Manual, Seventh Edition, volume 2; Bell Laboratories, Murray Hill, NJ (1980).

[Ganapathi and Fischer] Ganapathi, Mahadevan, and Charles N. Fischer. "Description-driven Code Generation using Attribute Grammars," Conference Record of the Ninth ACM Symposium on the Principles of Programming Languages, Albuquerque, NM (January 1982).

[Ghezzi and Mandrioli] Ghezzi, C. and D. Mandrioli, "Augmenting Parsers to Support Incrementality," Journal of the ACM, Vol. 27, No. 3 (July 1980).

[Goldberg] Goldberg, A. Smalltalk-80: The Interactive Programming Environment, Addison-Wesley, Reading, MA (1984).

[Habermann] Habermann, A. N. "An Overview of the Gandalf Project," CMU Computer Science Research Reviews (1980).

[Johnson1] Johnson, S. C. "A Tour through the Portable C Compiler," Unix Programmer's Manual, Seventh Edition, volume 2; Bell Laboratories, Murray Hill, NJ (1980).

[Johnson2] Johnson, S. C. "YACC: Yet Another Compiler-Compiler," Unix Programmer's Manual, Seventh Edition, volume 2; Bell Laboratories, Murray Hill, NJ (1980).

[Johnson and Fischer] Johnson, G. F., and C. N. Fischer. "Non-syntactic Attribute Flow in Language-based Editors," Conference Record of the Ninth ACM Symposium on the Principles of Programming Languages, Albuquerque, NM (January 1982).

[Joy, Graham, and Haley] Joy, William N., Susan L. Graham, and Charles B. Haley. "Berkeley Pascal User's Manual Version 3.0," Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley (July 1983).

[Joy and McKusick] Joy, William N., and M. Kirk McKusick. "Berkeley Pascal PX Implementation Notes Version 2.0," Technical Report, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley (January 1979).

[Kessler] Kessler, Peter B. "The Intermediate Representation of the Portable C Compiler, as used by the Berkeley Pascal Compiler," Technical Report, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California at Berkeley (April 1983).

[Kirslis, Terwilliger, and Campbell] Kirslis, Peter A., Robert B. Terwilliger, and Roy H. Campbell. "The SAGA Approach to Large Program Development in an Integrated Modular Environment," Proceedings of the GTE Software Engineering Environments for Programming-in-the-Large Workshop, Harwichport, MA (June 1985).

[Medina-Mora and Feiler] Medina-Mora, Raul, and Peter H. Feiler. "An Incremental Programming Environment," IEEE Transactions on Software Engineering, volume SE-7, number 5 (September 1981).

[Noonan and Collins] Noonin, R. E., and W. R. Collins. "The Mystro Parser Generator, PARGEN User's Manual, Version 6.3," College of William and Mary, Williamsburg, VA (1983).

[Paulson] Paulson, Lawrence. "A Semantics-directed Compiler Generator," Conference Record of the Ninth ACM Symposium on the Principles of Programming Languages, Albuquerque, NM (January 1982).

[Reiss] Reiss, Steven P. "PECAN: Program Development Systems that Support Multiple Views," IEEE Transactions on Software Engineering, volume SE-11, number 3 (March 1985).

[Reps] Reps, Thomas W. Generating Language-Based Environments. MIT Press, Cambridge, MA (1984).

[Richards] Richards, Paul G. "A Prototype Symbol Table Manager for the SAGA Environment," Master's Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (1984).

[Teitelbaum and Reps] Teitelbaum, Tim, and Thomas Reps. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Communications of the ACM, volume 24, number 9 (September 1981).

[Teitelman] Teitelman, Warren. "A Tour Through Cedar," Proceedings of the Seventh International Conference on Software Engineering, IEEE; Orlando, FA (March 1984).

[Teitelman and Masinter] Teitelman, Warren, and Larry Masinter. "The Interlisp Programming Environment," IEEE Computer, volume 14, number 4 (April 1981).