INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
RICE UNIVERSITY

A Single Intermediate Language
For Programming Environments

by

Michael Alan Caplinger

A Thesis Submitted
In Partial Fulfillment Of The
Requirements For The Degree

Doctor Of Philosophy

Approved, Thesis Committee:

Robert S. Cartwright, Director,
Professor of Computer Science

C. Sidney Burrus,
Professor of Electrical and Computer Engineering

Robert T. Hood,
Assistant Professor of Computer Science

Kenneth W. Kennedy,
Professor of Computer Science

Houston, Texas

May, 1985
A Single Intermediate Language for Programming Environments

Michael Alan Caplinger

Abstract

Programming by the traditional methods of text editing and batch compilation is no longer necessary. The advent of the personal workstation and bitmap graphics allows us to construct integrated programming environments, which actively aid the programmer in viewing, creating, and debugging programs. But despite many attempts to build such environments for the widely used Algol family of languages, none is in production use today.

Our system, Yggdrasil, is an attempt to answer the question of how to build an effective IPE for Algol-like languages. It consists of a language-independent framework for editing and executing programs, and a language-dependent part driven by specifications. An Yggdrasil environment for a particular language is built by writing a specification for that language. The specification must define three fairly separate components of the language: its abstract syntax, its static semantics, and its execution semantics. The first two are used to enforce the static validity of the program as it is edited, while the third allows the system to execute the program via interpretation.

The central part of the specification is a description of the programming language's abstract syntax. Unlike other systems, which may use several different program representations, only a single tree form is maintained. This insures that the environment will not have distinct "modes," since there is no need to translate...
between representations; it also lets different parts of the system share common functions.

Other specifications take the form of static annotations for each constructor in the abstract syntax. Such specifications control the user's view of the program, some checking of its validity, and the treatment of program names.

Some aspects of the programming language cannot be expressed statically. The system also allows constructors to be annotated with functions; each function is executed in response to a message sent to a particular node in the program tree, as in an object-oriented system.

The functions are used to complete the static checking of the program during editing, and to execute it. One of the function annotations specifies the execution semantics of a constructor, in terms of a composition of primitive operators provided by the system and the semantic functions of the constructor's sons. Execution is performed by repeatedly calling the semantic functions.

The combination of the editing and execution framework and the semantic primitives allows the simple specification of a powerful programming environment for Algol-like languages. We pay special attention to making the user interface for program composition, editing, and debugging execution convenient, while avoiding the problems of earlier IPE systems.
Acknowledgements

Writing this thesis has been a tedious, frustrating, yet occasionally enjoyable experience. The fun parts were due in large part to my friends in the LCSE at Rice, some of whom I would like to thank here.

My officemates David Chase and Scott Comer gave me four years of strange but enlightening companionship, as well as a few great technical arguments and some excellent music. Bill LeFebvre answered what must have been several thousand stupid questions about troff; most of what I know about text processing is due to him. Keith Cooper and Linda Torczon were islands of sanity in an often crazed world; I will miss their calming effect. Bob Hood has been a good friend and source of advice throughout my graduate career; Yggdrasil's name is his fault.

I would also like to thank the other members of my committee: Ken Kennedy, for welcoming me back to Rice and giving me innumerable toys to play with; Corky Cartwright, for his belief in notions even more radical than mine; and Sid Burrus, who always found time to help me navigate through a maze of AS&R paperwork.

I am grateful to the National Science Foundation and IBM for allowing me to maintain a standard of living far above the grad student norm (at least potentially).

Finally, much love to my parents, my brother, and Elaine Hill, for helping me to keep a sense of perspective about it all.

For those interested in matters typographical: most of this thesis was composed on an Apple Macintosh with MacWrite, then translated to ditroff, edited with Gosling's Emacs, and printed on an Imagen 8/300. My thanks also go to the authors of at least some of the software involved.
Table Of Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>Introduction and Previous Work</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>The Abstract Syntax Tree</td>
<td>23</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>Syntax and the User Interface</td>
<td>38</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Names and Scopes</td>
<td>49</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Unparsing</td>
<td>67</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Static Checking</td>
<td>84</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Execution Semantics</td>
<td>95</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Runtime Storage</td>
<td>128</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>The Runtime Environment</td>
<td>142</td>
</tr>
<tr>
<td>Chapter 10</td>
<td>Textual Considerations</td>
<td>163</td>
</tr>
<tr>
<td>Chapter 11</td>
<td>Conclusions</td>
<td>175</td>
</tr>
<tr>
<td>Appendix A</td>
<td>A Sample (Y) Specification</td>
<td>183</td>
</tr>
<tr>
<td>Bibliography</td>
<td></td>
<td>193</td>
</tr>
</tbody>
</table>
## Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>A programming environment design space</td>
<td>7</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Two tables of node usage</td>
<td>34</td>
</tr>
<tr>
<td>Figure 3</td>
<td>A cascaded menu tree</td>
<td>47</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Name lookup algorithms</td>
<td>61</td>
</tr>
<tr>
<td>Figure 5</td>
<td>A $Yg$ name display</td>
<td>65</td>
</tr>
<tr>
<td>Figure 6</td>
<td>The incremental unparser algorithm</td>
<td>72</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Samples of holophasm</td>
<td>74</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Holophasm in the incremental unparser</td>
<td>79</td>
</tr>
<tr>
<td>Figure 9</td>
<td>The top-level interpreter</td>
<td>113</td>
</tr>
<tr>
<td>Figure 10</td>
<td>The doGoto function</td>
<td>115</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction and Previous Work

You taught me language, and my profit on't is, I know how to curse.

_The Tempest_, I, ii

Every program ever written was developed in some kind of a programming environment. The earliest "environments" were based on punched cards or paper tape, and a simple operating system that allowed a single user to access the entire machine during a brief time slot. As machines became more powerful and more expensive, batch operating systems became important. These allowed many programs to be run without user intervention, in rapid succession. This had the advantage of making the best use of limited machine resources, but programmers lost the ability to examine their programs during execution. Fortunately, the steady growth of computer power made time-sharing and reasonably interactive program development possible, but much of the batch flavor of programming remained.

The advent of powerful, inexpensive microprocessors has opened up new possibilities for programming environments. A new generation of programming tools can be used now that we have come full circle to the point where each user again has the entire machine to himself. The easy availability of high-resolution graphics and fast distributed file access motivates the construction of _integrated programming environments_, which would perform all of the tasks required in programming, from editing and source control to debugging. In more conventional environments, each separate task is performed by an independent tool, which is often not able to
communicate with the other tools in the system. The term “integrated” implies that in such an environment the tools work so well in concert that the user may not even be aware they are separate, if in fact they are.

Unfortunately, despite many projects that designed and built IPEs, none has ever received widespread acceptance and use.\textsuperscript{1} There are several possible reasons for this:

(1) Inertia. By their nature, integrated tools often have to be implemented from scratch. This means that existing separate tools must be discarded or heavily modified, and users must adapt to the new system. Programmers are often resistant to such environmental change.

(2) No production-quality systems have been built. Most of the IPEs constructed to date have been research prototypes, with all the problems in robustness and usability that implies. Few have run on common machines, under common operating systems, nor have many been given general distribution.

(3) No IPE has demonstrated clear superiority over existing tools. Most efforts have concentrated on particular aspects of programming, to the exclusion of others. For example, the CMU Gandalf system [Hab82] is primarily a structured editor, with minimal support for execution and program management.

We believe that the last point is the real reason why IPEs are not commonly used. A system that obviously allowed easier program development could certainly overcome inertia and production problems. Therefore, we must try to determine what features can be provided only by an IPE, and how these features can be integrated in

\textsuperscript{1}We are speaking of environments for languages in the Algol family. Various Lisp environments have been used for production programming for some time—and we will discuss the reasons for that shortly.
a coherent manner. To do so, we must examine the nature of existing programming languages.

1.1. Current languages

It is ironic to consider that almost all of today's programming languages grew out of the two branches of language design present at the end of the 1950s. These two families are, of course, Fortran and Lisp. Their characteristics were largely determined by the goals of their designers.

1.1.1. Fortran

The goal of Fortran was to ease program development for applications programmers, specifically in the field of scientific and mathematical computing. At the time, most programming was still being done in assembly language, with its associated problems of complexity, non-portability, and maintainability. Fortran was to provide a largely machine-independent way to write programs that were competitive in efficiency with the best assembly language. This efficiency was the driving force of the project.

Because of the batch facilities of the time, an interactive compiler was essentially impossible. Instead, the program was represented as text, punched on cards, and fed to the compiler. The results were either an object module, or a series of error messages describing the problems the compiler encountered with the input text. Once a successful compilation was completed, the object code could be executed, and any problems found there resulted in another round of editing of the program cards. Thus was born the familiar “edit-compile-run” cycle still very much in evidence today.
The critical feature of this environment is the essential batch nature it imposes on program development. Programs are edited in text form by a general text-editing process—initially cards and keypunching, more recently text editor programs. The compiler is consulted in an off-line manner, and any static errors corrected, often over several passes. Once the program is ready to run, it has been transformed away from the program text into a quite different object representation, making mapping between the executing program and the program as seen by the user difficult. Separate compilation, required when programs become large, makes it necessary to provide a way for each program part to tell the compiler about the other parts it may reference. Each step in the development process is clearly delineated, whether or not the programmer's thought processes are similarly broken up.

These steps have changed little with the development of new languages. Environments for languages like Algol, Pascal, C, Modula, and Ada are still designed around the notions of text editing, batch compilation for separate files, and object code debugging with mapping back to the text program. While the features provided by the languages have changed radically (data structures, structured control flow, and concurrency are examples of such changes) the nature of the development process has hardly changed at all.

1.1.2. Lisp

The other branch of language design in the 1950s was the development of Lisp. Here the goal was much less clear than that of the Fortran project. Lisp began as a
fairly conventional language in the spirit of "Fortran with lists." It was only realized later that a Lisp interpreter could be written in Lisp by making Lisp programs be legitimate Lisp data objects.

The almost accidental character of Lisp pointed to a mode of program development radically different from that of Fortran. Because programs were directly interpreted from their representation as entered by the programmer, there was no real notion of compilation. For the same reason, errors in syntax were discovered immediately upon entry. The program was never stored in textual form once it was in the system, and could be entered right from a keyboard. Since execution used the same program representation as that seen by the user, the mapping problem disappeared, and debugging was enormously eased. Lisp seems tailor-made for interactive use, even though it was first developed when hardware to support such use was only just becoming available.

However, interpreted Lisp is not without its problems, and its major one is inefficiency. Because it uses no object code form for the program, it cannot be directly executed by standard hardware. Also, its typeless, dynamic nature makes the detection of many errors impossible until execution. In a dynamically-scoped Lisp, even a variable reference cannot be checked for validity until run-time. Attempts to increase the efficiency of Lisp programs have been surprisingly "Fortranesque"—the use of batch compilers that view the program as lexically scoped and accept variable declarations are typical. Even so, the best IPES available today are outgrowths of the interactive Lisp tradition.
1.2. Previous work

Before attempting to state the desirable features of an integrated programming environment, we will examine a few previous attempts to build one. We begin by classifying each one by the point it occupies in a design space, depending on its approach to various problems. Our design space has four dimensions. These are:

- **Retargetable**
  
The system has been designed as a specification tool such that different languages (possibly restricted to a particular class) can be supported without major reorganization of the system. The opposite of this is "hand-crafted" to support only a single language.

- **Structured**
  
A structure-based editing model is used, as opposed to a text-based model.

- **Compiled**
  
Execution is based on a compiled representation rather than an interpreted one.

- **Language-enhancing**
  
The system supports language constructs over and above those found in the standard definition of a language, instead of supporting only the language standard.

Obviously, we cannot assign numeric values for each of these dimensions for each system. Instead, we place each system in one of three bins along each axis. The reasons for selecting a particular bin will be explained in greater detail for each system. Figure 1 shows the positions of several systems in design space.
<table>
<thead>
<tr>
<th>system</th>
<th>retargetable</th>
<th>structured</th>
<th>compiled</th>
<th>enhancing</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPS</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>partially</td>
</tr>
<tr>
<td>Cedar</td>
<td>partially</td>
<td>no</td>
<td>yes</td>
<td>partially</td>
</tr>
<tr>
<td>Interlisp</td>
<td>no</td>
<td>partially</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Smalltalk</td>
<td>no</td>
<td>partially</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>PECAN</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Magpie</td>
<td>no</td>
<td>partially</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>CSG</td>
<td>yes</td>
<td>yes</td>
<td>--</td>
<td>no</td>
</tr>
<tr>
<td>Gandalf</td>
<td>yes</td>
<td>yes</td>
<td>--</td>
<td>no</td>
</tr>
</tbody>
</table>

Figure 1: A programming environment design space

1.2.1. Cornell Program Synthesizer

The Cornell Program Synthesizer [TeR81] was the precursor to most of the later structured environments. It was initially designed to support student programming in the PL/C dialect of PL/I. In such a learning atmosphere, structured editing via template replacement was viewed as valuable because it freed the programmer from small concerns of syntax.

The CPS was implemented on a small personal minicomputer, the Terak. It used template expansion and tree manipulation for editing at the statement level, switching to text entry for expressions. Motion commands were structured as well, and linked to special function keys. (More sophisticated pointing devices were unavailable.) The convenience of the environment was greatly enhanced by the ability to execute the program. An intermediate code produced from the tree was interpreted, and the state of the program was visually presented. Execution speed could be stepped, varied in speed, and even reversed.
Another important feature of the CPS was level-hiding. This allowed blocks of code to be hidden, displayed on the screen only by an introductory comment. The execution display was integrated with this, so that the debugging of one block could be concentrated on while the others were hidden.

Despite its great success as a teaching tool, the CPS was never intended for use in a production environment. The maximum program size supported was on the order of 100 lines, and there were no facilities to aid the development of larger programs. Because of the severe memory limitations of the Terak it was completely hand-coded.

1.2.2. Gandalf

The Gandalf project at CMU is an attempt to build a powerful programming environment supporting program management and module interconnection as well as structured editing and debugging [HaN82]. The system uses structured editing to provide a common user interface. The structured editor, called an ALOE, is automatically generated from specifications [MeN81].

An ALOE is composed of two parts: a standard framework to perform editing, tree management, and unparsing, and a series of user-written action routines. The framework provides a basic user interface based on structured motion and tree manipulation, and driven by static specifications for the language to be edited. The action routines are called by the framework when the user performs modifications, certain tree motions, or "extended commands." The action routines have been used to do non-syntactic checking, add additional information (such as symbol tables) to the tree, and generate code. They are extremely general; in the authors' words, "Indeed,

\footnote{A Language-Oriented Editor.}
the limits of what can be done by action routine calls have not been explored completely, but certainly do promise a wide variety of possibilities” [MeN81, page 39].

Unfortunately, because of its very generality it may be quite difficult to use ALOE to build an editing environment with more capability than the simple syntax-directed editing provided by the basic framework. For example, adding an interpreter to the editor would require the complete definition of the interpreter, in C, to be added as an extended command. An ALOE-based prototype debugging system used an existing compiler and ran the program in a separate address space [Fei82]. While this approach builds on existing tools, the tools are often unable to intercommunicate adequately.

An attempt to build a “meta-system” to build ALOE specifications based on a particular discipline is described in [AKE84]. While it addresses a number of problems related to non-syntactic checking, it fails to provide a more convenient way to specify execution.

1.2.3. Cedar

The Xerox Cedar Mesa project [Tei84] is a good example of the “integrated toolbox” approach to programming environments. Rather than build a single system to provide all the functions of the environment, it uses existing compilers and debuggers which can produce links back into the text as manipulated by the system’s editor, which is called Tioga. Tioga is a text editor that provides a structuring layer on top of the text; a file can be viewed as a collection of nodes in a hierarchical structure, with text associated with each node. When exported to a particular tool such as a compiler, Tioga produces only the text associated with language nodes,
leaving other nodes’ text out. This allows the programmer to mix documentation with code in a way similar to Knuth’s WEB system [Knu83]. Tioga thus serves as a partial user interface for a number of non-interactive tools.

This approach capitalizes on the presence of existing tools, but is only as strong as the connections between the tools, which may have to be modified in order to communicate with Tioga. It also fails to hide the distribution of work among the tools adequately; the system is still quite modal.

1.2.4. Interlisp

The Interlisp-D environment is probably the most successful and widely-used integrated environment in use today [Tei78]. It is a very large and interrelated collection of Lisp functions in a single workspace, providing text and structured editing, program management, static program analysis, debugging, documenting, language extension, and correction.

We call Interlisp a partially-structured environment because it does provide a structured Lisp editor. However, like most of the rest of the Lisp community [Woo81, Sym84] a form of augmented text editing is more often used. Because of Lisp’s simple syntax, a text editor can easily provide facilities like prettyprinting and parenthesis balancing, which correspond to the more complicated syntax checking required by other languages. Text entry is also eased in Interlisp by the presence of a powerful input correcting system called DWIM.\textsuperscript{3}

DWIM is also used to allow the user to use Algol-like syntax and to define new syntax. Such syntax is considered an error by most of the system, which calls DWIM

\textsuperscript{3}Do What I Mean.
to correct it. DWIM then recognizes the new syntax and generates normal Lisp from it. The system's unparsert can also do the reverse translation.

Interlisp is the ultimate expression of the Lisp philosophy of programming. Functions can be executed interpretively or compiled, and each form can call the other. Extensive run-time monitoring and debugging is provided. The boundary between editing and execution is nearly invisible. While it serves as a useful measure of what is possible for an IPE, much of the Interlisp system would be impossible to implement in an environment that did not use a single language for programming and implementation, all executing in the same workspace. Lisp is far more amenable to such use than any member of the Algol family.

1.2.5. Smalltalk

Smalltalk-80 [GoR83] is a self-contained programming environment built around an object-oriented language paradigm. The environment is entirely written in Smalltalk [Gol84]. The system uses a compiler to translate to an intermediate bytecode form, which is then executed by a virtual machine. This approach has made possible a number of ports to standard hardware, which unfortunately cannot compete in speed with the microcoded Xerox versions [Kra83], at least without translations like those described by Deutsch in [Deu84].

The uniformity of the language paradigm allows the system extreme flexibility. As in Lisp, Smalltalk programs are legitimate Smalltalk data objects. The system supports text editing with recompilation on function boundaries. It also arranges functions into a hierarchical structure which is viewed by a browser; the browser can also display information about the relations between functions. For example, a single
command can display all the places that a particular function is used.

1.2.6. PECAN

PECAN is the result of a project at Brown University to build a production IPE with structured editing, debugging execution, and graphical display [Rei84b]. PECAN is specification-based, but unlike other environments, the specification builds not only an editing environment (which allows both structured and text-oriented editing), but an incremental compiler as well.

Of the systems we have examined, PECAN is closest in spirit to the approach used in this thesis. We will make extensive comparisons between our work and PECAN later.

1.2.7. Magpie

Magpie, a Tektronix project, builds an IPE based on incremental parsing instead of structured editing [SDB84]. The program is entered textually, and purely syntactic checking is done on a token-by-token basis. Static checking is performed less frequently, after each line of input.

Magpie also supports the execution and debugging of programs. Rather than interpret the program, it translates it into throwaway machine code. The code is produced incrementally on procedure boundaries, making use of the user's "think-time" and dual-processor hardware to hide the recompilation. The system is hand-crafted rather than specification-based, and uses fairly traditional compiler technology. It is interesting for its view of program debugging actions, which we will discuss in more detail in Chapter 9.
1.2.8. Cornell Synthesizer Generator

The success of the Cornell Program Synthesizer has led to a project to build a retargetable environment-building tool, the Cornell Synthesizer Generator [ReT84]. The CSG uses attribute grammars [Knu68] to build language-based editors. The program is represented by an attributed tree, and the system uses an optimal reevaluation strategy to incrementally update the attributes in response to certain editing operations. The technique is based on the use of a specific set of structured motion commands, which are the only way a user can change viewpoint in the tree.

The attributes in an AG can be used to denote symbol tables. In a compiler framework, pure syntax is handled by the context-free grammar, while "semantic" issues like variable declaration and scoping is done with attributes. For example, at a subtree root an input symbol table is provided as an inherited attribute, and that table is updated by the attribute functions within the tree, appearing as a changed table in a synthesized attribute. Another possible attribute in a compiler is generated code, which is elaborated in each subtree. The entire program's code would appear as a synthesized attribute at the root of the tree.

Reps' work demonstrates how attributes can be updated efficiently after an editing change, where changes are limited to subtree deletion and replacement and elaboration of nonterminals. As an extension to his work, one can envision a programming environment using incremental compilation, where compiled code would be an attribute or set of attributes, and changes could be made with no extraneous processing.
However, this approach provides minimal support for the implementor. It can update attributes efficiently, but the real work of the system is being done by the attributes and their functions, which still must be defined. Because the design of the right set of attributes could be a substantial effort, attribute grammars by themselves do not solve the problems of a multilanguage system. Also, the structured motion commands are more restrictive and inconvenient than we feel is desirable.

Theoretically, attribute grammars could be used to evaluate the program represented by the tree, making it a true semantic object, but the means to do so is still not clear. It seems to us that because attribute evaluation is a much different and more abstract model of computation than conventional automata, any attempt to use AGs to evaluate the tree directly will, at best, be very inefficient, and will probably involve changes to the AG formalism.

1.3. Implications of language design

Our survey of previous systems indicates that the highly interactive, late binding language style pioneered by Lisp seems to need little help in the area of IPE design. The situation is much less favorable for the descendents of the Fortran style, which we will call the "Algol-like languages." There, the batch nature of typical compilers has intruded into environment design; the boundary between editing and compilation is often still visible.

The great advantage conferred by the Lisp style is the fact that there is only a single representation for a program. The disadvantage is that much of the program
can only be understood dynamically; it is difficult to statically check many parts of a
program for correctness, as can be done in an Algol-like language.

We propose to exploit the best parts of each philosophy to build an IPE for
Algol-like languages. This implies that our environment has two aspects; it should be
a powerful tool for constructing, examining, and modifying the static program, and it
should combine this ability with direct execution of the same program representation.
By doing this, the distinction between editing, compiling, and executing the program
will disappear.

1.4. Yggdrasil

We call our system Yggdrasil\(^6\) (or simply \(Y\)) because its central notion is to use a
single program representation, the abstract syntax tree, throughout program editing,
execution, and debugging.

A \(Y\) environment for a particular language is built by providing a specification
for that language. The specification defines three fairly separate parts of the
language: its abstract syntax, its static semantics, and its execution semantics.\(^5\)

Central to all three facets is the abstract syntax tree representation of the
program being operated on by the environment. The first part of the specification

\(^4\)This is historically true. Recent Lisp descendents like Common Lisp [Ste84] and T
[Ree84] use lexical scoping, but support typing only to allow compiler optimization. Lisp exten-
sions like flavors [MoS83] are specifically resolvable only at run-time.

\(^5\)In Norse mythology, Yggdrasil was the World Tree—a great ash that bound together
Earth, Heaven, and Hell [Bul62].

\(^6\)An analogy with yacc [Joh78] may be helpful. \(Y\) is to programming environments as yacc
is to parsers; just as yacc restricts itself to LALR(1) grammars, \(Y\) accepts only Algol-like
languages. Finally, \(Y\)'s static specifications correspond to grammar productions, while its
dynamic specs are code fragments just like yacc actions.
defines the constructors of the abstract syntax. Each constructor is viewed as a tree
node with a fixed number of sons. Nodes may also be grouped into explicit lists; the
list is a constructor built into the system, and only it has a variable arity. Unlike
many other tree representations for programs (Lisp S-expressions, for example) every
link in the tree can be traversed in either direction. This allows many portions of the
environment to easily exploit the structure of the tree in an efficient manner.

Other parts of the syntactic specification allow the construction of a user
interface for the editor portion of the environment. Yy uses a structured editor with
screen-oriented positioning and tree-oriented editing. An incremental unparses is used
to build the display presented to the user. Editing operations deal with the copying,
clipping, and replacement of selected subtrees in the AST. The syntactic specification
is used to determine which operations are syntactically valid. Menu-driven addition of
new constructs is also generated from the syntax.

Of course, simple syntactic validity does not insure a well-formed program.
Further checks (which we call static-semantic checks for want of a better term) are
also performed in the editing phase. The checks are specified by annotating each
constructor in the abstract syntax with a checking function. The system invokes these
functions for the appropriate constructors when the tree is modified in the editor. The
functions examine their subtrees to find errors that are not caught by the syntactic
phase. The system can store the errors for later correction by the user.

Regulating the use of names is a major part of the environment. Name usage is
partially controlled by the abstract syntax, and partially by the static-semantic
checkers. A large part of the built-in system is concerned with naming. We provide a
general mechanism for building programs with modularity and data abstraction, by allowing users of the environment to partition a program into distinct scopes. We believe that restricting name usage in the interactive framework of the environment is more desirable than building new syntax for modularity into the language.

Once a correct AST is built, it can be executed directly from the language specification. To do this, each constructor is also annotated with a semantic function. This function specifies the execution semantics of a constructor in terms of a composition of primitive operators provided by the system, and possibly the results of evaluating the semantic functions of the constructor’s sons. Each function also indicates where in the AST execution is to continue. An interpreter is built up by repeatedly calling semantic functions. Since the “location counter” of the executing program is merely a pointer into the AST, the editor can be trivially extended into an execution monitoring facility.

The provided system primitives can be viewed as the instructions of a fairly low-level machine, with typical arithmetic and logical operations and a linear memory. Since the goal is to provide an excellent debugging environment, the primitives are capable of checking for many run-time errors that cannot be detected statically, such as out-of-bounds array subscripts and pointers. The primitives also allow the user to monitor the changing state of memory.

Because many programs of realistic size cannot be executed fast enough to allow reasonable debugging with an interpretive evaluator, we also provide a mechanism for executing compiled code. The debugging of compiled code is possible because the memory layout of the variables of the interpreted program is identical to that
assumed by most compilers. Dynamic linking is used to make calls from compiled to
interpreted code.

Finally, the specification addresses the problem of mapping the AST
representation of a program, along with naming restrictions, to a valid syntactic
representation suitable for compilation. The structure of the program as constructed
by the environment is preserved as much as possible, keeping in mind that many of the
constructs used in the AST may have no analog in the language’s external syntax.

1.5. Research goals

YG has several research goals. These are

(1) Given the desire for an IPE for Algol-like languages, and the design concept of a
single intermediate language, how can the system be structured? This represents
an attempt to contribute to the maturity of the area.

(2) How can the system be designed to support a wide class of languages of the Algol
family through a mixture of primitives and specifications? (We are especially
concerned with how the execution semantics of the language will be specified.)

(3) What language features make more sense and can be better supported in the
context of an interactive program editor which does direct static checking? Of
particular note are features for data abstraction and modularity, as found in Ada
and Modula-2.

(4) The user interface for such a programming environment is a radical departure
from conventional text editing. What new display facilities does this make
possible, and how can the user interface be made more effective than existing
ones? Obviously, a powerful system with an unnatural user interface is not very useful.

1.6. Why specifications?

We choose a specification-driven approach rather than simple ad hoc construction for several reasons. First, we want to avoid making design decisions based on the character of a particular language. Also, it seems that the class of Algol-like languages have so many features in common that a general framework to specify all of them would involve little more work than building a hand-crafted system for some individual member of the class.

It is important to recognize the limitations of the specification-based approach. A specification may be completely general from a theoretical viewpoint and yet be too unwieldy for real use; consider the single-tape Turing machine. On the other hand, we must be careful not to make our specifications so tied to particular features that they are incapable of handling new and unanticipated ones. We make no claims that $\mathcal{Y}$ specifications can build an environment for every programming language. Indeed, even within the Algol family we may often support a particular discipline of programming in a language-independent way, rather than provide a mechanism to support all of a particular language's quirks. Our ultimate goal is to allow the fairly easy specification of environments for languages like C, Pascal, and Modula-2, as well as extensions to well-established languages like Fortran and Cobol.

We are also not claiming that a complex language can be easily specified. No matter what the power of a specification method, nothing will make the specification of a language with the complexity of, for example, full Ada easy.
1.7. Why Algol-like languages?

In view of the success of environments for Lisp, and the development of new languages designed with IPEs in mind, such as Smalltalk-80, the question can be asked of why we should continue to bother with Algol-like languages at all. There are a number of compelling reasons to do so.

First, it is unlikely any language radically different from the Algol-like class will dominate programming in the foreseeable future. Despite their obvious advantages, late-bound languages like Lisp and Smalltalk are too demanding of hardware resources with current technology. (Consider that the existing Smalltalk-80 environment needs a $150,000 microcoded workstation to run at effective speeds.) Nor have such languages demonstrated their utility for extremely large projects. The sheer inertia of millions of man-hours of programming in Algol-like languages, which has produced the vast majority of all the programs in existence today, will keep those languages important for decades.7

More importantly, however, is the fact that the nature of programming is largely the same regardless of the language used to express the program. The concepts of abstraction, modularity, data structure choice, and the use of efficient algorithms, which are the very essence of programming, transcend the language used for their implementation. What we are really trying to investigate in this thesis is how people program, and what a programming environment can do to ease their efforts.

---

7In fact, most programs today are still written in Fortran or Cobol. It may be some time before even other languages from the Algol family become heavily used.
1.8. The plan of this thesis

As we saw above, the problem of specifying a programming environment can be split naturally into three subproblems: specifications for syntax, static checking, and dynamic semantics. The thesis follows this subdivision.

1.8.1. Syntax

In Chapter 2 we discuss the basic abstract syntax tree, while Chapter 3 explains how programs map to this tree and how users edit programs through tree manipulation. Chapter 4 focuses on the treatment of name definition in the program. Chapter 5 describes the unparsing mechanism used to build a textual view of the tree for display to the user.

1.8.2. Static checking

Chapter 6 describes the static checking mechanism required to enforce language restrictions which cannot be expressed through the syntax. We also introduce the general form of dynamic specifications in the system here.

1.8.3. Execution semantics

The largest part of the design of \( Y \) deals with the specification of language execution semantics. We introduce our specification technique in Chapter 7, after a brief discussion of the approaches used in previous work. Chapter 8 contains a treatment of how the program's execution state is manipulated. In Chapter 9, we explore the requirements of the debugger user interface, and describe how run-time errors are detected and handled.
1.8.4. Textual considerations

Chapter 10 is a discussion of the textual aspects of programming languages, and how these aspects are handled in \( Y_g \).

1.8.5. Conclusions

Finally, in Chapter 11 we examine the successes, failures, and omissions in the \( Y_g \) design.
CHAPTER 2

The Abstract Syntax Tree

Trees shall be my books, and in their barks my thoughts I'll character.
As You Like It, III, ii

YG is a structured editor, in that it manipulates not the source text of a program, but the program's abstract syntax tree representation. In contrast to other structured editors, YG uses the AST representation for the entire program, without auxiliary data structures like symbol tables. As such, the AST and its associated functions are critically important.

We begin by describing the AST in terms of a high-level abstract data structure. Because YG is designed to be implemented in a lower-level language like C or Pascal, that have no support for the kinds of high-level operations we will use, we then describe the AST in terms of a concrete implementation based on pointers.

2.1. Abstract view

From an abstract viewpoint, the tree is built from constructors, each of which has a fixed number of arguments identified by selectors.1 (We will frequently refer to constructors as nodes.) The only exception to the fixed arity of constructors is a single built-in list constructor, which can have any number of direct descendents. Each argument is also typed with a set of constructors which may be used in that position.

A particular constructor in the tree is identified by a path to that constructor [HoC84]. The path consists of a sequence of selectors, and is traversed to the

1This terminology is borrowed from [Car80].

23
constructor it identifies by choosing the specified selector as each node from the root to the identified node. The path can be extended to identify an argument of some node, or retracted to identify a node's father. The space of constructors can contain several rooted trees, and is therefore a forest. Each path explicitly contains the root of some tree in the forest.

Modifications are somewhat more difficult to characterize. To remain purely functional, a modification operation has to produce an entirely new copy of the tree with the appropriate modifications made. This may result in some paths that were previously valid to become invalid, as they would specify selectors at nodes which do not exist in the new tree.

`clip(path)` causes the entire subtree at path to be detached from the tree. Two results are returned; a path to the subtree, now a separately rooted tree, and a path to the subtree's father in the new version of the initial tree.

`replace(path1, path2)` replaces the subtree at path1 with (a copy of) the subtree at path2.

`copy(path)` returns a path to (a copy of) the subtree as a rooted tree. Note that copying may not actually be necessary as we could do some kind of sharing in the functional framework.

The list constructor is introduced so constructors can appear to have arbitrary arity and still have a fixed number of selectors. Lists have implied selectors 1..length of list so that paths are capable of leading to any element of a list. Functions exist to add elements between existing ones. These are viewed like the other modification functions as producing new trees.
2.2. Low-level tree

The path model is attractive as a data abstraction, but it cannot be implemented efficiently without language support [Hoo82]. We define low-level tree manipulation functions in terms of pointers to nodes and destructive updating, while making the connections between the high and low-level operations clear. We will present these functions in terms of data structures and functions that could be easily implemented in an Algol-like language without garbage collection.

The basic element of the tree is the node. The node is a record with several fields:

- **type** is an integer denoting the type of the node. This is used to interpret the values of the sons and defines their number.

- **father, next, and prev** are used to link nodes together. Note that the tree is always doubly-linked; given any node, it is possible to traverse the whole tree. This allows us to simulate the behavior of *retract* and *extend* in constant time, and using only constant space for a "path," which now just becomes a pointer to a node.

- **son1...sonN** point to the node's sons. The arity of a particular node is defined by its type.

Nodes are taken from a flat storage space, and each node is identified by an integer number. There is also a single string space, which contains unique strings of varying length also identified by integers. The son fields contain an object called a *tree pointer*. This pointer consists of two parts: tag and data. The tag specifies how the data is to be interpreted, as follows:
• **tree** denotes that the pointer refers to another node that is either the containing node's father or one of its sons. We call this pointer a *tree edge*.

• **graph** indicates that the pointer refers to a node not a direct descendent or ancestor of the containing node. We call this a *graph edge*. Note that the undisciplined use of graph edges is dangerous, because a node can be deleted or changed without altering any of the graph edges. Graph edges are introduced as a mechanism to share data efficiently. They also make it possible to update many potential uses of a subtree in constant time, without actually modifying the usage sites.

• **string** indicates that the data is an offset into the string space.

• **integer** indicates that the data is to be interpreted as an integer. This can be used to associate auxiliary information with the node.

Any other tag value can be used by the application. The system interprets all such tags as integer. There should be at least eight available user tags; the Yg design currently uses four of them.

The AST package distinguishes the value (tag 0, data 0) as the nil pointer.

Yg stores entire programs in a single node space. Therefore, the data fields should be as large as possible, as small numbers of nodes will severely limit the maximum size of an editable program. There need be no relation between the tree pointer size and the hardware size of an address, though many languages require function results to be scalars and choosing a scalar representation may ease programming. The initial implementation of the package for 32-bit machines uses an 8-bit tag field and a 24-bit data field.
2.2.1. Constructor functions

Nodes, integers, and strings are built by constructor functions as follows.

treePtr makeNode(type, son1, ..., sonN) builds a new tree node with the
specified type and the specified N descendents. The value of N is defined by the
node type by defineNodeType. If the implementation language supports the
feature, it is permissible to pass less than N sons, in which case the remaining
sons are set to NIL. A pointer to the new node is returned.

defineNodeType(type, arity) specifies that the integer type is a node type
with number of sons arity.

treePtr makeInt(int) returns a tree pointer to the scalar integer value.

treePtr makeString(string) adds string to the single string space if not
already present and returns a tree pointer to it.

treePtr makeGraph(tree) makes a graph edge from the specified tree edge and
returns it. If tree is not a tree edge, the argument is simply returned.

treePtr copy(tree) copies the entire subtree under tree and returns it as a
rooted tree. If tree is not a tree edge, the argument is simply returned.

2.2.2. Positioning

Motion from one tree node to another is accomplished by the following functions:

nodePtr getField(node, field) returns the field\textsuperscript{th} son of node.

nodePtr getFather(node) returns the father of node.

integer whichSon(node) returns the number of the field which contains the tree
edge to node in node's father.
2.2.3. Modification

The tree structure can be modified by these functions:

clip(node) detaches node from the existing tree structure and returns it as a rooted tree, updating all appropriate fields. (i.e., sets the reference to node in node's father to NIL and sets node's father to NIL.)

replace(node, field, new) replaces the field\( ^{th} \) subtree of node with the subtree rooted at new and updates all appropriate fields. New is returned. The subtree present in that field before is lost. There are two possibilities for how to do the update: either the old subtree is simply thrown away (possibly after reclaiming its storage), or the old node is modified in place to point to the new node's fields. The latter has the advantage that any graph edges that previously referenced the old node would then painlessly reference the new one instead of being invalid. On the other hand, this is confusing and possibly leads to a false sense of security as regards graph edges.

2.2.4. Lists

Nodes may be arranged in pure trees, or they may contain lists of nodes. For reasons of efficiency, the lists are represented explicitly. For this purpose, the node type LIST is reserved by the system. Each node contains a pointer to the previous and next node in its list. The LIST node has two sons. The first is a tree edge that points to the first node in the associated list, and the second is a graph edge that points to the last node in that list. (The second pointer exists so that appending and positioning to the end of a list may be done in constant time.)
The package provides a series of primitives that construct, modify, and position in list structure, maintaining the fields of the LIST node. These are:

`treePtr makeList(node)` builds a list with a single element specified by `node`, and returns it.

`treePtr appendToList(list, node)` adds `node` to the end of `list`, and returns `list`. This function is mainly provided for use in the noninteractive context of a parser; it is also the only way to add elements to an existing empty list.

`treePtr insertAfter(node, new)` inserts `new` into the list of which `node` is a member, immediately after `node`. An error results if `node` is not the member of a list. The new node is returned.

`treePtr insertBefore(node, new)` is as `insertAfter`, but `new` is inserted before `node`.

`treePtr copyRange(first, last)` The nodes between `first` and `last`, inclusive, are copied. A new list containing the copied elements is constructed and returned.

`treePtr clipRange(first, last)` The nodes between `first` and `last`, inclusive, are removed from their containing list. A new list containing the removed elements is constructed and returned.

`treePtr spliceAfter(node, list)` The elements of `list` immediately after `node` are inserted into the list of which `node` is a member, immediately after `node`. The first element of `list` is returned.

`treePtr spliceBefore(node, list)` As `spliceAfter`, but the elements of `list` are inserted before `node`. 
treePtr next(node) returns the element after node in its containing list. If node is the last element, NIL is returned. If node is not in a list, an error results.

treePtr prev(node) returns the element before node in its containing list. If node is the first element, NIL is returned. If node is not in a list, an error results.

2.2.5. Storage Management

Since a tree can be easily traversed and marked, the space of tree nodes could be garbage-collected by a conventional mark-and-sweep algorithm. A complication is introduced by the existence of functions like makeNode and clip, which produce detached, or separately-rooted, trees. Since all trees coexist in a single space, the root of each one would have to be known for a garbage collector to work correctly. This would imply that the system keep a list of the root of each tree stored in the node space. Since there is no way to tell from the tree package when pointers to such trees (which are simply variables in the implementation language) are no longer of interest, an explicit release primitive would still be necessary for the garbage collector to be effective.²

An alternate approach would be to provide a primitive to release the storage used by an entire subtree. This requires more work on the programmer’s part, but our experience with the IR² editor indicates that such freeing needs to be done only at a few well-defined places in the editing process, and is not an unreasonable programming

---

²Remember we are assuming the absence of garbage collection support from our implementation language. If such support was provided, we could probably exploit it.
burden.

2.3. External representation

Trees may be described textually in a fashion similar to the Lisp S-expression. Integers and strings appear as numeric and quoted atoms, respectively. Graph edges are represented as labels. A label is an atomic name with a colon as the last character. Lists are represented as LIST nodes with an arbitrary number of sons. For example,

```
(scope
  (list
    1: (name "i")
    2: (name "j")
    3: (name "foo")
  )
  (list
    (do (< 1 2:)
        (list
          (set 1: (invoke 3: 1:))
        )
    )
  )
)
```

2.4. Higher level positioning

While the low-level functions are sufficient to perform any possible manipulation of the tree, several parts of Y9 find it convenient to use a somewhat higher level notion of positioning in the tree. This interface is based on a notion of structured motion functions, which are similar to the user interfaces of earlier programming environments [Alb81, MeN81].

Each constructor in the supported programming language's abstract syntax is represented as a node. These nodes are separated into two sets, statement and
substatement. (The exact distinctions between the two sets are explained in the next several chapters.) The statements are again divided into two groups, simple and compound. Simple statements have no statements in any subtree. Compound statements have a single list of statements as one subtree, possibly along with other nonstatement subtrees.

We define the following positioning functions on statement nodes:

\texttt{in(node)} returns the first node in a compound statement's sublist, or nil if the node is simple or nonstatement.

\texttt{inToEnd(node)} returns the last node in a compound statement's sublist, or nil if the node is simple or nonstatement.

\texttt{out(node)} returns the compound node containing the list that the node is a member of, or nil.

\texttt{next(node)} returns the next element of the list the node is a member of, or nil.

\texttt{prev(node)} returns the previous element of the list the node is a member of, or nil.

It is extremely important that each of these operations be implementable in constant time; this condition is easily met by the low-level package.

2.5. Storage Efficiency

In contrast to text-based systems that separate a program into several disjoint text files, \texttt{Yg} uses a single tree space to store the entire program. This has the important advantage that any desired information about the program is directly accessible, but it has the obvious problem of making the maximum program size a
function of the maximum number of nodes that can be contained in memory.

Storage efficiency has been a problem for other structured integrated environments. For example, the Magpie system [SDB84] uses about 1500 bytes per line of Pascal source code (including throwaway machine code), imposing a maximum program size of 5000 lines in the entire 24-bit address space of a 68000 processor. The Gandalf ALOE editor described in [MeN81] uses a minimum of 40 bytes for each tree node. The current IR^n editor can edit a maximum module size of perhaps 2000 lines.

The initial implementation of the \( Yg \) tree package uses the same storage technique as the IR^n Fortran editor. Nodes are allocated from a single node array. For simplicity, each node in the array is fixed size and has enough son fields to accommodate the maximum arity of any node. This is likely to waste a great deal of space, and seems to in practice; measurements of stored IR^n trees indicate that they are 70-90% unused fields.

We have found little data on the relationship between program size by a conventional metric, such as number of lines, and the number of nodes required by that program's AST representation. In order to estimate the storage requirements of a structured editor, we built a simple parser for C, using the Unix yacc and lex tools. Rather than actually build the AST, it counts the number and arity of each node that would have been used. The tree format was taken from the \( Yg \) C specification found in Appendix 1. This parser was run on a diverse set of programs taken from the Unix toolset and several other C programs in local use. The results are shown in Figures 2a

\[ \text{All comments and blank lines were removed from the input source, but any include files were also processed. The space used by an actual program in } Yg \text{ format would be somewhat less because global declarations would be shared. The source line count is a questionable metric; most code probably has three times as many lines taken up by white space and com-} \]
and 2b.

It should be noted that for C, a language that is extremely "expression-oriented," over 80% of the nodes have either unary or binary arity. (Nodes with arity 4 are more frequent than arity 3 nodes because the variable definition node we are using has arity 4.) This suggests that a better strategy is to use multiple node spaces, one for each different arity. This could be easily implemented by using different tags for each node space, and for our observed node distribution, would result in space savings of a factor of about three. We expect similar distributions of node use for other Algol-like languages.

<table>
<thead>
<tr>
<th>arity</th>
<th>number</th>
<th>fraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>13261</td>
<td>8.2</td>
</tr>
<tr>
<td>1</td>
<td>98351</td>
<td>60.5</td>
</tr>
<tr>
<td>2</td>
<td>42449</td>
<td>26.1</td>
</tr>
<tr>
<td>3</td>
<td>1772</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>6686</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Figure 2a: Distribution of node arity in C programs

<table>
<thead>
<tr>
<th>program</th>
<th>lines</th>
<th>space</th>
<th>per line</th>
</tr>
</thead>
<tbody>
<tr>
<td>csh</td>
<td>12863</td>
<td>973181</td>
<td>75.66</td>
</tr>
<tr>
<td>emacs</td>
<td>27861</td>
<td>2595724</td>
<td>93.17</td>
</tr>
<tr>
<td>led</td>
<td>1673</td>
<td>163422</td>
<td>97.68</td>
</tr>
<tr>
<td>most</td>
<td>3940</td>
<td>273911</td>
<td>69.52</td>
</tr>
<tr>
<td>nroff</td>
<td>8574</td>
<td>618053</td>
<td>72.08</td>
</tr>
</tbody>
</table>

Figure 2b: Space used by the AST representation of C programs
2.6. Tree Segmenting

We have so far assumed that virtual memory allows us to store arbitrarily large trees entirely in memory. Assuming a typical space usage of 100 bytes/line, this leads to a maximum program size (assuming a 24-bit address space and a 1 meg program, leaving 15 megabytes for tree storage) of 150000 lines, approximately.

However, in typical editing most of this storage will not needed at any given moment. Most editing activity will be constrained to a small subset of the entire number of procedures and modules, and the other modules are not really required. The only parts of modules that are needed are those that might be referenced during editing—the public definitions made by the modules. These might include the types of procedures, parameters, and global data, but not the bodies of procedures.

Many structured editing systems handle this by explicitly breaking the tree into discrete editable units. We prefer to avoid this solution, for the following reason: after the modification of a global object, we will want to examine all of the usage sites of that object. If the entire tree is resident, we can easily find each site, as the usages are directly linked together. A system that used discrete editable units might be unable to do this without forcing the user to load each unit and examine it.

An alternate scheme would be to load a nonresident body when some need for it was detected. There are two ways to do this: by lazy loading of bodies, and by tree paging.

---

4 The Gandalf system does this explicitly through the use of the FAILUP and FAILDOWN action routines.
In lazy loading, the global part of the tree contains no direct pointers to the unneeded body of a tree construct. Instead, it merely contains "soft links" that indicate that given names are referenced in the body. If the request is made to examine those references, the body is loaded, and the soft links are replaced with direct links to the references in the body. When storage runs low, inactive bodies are unloaded and their references replaced with soft links again. This scheme implies that the bodies as stored on disk have no explicit pointers to storage other than that used by global declarations, and that these global declarations have invariant addresses. Some manipulation of the node space, or the use of several node spaces, may be necessary to achieve this.

A second scheme uses a paged node space. Just as typical virtual memory hardware splits an address into a page number and an offset within the page, paging of the node space splits the data part of a tree pointer into these two parts. The available main memory is used to store active pages, and a conventional page replacement algorithm is used to flush less-used pages when others are requested. In order to avoid thrashing, nodes in the same part of the tree should be allocated in the same pages when possible. This would tend to happen naturally, since editing (and associated tree manipulation) will typically occur in the same part of the tree. With some care, this scheme could achieve the same performance as explicitly unloading subtrees without the complications required by that method.

A slight variant of paging is to use segmentation. Rather than splitting the node space into small pages, we break it into larger segments, each segment containing the nodes of some logical subtree in the program, such as a procedure. This simplifies
allocation of nodes, since all of the nodes likely to be referenced in the editing of a procedure would be found in the same segment, rather than in several pages. This approach requires us to set limits on the maximum size and number of such logical subtrees. Such segmentation could be easily added by designating certain node types as *segment leaders*, such as *procedure*, *function*, *module*, and *program*. When such nodes were allocated, they would immediately be placed in a new segment.
CHAPTER 3

Syntax and the User Interface

You do so grow to my requital, as nothing can unroot you.

All’s Well That Ends Well, V, i

We have now discussed a package for manipulating the abstract syntax tree representation of a program. The Yg system builds such a representation from a specification of the supported language's abstract syntax. The editor in Yg directly manipulates this representation, relying on a display module to display the tree to the user.

Most user editing actions consist of adding, deleting, or replacing elements of the tree. Addition and replacement is done by selecting a particular construct from a menu of valid items. This menu is generated by examining the position of the current focus, and using the language's specification to build the list of possible alternatives. Because the editor only allows menu selections to be inserted, a large degree of correctness can be insured simply by requiring that the menus contain valid possibilities. This chapter will discuss the part of the language specification that deals with this first line of defense in insuring correctness—enforcement of purely syntactic constraints.

3.1. Basic user interface

Each editor instance maintains two pointers to locations in the AST; these pointers specify the focus. The focus is the region of the tree that the user is currently interested in; the display arranges to highlight this region. All commands that act on
the tree use the focus as their operand. The two pointers, startFocus and endFocus, are interpreted as follows:

- if startFocus = endFocus, the focus is the subtree rooted at startFocus.
- otherwise, the focus is the list of nodes between and including startFocus and endFocus. In this case, startFocus and endFocus must both be in the same list, and startFocus must be before endFocus in that list. The editor makes sure that the focus meets these conditions.

The basic editing operations on the AST are

1. insertion of a new tree element, either before or after the focus.
2. expansion of the focus, if it is an incomplete field in the tree.
3. deletion of the focus. The focus is saved in a detached tree for later use. If the focus was a sublist, then the detached tree is “listified”, as the tree package does not allow a disembodied list without a LIST node.
4. copying of the focus, creating a new detached tree.
5. insertion of an existing detached tree either before or after the focus.
6. replacement of the focus with a detached tree.
7. “imbedding” — making the current focus a subtree of a new node.
8. “extracting” — deleting the focus' parent and inserting the focus in the parent's old position.
9. “mutation” of the node type of the focus to another, “conformable” type.

Several of these operations require that a menu of valid constructs be constructed and displayed to the user. The valid constructs are defined by a syntactic
specification of the AST, described next.

3.2. Syntactic specification

The source language is specified to contain a number of particular node types. For each node type, the number of fields and the name of each field is given. Finally, each field is given a syntactic type, which specifies what type or types of nodes may be valid descendents of the node at that field. From this simple specification, menus can be constructed and syntactic validity insured.

The specification format is simple. A node definition looks like

\[
\text{node-name field-name:syn-type field-name:syn-type...}
\]

Nonterminal names are defined by

\[
\text{nonterminal-name} = (\text{nonterminal-name} \mid \text{node-name})^*
\]

The same nonterminal name may not appear on both the left- and right-hand sides.

Syn-type may be the name of a node, a nonterminal name, or a node name or nonterminal name with the suffix "-list" appended. The list suffix simply indicates that a list of such nodes is the valid reference for the field. (In cases where both singletons and lists may be valid, a list of length one is used to represent the singleton.)

3.3. Menu generation

If the user wishes to expand a field or add a new item to a list, the system must build a menu containing the valid alternatives at the location indicated by the focus. It does so by examining the syntactic type of the field to be modified. If that type is a node name, there is only one valid expansion. If it is a nonterminal name, the menu is
constructed from the nonterminal definition. Node names within the nonterminal correspond to immediate menu selections, while nonterminal names, if selected, yield another menu.

If the operation to be performed is a list operation such as insertion, then the field must have a list syntactic type. If not, the system disallows the entire operation. Otherwise, a menu is generated as before.

3.4. Tree modification

The tree is modified in response to menu selection. When a alternative is chosen, a possibly incomplete node is added in the desired place in the tree. The incomplete node is represented in the tree by a node of the selected type, with descendent meta nodes, possibly in list structures. A meta node has one field, which refers to the syntactic types of nodes that it could be legally replaced by. The meta nodes simply serve as placeholders in the tree for unexpanded tree structure. The new node and its descendent lists and/or metas is called a template.

Templates are generated from the syntactic specifications by the following algorithm, which returns the completed template.

```plaintext
buildTemplate(nodetype) ≡
    root = makeNode(nodetype)
    for i = 0 to nFields(nodetype)
        if nonTerm(fieldType(nodetype, i))
            if isList
                field(i) = makeList(meta)
            else
                field(i) = meta
        else
            if isList
                field(i) = makeList(buildTemplate(fieldType))
            else
                field(i) = buildTemplate(fieldType)
    return root
```
Note that if there are no choices in a field's syntactic type (that is, it is a single node type or a list of a particular node) the corresponding structure is built, rather than a meta. This minimizes the number of single-choice decisions the user is asked to "make."

After the template is inserted, the focus becomes the first meta node in the new subtree. This saves the user from having to manually position to the next possible expansion point, making the construction of new code the repeated expansion of metas.

3.5. Deletion

A shorthand user action is replacement—the focus is deleted and then the system queries the user for a new construct. Simple deletion also causes the focus to be deleted. When the deletion occurs, it may be necessary to recreate the original template structure. This is done by the following algorithm.

```
fillField(synType) ≡
    if terminal(synType) return buildTemplate(synType)
    else return makeNode(META)

deleteAndTemplate(node) ≡
    get father, fieldInFather
    new = fillField(getSynType(type(father), fieldInFather))
    if notInList(node)
        delete(node)
        field(father, fieldInFather) = new
    else
        if lastInList
            insert(new)
        delete(node)
```

If the user wanted to replace the deleted node, the system simply goes through the expansion process on the new template (without redisplaying the tree).

1"First" should be defined as the first appearing in the textual representation of the subtree. This will usually correspond to the first encountered during an inorder walk of the tree.
3.6. Adding existing trees

The detached trees created by deleting or copying a focus can be inserted into another part of the tree. If the focus is in a list, and the detached tree is a list, then the list's contents are spliced in. If the focus is a meta node, then the meta node is replaced by the contents of the detached tree.

In all cases, the validity of the detached tree in the new position can be determined. For simple nodes, the type of the node must be contained in the syntactic type of the field where the addition is made. For lists, the field must have a list syntactic type, and each node in the list must be contained in that type. The system does not allow insertion operations when the focus is not in a list.

3.7. Imbedding and extracting

Imbedding can be accomplished by taking the current focus and examining all nodes to see which nodes have fields that the current focus might occupy. A further restriction can be made because the candidate nodes must also be valid in the same position as the current focus.

In general, there might be several fields in any given node that the focus might occupy. However, we expect that this action will be used primarily at the statement level, and since we have made the restriction that any given statement have only one sublist, this ambiguity does not arise.

We also make the assumption that even if the focus is a single node, the user might still intend for it to be made into a singleton list. Therefore, both list and nonlist fields are considered valid positions, and the single element is listified if a list is required.
Extracting is somewhat simpler. We simply check to see if the focus can validly occupy its father’s position, as in an insertion. If so, we delete the focus, insert it before the father, and delete the father. Any fields that were lost by this operation are still accessible in a detached tree.

3.8. Mutation

A major objection to structured editing environments is that seemingly simple transformations—such as changing an if statement to a while statement—could be easily made by a text editor, but involved complicated, non-intuitive manipulations of the tree. We solve this problem by allowing the definition of a series of node-specific transformations, called *mutations*, that can be made by the user in a single operation.

Consider the if statement

\[
\text{if } b = 0 \text{ then } b = f(b)
\]

which has a tree representation of

\[
\begin{align*}
\text{(if } & \text{ (guard } (= b 0) \\
& \text{ (list } \\
& \text{ (set b (invoke f b)) } \\
& \text{ ) ) ) )}
\end{align*}
\]

The mutation desired alters this construct to

\[
\text{while } b = 0 \text{ do } b = f(b)
\]

with tree structure

\[
(\text{while } (= b 0)
\]
(list
  (set b (invoke f b))
)
)

Therefore, the mutation can be described as

if => while
  while:cond = in:cond
  while:body = in:body

which specifies that the while should have the condition and body of the first guard in the if's list.

A guard mutation can also be defined as

guard => while
  while:cond = guard:cond
  while:body = guard:body

Note that mutations may not result in valid trees. For example, a guard=>while mutation might do what the user intended, but it would leave an illegal while node in a list of guards. In this case, the node is marked as in error and the user is required to move it. After all, a similar result is produced by simple text editing.

3.9. Errors

When any part of the system detects an error, the construct in error is marked. (Each node contains an error bit, and the user interface displays nodes whose bit is set differently.) Using purely syntactic checking as described so far, the only user actions that can produce an error are moving an existing piece of the tree to a position where it is invalid, and mutation.
3.10. The role of menus

We have mentioned the obvious connection between template-based expansion and nonterminals in the definition of a language's abstract syntax and menu selection. Nonterminals in the syntax represent choices which can be presented to the user in menu form, and indeed, this mode of interaction is available in $Yg$.

However, informal studies done with the IR$^2$ structured editor indicate that standard menu selection is undesirable for the experienced user; in particular, it is too slow during initial program entry. This fact has led the designers of several systems to use text-based incremental parsers [DMS84, HoM84]. $Yg$ does not go to this extreme because of the great advantages of tree-based entry. Instead, we use two techniques that preserve the flavor of tree editing while attaining more entry speed.

The first is to avoid the proliferation of menu selections. The danger of a series of successive nonterminal selections is that every nonterminal encountered results in another menu to be responded to. We avoid this by using cascaded menus.\(^2\) Rather than using successive menus that simulate the tree structure of the nonterminals, we generate a complete menu tree initially, and display it as shown in Figure 3. This provides the descriptive power of the menu tree without requiring more than one user response.

The second tactic is to avoid menu selection entirely by using token entry from the keyboard. Each constructor in the syntax can be given a textual name used to generate its menu entry. If this name is entered from the keyboard, it is treated the same as if the user requested the menu and then ultimately selected that constructor.

\(^2\)Similar menus are used in the Interleaf document preparation system.
expr = lval binop assign assign-op
lval = deref name subscript
binop = plus minus times div
assign-op = plusab minusab timesab divab

Expansion of an expr meta node produces the menu:

<table>
<thead>
<tr>
<th>lval</th>
</tr>
</thead>
<tbody>
<tr>
<td>binop</td>
</tr>
<tr>
<td>assign</td>
</tr>
<tr>
<td>assign-op</td>
</tr>
</tbody>
</table>

and further selections cause subsidiary, "cascading" menus to appear.

Figure 3: A cascaded menu tree
(This implies that terminals in the syntax must have unique textual names.) The same mechanism can be used to enter name usages, as individual names behave just like constructor strings in the syntax. Menus for names will be discussed further in Chapter 4.
CHAPTER 4

Names and Scopes

Didst thou hear, without wond'ring, how thy name should be hang'd and carv'd upon these trees?

As You Like It, III, ii

So every scope by the immoderate use turns to restrain.

Measure for Measure, I, iii

Although the abstract syntax tree represents the structure of the program, leaves in the AST are not simply strings representing the program's identifiers and constants. Instead, the AST structure is used itself to store information, both about identifiers and constants, and their scoped visibility. This approach differs from a traditional compiler's symbol table, where a data structure separate from the AST stores such information. The reasons for this difference are manyfold:

- Using only a single data structure for all purposes simplifies the system's design.

- Because the tree contains scope information, a position in the tree uniquely identifies the active scopes where names might be found. With a separate symbol table, changing positions in the tree would necessitate the updating of the symbol table as well, and might cause significant delays when the user changed focus. Also, the system will typically have many foci active simultaneously, causing problems for a single symbol table.

- Many constructs that would appear in the symbol table (such as types, record definitions, and initializers) are themselves trees.
• Because a usage site for a name in the tree is directly linked to the correct
definition, the interpreter for the AST need not worry about name binding at run
time—a traditional problem in Lisp interpreters.

Names and scopes are partially specified by the environment implementor. However,
the lookup mechanism for names is hardwired into the system. It is sufficiently
general to handle the naming features of most languages.

4.1. The philosophy of naming

Naming is perhaps the single most important abstraction feature in programming
languages. At the highest level, a program is simply a collection of names; each name
represents a language object such as a procedure, a type, or a variable.

However, most programming languages fail to view names as the important
concepts they are. Instead, names are buried in a morass of declaratory syntax. This
has become even more true with the introduction of languages like Ada and Modula-2,
which control the use of names by adding even more declaratory syntax. The problem
with these syntactic extensions is that it becomes more difficult for the programmer to
provide the right syntax to express his intentions. The compiler can check to see if the
program is legal, but if not, it can only point out errors in consistency present in the
declaratory syntax. Rather than an assistant, the compiler is an “off-line” enforcer of
syntactic rules. In the multiple-programmer environment for which Ada is
particularly intended, those errors may not even be the responsibility of the
programmer requesting the compilation.

In addition, languages like Ada and Modula often require a great deal of
redundant information to be specified. For example, a user of a Modula module must
write declarations of that module's contents, and the definer of the module must also
give them, in a slightly different form. The combination of complicated syntax and
redundancy increases the amount of detail work required of the programmer, and only
makes it harder to get a program past the compilation phase. By strongly decoupling
execution from editing, the compiler only distracts the programmer from the real task
of getting the program running.

The tree-editing framework of Yg gives us an opportunity to change the role of
the computer in the language user's view of the programming process. Rather than
operate in a batch-oriented mode, where the compiler looks at textual input and
reports errors in it, the editor can quietly point out potential errors while the actual
programming is taking place. This can be a fundamental difference, particularly when
issues of modularity and data abstraction are involved.

What modularity and data abstraction really do is limit our ability to see other
parts of a program from a particular locale. In a text editor, we use variable names
without any constraints except our own beliefs; we may think that a particular
variable is visible from the function we are composing, but the system is not going to
confirm that belief until we try to compile. In Yg, composing a function can be quite
different. Immediately after we try to reference some name the system can check to
see if it is already visible, and if not, may allow us to import it from elsewhere or
define it locally. The editor can even construct menus of accessible names.

From here it is an obvious extension to give the structured editor the ability, not
only to facilitate, but to control a programmer's access to names. We can provide
editor features to encourage modularity and data abstraction without having such
features defined in a language's purely textual representation. This aspect of the editor is largely language-independent.

4.2. Names in the AST

Leaf nodes in the AST are represented as nodes of type nameref. This node has one field, name, which is a graph edge to the corresponding name definition. (A field requiring a name usage cannot simply contain the graph edge itself, since the system must have a unique pointer to every distinct use of a name for editing purposes.) A name definition consists of a name node with three fields.

- The string field is a tree pointer into the string space, representing the name's external ("print") string representation. We will call this the name's "string."

- The value field is a subtree that further describes the name. For each type of named object in the language, there is a specific node type to be used as the value subtree; for example, a typical language might have value subtrees of variable, constant, type, and procedure. Following the terminology of [Rei83], we will call the type of the value subtree's root node the class of the name.

- Because leaf nodes are graph edges, there is no way, given a name definition, to find all instances of the name's use in the tree. Since this could result in laborious searching of the tree if a name were modified, the use field is a list of ref nodes containing graph edges to the name's various uses. The system maintains this list as the tree is modified and name references are added.

For documentation purposes, a block of text can be associated with each name; the system allows the user to view and modify this text.
4.3. Scopes

Constructors that contain name definitions are collectively called scope nodes. Every such node type has a field called names, which is a list of name nodes defined in the scope. Some scope nodes will also have other fields.

For example, the C fragment

```c
{
    int a, b;
    a = 0;
    b = 1;
    {
        char b;
        b = 'b';
    }
}
```

would be represented as

```
(block
  (list
    1: (name "a" ...)
    2: (name "b" ...)
    3: (name "0" ...)
    4: (name "1" ...)
    6: (name "b" ...)
  )
  (list
    (set (nameref 1:) (nameref 3:))
    (set (nameref 2:) (nameref 4:))
    (block
      (list
        5: (name "b" ...)
      )
      (list
        (set (nameref 5:) (nameref 6:))
      )
    )
  )
)
```

where block is a scope node with both a list of defined names and a body of
statements.

4.4. Name lookup

When a name is to be referenced in the program, the user enters some information about it; for example, the name's string. It is the job of the naming mechanism to match this information with the definition of some name, and enter a direct link to that name in the tree.

Given the correct matching function, lexical scoping tells us how the search should be performed. The system simply searches linearly through the name list of the scope that contains the future reference site, looking for a matching name. If no matching name is found, then the search is continued in the next enclosing scope, and so on. If the root scope is reached before any name matches, then a new name must be defined in some scope; the system stops to allow the user to do this.

The most primitive matching function requires only the string of the name to be known. Then, simple string comparison between the desired and the candidate name strings is used. However, this is inadequate for nearly all languages, since they usually have some notion of overloading—that is, two strings may appear in the same scope, but with different classes.

The second-order approach to matching overcomes this problem. We also require that the class of the desired name be known; then two items match if they have the same string and the same class. For example, consider the C language, where a structure name may have the same string as a variable:
struct ex {
    int a;
    char b;
};

struct ex ex;

These declarations would be represented as the two name nodes

(name "ex" (struct ...))

(name "ex" (var ...))

When the editor needed a variable name, it would specify both a string of “ex” and a class of var, causing only the second name to match.

A still more complicated notion of matching is required to handle overloading as seen in Algol 68 and Ada, where many procedure definitions are given the same string and a specific call site is matched with the correct definition based on the argument and result types of each definition. Even without such overloading, we will still want to do matching more sophisticated than matching by class. For example, all names that define variables are likely to have class var, but such names also have types, and we would like to restrict matching to those names whose types match those needed at the reference site. We assume the existence of a supplied function that tells us if a particular name definition is valid at a particular use site. This function is also used for general static checking, and is described in Chapter 6.

Regardless of the type of matching used, the following property holds—no two names defined in the same scope will match. This prevents “redeclaration” errors.
4.5. Lookup with incomplete information

Matching by string, class, and user-defined function works well in a compiler environment, because the compiler has all the information available about the restrictions on a particular use site. However, in the editor use sites may be only partially expanded, and all the information may not be available. A single match may not be possible. In such a case, the system must respond with a menu of the possible names.

The user will typically enter names in one of three ways: by string entry, by name selection, and by menu selection. What method is used affects the information available for name matching, so the system must cope with each method.

(Remember that we have two choices about how to handle reference errors; we can either insert the reference and mark it as in error, or we can reject it completely. The former method makes sense if the user added information that would be lost if the addition were rejected. On the other hand, an ideal menu never contains choices that can be rejected by later checking.)

When the user enters a name reference textually, we have about as much information as a compiler would have; the valid classes for the name are known from the abstract syntax specification and the name's string is provided by the user. The only difference will be that complete information about types may not be known, since the user may be entering a name in a location where there are unexpanded meta nodes in the same subtree. The checking function is only called if there are several names that match in string and class. (If no possibility matches completely, the reference can be left in the tree as text; this is called a soft reference, and will be described later.)
In the second method of name entry, the user selects a particular name definition from the display. This inverts the matching problem; instead of finding a specific name, the system must confirm that the specific name selected is valid. To do this, it must determine that the definition is accessible from the use—that is, that the use is lexically enclosed from the definition, and that no shadowing has occurred. To check this, the system simply uses the class known from the usage site, and the string taken from the selected name, and performs the search already described. If the result is the same as the selected name, the use is added.

The final way to enter names is to pick them from a menu generated by the system. If the system never displays an incorrect name in a menu, then no checking need be done after one is selected. The problem then becomes one of generating the menu in the first place.

Remember that there are several levels of validity for names. We could generate a menu by simply finding all names of the correct class, but this would mean that the types of the names might still be wildly wrong. We can correct this by synthesizing a reference to each candidate name and checking it, then putting only the names that do not cause errors in the menu.

4.6. Name reference expansion

Many names have associated reference operators. For example, the name of an array might only appear as the son of a subscript node. There are two ways we might expect such name references to be built. The first forces the user to explicitly build the subscript node, and then add a name reference to it. This seems undesirable; when a user wants to use a name, he would prefer to use it and then build the
reference structure.

We would like to support fast name entry by automatically building the correct reference structure. For example, procedure name references would automatically construct invocation nodes; this could be used to provide procedure call templates, with nonterminals for the types and number of parameters.

To allow such entry, each name must provide a list of its reference subtrees, and the resultant type of those trees. (It is actually sufficient for each type to provide this information, since checking is usually done only on the basis of type.) Each reference subtree is checked for validity, and the valid ones are further considered. This may mean that string entry causes a menu to appear; for example, consider the C declaration

\[
\text{int a[50], b;}
\]

and the reference

\[
b = \text{<name>};
\]

might be satisfied by either one of

\[
b = *a;
\]

\[
b = a[\text{<expr>}] ;
\]

Since there are two reference subtrees with the same type (or valid types), the choice between them must be made.

4.7. Properties of nested scopes

The discussion above implies that names are to be chosen from a single scope—a flat list of candidates. This is overly simplistic, since nearly all languages employ
multiple scopes and visibility rules across scope boundaries. Since names are clustered into scopes, nested scopes are easily represented as subtrees.

The most common form of scoping visibility is inheritance; a name is visible in any subtree unless it is shadowed by another name definition in an enclosed scope. This form of scoping is used in Pascal, C, Algol, and most other Algol-like languages without modules. A second kind of scoping, introduced in Modula, Ada, and other languages with facilities for data hiding, is concerned with exporting and importing names. In these schemes, a name can be made visible outside its defining scope through the use of explicit or implicit qualification.

Rather than attempt to encompass all possible variants of modular scoping, we provide a single mechanism to support data hiding. Remember that ordinarily, a name is looked for lexically in every scope from the usage site to the root of the tree. In order to implement the concept of restricted visibility and explicit qualification, searches may also be qualified. A qualification list is a list of scope names, the first of which identifies the scope where the search is to begin. Further elements indicate the scopes that are to be entered to find the specific scope to be searched for the final name.

Several qualification lists may be separately entered by the user, in which case each name search is also made using these lists, after the standard lexical search is made. Language constructs may also specify qualification implicitly. For example, the C "dot" operator, which takes the form <variable of type structure>.<name of field in structure>, causes the search to begin in the scope indicated by the structure variable's type. Qualifications that point directly to a scope are called direct
Each particular scope node type specifies whether qualified searches are allowed in its name list. For example, the module and structure scopes might allow qualification, while the function and block nodes would not.

If a scope allows qualified searches, all the names within it become potentially visible. Individual names can be made invisible to qualified search by marking them private. References located within the same "module" that defined the private name are permitted, while outside are not. Another attribute of each scope node type indicates whether it acts like a module with respect to private names.

4.8. Name lookup algorithms

The lookup algorithm shown in Figure 4a is used when the name's string and possible classes are known. That in Figure 4b is used when only the possible classes are known; it constructs menus for the user. Finally, Figure 4c is the algorithm used for qualified search.

In each case, the initial scope will be either the scope containing the reference, or if the lookup is directly qualified, the scope pointed to. If the name is not found in that scope, then a qualified search fails.

4.9. Moving references between scopes

Once a reference has been resolved, a direct link exists between the use and the definition. This is a desirable state of affairs until the reference (or more likely, a subtree containing the reference) is moved or copied. Then the decision needs to be made as to whether the use must be re-resolved.
findInScope(scope, string, classSet)
    for all names in scope matching string
      if class(name) in classSet and staticCheck = OK return name
      else
        if definition shadows any others return STOP
      else return nil // we could add it to this scope

lookup(string, classSet)
    scope = startScope
    while startScope ≠ nil
      name = findInScope(scope, string, classSet)
      if name = STOP return no such name
      if name ≠ nil return name
      else
        scope = next outer scope

Figure 4a: Name lookup with string and class.

lookupMenu(classSet)
    while there's another scope
      for all names in scope matching string and not in shadowing list
        if class(name) in classSet and checkStatic = OK add to menu
      else
        if definition would shadow any others
          add name to shadowing list
      display menu; if name picked, use it and stop
      scope = next outer scope

Figure 4b: Lookup by class only

lookupQualified(qualList, string, classSet)
    scope = lookup(first(qualList), moduleClasses)
    qualList = rest(qualList)
    while qualList ≠ nil
      scope = findInScope(scope, first(qualList), moduleClasses)
      qualList = rest(qualList)
    // We now have the scope to look in
    return findInScope(scope, string, classSet)

Figure 4c: Qualified lookup

If the new location is in the same scope as the old location, then nothing has happened to alter the validity of the link, and it can be left. Unfortunately, the typical editing operation creates a detached tree. When the tree is detached, the
information about its containing scope is lost.

In order to handle this difficulty, we introduce the concept of a *soft link*. This is a reference node (of type `softnameref`) whose contents is not a pointer to a definition, but instead is the string of the desired reference. A soft link must be resolved just like an initial, user-entered string.

Consider the more complicated example of a reference moved between scopes.

```c
proc1()
int a;

    a = 1;

proc2()
struct t a;
...
```

We copy the subtree for "a = 1" from its scope in proc1 and try and insert it in proc2. The first definition acceptable on syntactic grounds is "struct t a," and it shadows any other variable definition of `a`. Therefore, the soft link to `a` is "hardened" to point to the structure `a`, and static checking determines that this is an illegal reference.

Because soft links are just strings, they have no determinable type. They are exactly equivalent to the string typed by the user during program composition.

### 4.10. Name display

Earlier structured editors, such as GNOME [GaM84], and the Cornell Program Synthesizer, displayed and edited declarations just like any other piece of syntax. The user's view of name declaration and usage was almost the same as with a text editor; the only difference was that syntax errors in declarations were avoided.
In keeping with our design philosophy of emphasizing the name structure of the program, the top level of ยก displays the whole program as a collection of names. The interface is uniform regardless of what the names define; any named object is viewed and edited in the same way. The definition of any name can be viewed simply by pointing at it. The comments about the name are also displayed.

Other programming environments, including conventional text-editing systems, also define an editable unit. Typically, an editor session is restricted to one editable unit, and communication between sessions is difficult. ยก is quite different, since there is no notion of an editable unit. The user may see as much or as little of the program at any one time as desired.

However, the fact that there are no real boundaries in the program's representation does not imply that it visually looks like only a single editing session exists. This view is rejected simply because the entire program cannot be displayed on the screen at once. In order to visually segment the program by its logical components, each definition of a name is usually presented in its own window, called (naturally enough) a definition window. The textual view of the tree being edited is generated by the unparsing mechanism described in Chapter 5. The user manipulates the tree using the commands discussed in Chapter 3; each window possesses its own focus. A definition window is created simply by pointing at a name usage and requesting its definition.

Since, in the course of programming, defining one name may require the examination or definition of many others, the editor must manage many windows
simultaneously, one for each name being viewed or edited.\textsuperscript{1}

In a flat space of names, this view would be sufficient. However, languages are scoped; some names, such as module scopes, have definitions that are themselves lists of name definitions. A window that displays the names in such a list is called a name window; only the names (and perhaps a shorthand view of each name's definition) are displayed. For example, a entry for a procedure appearing in a name window contains the procedure's type, arguments, and result, but its body is not shown. To see the body, a definition window for the procedure must be created.

Many constructors, such as blocks and procedures, have both an associated list of name definitions and a body where those names are known. Since this list is not the definition of some name, it cannot be directly displayed in a name window; however, the user typically wishes to see a list of accessible names while viewing a part of the body. These are displayed in a pane of the window editing the constructor; apart from its visual connection with a specific definition window, the contents of this pane are equivalent to those of a name window. The names so displayed can be just those in the innermost containing scope, or those in a number of ancestor scopes. Mixtures of name lists and the bodies of name definitions never appear in the same window; this is due both to organizational considerations and limitations of the unparsing algorithm.\textsuperscript{2}

Figure 5 shows a typical display as seen by the \texttt{Yg} user. The top two windows display scopes and shorthand views of the names contained in them; the bottom two

\textsuperscript{1}It remains to be seen if conventional window manipulation commands, which allow the hiding, closing, movement, and size alteration of windows, will be adequate for the \texttt{Yg} system. Typically, \texttt{Yg} creates many more windows than other applications, and it may be necessary to automatically manage windows in order to avoid display clutter.

\textsuperscript{2}The unparsers must have a unique \texttt{in} for each statement-level node. Displaying both the name list and body of a node would require two} base.
**Program yg**

<table>
<thead>
<tr>
<th>Module lists</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef list</td>
</tr>
<tr>
<td>list makeList()</td>
</tr>
<tr>
<td>list insert()</td>
</tr>
<tr>
<td>list delete()</td>
</tr>
<tr>
<td>list append()</td>
</tr>
<tr>
<td>list copy()</td>
</tr>
<tr>
<td>node next()</td>
</tr>
<tr>
<td>node prev()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Module trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>typedef tree</td>
</tr>
<tr>
<td>tree makeTree()</td>
</tr>
</tbody>
</table>

**Module lists**

Basic list manipulation routines.

```c
typedef list
typedef struct {
    node *first;
    node *last;
} list;

list makeList(node elem) |
list insert(list l, new) |
list delete(node) |
```

**Function makeList**

Construct a one-element list from the given node.

```c
list makeList(node elem) {
    new = malloc(sizeof list);
    new->first = elem;
    new->last = elem;
    elem->next = nil;
    elem->father = new;

    return new;
}
```

```c
list new;
```

Figure 5: A Yg name display
are definition windows for a type and a function. Note that the function definition window contains an additional pane that contains the declaration for a variable local to that function.
CHAPTER 5

Unparsing

Wherefore are these things hid?
_Twelfth Night_, I, iii

Critical to the notion of directly editing the AST representation of a program is the unparsers for the editor. The unparsers produces a textual view of the tree, guiding the user's manipulations and providing enhanced display capability. Since it is so critical to the user interface, the unparsers needs to be as fast and flexible as possible. This chapter describes the unparsers algorithm, and the unparsing specifications used to build an unparsers for a particular language. The algorithm takes as an argument an arbitrary location within the abstract syntax tree, and unparses the tree around that location. It does only as much work at each redisplay request as is necessary to fill one screen with text. This remains true even in the presence of code elision and other forms of information hiding.

5.1. Previous work

Many previous unparsers, such as the one used in the Cornell Program Synthesizer, have been non-incremental in nature; whenever a change was made to the program, they simply regenerated the text form of the entire program, and then put the desired portion on the screen. While adequate for small programs, or languages where the editable unit is typically small, such an approach is not feasible in $\mathcal{Y}$ since the editable unit might be arbitrarily large.
As part of his work on programming environments for Lisp and PL/I, Mikelsons has developed an incremental algorithm for prettyprinting [Mik81]. His algorithm uses quite a different approach toward code elision than the one presented here; the result is that less user specification is allowed (and less required as well). One phase of his algorithm requires time proportional to the square of the size of the display. Although sufficiently fast for standard 24×80 terminal screens, his algorithm may prove too slow for larger displays.

5.2. Basic philosophy

The unusual goals of $Y$ have necessitated some departures from the design of earlier structured editors. The two major differences which have an impact on unparsing are an absence of structured motion commands and the ability to edit large programs.

The implementors of most of the earlier systems have come to the realization that tree-oriented motion commands can be clumsy. The average user of our system may not even be aware of the tree representation of his program. Because of this, all motion in our editor is text-based; the user points at a spot on the screen and is immediately positioned there, and scrolls by text pages. However, tree modification is still tree-based; commands act on the highlighted focus of attention, which can be a single subtree or a sublist.

Although the use of modularity and small subroutines is a highly desirable goal, it is often unrealized. In addition, even in a highly modular program it is desirable to be able to edit an entire set of routines in the same editing context. Therefore, our system must be efficient enough to handle large editable units, and provides powerful
information-hiding mechanisms in order to make large programs more intelligible.

5.3. Comparison with Lisp

The prototypical unparsing algorithm is of course the PRINT function common to all Lisp systems. Users dissatisfied with PRINT’s output created a large number of “prettyprinting” functions, culminating in Waters’ GPRINT [Wat83]. GPRINT not only contains built-in information about the structure of Lisp, it allows the user to add formatting rules for his own constructs. While it is extremely flexible, its algorithm is rather complicated, and not readily adaptable to an incremental strategy.

Fortunately, the formatting problem for the typical Algol-like language is much simpler than for Lisp. An Algol-like language is characterized by lists of short statements, grouped with blocks and control constructs like if’s and while’s. Expressions tend to be short, and function arguments which are themselves other function invocations tend to be rare. Because of this, a small, rigid set of formatting rules can be identified.

Lisp, on the other hand, is much more varied. Control constructs can be defined by the user, and function invocations are likely to be deeply nested. Because everything is a function call, the “statement” boundary is much fuzzier than in Algol-like languages. For example, contrast a short COND like

\[ \text{(cond ((gc-message) (print "GC"))}) \]

with a much deeper COND like

\[
\text{(COND}
\quad (\text{(null? body) (append1 label-procs}
\quad \quad (\text{(finish-block current Proc (return nil))})
\quad \text{((atom? (car body))})
\text{)}
\]
(build-prog
  (append1 label-procs
    (finish-block current-proc 'go , (car body)))
  (list (list (car body))
    (cdr body)))
(t (build-prog label-procs
  (append1 current-proc (car body)) (cdr body)))

It becomes obvious that no fixed rule for COND is going to result in a pleasing output format in all cases. An Algol-like language is more rigid, and we can exploit that rigidity in the following way.

(Remember that the abstract syntax tree is built from lists of nodes which can themselves have list descendents. We class each type of node as either a statement or a substatement node. Statement nodes are further classed as either simple statements or compound statements. As a special case, compound statement nodes have only one list descendent. An if–then–else construct which appears to have two sublists is instead viewed as a special case of the more general construct if–then–elsif–then–...–elsif–then–else. In this context an if statement has a single list of guarded commands of the form <condition> then <statement list>. In addition, as shown above, each guarded command has a single list of statements. The final else clause is represented as a guarded command whose condition field is true.)

Categorizing nodes in this way allows us to define a node’s unparsed representation on a line basis, according to the following rules. No two statements appear on the same line. Simple statements occupy a single contiguous group of lines. Compound statements occupy two contiguous groups of lines, one for the statement’s opening bracket, and one for its closing bracket. The space between these two groups is occupied by its descendent statements.
As an example, consider the program fragment

```plaintext
sum = 0
i = 1
while (i <= n)
    if a(i) > 0 then
        sum = sum + a(i)
    i = i + 1
print(sum)
```

In order to express our unparsing algorithm more clearly, we use the structured positioning commands `in`, `out`, `next` and `prev`, as defined in Chapter 2. We also define some abstract output functions:\(^1\)

- `displaySimple(simple_node)` generates the text representation for a simple statement.
- `displayOpen(compound_node)` generates the text representation for the opening bracket of a compound statement.
- `displayClose(compound_node)` generates the text representation for the closing bracket of a compound statement.

Armed with these definitions, we now give our first incremental unparsing algorithm in Figure 6. This algorithm is given a statement node in the tree to unpars around, usually the editor's focus of attention. The screen is logically divided into two parts; the bottom part, displaying the part of the program starting with the center statement, is filled by the first while loop. The second loop displays the part of the program before center. The unparsers thus fills the screen starting from the center, going forward to the screen's bottom, and then proceeding backwards from the center.

\(^1\)Later we will describe how to construct these functions from specifications.
display(center)
   // display bottom half of screen, starting with center
   // and going forward.
   node = center
   TopOfList = node
   while node ≠ nil out(node) ≠ nil
       if class(node) = simple
           displaySimple(node)
           node = next(node)
       else if class(node) = compound
           displayOpen(node)
           node = in(node)
           TopOfList = node
       else /* node must be nil */
           node = out(TopOfList);
           displayClose(node)
           TopOfList = node
           node = next(node)

   // display top half of screen, starting with prev(center)
   // and going backward.
   node = prev(center)
   BottomOfList = node
   while ( node ≠ nil out(node) ≠ nil)
       if class(node) = simple
           displaySimple(node)
           node = prev(node)
       else if class(node) = compound
           displayClose(node)
           node = inToEnd(node)
           BottomOfList = node
       else /* node must be nil */
           node = out(BottomOfList);
           displayOpen(node)
           BottomOfList = node
           node = prev(node)

Figure 6: The incremental algorithm skeleton

to the screen's top.²

²This strange display order would be hidden from the user by an optimal screen redrawing
The simple, open, and close display routines can be thought of as producing a contiguous block of lines, which are then glued to the bottom or top edge of the screen area already filled, according to whether the display has been reversed or not. In order to avoid a clumsy test before every call to a display routine, the while loops are implicitly terminated when the appropriate half of the screen is completely filled; the display routines trim any lines which hang off the screen extents.

5.4. Highlighting and Holophrasm

The simple unparsing algorithm is sufficient for regenerating a simple screenful of textual information. We now want to modify that algorithm to do holophrasm, or selective information hiding. We also want to provide the user with visual feedback about the location of the focus, by highlighting any text which is in it.

Holophrasm is mostly automatic, and is based on indenting levels and user-definable parameters. There are several of these parameters.

(1) If the user has placed the focus on a particular statement, we may suppress levels of detail below a certain indenting level. The InLevel parameter is the number of deeper levels, relative to the focus level, to display. Similarly, OutLevel is the number of levels closer to the root from the focus to display. The subtree whose root is OutLevel indenting levels higher in the tree than the focus is called the nearFocus region of the tree. Unless qualifying for display because of some other criterion listed below, nodes outside of the nearFocus region are not shown.

(2) Outside the nearFocus region of the tree, we may want to obscure detail below a particular indenting level. The TopLevel parameter is the level number of the package between the unparsers and the display.
deepest level to display for nodes outside nearFocus.

(3) Because many programs do not exhibit deep nesting levels, there is a mechanism to hide statements within a given list. The SameLevel parameter is the number of statements on either side of the focus to display.

(4) Finally, the user may elect to obscure parts of the program by simply marking them as non-displayable.

For example, the subroutine of Figure 7a is displayed with TopLevel=1,

Figure 7a: A sample program

\[\text{with the IR}^2\text{ Fortran editor, which uses this unparsing algorithm.}\]
OutLevel=1, InLevel=0, and SameLevel=∞ in Figure 7b.

In order to implement this enhanced unparsers, we assume the existence of the following functions on nodes.

- `indentLevel(node)` gives the textual indenting level of the node.

- `inFocus(node)` is a predicate indicating whether the node is in the focus. The focus itself is designated by two tree nodes, `beginFocus` and `endFocus`. If these two nodes are different, then the focus consists of all nodes and their subtrees in the list between the two focus nodes (inclusive). Otherwise, the focus is the subtree rooted at `beginFocus`.

---

**Figure 7b:** The sample program with holophrasm
inSameListAsFocus(node) is a predicate indicating whether node is in the same list as the focus.

onPathToRoot(node) is a predicate that is true if node contains the focus as a descendent.

nearFocus(node) is a predicate that is true if node is in the nearFocus region of the tree, i.e. if it is in the subtree with root OutLevel indenting levels higher in the tree than the focus.

focusDistance(node) for node in the same list as beginFocus is a function returning the number of next's or prev's necessary to take us from node to the closer of beginFocus or endFocus. If node is not in the same list as the focus, then the function value is infinite.

userHidden(node) is true if the node has been explicitly marked by the user to not be displayed. This is simply implemented as a bit in the tree node.

Clearly, since these functions will be invoked frequently by the unparsing process, they must be efficiently implemented, preferably in constant time. Although we might solve this problem by caching information in auxiliary fields in the tree, the fact that we may have multiple active foci in the same tree makes this impossible. Maintaining such fields would also increase the time required to change even a single focus. Focus changing should be a rapid operation, since we expect the user to do it frequently.  

4It may seem initially that at least highlighting can be done without using any of these simple predicates, by checking in the unparsing algorithm to see if each node is equal to begin-Focus or endFocus as it is displayed. Unfortunately, this approach fails, because there is no guarantee that either end of the focus range will be crossed on a particular screen unpars; the ends may be off the screen entirely.
However, the obvious implementations which use no auxiliary tree fields are
typically $O(\text{depth of node})$. For example, consider a naive inFocus:

\[
\text{inFocus}(\text{node}) \equiv \\
\text{while depth(\text{node})} > \text{depth(\text{Focus})} \\
\text{node} = \text{out(\text{node})} \\
\text{if inSameListAsFocus(\text{node}) and} \\
\text{betweenNodes(\text{node, beginFocus, endFocus})} \\
\text{return true} \\
\text{else return false}
\]

where \text{betweenNodes(\text{node1, node2, node3})} is a predicate that is true if \text{node1} is in
between \text{node2} and \text{node3}; i.e. \text{0 or more prevs applied to node1 should get us to node2,}
and \text{0 or more nexts should get us to node3}. This implementation of \text{inFocus}
has complexity $O(d + 1)$, where $d$ is the relative depth of the node and the focus, and $1$
is the length between the start and end of the focus. It is too inefficient to call many
times, as we will want to.

Fortunately, given this information for one node, we can easily compute it
incrementally, in the tree motion functions. For example, after applying a next
operation we can determine the truth value of \text{inFocus} as follows:

\[
\text{nextNode} = \text{next(\text{node})} \\
\text{inFocus(\text{nextNode})} \equiv \text{inFocus(\text{node}) and} \\
\text{node} \neq \text{endFocus or nextNode} = \text{startFocus}
\]

We can use these incremental rules to evaluate all of our functions during the unparse.
We simply use the naive tree searching methods to get the information for the node
around which the unparse will be done, and let the motion functions of the unparsers
update the information for all other nodes rapidly.

Once armed with these functions, we can build a visibility function that tells us,
at each step in the unparse, whether to display the given node or not. This function is
isVisible(node) \equiv
  \text{not userHidden(node) and}
  (\text{onPathToRoot(node)} \text{ or}
  \text{indentLevel(node) < TopLevel or}
  \text{not inSameListAsFocus(node) and}
  \text{in truncated nearFocus subtree or}
  \text{inSameListAsFocus(node) and}
  (\text{inFocus(node) or focusDistance(node) \leq SameLevel}))

where \text{in truncated nearFocus subtree} is defined as:

\text{indentLevel(node) \leq indentLevel(focus) + inLevel and nearFocus(node)}.

In order to add holophrasm to the skeleton unpars (node) to be displayed. If isVisible is false,
then the displayEllipsis procedure is called to print an ellipsis on the screen. In order
to save on checking in higher-level routines, displayEllipsis never prints two ellipses on
consecutive lines. Since we know that if a statement is not visible then none of its
descendents is visible, we can prune our tree traversal when we discover a compound
statement that doesn't get displayed. In Figure 8 we add holophrasm to the forward
direction part of the skeleton algorithm of Figure 6. We can implement the
highlighting of the focus by having the lower-level display routines test the value of
inFocus for the node they are printing. If inFocus is true, the routines display text in
a bold font, so that the focus is visually indicated.

To make isVisible and inFocus constant time operations in the context of the
skeleton unpars, we add more information to a node pointer. For each node, this
expanded node pointer contains the actual AST position, as well as the value of
inFocus, nearFocus, indentLevel, and the distance to the beginning or end of the focus.
Given this information for the node around which the unpars proceeds, we compute
// display bottom half of screen, starting with center and going forward
node = center
TopOfList = node
while node $\neq$ nil  out(node) $\neq$ nil
    if not isVisible(node)
        displayEllipsis()
        node = next(node)
    else if class(node) = simple
        displaySimple(node)
        node = next(node)
    else if class(node) = compound
        displayOpen(node)
        node = in(node)
        TopOfList = node
    else // node must be nil
        node = out(TopOfList)
    if isVisible(node)
        displayClose(node)
    else
        displayEllipsis()
    TopOfList = node
    node = next(node)

Figure 8: Holophrasm in the incremental unparsers

and record new values incrementally in the motion operations in, inToEnd, out, next, and prev.

5.5. The Screen Map

In order to support a text-based user interface, we must maintain a screen map, which is essentially a two-dimensional array which maps coordinates on the screen to positions in the AST. This map is loaded by the display routines as they place formatted text onto the display.
In order to decrease the time required for an unparses even further, we can eliminate the tree search to calculate the initial function values for the center node, by caching this information in the screen map. Since most of the unparses will be done in response to the user clicking on a node as displayed on the screen, the cached information is usually available. If the center node is off the screen, as might occur after a find command, the tree traversal can still be done.

Scrolling forward on the screen is done by simply unparsing around the last node on the screen, with the unparsing beginning on the first line; since this node is always on the screen, its cached information is always available. Scrolling backwards is treated similarly.

5.6. Unparsing specifications

The unparsing algorithm above requires that each node type be tagged as being statement or substatement, and each statement node be further classified as simple or compound. Compound statements have one of their descendents marked to be the result of an in positioning command on that node, and also specify the change of indenting level to occur between the node and its in. All of this information can be gathered from static information in the definitions of the abstract syntax.

The unparsing algorithm makes use of three functions, displayOpen, displayClose, and displaySimple. Collectively we call these the display primitives. These are also built from specifications, but static information may not be sufficient.

Most unparsers have used a "format string" approach, as used in the Gandalf system, DICE [Fri84], and the IR insert editor. The format string is composed of simple commands and requests to recursively unparses subtrees; some simple commands from
Gandalf are

"@n"

Insert a newline in the output.

"@+"

Increase the indenting level, to take effect at the next "@n."

Unfortunately, unless the abstract syntax of the language is inconveniently close to its actual concrete syntax, this approach is inadequate. One of the Gandalf commands calls an implementor-supplied routine, allowing special actions to be performed outside the pure format string framework. IR[^2] provides a number of predicates and a conditional command in the format string, to allow options to be specified without additional code; even so, several special cases have had to be imbedded into the system.

The format string method only seems acceptable for simple unparsing schemes; in order to accommodate more complicated schemes, it must be extended until it becomes a small programming language in its own right. We adopt a hybrid scheme. A display primitive can be specified either with a simple format string, or by a piece of executable code. The code fragment is defined as a method for the display message; methods will be discussed in Chapter 6.

5.6.1. Format strings

Format strings are intended to handle simple cases, such as those found in most expressions. They contain the following commands:
%filename
Invokes the unparsing method of the filename field of the current node.

\n Causes the next output character to appear on the next line, at the current
global indenting level.

\c Displays a "conditional space." A conditional space acts either as a space or a
"\n\", depending on how much space is left on the current line. Conditional
spaces are used to indicate good locations for line breaking.

\r Causes the next output character to appear at column 1 of the current line. Any
output already on the line is lost.

\> Increases the indenting level of the current line by one. Any output already on
the current line is lost.

\< Decreases the indenting level of the current line by one. Any output already on
the current line is lost.

\% Displays a percent sign.
\\ Displays a backslash.

Any other character is simply displayed. The screen map entry for each character
output by a format string always points to the current node.

As an example, consider an assignment statement of the form

    simple statement
    assignment lhs rhs

The displaySimple routine for this node could be simply specified with the format
string
"%lhs := \%rhs\n"

A simple while statement,

   compound statement, indent +1 IN body
   while cond body

might have a displayOpen of

   "while %cond do\n"

and a displayClose of

   "enddo\n"

5.6.2. Unparsing methods

   An unparsing method is simply a method (as described in Chapter 6) answering
the messages displayOpen, displayClose, or displaySimple. Since they are code
fragments, such methods can examine the tree structure using system routines, and
make decisions using ordinary control flow. They produce output by calling the
routine mapf,

   mapf(node, formatString)

where formatString is identical to the format string described above. Every
color character output by the format string maps to node in the screen map.
CHAPTER 6

Static Checking

I have, sir, a son by order of law.

*King Lear*, I, i

The syntactic type mechanism described earlier provides a measure of security that the user’s AST represents a valid program. The name scoping mechanism further guarantees that all names used are declared. Obviously, however, these two mechanisms are insufficient to insure that the program is completely correct. There may still be typing and other “semantic” errors present. This chapter discusses the static semantic checking mechanism in $Y_g$.

As might be expected, unlike the earlier checking provided by syntactic types, semantic checks are not statically specified. Instead, the specification for each node includes a *method* that checks the validity of its sons. The notion of a method is a common feature of all dynamic specifications in $Y_g$, so we begin by explaining it.

6.1. Methods and messages

Associated with each node type in the $Y_g$ system are a number of methods. Like the message-passing paradigm of Smalltalk, methods allow each node to treat a particular action requested of it in a node-specific way. In object-oriented terminology, each node type defines an *object* type, and the node’s fields are the type’s *instance variables*. A method definition looks like:

message-name args...

body
Invocations of other methods may appear in the body as follows:

\[ \text{message-name} > \text{node-expression} \ [\text{with args...}] \]

A method is either invoked by the system framework itself, or by other methods. The invocation mechanism is simple. The receiving node's type is looked up, and that node's specification is examined to see if it has a method of that particular message. If not, a run-time error occurs. Otherwise, the formal parameters of the method are bound to the invoked arguments, and the body is executed.

The body is simply a code fragment of the implementation language. That language might be compiled or interpreted, and we assume that it will be an Algol-like language, perhaps C. A compiled system would use a scheme similar to that of the Unix tools \texttt{yacc} and \texttt{lex}, where the specifications are either fragments that are imbedded into an existing framework, or separate subroutines. In this work, we will use a C-like pseudo-language.

The body may declare local variables may be declared, if desired. The method result is specified for each specific message, and must be a valid returnable type in the implementation language. When the system reads the method specification, it constructs a function header and any necessary local variables, and then inserts the body. The following names are bound in the body:

- \texttt{self} is bound to the node whose method is being invoked.
- \texttt{field1...fieldN} are bound to the objects in the corresponding fields of the of the object being invoked.

The syntactic sugar \texttt{node:fieldName} may be used in place of \texttt{field(node, fieldnumber)}. The system may arrange for a method to be invoked with different
objects bound to certain fieldnames. This implies that `self:fieldname` may not be equal to `fieldname`. This is done so that typechecking on a possible expansion can be performed before the field is actually replaced.

If the actual field number of a field name is required, the syntax `#fieldname` is used.

A method can forward a message sent to it to another node by substituting `forward>fieldname` for the method body. This behaves just as though the sender had sent the message and any arguments to the node specified by `fieldname`.

Note that there are two possible implementation strategies for methods. The first is to build a function in the implementation language for each node's method for a particular message, and a table of such functions for each message, indexed by node type. The appropriate function can be invoked at run-time by the indirect procedure call of C:

```c
/* message>node with args */
result = *(message_table[type(node)])(args, instance_variables);
```

This will incur a large amount of procedure call overhead, however. An alternate formulation which can be used if method invocations are not recursive (that is, a particular method never sends the same message it was invoked with) would involve a simple `switch` statement:

```c
/* message>node with args */
/* assignments to argument and instance variables */
arg1 = val1;
argM = valM;

switch(type(node)) {
  case TYPE1:
```
/* Type1's method for message */
/* may reference arg1...argM */
break;
...
case TYPEN:
    /* TypeN's method for message */
    break;
}

We will be able to use this technique in the eval methods of Chapter 7.

6.2. Type checking

There are a number of user actions that invoke static semantic checking. Name usage is the most common of these. While names can be statically restricted by class (so that, for example, the name of a constant will not be used where the name of a variable is required), class restrictions are not sufficient to enforce other constraints, such as type correctness. In order to maintain this kind of correctness, the specifications provide a check method for nodes.

Note that static semantic checking is somewhat analogous to "type-checking" in compilers. However, it encompasses not only type checking, but any checking that can be performed without execution, not including checking that can be enforced by the simple syntactic rules of the editor alone. Also, a distinction is made between a "syntactic type," as discussed earlier, and a "semantic type." Semantic type is what we typically think of when considering the type of an expression. For the sake of brevity we may use simple term "type" when referring to semantic type.

In general, the modification of a subtree in the AST might result in incorrectness at any point above in the tree, even causing the entire tree to become incorrect. The simple scheme outlined here would be extremely inefficient in such a case, since the entire tree would have to be checked after every modification. However, we will again
exploit a property of Algol-like languages to make the job easier. That is, no error caused by a subtree's contents can be propagated beyond the containing statement. This is a natural consequence of the fact that statements do not have values, and so therefore are typeless (or, alternately, uniformly of type void.) Also, because types in the class of languages being discussed are statically determinable, a change in the type of an expression cannot result in the change of the type of other variables.

Because of this condition, a single static check can result in only as much work as is required to check a single statement's subtree. This is done in the following framework. When a subtree in the AST is entered or modified, the system sends a check message, with no arguments, to the statement level node that contains that subtree. The response to that message should be one of {OK, ERROR}; OK indicates that the node is legal, while ERROR indicates that it is in error. The statement's check method is responsible for checking the entire subtree, which it typically does by invoking the check methods of its sons, and those sons checking deeper nodes recursively until the leaves of the subtree are reached.

In order to check the types of expressions for correctness, the check methods of expressions should return one of {ERROR, UNKNOWN, <type>}. UNKNOWN indicates that the type of the expression cannot be determined, either because some of its sons are incomplete (contain meta nodes) or because there are errors deeper in the tree. If the type is determinable, <type> will be a tree pointer to the type's representation. These types will be used in higher levels of the tree to determine the result type and whether it is valid or not.
In an abstract sense, the checking function for a constructor of arity \( k \) is a function of \( k \) arguments, mapping the check results for its arguments into the values \{OK, ERROR\} for statements and \{UNKNOWN, ERROR, <type>\} for expressions.

6.3. Marking errors

Rather than frustrating the user by refusing to insert an incorrect construct into the tree, we instead mark the incorrect subtree as being in error for later correction. The editor provides a user command for chaining from one error to the next, and highlights incorrect subtrees visually.

There are two approaches we could take in order to mark errors. In the first, the specific checking functions are unconcerned with results of UNKNOWN or ERROR from their subtrees. Instead, the system itself monitors the return values from each check method. If a method returns ERROR, the associated node is marked by the system. We define the checking function as follows:

For expressions:

\[
\text{check}(\text{res1}, ..., \text{ERROR}, ...\text{resN}) = \text{UNKNOWN} \\
\text{check}(\text{res1}, ..., \text{UNKNOWN}, ...\text{resN}) = \text{UNKNOWN}
\]

For statements:

\[
\text{check}(\text{res1}, ..., \text{ERROR}, ...\text{resN}) = \text{OK} \\
\text{check}(\text{res1}, ..., \text{UNKNOWN}, ...\text{resN}) = \text{OK}
\]

In this scheme, the first node whose check method reported ERROR or UNKNOWN would immediately determine the results for every parent node to the statement root, and there would be no point in evaluating their check methods.

This method allows the check methods to be quite simple (they can be partial, taking only arguments of known types, and ignoring errors) but they cause information
to be lost. For example, consider a binary constructor whose type is only dependent on the type of its left son. In spite of the fact that the right son's type may be unknown or in error, the resultant type is still known.

We can provide for both approaches. The system still marks every node whose check method returns ERROR, but a mark primitive is provided for check methods to call. Nodes which wish to mark errors themselves will be able to do so, and will simply never return ERROR. This also allows for the common case where a node is legal in isolation, but inserted in an invalid place. Typically, such an error will not be detected until the parent's check handler has completed, but the node to mark in error is not the parent. In such a case, the offending son will be marked, and the check method will return UNKNOWN rather than ERROR. This makes sure that errors will not cascade up the tree and cause a large number of spuriously illegal nodes.

6.4. Examples

To give an example of this scheme, let us examine the chain of events for the checking of a simple assignment statement. (Suppose our language contains three data types, int, string, and file, and that file variables may not appear as operands of the plus operator.) The assignment node type will be

```
assign lhs:ival rhs:expr
check
  if check>lhs = check>rhs then return OK
  else return ERROR
```

Suppose that the assignment

```
(assign (var a (int)) (+ (const 1) (string "foo")))
```

is entered. The check message, when sent to the left hand side, will yield "(int)," while
the right hand side will invoke plus's method:

```plaintext
plus lh:expr rh:expr
check
  lht = check>lh
  rht = check>rh
  if lht = (file)
    markError lh
    return UNKNOWN
  if rht = (file)
    markError rh
    return UNKNOWN
  if lht = (int) and rht = (int) then return (int)
  else if lht = (string) and rht = (string) then return (string)
  else
    return ERROR
```

Obviously, the `plus` node's check method will detect the type mismatch. Since both `int` and `string` are candidate types for the "plus" operator, neither subtree can be said to be in error. (This points out a deficiency in the mechanism. If the checking at this level knew that the left-hand side of the assignment was of type `int`, then it would be a reasonable assumption to minimize the number of errors by indicating that the string expression was in error. Since our decision is based on purely local information, that is, information flowing into our node from below, we can't detect this fact.) The node marks itself in error and returns a OK.

6.5. Complexity

It is important to examine the algorithmic complexity of our semantic checking scheme. Because of the language properties discussed above, the checking of a single subtree modification only requires work proportional to the number of nodes in the containing statement and its subtree. For nearly all realistic programs, this will be
small.¹

The only modifications that will require more effort occur when a name is altered in type. Because name references are chained, the system can respond to a change by checking only those statement subtrees that contain references to the affected name, avoiding the rechecking of the entire tree. It is hard to see how any scheme could hope to do better than this, without recording substantial information about the types required at each site.

Finally, names can themselves have imbedded name references. When a name is changed, the usage sites of all affected names must be checked. Nodes that define names have check methods themselves, and an illegal name has no type, just as an illegal expression has no type. Names will be discussed in more detail later.

6.6. Types as trees

Like everything else in the AST, types are represented as trees. This means that the type expressions produced by check methods are tree objects, and can be manipulated as such. Types are created and edited syntactically from node specifications just as other nodes are.

For example, consider the "*" or dereferencing operator in C:

deref expr:expr
   check
       t = check>expr
       if nodeType(t) = pointer-to then return field(t, expr)
       else return 0

¹In the static analysis of SAL programs described in [Tan78], only 0.3 percent of all assignment statements had more than 4 terms on the right hand side. A survey of XPL programs [AJW75] yielded an average of 0.76 operators per expression (again, assignment was not considered an operator.)
This method simply checks to see if the expression to be dereferenced is in fact a pointer to some type; if it is, then the resultant type is that type. Otherwise, the dereference is an error.

6.7. Scalar types

There are an infinite number of types in a language like C, Pascal, or Algol 68. They are built up out of type constructors like `array-of`, `pointer-to`, or `record`, from a collection of primitive scalar types.

Primitive scalar types appear as nodes without descendents. These will typically be associated with the system semantic primitives for common scalars such as `shortint`, `longint`, `shortfloat`, `longfloat`, and `char`, but any type can be made scalar as a constructor.

Types may be named. Language-specified named types may be defined as names that reference primitive types or type constructors. For example, the C declaration

```c
typedef *int BLOB;
```

would be represented as

```plaintext
(name "BLOB" (type (pointer-to (int))))
```

while the declaration

```c
struct foo { int a; float b; } bar;
```

would be represented as

```plaintext
(name "bar" (var
  type: (name "foo"
    (struct
      (name "a" (field (longint)))
      (name "b" (field (float)))))
```
where, of course, the edges to the various name nodes are graph edges. A name can be used anywhere a subtree with the name’s value can appear; the name node forwards all messages to its value subtree.

For the purposes of storage allocation, each storable type constructor must answer the sizeof message, which should return the number of bytes needed to store a variable of that type. Typically, the scalar types will return constants, while aggregate types will call the sizeof methods of their descendents recursively. For example,

```c
int /* primitive scalar */

sizeof
    return 4

array-of elem-type:type dimension:name{constant}

sizeof
    return sizeof>elem-type * ceval>dimension
```
CHAPTER 7

Execution Semantics

Speak what terrible language you will; though you understand it not yourselves, no matter; for we must not seem to understand him, unless some one among us whom we must produce for an interpreter.

All's Well That Ends Well, IV, i

We have now described the $Y_f$ facilities that construct a syntactically and static-semantically correct AST. However, simply editing and maintaining the correctness of an AST is only minimally useful in the context of real programming. Without a means to execute the program, the most we can expect from the editor is the elimination of errors traditionally detected by the compiler. By definition, debugging is not concerned with locating such errors. Instead, we want to allow the user to switch between editing and execution rapidly.

Adding a compiler to the system transparently might help, were it not for the fact that any time spent waiting for the system to start executing is damaging to the programmer’s train of thought. The time elapsed between the completion of an editing pass and the start of execution should be as short as possible, and preferably unnoticeable. Many systems relying on existing compilers attempt to provide this kind of speed by making the unit of compilation a single procedure. Unfortunately, procedures are often not small.

Another consideration is the kind of facility provided for the programmer to examine the state of the stopped program. While system tracing of variables is very useful, no prebuilt system can really anticipate the exact form of debugging output
desired by the user. It may be that the programmer himself is not entirely sure what
he wants to examine before the program is stopped; he will more likely make dynamic
decisions about what to examine next as he examines the program's state, each action
motivating the next. In order to facilitate this, we want to give the programmer
access to the full power of the language being debugged, to the point of allowing him
to type expressions and even complete code fragments at will. In order to support this
kind of interface reasonably, the time gap between entering of breakpoint code and its
execution must be very brief.

In order to meet these requirements, جة uses a purely interpretive scheme to
provide debugging support. Each node type in the AST has an associated specification
function which, when executed, produces the actions of the program. There is no
intermediate code of any kind produced from the AST; instead, it can be viewed as
being directly executable by an abstract machine.

7.1. Previous work

Before we attempt to describe our semantic specification method, we should
briefly examine the techniques used in previous attempts to define the meaning of
programs. These earlier methods come from two separate areas: compiler
construction, and denotational semantics. Recently, these two areas have begun to
merge.

7.1.1. Compilers

Specifying the meaning of a syntactic object like the text of a program is the
fundamental problem of compiler construction. A compiler specifies a program's
semantics by translating it from its source form to machine language, which is in turn
defined by a particular piece of hardware. As a desirable by-product of this
translation, the machine code can also be executed.

Most compilers are largely ad hoc tools to perform this translation. In practical
use, ambiguities in the language definition are resolved by compiling sample programs
and either examining the output machine code or actually executing it. The compiler
itself serves as the only defining mechanism.

Unfortunately, the whole purpose of high-level languages is to eliminate
dependencies on particular hardware. Since most compilers are hand-crafted to
produce code for a particular machine, they serve as the definition not for a language,
but only for a pair <language, processor>. If program portability is required, such a
definition is inadequate, since language ambiguities may have been resolved in many
mutually incompatible ways in each compiler.

Another pragmatic concern arises from the observation that compilers can be
difficult to write. Having to write an entirely new version of a compiler for each new
processor is clearly undesirable. This has driven the development of “portable”
compilers.

7.1.1.1. UNCOL

One of the earliest approaches to compiler portability involved the definition of
an “UNCOL” intermediate language [Con58]. As originally conceived, this would be a
low-level language that could be easily translated to machine instructions for a
particular processor. Once a compiler producing UNCOL was built for a language, the
construction of the UNCOL-to-machine-code translator (presumably a task much
simpler than compiler construction) would complete the compiler for a specific machine. This approach in somewhat different guise has been used in a number of systems, such as BCPL [Ric71], Janus [HaW78], and the UCSD Pascal P-system. Several of these systems have even greater portability because they include a portable interpreter that can directly execute the intermediate code.

Despite their apparent success, UNCOL-based systems fail to solve many compiler problems. The UNCOL critically depends on its match with both processor architectures and language design. If the form of the UNCOL is too language-specific, it may be impossible to compile another language into it. On the other hand, if it is too processor-specific, the translation from UNCOL to hardware becomes very difficult. In practice, this mapping seems to be the bottleneck that keeps such systems from generating optimized code.

7.1.1.2. Retargetable compilers

Several "portable" compilers have been built using multiple compilation phases. The "front end" of such a system parses the input program, and produces some intermediate form, like an AST or a linear list of quads. Since no processor dependencies appear in the intermediate form, this phase can be machine-independent. There may also be portable optimizations that can be made at this phase. After the intermediate form is constructed, a "back end" translates it into machine code for the target processor. This approach differs from UNCOL in that the intermediate form may be as closely matched to the source language as necessary.

A well-known system based on this technique is the Unix Portable C compiler [Joh79]. The front end of the PCC produces a mixture of target machine code (for
control flow and storage management) and machine-independent expression trees. The back end then translates the trees into machine code. To build a compiler for a new machine, only the back end (and a small fraction of the front end) must be rewritten. The system has been ported to more than 30 processors.

The basic structure provided by the PCC makes porting quite easy, but it also implies that a number of important optimizations cannot be done; for example, global register allocation is nearly impossible in the framework of the PCC. Also, attempts to use the C back end with other language front ends have resulted in very poor code quality [FeW78].

7.1.1.3. Pattern matching

As long as portable compilers cannot generate high-quality optimized code for a given processor, handcrafted compilers will remain necessary. Apparently, such a compiler requires explicit knowledge of the available instructions of the processor. Therefore, an UNCOL-based compiler is incapable of portably producing good code, since optimal UNCOL code may not be transformable to optimal machine code.

Another portability approach is to build a system that, given a specification for a language, and a specification of a particular processor, produces an optimizing compiler for that <language, processor> pair. Such systems include the CMU Production Quality Compiler Construction project, its derivatives [Cat82] and Graham-Glanville code generators [GHS82].

Cattell's system first transforms the input program into a simple tree form called TCOL. The processor description consists of patterns of TCOL structure and specific machine code to be emitted for each pattern. Code generation then becomes a
pattern-matching problem, looking for optimal correspondences between the input tree and the patterns and associated code. Because the success of such a scheme is dependent on the available patterns, the patterns themselves are generated from a machine description like ISP.

7.1.2. Compiler limitations

Regardless of the techniques used to compile programs to machine code, compilers are difficult to fit into an interactive framework. Most programming environments based on compilation, such as Gandalf and Magpie, recompile on procedure boundaries. An execution request following an editing change results in the recompilation of the changed procedures, as well as any linking that may be required. While this allows the use of existing compilers, the resulting environment is likely to be quite modal, because there is an obvious user-visible point where compilation is done. In fact, Feiler's system [Fei82], which used a standard compiler, could only detect static errors on procedure boundaries, since it relied on the compiler to locate them.

Also, the use of a compiled representation makes it necessary to map from the representation being edited to the compiled code and vice versa, to support debugging. This mapping is often difficult, particularly if the compiler is not designed with the edited representation in mind. Our interpreter avoids this problem by using only a single tree representation for the program.

7.1.3. Denotational semantics

Pragmatic language definition tools like compilers are obviously essential for the mundane purpose of executing programs. However, they do not address other goals,
such as proof of program correctness. Nor have any of the specification techniques we
have examined so far really defined a language in an unambiguous or machine-
independent way. At best, they have raised the problem another level, by phrasing
the meaning of a language construct in terms of a series of machine operations. This
level of concreteness is unsuitable for purposes of reasoning about a program's
meaning.

*Formal or denotational semantics* [Sto77] is a method of describing the meaning of
a program construct, not in terms of its machine code representation, but through
mathematical functions operating on well-defined domains. Such definitions allows us
to apply formal mathematical techniques to prove assertions about program behavior.

A typical denotational definition has several elements [Gor79]:

- A number of *primitive syntactic domains*, such as identifiers, constants, and
  operators.

- A number of *compound syntactic domains*, such as programs, commands, and
  expressions. These are defined by syntactic clauses containing elements of the
  primitive domains and recursive instances of the compound domains. The
  syntactic clauses can be viewed as the definitions of constructors in an abstract
  syntax, each constructor a member of a particular syntactic domain.

- A number of *semantic domains*, such as values, environments, and continuations.

- A number of *semantic functions*, each of which maps some syntactic domain into
  some semantic domain, thus specifying the meaning of the syntactic object. Each
  function is defined in terms of a number of semantic clauses, each specifying the
  meaning of a particular constructor. The clauses contain recursive invocations of
the semantic functions, composed with lambda-calculus expressions and primitive functions.

Initially, denotational semantics was used as a reasoning tool. Nevertheless, a number of systems have been built to take denotational descriptions of languages and interpret or compile them. Examples are PSP [Fau84], SIS [Ple84], Plumb [Set83], and Wand’s system [Wan84].

Such systems use the term “compiler” rather loosely. SIS and PSP both transform programs into lambda-calculus; SIS uses direct reduction to interpret the result, while PSP uses code generated for the SECD machine. Wand’s system produces a series of Scheme functions (which can be viewed as a more expressive lambda-calculus) and relies on a Scheme compiler to translate them into instructions for a virtual Scheme machine. Plumb produces a flow graph, which can then be translated by a back end into actual machine code. None of these systems are comparable in efficiency to even naive conventional compilers,¹ so they are currently viewed as prototyping systems that can be used to “debug” denotational definitions.

There is also a fine line between denotational and operational definitions. This confusion led Stoy to comment on the use of denotational semantics to build interpreters:

Such a way of looking at our method reduces it to just another method of mechanistic definition, with all the problems that such methods bring. This was not our intention. Our “defining language” was not to be regarded as intended for mechanical evaluation, but rather as describing various relationships which hold in some value space. The fact that the expressions of these relationships are susceptible to manipulation according to rules is an uncovenanted extra... [Stoy, page 181]

¹Actually, Plumb is competitive with existing compilers, but relies on the handwritten back end. Also, so far it has only been used for control flow constructs.
We will not be overly concerned about which side of the line our definitions fall on in this thesis.

7.1.4. Lisp interpreters

Most Lisp systems are based on a function called eval, which takes a Lisp expression in S-expression form and evaluates it. This function can be viewed as a semantic definition for the syntax of Lisp. Because Lisp programs are representable as Lisp data objects, eval can be written in Lisp itself, and such a definition is often used as a documentation aid [Ree84, Appendix A] or as a guide to implementing an actual Lisp interpreter.²

The difficulty with a single eval function is that the relationship between a particular abstract syntax constructor and its meaning is often obscured. One of the advantages of denotational definitions is that the definitions are modular; each constructor’s meaning is defined by a separate semantic clause. This is the paradigm we wish to use for \( Y \); each AST constructor will possess its own semantic definition function. We now examine what those functions will look like.

7.2. The form of semantic functions

Given that we have made the decision to use interpretive execution, we need to determine how to build an interpreter from specifications. Following our method of associating a static checking function with each AST constructor, we will define the semantics of each node with a semantic function. There are several approaches we

²In practice, Lisp interpreters rarely have the exact structure of such an idealized eval, because such a definition would be far too inefficient for actual use. For example, the Franz Lisp interpreter [Fos82] has an explicit stack-based evaluator written in C.
could use to define this function.

If the constructor has fields field1...fieldN, we might make the function one of N variables, where the variables were evaluated before the function was invoked. The function would then resemble a Lisp lambda function. For example, consider a plus function:

\[
\text{plus}(lh \ rh) = \\
lh + rh
\]

The system invokes plus by evaluating the lh and rh fields and binding the results to the parameters of the plus function, then executing the body.

This seems to be a reasonable solution for expressions, but consider the example of a hypothetical if function:

\[
\text{if (test then-part else-part)} \\
\quad \text{if test then then-part} \\
\quad \text{else else-part}
\]

Since we have no way to specify that else-part is not executed unless the test is false, we cannot describe the semantics of a conditional. (Even in a side-effect-free language this remains a problem, because of termination properties. If the then part failed to terminate when the condition was false, the if function would not terminate even if the else part did.)

In order to correctly express the semantics of such constructs (which are usually control constructs of some kind) we might define our semantic function so that evaluation is explicit. This is called the fexpr model in Lisp. For example,

\[
\text{plus}(lh, rh) = \\
\text{eval}(lh) + \text{eval}(rh)
\]
and more interestingly

\[
\text{if} (\text{test}, \text{then-part}, \text{else-part}) = \\
\quad \text{if} \ \text{eval} (\text{test}) \ \text{then} \ \text{eval} (\text{then-part}) \\
\quad \text{else} \ \text{eval} (\text{else-part})
\]

This scheme allows us to specify the semantics of control constructs correctly.

Notice, however, that in the \textit{fexpr} model of a semantic function, \text{eval} is an explicitly recursive function; evaluating a particular semantic function results in that function calling \text{eval} recursively on other semantic functions. The control state of the interpreted program is thus inextricably tied up in the control state of the interpreter itself.

To avoid this explicit reliance on a recursive evaluator, we can use a \textit{continuation-based} interpreter. Rather than using recursion to remember where to continue execution after a recursive call, each semantic function is passed another function, called the continuation, which tells it what to do after the current computation is done. For example, consider an if function with no else part:

\[
\text{if} (\text{test}, \text{then-part}, \text{cont}) \\
\quad \text{expr-eval} (\text{test}) \ \text{then} \ \text{eval} (\text{then-part}, \text{cont}) \\
\quad \text{else} \ \text{cont}
\]

The continuation represents the "rest of the program" in some sense; in our case, it can be represented as a pointer to the "next" statement after the if statement. (For simplicity we assume that expressions like the test can be evaluated with no change to the flow of control, so we simply invoke an expression evaluator recursively. Later, we will modify this view.) Note that the last thing the semantic function does is either invoke another semantic function which will invoke the continuation, or invoke the continuation itself. Though these may seem to be recursive calls, they are only tail-
recursive; after eval or the continuation is called, the if function is no longer active.

In many cases, a single function is inadequate for such a formulation. Consider an until function:

\[
\text{until(body, test, cont) =}
\begin{align*}
\text{until1}() &= \\
\text{if not eval-expr(test) then eval(body, until)} \\
\text{else cont} \\
\text{eval(body, until1)}
\end{align*}
\]

We define an auxiliary function, until1, that does the test and passes control to the body. This function is made the continuation on the first execution of the body. This captures the semantics of until, but it is no longer tail-recursive; we must save the value of the continuation across executions. Consider the following evaluation order, where the body is executed once and the test then fails:

\[
\text{until(b, t, c1)} \\
\text{bodySemanticFunction(until1)} \\
\text{until1(c1)} \\
c1...
\]

Until1 is not passed an explicit continuation. Instead, it uses the continuation passed to until. This continuation must be saved before the body is executed.

We can eliminate continuation-saving in most cases by exploiting the structure of the AST. Remember that an AST constructor contains back-links to its parent. This link information can be used to avoid a continuation save. Again consider the until function.

\[
\text{until(body, test, from)} \\
\text{if from = nil then body} \\
\text{if from = body then} \\
\quad \text{if ev-expr(test) then body} \\
\text{else next}
\]
From indicates not the next function to evaluate, but the last function that was evaluated as part of this semantic function. The function starts with a from of nil, indicating that no part of it has been evaluated, and tail-recursively invokes the body. When the body completes, the AST is examined to find the constructor that contains the body, and the until function is again executed, now with a from of body. The function knows that it is now to evaluate the test. If the test is true, the body is again executed. Otherwise, the system calls next, which examines the AST structure and invokes the next statement's semantic function with the correct from.

(Note that we can view tags like body as either tail-recursive invocations of the constructors’ semantic functions, or as returned functions which are invoked by a top-level loop. The latter explicitly converts the tail-recursion to an iteration, and is the approach we will discuss later when we describe the implementation.)

Using this scheme, most of the information formerly maintained by continuation-saving is replaced by the static information in the AST. The only time continuations must be explicitly saved is when there is true recursion in the interpreted program; for example, during a subroutine call. Saving (or recursion) introduced only by nested constructors is eliminated.

As a final observation, note that it is actually not possible to build a purely recursive function like ev-expr to evaluate expressions. This is because expressions may be function calls, and as such, alter the flow of control. We can easily extend functions to treat expression evaluation in the same way as statement evaluation. For example,
until(body, test, from)
  if from = nil then body
  if from = body then test
  if from = test then
    if expr-stack-true then body
    else next

Note that expression results are not made available as parameters, but are left on a stack for later examination.

This definition method also has two important consequences for the implementation of the interpreter. First, the structure of the $\beta$ user interface is simplified, because the interpreter is responsible for its own execution state. This means that the system can be interrupted at any point (say, to await a response from the user) without complex state saving. In other words, the interpreter can be viewed by the user interface as a subroutine rather than a coroutine.\textsuperscript{2}

Secondly, many control constructs are clumsily expressed in a recursive framework. Gotos and catch/throws in Lisp require recursive invocations of the evaluator to return through several levels in order to continue execution at the proper place. Even less disciplined transfers of control, such as a jump into an inactive block (as allowed in C) are even harder to express in an explicitly recursive framework.

7.3. Implementing the specifications

Like those for static semantics, the specification of execution semantics for each node is expressed as a method, this time for the eval message. This method takes one argument, called whence, and returns one result, called next. Whence specifies what triggered the evaluation of this node, while next specifies what the next node to be

\textsuperscript{2}Such a view is required, for example, by the IR\textsuperscript{2} monitor [CuC83].
executed will be. The finest granularity available to in the interpreter is a single eval method execution. We will speak of executing an AST subtree, when in fact what we will do is invoke the eval method attached to the node type of the subtree’s root.

Conventional compiled machine code is linear in structure, with only explicit jumps serving to alter the linear flow of control. Since the AST is a non-linear data structure, every node execution directly specifies where in the tree to continue execution, but there is usually no history kept of what nodes caused the evaluation of any other. Instead, the structure of the AST is used to determine where to continue execution.

For example, suppose that an eval method wishes to evaluate one of its subtrees. A recursive interpreter would simply invoke itself on the appropriate tree node. An interpreter that eliminated recursion with an explicit stack might push the current tree location and continue at the new one. Our interpreter uses neither approach. Instead, an eval method simply specifies the subtree as the node to execute next and returns. After a node is completely evaluated (when its semantic function executes the "default continuation"), the static structure of the AST is examined to determine the next node to be interpreted. The only time an explicit stack is used is when there is true dynamic recursion in the program being executed, such as when a procedure is invoked.⁴

When a subtree has completed its execution, it will most likely designate its father to be the next node to execute. The whence argument to eval is used to

---

⁴The eval methods are not recursive, but it is sometimes necessary to evaluate constant expressions in other methods, such as sizeof or check. A special ceval message simply executes the receiving subtree until it is completed; the system provides for this message itself.
inform a node what previous computations have been completed. It may have any of
the following values:

nil

means that the node is being executed “fresh,” and none of its subtrees have been
executed yet.

sonN

means that the node’s Nth subtree has finished execution. The result of that
execution, if any, is to be found on the expression stack, which will be discussed
later.

out

For a statement node, means that the node’s statement sublist (its IN) has
finished.

aNode

An arbitrary subtree pointer means that that node, which is not a direct
descendant of the node being invoked, has finished execution. This will typically
be found when expression nodes have list descendants.

unwind

Any action associated with leaving the node, such as freeing or marking
variables, is to be performed. Typically, nil is returned.

An eval method’s return value, next, takes on the following values, which indicate the
node to be evaluated next.

nil  For statements, means to follow the linear order of the program and execute the
     “next” statement. For expressions, it means to evaluate the node’s father.
in for compound statements, continue execution at the first statement of the sublist, with a whence of nil.

sonN

continues execution at the Nth son of the node, with a whence of nil.

aNode:

means that execution is to continue at the specified node. If aNode is a statement, this triggers a “goto action” described later. If aNode is an expression, it is executed with a whence of nil. This is typically used by expression nodes that have sublists in order to specify the list evaluation order.

For expressions, aNode should be contained in some subtree of the current node.

To further explain the scheme, consider a typical WHILE construct. It would have an eval method like

```
while condition stmt-list
  nil:
    return condition
condition:
  if expr-stack-true return in
  else return nil
out:
  return condition
unwind:
  // trash any loop variables
```

A typical FOR construct would look like

```
for init test loop stmt-list
  nil:
    return init
init:
  pop // init’s result is ignored
  return test
test:
  if expr-stack-true
```
return in
else
    return nil

loop:
    return test
out:
    return loop
unwind:
    return nil

The GUARD node associated with an IF statement:

    guard condition stmt-list
    nil:
        return condition
    condition:
        if expr-stack-true
            return stmt-list
        else
            return nil
    stmt-list:
        return out

7.4. The interpreter top level

Interpretation is performed by a top-level algorithm that controls the successive eval method invocations. This algorithm appears in Figure 9. Note that the structure of the AST is used to determine the next node to execute if the eval method is completed, i.e., returns nil. In particular, knowledge of the evaluation order of list structure is imbedded in this algorithm. The LIST node is never evaluated, and has no semantic function associated with it. Instead, the interpreter treats lists separately. When a statement node's method returns nil, control passes to the next statement in the statement list, or to the compound statement containing the list.

Expression lists have to be treated differently because of the values they produce. Some list constructs expect these values to be retained, while others throw them away.
toplevel()
node = start; tag = nil
while node ≠ nil
    if type(node) = LIST
        // this has to be an expression list, as a stmt LIST is hidden
        if tag = nil
            node = first(node)
            tag = nil
        // actually do the single evaluation
        next = eval>node with tag
        if nextTag is a node pointer
            tag = nil
            doGoto(node, next)
        else
            switch nextTag
                nil:
                    // execute the "next" node.
                    if stmt(node) and next(node)
                        tag = nil
                        next = next(node)
                    else
                        if stmt(node)
                            next = out(node); tag = out
                        else
                            next = father(node)
                            if type(next) = LIST
                                whichNode = node; next = father(next)
                                tag = N, where node = sonN(next)
                in:
                    next = in(node)
                    tag = nil
                out:
                    next = out(node)
                    tag = out
                sonN:
                    next = sonN(node)
                    tag = nil
                unwind:
                    next = out(node)
                    tag = unwind
            node = next
            // wait for user if necessary

Figure 9: The top-level interpreter
The system allows each node that contains an expression sublist to choose its treatment of results by causing the completion of any expression in the list to invoke the eval method of the list's father with a whence of sonN, where the list is the Nth son of the node. An auxiliary variable, whichNode, contains a pointer to the expression node which completed. This allows a node which may have expression sublists to sequence their execution and perform necessary actions between evaluations.

Expression lists could be handled with a binary operator, like the comma operator in C. It would be defined as

```c
comma lh rh
nil: lh
lh:
    pop() // throw result away.
rh
rh: nil
```

We have not taken this approach because most languages have several constructs that resemble expression lists, but differ in some way. For example, in C the expression

```
sub1(a, b, 3)
```

is a procedure invocation. The parameter list does not contain any comma operators, though it is syntactically similar to expressions that do, such as

```
i = a, b, 3
```

If we forced the user to view parameter lists as different from comma-operator expressions (disallowing list operations like insertion from the latter) then great confusion would result. We prefer to represent all list-like constructs as pure lists. In order to provide the behavior of comma-lists in C, we define a special node to hold the
expression list.

As we mentioned before, if a semantic function indicates that a specific node is to be executed next, then we perform the goto action shown in Figure 10. In essence, we must make sure that all scopes that would be exited when moving from the current position to the goto position are correctly unwound. This definition does not handle the case of jumping into an inactive scope; to handle such a case, the system would unwind out to the first scope that was an ancestor of both the current scope and the target scope, and then activate each scope in turn until reaching the target scope.

7.5. Expressions and values

The eval method mechanism described so far can specify the dynamic control flow for the AST execution. However, it does not describe the results of the execution. Instead, it serves as the "glue" that holds invocations of the system's primitives together, and determines the order in which primitives will be executed.

```
// Unwind scopes until we reach the same scope as that containing
// the goto target.

doGoto(node, target)
    node = out(node)
    while !inSameScope(node, target)
        eval>node with unwind
        node = out(node)
        if node = nil then we tried to goto to
            an inactive scope.
```

Figure 10: The doGoto function
An eval method can be viewed as a function that takes a tag value and returns a new one, which is used to determine the next tree location. This is true regardless of whether the method was for a statement or an expression node. An statement has no other information associated with it, but an expression also produces a value. We have made many oblique references to the values produced by executing expressions. These values are not directly accessible to the semantic definitions; that is, they are not available as variables in the language used to express the semantic functions. Instead, they are held in a separate data structure that represents the program's run-time memory. This memory is segmented into four areas called the stack, static, heap, and temporary areas. The allocation of this space at run-time is described in detail in Chapter 8. For the present, we define primitives that operate on pointers into memory without worrying about how the memory is assigned.

The primitives operate on a stack, called the expression stack. Each primitive is defined to take a certain number of operands from the top of this stack, perform the given operation, and push the result. The elements of the expression stack will typically be pointers to some area in memory, though they may be any tree pointer.

Some of the primitives not only affect the expression stack, but return meaningful values to the method that invoked them as well. This allows methods to make decisions based on the state of the expression stack, typically in order to affect the flow of control in some way. An example is the stack0 primitive used in the example methods earlier in this chapter. This primitive examines the memory pointed to by the top of the expression stack and returns a boolean based on whether the value of that memory is zero or non-zero.
Primitives that produce new memory values rather than modifying existing ones push pointers into the temporary memory space. The temporary space can therefore be viewed as the register set of a machine with an unbounded number of variable-sized registers.

Some of the primitives are listed below. Most of them are relatively low-level, and similar to corresponding machine instructions on conventional processors. The scalars provided are short and long integers, short and long floating-point numbers, pointers,\(^5\) and characters. Unless otherwise specified, the primitives take arguments from the expression stack, and leave their results there. The length of operands is usually implied by the operation.

7.5.1. Comparisons

Comparisons are of the form top-1 \( op \) top. The operations are \(<, \leq, >, \geq, =, \neq\) for operands of type character, short integer, long integer, short float, and long float. Integer values may be either signed or unsigned. Pointers may be tested for equality and inequality only.

A general data comparison function takes a length as an argument, and bytewise compares the contents pointed to by top-1 and top.

These operations return a result directly to the semantic function, popping two descriptors from the stack.

---

\(^5\)Pointers are viewed as untyped; the type they point to is implied by their use.
7.5.2. Arithmetic operations

Binary arithmetic operations are of the form top-1 op top. The operations are
+, -, *, /, and mod for operands of arithmetic type. Additional logical operations and,
or, and xor operate on "integer" values only.

Unary operations are of the form op top. The unary operations are bitwise not
for integer operands, and negate for arithmetic ones.

"Mixed mode" operations take the same form as binary operators. For rightshift
and leftshift, top is a long integer, while top-1 is any integer value. For subPointer
and addPointer, top is a long integer and top-1 is a pointer; the result is a pointer to
the byte offset in the appropriate direction from the pointer.

With the exception of the pointer functions, a new item in temporary space is
created and a descriptor to it is pushed onto the stack.

7.5.3. Conversions

Conversions have the same form as unary operators. They convert short float to
long float, long float to short float, character to short integer, character to long, short
integer to long integer, long float to long int, and short float to long int. They all
create a new item in temporary space and push its descriptor onto the stack.

7.5.4. Byte moving

The general data move interprets top as dest and top-1 as source, and takes an
argument of n bytes to move. It leaves dest on the stack.
7.5.5. Dereferencing

contentsOf() replaces top with the contents of top; top must be a pointer.

7.5.6. I/O primitives

Input/output is a traditional problem area both for specifications and implementations. The Algol family treats I/O in one of two ways:

(1) Input/output *statements* are explicitly provided in the language syntax. This usually means that I/O is done in an operating-system-independent way, enhancing program portability, although nonstandard extensions to the syntax are common. Examples of such languages are Fortran and Pascal.

(2) The language itself does not define any I/O. Instead, a *standard library* of I/O functions is provided. In this paradigm, the I/O functions appear to be ordinary functions of the language, though they frequently must be implemented in assembly language in actuality. At least in principle, a language user could write a completely different I/O package. C and Modula-2 are two languages that use this approach. Unfortunately, since they rely on "standard" but operating-system-dependent packages, such I/O is often nonportable.

Since we define the semantics of syntactic constructions, the first approach is easily accommodated. The second requires us to make any I/O primitives we provide accessible not only from the environment specifications, but from the supported language itself, perhaps by defining global names of type BUILTIN, which are then trapped by a semantic handler that performs the requested function.
The I/O primitives we support provide a simple interface to I/O, based on a small number of *streams* and operations that transfer data from the streams into and out of the program memory areas. In form, this interface rather closely resembles the lowest-level Unix I/O interface [Tho78]. Streams are represented as small integers, and file names are represented as sequences of memory bytes of given lengths.

7.5.7. The procedure call

Much of Yg's advantage over earlier language environment systems is conferred by the fact that Yg defines a number of primitives valid for many languages. One of the most important primitives provided is that for procedure call.

Procedure call is done almost identically for nearly every Algol-like language in existence, and many other languages. The main mechanism is simple. The arguments are evaluated and pushed onto a run-time stack. The parameters in the called procedure are caused to refer to the values on the stack. Reference parameters are passed by address and either implicitly dereferenced, as in Pascal, or explicitly dereferenced as in C. Some way exists for a called function to return a value to the call site.

The invocation primitive handles these actions. A call site evaluates the arguments as desired and pushes the results onto the stack, first argument first. The invocation primitive `call(sub, nArgs)` is then called. Sub is a tree pointer to the first node of the procedure to be executed. nArgs is the number of arguments at the call site. The first nArgs items on the stack are reversed, and call returns sub after saving the current context on an internal control stack. The semantic function should simply return this value.
The inverse of call is provided by return. Return takes the topmost context from the control stack and returns a tree pointer to the call site.

This scheme allows enough freedom for the expression of individual language quirks about evaluation order of parameters, and so forth, while making procedure linkage simple overall.

7.5.8. Constants

Constants are represented as static memory locations, which are initialized before program execution begins. There is no visible difference between a constant and a variable at the system primitive level; the editor prohibits the use of constants at positions in statements and expressions where only variables are allowed.

7.5.9. External representations

Each primitive data type is provided with functions that take character strings and return the actual value, and vice versa. These can be used to build up representation functions for other data types, as necessary.

7.6. Relationship with denotational semantics

We have chosen this specification method for pragmatic reasons; it can easily be interpreted and stepped, and the available primitives allow us to build descriptions for Algol-like languages fairly easily. However, there are obvious similarities between the semantic functions of Yg and denotational semantic definitions. This should not be surprising, since Yg definitions have been “inspired” in large part by work in denotational semantics. We discuss the similarities and differences here.
Consider the following continuation-based denotational definition for a simple, one-argument procedure call [Gor79].

\[
D[\text{fun } I(I_1); \ E]ru = u(p/I) \\
\text{where } p = \lambda ke. E[E[r[e/I_1]]k]
\]

\[
E[E_1(E_2)]rk = E[E_1]r; \text{Fun}; \\
E[E_2]r; f; k
\]

where \( E \) and \( D \) are the semantic functions that map syntactic objects to meanings (\( E \) for expressions, \( D \) for declarations), \( r \) is an environment, \( k \) is an expression continuation, and \( u \) is a declaration continuation.

The first definition contains the semantics of the declaratory syntax that builds a function to be called. A function with name \( I \), formal parameter \( I_1 \), and body \( E \) (a single expression) is defined by binding a lambda function to \( I \) in the current environment. When evaluated, the lambda function (which will take two arguments, the actual parameter's value and an expression continuation) binds the actual to the formal parameter, and invokes the meaning of the body with the resulting environment and the continuation.

The second expression defines the semantics of the procedure call at run-time. \( E_1 \) is evaluated and the result is checked to see if it is a function. If so, then the parameter is evaluated, its result is passed to the function, and finally the expression continuation is invoked.

In comparison, let us examine a \( \lambda \) definition for a similar construct:

\[
\text{function arg:name\{param\} body:expr} \\
\text{nil:} \\
\text{arg:} \\
\text{arg:} \\
\text{body}
\]
body:
return-primitive
return nil

invocation func:expr arg:expr
nil:
  func
func:
  arg
arg:
  return invoke-primitive(func)

These definitions are somewhat equivalent; each defines the semantics of function
definition and invocation in terms of some actions composed with the semantics of the
fields of the constructor. For example, both definitions for invocation first evaluate
the expression being applied to obtain a function object, then evaluate the argument,
invoke the function object with the result, and finally call the initial continuation.

However, denotational definitions are more general than those of $\bar{V}$. As an
example, consider that the denotational definitions, which incorporate static binding of
names, could easily be altered to use dynamic binding instead [Gor79, pages 102-105].
$\bar{V}$ definitions represent a subset of the possible denotational definitions—those that can
specify the semantics of the Algol-like languages. The limitations on what can be
defined include:

- The binding between name definitions and name usages is done statically, at
  "edit-time." No new names can be generated during run-time.

- Function objects are simply pointers to the syntactic definitions of functions. No
  attempt is made, as it is in functional languages like Lisp, to save the values of
  free variables in a function closure.
• Denotable values are all represented as fixed-size pointers; this requires the use of a two-level mapping of the form names→locations→values. (Actually, if all values were of fixed size, this would not be strictly true; but no real Algol-like language has this property.)

• Heavy use is made in the definitions of primitives that directly access a low-level data structure. For example, the definition of invocation above called a primitive called invoke to save the call site location and continue at the invoked function. This is in contrast to the denotational definition, which defined the semantics of the function by constructing a lambda expression in the meta-language.

Denotational definitions almost exclusively use abstract higher-order objects like first-class functions and full continuations. These objects have the advantage that their semantics is easily expressed in a formal functional framework. However, they are extremely inefficient to simulate directly. In contrast, the traditional data structures used by programming language implementations are highly efficient, but their semantics is nearly impossible to define.

There are two approaches to efficiently executing denotational definitions. The first is adopted by Wand’s system; it relies on a compiler for Scheme that can take the abstract objects and translate them, in many instances, into exactly the efficient operations that a low-level compiler might have used. The techniques used in the Scheme compiler are discussed in [Ste78].

The second approach, and the one which we adopt, is to hide the low-level operations as primitives available to the specification writer. Since the primitives can be efficiently executed, the specification can be used directly. This approach has the
danger that too much of the low-level representations used by the primitives will become visible in the specification. This is a risk that we are prepared to take, since our primary aim is to build an efficient interpreter.

7.6.1. Analogies

Several analogies can be made between the structure of $\mathcal{Y}$ definitions and primitives and denotational definitions. These are:

Command continuations

Command continuations are passed and stored in denotational definitions to define control flow in functional terms. A semantic clause which does not alter the flow of control simply finishes by invoking the continuation it was passed. Similarly, the tags returned by a $\mathcal{Y}$ semantic function indicate where execution is to continue; a tag of nil corresponds to invoking the default continuation. However, in $\mathcal{Y}$ many of the continuations are implicit in the AST structure. The default continuation invoked by a nil tag is not actually one passed into the function; instead, it is the parent of the node being executed.

Expression continuations

Expression continuations also allow a value to be passed through the semantic clauses. In $\mathcal{Y}$, the values resulting from one semantic function are saved on the expression stack, where they can be used by later functions.

Environment

As we mentioned before, the binding between name usages and definitions is built into the AST by the editor. This is analogous to the $\mathcal{D}$ function in the denotational definitions above; bindings are made before execution by a separate
declaration semantic function. In $\mathcal{Y}$, this $\text{D}$ function is a basic primitive in the editor.

Store

$\mathcal{Y}$ associates most name definitions with locations in memory. (Exceptions are storable values of tree locations, such as labels and functions.) This association is made at run-time. Each storage-defining construct uses one of the memory primitives to obtain its storage. Since scoping requires these values to be saved, changed, and then restored if a scope is recursively invoked, primitives exist to do this as well. Denotation semantic definitions often have similar primitives, for example, see [Gor79], page 63. They often cannot express the stack nature of allocation, which our primitives explicitly do.

We will say little more about the role of denotational semantics in this thesis. In particular, we will not present a mechanical translation mechanism to produce more concrete $\mathcal{Y}$ definitions from abstract denotational ones, though such translation seems possible.

7.7. Comparison with PECAN

In Chapter 1 we mentioned the PECAN programming environment. PECAN is germane since it is the only existing programming environment that uses semantic specifications for execution. The specifications are descendents of those used in earlier compiler work at Brown; those for the symbol table are described in [Rei83], while the semantic specifications are first discussed in [Rei81]. The system uses a tree representation of the program during editing, but this is transformed by a true incremental compiler to a flow graph representation [Rei84b]. Actions at each node
are specified as a series of invocations of system primitive operators on predefined types.

In contrast to $\mathcal{Y}$, PECAN is a nonprocedural system. The specifications are treated as static descriptions of program behavior, instead of small fragments of executable code as in our eval methods. The PECAN specification is very dependent on the system's predefined framework. The part of $\mathcal{Y}$ that controls the invocation of implementor-defined methods is quite small, while the translation of PECAN specifications to a final environment is likely to be complicated. We contend that $\mathcal{Y}$ specifications are at least as simple as those of PECAN, while $\mathcal{Y}$'s supporting system is much simpler. Any advantage PECAN has is in the area of speed, since it does not rely on the direct execution of the methods of the specification to perform execution. As we saw with compilers, the price of this speed is the difficulty of mapping between many representations.
CHAPTER 8

Runtime Storage

Canst thou not pluck from the memory a rooted sorrow?
Macbeth, V, iii

The AST represents the static portion of a program. The last chapter discussed how eval methods are used to define the control flow and computational aspects of program execution. Control flow was managed by an interpreter that used the AST structure and a control stack to determine the next method to invoke. While primitives to perform computations and leave results on the expression stack were mentioned, most of the details of how the result values were stored were ignored.

Many objects in the AST have associated state not stored in the tree itself. Some examples are variables, parameters, and constants. These actually refer to locations in a memory. Such objects also have eval methods, which must be prepared to produce a memory location when invoked. This chapter describes how the memory used during execution is managed.

8.1. Storage types

In order to know what memory access primitives must be supported, we first examine what styles of memory reference our languages provide. Typical Algol-like languages have the following notions of storage space.

- static, for external variables and statics. This storage is allocated once at the beginning of execution, and is never changed in size or reinitialized.
- stack, for local variables within procedures and blocks. This storage is created on block or procedure entry, possibly initialized, and exists until that invocation is exited. While it continues to exist, it may not be referenced directly by name in a recursive call to the same procedure.

- heap, for programmer-managed dynamic variables. This storage is allocated and initialized by the programmer. It exists until either explicitly freed, or, in some languages, until there are no references anywhere in the program to it (garbage collection).

- procedure space. We assume that "Algol-like" languages do not view procedures as first-class objects in the sense of a language like Scheme. That is, no attempt is made to produce closures or solve the "funarg problem." Instead, storable procedure values are pointers to procedure code. The language allows the programmer to pass procedure pointers as parameters, to store procedure pointers in variables, and to invoke such variables.

- register space. Most compilers rely on the ability of the underlying hardware to store temporary results in registers. Some languages allow direct access to registers; that is, the programmer can specify that particular variables be kept in registers. Such register variables are usually limited in size and accessibility; for example, they cannot be referenced by pointers.

- parameters. Parameters to procedures resemble local variables that are initialized to the values at the call site. They otherwise have the same lifetime as locals. Call-by-reference parameters can be thought of simply as call-by-value parameters that are implicitly dereferenced within the procedure. Alternatively,
all parameters can be viewed as addresses, with value parameters the addresses of temporary storage, while reference parameters point at the storage of the actual passed variables themselves.

8.2. Typical implementation

The above memory access styles have been motivated by their implementation by conventional compilers and typical machine instruction sets. In keeping with the goals of the Algol family, efficiency at run time is the overriding concern, which tends to restrict the kinds of memory access supported. Some of the implementation strategies used in compilers are discussed below.

- Static storage is allocated at compile time, and the generated executable has the storage areas preinitialized. Essentially, the initialization occurs at load time.

- Local storage is allocated on the run-time stack. On block entry, the stack pointer is incremented beyond the local storage, and on block exit, the stack pointer is restored. In languages where the size of a block’s local storage is a compile-time constant, references to locals are made as static offsets from the current stack pointer. Thus, the references are completely static, allowing very efficient access. Variable-sized locals are typically handled with an extra level of indirection, which may require the explicit saving and restoring of the base address of a particular instance of the local variable. In languages with nested procedures, several lexical levels of variables may be accessible, requiring multiple base addresses; these are provided either by a table of active base addresses (a display) or by multiple levels of pointer indirection at run-time (static links).
• Heap storage is allocated in a segment separate from statics and the run-time stack. Typical languages reference all such storage through pointers, and the programmer is required to explicitly allocate and free pieces of heap space. Some languages, like Cedar Mesa, Algol 68, and Algol-W, use an automatic garbage collection strategy similar to Lisp's.

• Procedure pointers are simply addresses in the program's text segment. They are usually invoked by some kind of indirect procedure call instruction provided by the hardware.

• Parameters are pushed on the run-time stack at the call site, and usually referred to as local variables in the called procedure. Variable numbers of arguments can be accommodated by having a separate pointer to the base of the parameter area, called the argument pointer. Often, the number of parameters is explicitly imbedded in the stack frame, which also appears on the run-time stack and holds the previous values of crucial registers such as the stack pointer. The values returned by procedures are either left in specific registers, pushed onto the runtime stack, or left in static areas either in the called function (which prohibits recursion) or in the calling block.

8.3. Interpreter storage allocation

In the environment of the interpreter, many tricks used in compilers are no longer applicable. Since there is no explicit "compile time," allocation and initialization cannot be done separately from execution. (We could simulate "compile-time" by doing a prepass over the tree before execution, but this would probably lead to unacceptable interactive response. We want execution to begin immediately.)
Because there is no separate compilation phase, and because initialization may be dependent on run-time execution, we combine the aspects of allocation and execution. That is, executing a variable may cause it to be allocated or initialized as well. In order to choose appropriate action, each allocatable object contains some information about its allocation state.

8.4. Areas

YG accommodates all these notions of storage by dividing program memory into areas. There are four areas: static, stack, heap, and temporary. These areas are allocated by the system, and are referred to with tree pointers whose tags are STATIC, STACK, HEAP, and TEMP, respectively.

There are no explicit references to values in the memory; instead, all references are via pointers, with the references implicitly indicating the size and type of the value. For example, the constant "1" is denoted by a pointer to a static location with the value 1. At least part of the "descriptors" present on the expression stack are such tree pointers.

Rather than attempt to define the mapping of objects in the AST to storage locations as a fixed part of the system, we provide a set of basic primitives to manipulate memory. The actual mapping is then performed by the eval methods of nodes that define stored objects. These primitives are used by such methods.

newStatic(size) returns a pointer to size bytes from the static space.

newScope(oldbase) indicates that a new scope that will allocate stack storage has been entered. It pushes the old scope base onto the control stack and returns the new base address for that scope, which is simply the current value of the
stack pointer for stack space.

restoreScope() pops the top element of the control stack, restoring the stack pointer to that value and returning it.

newStackOffset(size) allocates a local variable of size bytes, updating the system’s notion of the size of the current scope. It returns an offset from the current scope’s base as returned by newScope. This offset can be used with pushOffset, see below.

newStack(size) also allocates a local variable, but returns the absolute address of the new storage. This would be used if local storage was variable-sized and offsets could not be employed.

saveAddress(addr) saves the given address on the control stack. This can be used by scopes or variables to save their old address for later restoration.

restoreAddress() pops the topmost address from the control stack and returns it.

newHeap(size) allocates size bytes from the heap area, returning its address.

freeHeap(addr) frees the address allocated by newheap. Note that since the size of the storage is not given, the heap storage manager must record such information.

Each of these primitives returns a tree pointer to the specification language. Objects can also be moved from the specification language to the memories, by the primitives below.
pushPtr(tree) pushes the given tree pointer onto the expression stack.

pushOffset(offset) calculates the address pointed to by offset in stack space, using the containing scope’s base, and pushes the result.

pushLong(value) creates a new value in temporary space and pushes a pointer to it onto the expression stack. Similar functions exist for character, short, and short and long floating-point values.

makeTemp(size) creates an uninitialized area of size bytes and pushes a pointer to it.

popPtr() pops the top of the expression stack and returns it as a specification language object. Similar functions exist for character, short, long, and short and long floating-point values.

Several primitives exist to manipulate the expression stack directly. These are:

dup() creates a copy of the descriptor on the top of the stack and pushes it. Note that in all of the stack manipulation functions, only the stack is affected; memory is not.

rotate() "flips" the first three elements of the stack. That is, top ← top-1, top-1 ← top-2, and top-2 ← top, assuming the assignments are done in parallel.\(^1\)

reverse(n) inverts the first n values on the stack. Note that rot() is not equivalent to reverse(3).

\(^1\)The astute reader will notice a similarity between this function and the rot word in Forth-79.
8.5. **Temporary space management**

Unlike the other memory areas, temporary space has no explicit free operation defined for it. Instead, space is automatically reclaimed. There are three potential strategies for doing this.

**flushing**

If clear boundaries could be located when the entire contents of the temporary space became unused, then the space could be emptied on these boundaries. This approach is used for register allocation in naive compilers; registers are marked unused between statements. In the $Yg$ context, this would require a mechanism to save and restore the contents of temp space across procedure calls, and would complicate the passing of information across such boundaries.

**stack allocation**

The second scheme exploits the size information about temp space that can be derived from the pointers on the stack. When a pointer is popped from the stack, its storage is reclaimed. Unfortunately, there can be multiple pointers to the same space on the stack simultaneously, requiring us to check the entire stack for overlaps when we perform a free.

**garbage collection**

A final method uses garbage collection; when the temp space is exhausted, the stack is examined and any space not pointed to is reclaimed. Note that both this scheme and stack allocation may cause fragmentation of the temp space, but because pointers to temp space only appear on the expression stack, those pointers can be arbitrarily readjusted to compact the space.
We will assume the use of some method that frees us from worries about temp space allocation.

8.6. A typical storage scheme

There are no node types with "wired" execution semantics; even nodes associated with storage will define their semantics by using the memory management primitives. However, most Algol-like languages will be able to use similar definitions for such nodes. The definitions given here will serve as a model for a Yg definition of a language with static variables, fixed-size stack variables, and call-by-value parameters.

8.6.1. Variables

Variables in the AST are represented as name nodes with a var node in the value field. Each var node contains several fields that contain information about the variable.

- the addr field is a memory pointer to the name's storage. For static storage, this is an absolute pointer to STATIC space, while for local stack storage and parameters, it is an offset from the containing scope's base.

- the persistence field indicates what kind of memory is to be used for the variable. It may take on the values STATIC or STACK. (Parameters are handled separately.)

- the type field points to a name node that describes the semantic type of the variable.

- the initial field is a subtree that contains a description of the variable's initial value.
As mentioned before, storage is allocated as a consequence of evaluation; declarations of variables are treated as executable. Scope nodes will evaluate their name lists before continuing to the body. Since name nodes forward the eval message to their value subtrees, the method for var performs variable references. The status and addr fields are examined by var's method to determine the correct action to take. The definition of var's eval method will be

```
init:
    move(sizeof>self)
    status = INITIALIZED
    // fall through

nil:
if status = NEW
    switch persistence
    STATIC:
        addr = newStatic(sizeof>type)
    STACK:
        addr = newStackOffset(sizeof>type)
    status = ALLOCATED

if status = ALLOCATED
    // evaluate initializers
    return init

if status = INITIALIZED
    switch persistence
    STATIC:
        pushPtr(addr)
    STACK:
        pushOffset(addr)
```

Variable definitions will only occur as members of the name list of a block node. This node is responsible for the initialization of the scope base, and for the setting of the status fields of local variables when the block is first entered. The eval method for block will be
nil:
    base = newScope(base)
    // set all the status fields for local vars to UNALLOCATED,
    // so they will allocate themselves on evaluation. Then
    // evaluate each in turn.
    return first(names)
names:
    pop
    if next(fromNode) return fromNode
    else return nil
unwind:
    base = restoreScope()

Every entry to the block requires the evaluation of the name list. This is
necessary both to insure that initializations are done, and to allocate the space for the
block. Although the offsets generated for each local variable are always the same,
they must be recalculated so that the system knows the size of the block. Information
about the organization of storage is also recorded during allocation, and is used by the
run-time checks described in Chapter 9. (This is in contrast to a compiled system,
which would do the allocation once, and generate enough code to recreate the actions
efficiently at run time. For example, a typical block entry would simply increment the
stack pointer. We are unable to do this because of the large amount of state we must
build for run-time checking; otherwise, it would be a simple matter for the scope to
record its total size and simply allocate it in one operation.)

Initializations are done in the order that the variables appear in the name list.
Each initialization assumes that all of the variables that it references have already
been allocated. The definitions for languages that allow initializers to contain
references to variables declared in the same scope may have to go to additional effort
to reorder the evaluations, since the editor makes no effort to reorder them itself.
Variables maintain internal state about their allocation status, allocating themselves when necessary, so they can be referenced before their defining scope is even entered. This allows us to allocate statics "lazily," rather than having to allocate all statics at once before execution can begin. Because the initialization of a static variable must be a compile-time constant, there is no problem caused by evaluating the initialization at different times. Since it makes no sense to reference a stack variable outside its definition scope (by export, say), the editor does not allow such references.

Local storage of dynamic size can easily be handled by using the save and restoreAddress primitives to save the offsets of any dynamic variables and restoring them on block exit. In order to preserve the invariance of offsets to the fixed-size locals in the same block, the block method would have to be changed to make two passes through the name list, once for fixed-sized and once for variable-sized objects.

8.6.2. Parameters

In our example, parameters are passed by value and cannot vary in size or number. They are initialized only by the actual parameter expressions at the call site. Because parameters are simpler than variables, they are defined by a separate node.

param addr type

No persistence is specified, as parameters are always local. Similarly, no initialization is allowed. Allocation of parameters is similar to variable allocation, except that the initializers need not be executed; the initial values are already on the expression stack, having been put there by the invoking procedure call. The eval method is
nil:
    if status = UNALLOCATED
        addr = newStackOffset(sizeof>self)
        status = ALLOCATED
        pushOffset(addr)
        swap()
        move(sizeof>self)
        pop()
        pushOffset(addr)

8.6.3. Constants

Constants are simply viewed as static memory locations that are never changed by program execution. Constants appear as names with the class const in the program's outermost scope. (Constants could be defined in the scope where they were referenced, but this would lead to a proliferation of identical constants in several scopes, so constants are treated as globals instead.) A const node resembles a var node, except that the const node has no persistence field; constants are always STATIC.

The init field of a const node must be evaluable before any variable has been initialized; as such, it cannot contain references to var nodes.

If the init field of a constant is null, it denotes a "primitive constant" like an integer, character, or floating point number. In this case, the initialization is done by using system primitives to convert the constant's string into a value. Constants whose values are constant expressions or aggregate constants such as arrays or structures have init fields which generate the constant's value on evaluation.
8.7. Aliasing

Statically-aliased names (such as those defined by FORTRAN equivalence, Ada rename, and various forms of overlay definition) require special treatment. They can be represented by var nodes with persistence ALIAS. The init field should then contain information about which name serves as the base address for the alias. When evaluation is performed the eval method for var can examine the initialization information and set the addr field to point to the correct storage, as defined by the base variable.
CHAPTER 9

The Runtime Environment

'Twill be recorded for a precedent,
and many an error by the same example
will rush into the state.

_The Merchant of Venice_, IV, i

The dynamic specifications formed by `eval` methods allow Yg to execute a
program interpretively. The motivation for this interpretive execution is to support
convenient ways to debug the program, mixing editing with execution and execution
monitoring. This chapter discusses how the basic evaluator interacts with the user
interface and the editor to provide such features.

9.1. Stepping

The top-level evaluator has been carefully designed so that the program can be
stopped after each atomic unit of evaluation. Each cycle of the evaluator merely
checks to see if control should be returned to the user. The two basic conditions that
cause a stop are single-stepping and user-set breakpoints.

Single-stepping can be set to a number of different granularities. At the finest
level that the evaluator is capable of, execution is suspended between each method
invocation. For example, the evaluation of the simple expression “1+2” results in the
following method invocations:

```
> eval > plus with nil
> eval > 1 with nil
> eval > plus with 1
> eval > 2 with nil
> eval > plus with 2
```
This is likely to be as fine as the user will ever want, and frequently it will prove too fine.

A higher grain of evaluation suspends before each invocation of a statement node's method. This has the effect of stopping before each expression evaluation, but not within expression evaluations. For example, consider the evaluation of the assignment "a = 1:"

stop
eval>set with nil
eval>a with nil
stop
eval>set with a
eval>1 with nil
stop
eval>set with 1

Of course, if an expression evaluation involves a statement execution (as in the case of a function call, for example) stops will occur in the function body as well.

The next grain corresponds to conventional statement single-stepping. A stop occurs before each invocation of a statement method, but only if the argument is nil.

Execution can be stopped when particular nodes are invoked. For example, stopping only on function and return nodes stops the program only at function-call boundaries.

Finally, a bit in the nodes themselves can provide for user-set breakpoints.

9.1.1. Actions at breakpoints

When execution has been suspended, control is returned to the user. At this point, the user will usually want to examine the state of the stopped program. In Y9, most such examination will be made by entering and executing small code fragments
of the language being edited. We call these pieces of code *break fragments*. (This is in contrast to most of today's debuggers, which have a command language not related to that of the program being debugged.)

The fragments are executed by saving the stopping tree location and then resuming evaluation at the fragment's body. Since break fragments are subject to breakpoints themselves, each stop saves the stopping location and enters a new *break level*. Successful completion of the current break fragment causes execution to be resumed at the previous break level.

As implied by the name, break fragments make no sense without an associated position in the tree. This is due to the static, lexical nature of name resolution in the Yg AST. Unlike expressions typed at a break in a Lisp environment, where name resolution is all done at run time, names in Yg break fragments must be linked to their definitions.

A break fragment entered during execution is inserted directly after the tree location that caused the breakpoint, in exactly the same way as the fragment would be entered during editing. After the system completes the fragment's execution, it can delete it, retain it in the tree as debugging code, or simply leave it as part of the program. There is otherwise no difference between a break fragment and a piece of code added with ordinary editing operations during the break.

Note that during the break, the user is simply in an editor session that is slightly extended by the ability to enter and run break fragments. All of the editor facilities to examine and modify the static program are still present, with the proviso that some modifications may cause the evaluator to lose its ability to continue the program.
Those modifications are any that would require the program state to be altered in a nontrivial way. This typically means that declarations of active variables may not be altered.

9.1.2. Tracing

Often, the user is not concerned with stopping the program and examining its state. Instead, he only wants visual feedback about where the program is executing. Yô supports a simple kind of visual feedback trivially, by causing the editor to redisplay the program at each step, with the current evaluation location highlighted. This redisplay can be controlled at the same levels of granularity that govern breaks.

Holophrasm in the display interacts nicely with stepping. We can easily determine if a given node is being hidden or not, and suspend execution there only if it is visible. This means that the granularity of stepping can be controlled, not by breakpoints, but by changing the display parameters. Code that isn't displayed is simply not stepped into.

9.1.3. State monitoring

An important facility for debugging is the display of pieces of the program state, such as the current values of variables. A related problem is the suspension of execution when a particular variable is changed. Typical object-code debuggers have to redisplay the values of all variables at each granularity step, since they do not know the relationships between variable names and memory. To stop when a particular variable is modified, they must single-step the program at the granularity of a single instruction. In fact, this may result in slower speeds for compiled systems;
Fieler comments, "LOIPE cannot compete in continuous tracing with interpretive systems like the CPS because of the context switching overhead."

The situation is appreciably better in \( Y \), because the modification of memory is only possible through system memory access primitives. (The same functionality could be supported in object-code debuggers if hardware could trap specific memory references. Typical hardware can only do so in units of memory pages.) This fact allows efficient treatment of variable redisplay and breakpointing, as follows. The user specifies that he is interested in monitoring the values of a set of variables, either by asking for their values to be displayed or setting a breakpoint for their modification. The memory locations of the variable set are stored by the evaluator. When an access or modification of some location in the set occurs, the memory access primitive tells the evaluator to stop, and appropriate action is taken. The mapping of variable names to memory locations is many-to-one, but it is possible to determine from the memory location which variables are affected, and only their values are redisplayed. In a later section we discuss how the memory primitives detect references to the "interesting" areas of memory.

The actual display of variable values has typically been a fixed debugger primitive. Data types were displayed in a canonical form defined by the debugger, which often yielded too little or too much information. (Consider a circularly-linked list viewed first with a debugger that displays pointer values in hexadecimal, and then with one that follows pointers naively.) In the presence of user-defined data types and data abstraction, such displays make little sense. \( Y \) allows the programmer to define how data values will be displayed, through the use of the standard break fragment
mechanism. Associated with each data type and variable is a code fragment that displays the value. When the value is requested, the fragment is simply executed and the results displayed.

This facility is a generalization of the *demon* in Magpie [DMS84]. Because Magpie uses incremental compilation on procedure boundaries, the demon must be a procedure. In Y9, we can make the code associated with a variable reference be any language construct, as long as it is valid at any location in the program where the variable might change. Also, Magpie does not attempt to execute the demons of variables that are modified through indirect references like pointers or aliases.

9.2. Run-time checks

The fact that Y9 is an interpretive system allows us to support an extensive facility for run-time checks. Because each operation is filtered through a method that can perform any checking necessary, the checks can be very comprehensive. We now describe the run-time checks that the system supports directly.

9.2.1. Storage validity

One of the principle sources of error in an Algol-like language is caused by incorrect reference to program memory. Out-of-range array subscripts are the prototypical storage error. The addition of pointers to a language provides an even richer source of hard-to-detect errors, and pointer computation causes still more.

9.2.1.1. Pointers

Pointer errors fall into two different classes:
• The pointer refers to storage that was valid at the time of the pointer's assignment, but which has since been freed (and possibly reallocated). This error can be caused either by setting global pointers to local storage, or by using explicitly-deallocated heap storage. Since heap storage is typically referenced through pointers, pointer errors abound in its use.

• The pointer is manipulated in such a way that a previously valid reference is made invalid. A typical cause of such error occurs in languages, like C, that support pointer arithmetic. A pointer is initialized to the beginning of a region of memory, and incremented past the end of that region, until it points at storage that may be of completely different type.

Although memory management hardware on most of today's processors can be set to produce an error when a nonexistent area of memory is accessed, few errors fall into that category. Such hardware almost invariably cannot detect a reference to logically-unallocated memory, as it is only allocated once by the hardware. Even if such restrictions were not present, the second type of error is much more difficult to detect, since the hardware is unaware of the types of variables in memory. (While tagged architectures can solve this problem for a small subset of types, most languages have an infinite number of types; consider the infinite number of record types definable in Pascal. Some hardware uses descriptors for storing the base and range of aggregates like arrays. Typically, array elements are only accessed via the descriptors, with special instructions. This scheme only works completely in languages with severe restrictions on pointers; [BiB81] contains a discussion of the descriptor implementations on several machines.)
Errors of the first kind can be detected by keeping track of all pointer variables in the executing program. When storage is freed, any pointer that references that storage is destroyed, by setting it to an illegal value. This involves examining the values of all pointers when a piece of memory is deallocated, as would happen either in an explicit free or when a block or procedure was exited. (For the present at least, we ignore the possibility of using dataflow techniques to reduce the number of candidate pointers that must be examined.)

This technique is sufficient to trap pointer errors in languages that do not allow pointer computation. However, illegal pointers generated as a result of pointer arithmetic cannot be detected in this way. In fact, in any language with overlays or pointer type conversion errors can arise that are not detected by the first method.

To detect the second kind of error, memory is logically partitioned by *fences*. The region of memory between two fences can be viewed as a collection of objects of the same type, which are guaranteed by the semantics of the programming language to be contiguous. A fence pair also has associated with it the type and size of the objects contained within it. In order to support this, the memory allocation primitives must be modified to take type and array size information; for example, `newStatic(size, type, number)` would allocate `number` contiguous instances of size `bytes` of the given type.

When a pointer is initialized, it must lie on an object boundary within two fences, and those fences must be of the correct type for the pointer. Thereafter, any pointer arithmetic must leave the pointer on an object boundary within the same two fences. If the pointer leaves the fenced-in area, it is immediately marked invalid. The next
attempt to dereference it will produce a run-time error.

The fence scheme detects the second type of pointer error. Additionally, the first error can be detected by always verifying, at dereference time, that the pointer is correctly within some fenced area. A fence pair is deleted if the storage it fences is freed. (Marking pointers invalid after computation is merely an optimization.)

Obviously, one error will be undetected by this scheme. If a pointer refers to a valid element of storage that is then freed, and a later allocation puts a different variable in the same memory location, a dereference after the second allocation will be seen to be valid, as long as the types match. The only alternative seems to be explicitly scanning all extent pointers of the correct type following a deallocation of storage.

9.2.1.2. Arrays

Arrays are a rather simpler matter, at least at first glance. If the bounds of an array are known, then the subscript operation need only check to see if the subscripts are within bounds. Unfortunately, array bounds are not always directly known. For example, most languages allow arrays to be passed as parameters without specifying the last bound; this makes it impossible to do bounds checking without recourse to some invisible parameter like a dope vector. If we treat arrays as pointers to some block of storage, then the techniques we used to detect pointer errors work as well for arrays. In a language like C, where all array references are identical to pointer computations, such an approach is essential.
9.2.1.3. Aliasing and overlays

Earlier, we alluded to difficulties caused by overlaying and pointer type conversion. In a language like C, where pointers of a particular type can be coerced ("cast") to any other pointer type, invalid pointers can be generated because the storage described by the initial type may be smaller than the storage required by the new type. The semantics may even specify that the overlapping storage be usable, even if trying to use the remaining storage is clearly illegal.

Some coercion errors can be detected statically. Let us consider the problem of casting in C. Suppose the following cast were made:

```
    type1 *l;
    type2 *r;
    *l = (type1 *) r;
```

This cast is certainly valid if type1 and type2 are the same, or are derived from the same base type. It is probably valid if type1 and type2 are the same size, though there may be problems of underlying data organization, such as padding or byte swapping. If type1 is smaller than type2, then at least no storage outside the bounds of the storage pointed to by r will be referenced via l. However, if type1 is larger than type2, some additional storage may be referenced. Such checks can easily be made at "compile time."

The problem with such a static scheme is that it cannot judge what storage will actually look like during execution. The last case mentioned above, which certainly seems statically invalid, is in fact a common occurrence in C programming:

```
    type1 *l;
    char *malloc();
```
l = (type1 *) malloc(sizeof type1);

Here, the standard system function malloc is being used to dynamically allocate a variable of type type1. Malloc has a type of char *, and expects users to cast it to the correct type. However, a pointer in C is not simply a guarantee that one element of the pointer's type is referenced; there are also contiguity restrictions, and these are often very subtle. Put another way, a pointer in C does not contain information about the size of the aggregate it may point to.

9.2.2. Uninitialized storage

Another common error is failure to initialize storage. Several approaches can be taken to detect this problem. First, dataflow analysis can be used to find many common cases where variables cannot or might not be initialized. This is a static technique, and is discussed by Zadeck in [Zad83]. However, many initialization failures cannot be detected statically, so we prefer to concentrate on dynamic detection of uninitialized storage.

There are two methods that can be used to detect such errors. The first tags every memory location with a bit indicating whether that cell has been initialized. The bit is cleared when the associated storage is deallocated, and set when the storage is assigned to. If an attempt is made to reference a piece of storage with a clear bit, an error is signaled. (If all variables were scalars, then the bit could perhaps be associated with the variable. This is not true for aggregates, which can be partially initialized, or for languages where aliasing is possible.)

This scheme is simple, but has the tremendous disadvantage of requiring large amounts of additional storage. An alternate method involves choosing an illegal
values for variables. Variables are initially set to the illegal value, and attempts to access variables that still have this value produce errors. This approach is used in some parts of the PL/I Checkout Compiler [IBM81].

The obvious flaw here is that the illegal value is difficult to find. Imagine trying to choose such a value for integer scalars. One solution is to reduce the allowable range of the integers, in essence reserving one of the data bits for the uninitialized bit. This is a restriction that would not be found were the program conventionally compiled on the target machine, and many overflows might occur in the debugger that would not in the compiled program. An alternate approach, to reserve a single value as meaning "uninitialized," causes similar difficulties.

Rather than use the memory-intensive tag bit scheme, or the overly restrictive illegal value scheme, we can again use the fence mechanism to detect uninitialized variables. Now the fences indicate areas of memory that are not initialized. Initially, the area indicated is all of memory, and as regions are initialized, that area is split, creating more fences. All we need do to detect uninitialized variables is make sure that the resultant memory references are not within any fence.

9.3. Implementing fences

We rely on the same fence mechanism to insure valid storage references, trap uninitialized variables, and display and monitor memory use. We now discuss how to implement fences efficiently. Note that an efficient decision algorithm is essential, since every memory reference will invoke it several times, once for each separate use of the fence mechanism.
The problem can be stated in an idealized way as follows:

Given a set \( F \) of \( m \) elements, each of the form \([i, j]\), give an algorithm to determine which elements of \( F \) a given integer \( k \) falls into. We assume that the maximum number of overlapping ranges is bounded by a constant. Ideally, we would like this algorithm to have time complexity \( O(1) \) and space complexity \( O(m) \). At the time of this writing, we do not know of an algorithm with such bounds.

Note that each fence pair is an element of \( F \). The integers \( i, j, \) and \( k \) are memory addresses and can be represented in a constant but possibly large number of bits \( N \). Therefore, there are \( 2^N \) possible memory locations.

9.3.1. Tag bits

The tag bit scheme mentioned above solves the problem in time \( O(1) \), but unfortunately takes space \( O(2^N) \), and is therefore unacceptable.

9.3.2. Simple hashing

A hash table containing all of the addresses in any range could solve the problem in time \( O(1) \), but would take space \( O(R) \), where \( R \) is the total number of addresses falling within any range. Since \( R \) might well be nearly equal to \( N \), this is also not a suitable solution.

9.3.3. Binary search

A simple sorted table containing the endpoints of all ranges could be stored in space \( O(m) \), as desired. The time complexity of this solution is, of course, \( O(\log m) \).

For some applications, \( m \) would be sufficiently small for this scheme to be effective. However, use of fences to enforce storage validity creates one fence pair for every
contiguous range in the program. This might cause \( m \) to be too large for binary search to be allowable.

9.3.4. A hybrid scheme

The method we adopt is a hybrid of the binary search and the tag bit schemes. We pick some fraction of \( N \), called \( P \), and create a table with \( 2^P \) entries, which we call *buckets*. Each bucket contains either a sorted table of the ranges contained in that segment of the memory space, or an indication that there are no such ranges. Each memory reference is examined by taking its high \( P \) bits and indexing into the table. If there are ranges in the bucket, they are searched using the binary method. Otherwise, there is no applicable range and memory access continues.

This technique has space complexity \( O(m + 2^P) \). The best case time complexity is \( O(1) \), and the worst case is \( O(\log m) \). Assuming an even distribution of ranges, average case complexity is \( O(\log(m/2^P)) = O(\log m - P) \).

Obviously, the choice of \( P \) has a lot to do with how effective this scheme will be. If \( m \) is expected to be small, \( P \) should also be small, perhaps 0. (If \( P = 0 \), the method reduces to straight binary search.) If \( m \) is expected to be roughly the same size as the total number of addresses, \( P \) might be chosen to be about \( N \). (If \( P = N \), the method reduces to the tag bit scheme.)

In practice, we might expect the following values of \( m \). For data monitoring, only a small number of variables are likely to be monitored by the user at any given time, so \( m \) will be quite small, perhaps on the order of 10 or less\(^1\).

\[^1\text{It is interesting to note that one commercial microprocessor, the NS 16032, implements a single fence pair in hardware. This feature has been used to great advantage by the ddt debugger for the machine [SkJ83].}\]
For the purpose of logically segmenting memory to determine the validity of memory references, \( m \) is equal to the total number of distinct active variables in the program. In the degenerate case, where every variable in the program is a scalar and the language makes no guarantees about the placement in memory of adjacently declared scalars, \( m \) might be quite large, and the fence data could easily occupy more storage than the variables themselves. Fortunately, most program memory is used by aggregate structures such as arrays or records, where some contiguity of storage is defined by the language. This radically decreases the ratio of fence storage to actual variable storage.

Finally, the number of fences required to detect uninitialized variables varies rapidly over the course of execution. We have so far said nothing about the time complexity of adding new fences. If an array of \( s \) elements were used to store the range data for a particular bucket, then adding a new element would involve \( O(\log s) \) time to find the insertion point, and \( O(s) \) time to insert it. Typically, variable storage is allocated and immediately initialized. The total number of uninitialized variables at any given point after this "prelude" is hopefully small. This implies that \( m \) will be small except immediately after entry into a block or procedure. Therefore, it might be desirable to defer the entry of fence information until after the "burst" of initializations are over. This is possible if initializations of some variables in a block do not depend on others in the same block. This restriction is often made by the language.
9.4. Runtime display

The interpreter during execution can be viewed by the display system as a user who moves the focus to indicate the next tree construct to be executed. As such, a display tracing the progress of the computation is simple; that focus is redisplayed after each discrete step of the interpreter. This may cause name and definition windows to be created and destroyed, if code that was not being viewed or edited is entered.

The composition of break code is likewise handled by the standard editing interface.

The input and output of the program being debugged must also be displayed. The inability to multiplex between program I/O and debugger I/O has usually been a major stumbling block for debuggers, particularly when used with programs that manipulated the user’s terminal in non-trivial ways (such as screen text editors). The problem has become more pronounced with window systems; debugging an application that wishes to use the entire frame buffer from within a window system is a near impossibility.

To handle output multiplexing, the I/O primitives allow for several classes of output display; each class corresponds to a particular output device. Some of these classes are:

9.4.1. Pads

*Pads* [LaR79] are the lowest common denominator for output. They represent semi-infinite pieces of line printer output, which can be scrolled back and forth by the user. A typical batch application might print its results via such a window.
9.4.2. Terminals A *terminal* is a window that emulates a conventional ASCII terminal with cursor positioning and editing commands imbedded in the output stream. Such a window would be used by an application, like a text editor, that was designed to run on a terminal.

9.4.3. Bitmaps

A *bitmap* is a window that allows the application to manipulate its underlying piece of the full display bitmap directly. Bitmaps could be used by graphics and user interface applications that expected to access the entire bitmap of the workstation.

9.4.4. Files

Scrolls, terminals, and bitmaps are all represent interactive devices, in that they are intended to be visible to the user by the application. Many applications also perform I/O to *disk files*, which are assumed to be hidden from the user; the application's user has no interest in their contents, but a programmer interested in debugging is likely to be very interested.

A *file browser* is an interactive display of a disk file's contents. This might be a hex or a formatted dump of the file's contents, or simply a display of which blocks of the file have been written, and the position of the file pointer. Alternatively, a piece of break code can be executed to display relevant information about the file.

9.4.5. Data displays

In order to generate displays of the current values of program data, we simply invoke pieces of system-defined or user-entered break code. Each changed object causes a separate invocation of the break code for that object. However, the output
from such code appears not in the window associated with the program's output
device, but in a pane of the definition window for the object's name.

9.4.6. Input multiplexing

Some of the output window classes also allow input. Pads allow keyboard input,
while bitmaps and terminals may provide different kinds of mouse input as well. This
input needs to be separated from input intended for $Y_g$ itself. We use the same
mechanism as is used to direct system input to one of the several definition windows
on the screen; this is likely to be influenced in turn by the operating system underlying
$Y_g$.

9.5. Mixing compiled and interpreted code

We have advocated an interpretive strategy so we could insure a high degree of
controllability in the execution process. However, executing an entire program
interpretively is likely to be quite time-consuming. Taking our lead from the Lisp
environment, we want the ability to mix the execution of compiled and interpreted
code. This would allow sections of the program to be independently debugged, and
then compiled for speed.

The Lisp environment has the substantial advantage that their Lisp compilers
were designed in concert with their interpreters. This allows compiled code to contain
hooks to the interpreter. For example, in the Franz Lisp system [Fod82], function calls
in compiled code are not made directly, but through a transfer table. The transfer

---

A preliminary implementation of the top-level interpreter for a simple subset of C can ex-
ecute code at about one-hundredth the speed of compiled code, for simple loops. This is rough-
ly the same speed as the Franz Lisp interpreter.
table contains pairs of the form \(<\text{symbol}, \text{code address}\>\). Other compiled functions have the address of their code in this table, while interpreted functions have the address of an interface routine that invokes the interpreter system. (The interface routine determines the name of the function to be invoked by finding the table entry from the call stack and looking at the name contained therein.) For speed, Franz also arranges for functions that call only each other and are compiled from the same source file to call each other directly, using a much faster calling sequence.

Another important fact is that all access to non-local names is done symbolically. This means that compiled functions do not have actual addresses for non-local names; instead, they must go through the same symbol table used by interpreted code. Only local names are referred to directly by address.

Because of the nature of compiled code, Lisp allows tracing and breakpoints on entry and exit to compiled functions, but not within them. The compiled code is treated as an atomic action by the interpreter, just like a function that could be written in Lisp but which for reasons of efficiency was coded into the Lisp kernel. The rationale is that interpreted functions are used during debugging, and then replaced. Since the interpreted version is still available, switching back and forth is not difficult.

Unfortunately, we may not have such control over the compiler. In fact, we would like to use existing batch compilers and avoid the effort of writing a special-purpose compiler at all. In some ways, this is not difficult, since we have been careful to use the same storage layout as that generated by standard compilers.

---

This "optimization" is what causes most Lisp compilers to treat names as being lexically scoped, while the interpreters use dynamic scoping.
Remember that in order to express pointer aliasing in the supported languages we chose a simple linear memory organization, with all knowledge about that memory in separate data structures. This allows us to execute compiled code by simply linking it such that external data references point to the locations of the memory within the environment. A standard compiler object module contains all the information needed to do this; to the compiled code, the environment is just a large executable it is being linked into.

An additional complication arises, however. Because we have represented locations in one of the four memory areas as tagged offsets, the values of pointer variables in memory are not legal hardware addresses. Compiled code references to external pointers would become confused by this. The only solution seems to be to translate the offsets to actual hardware addresses before storing them in the memory array, and convert them back again in the interpreter's dereference primitive. This action could be taken only when compiled code with pointers was being present in the system, because it might be quite time-consuming. Alternatively, all pointers could be translated before branching to compiled code. The latter approach would be preferable if switches from interpreted to compiled code were infrequent.

9.5.1. Transfers out of compiled code

With the method described so far, a single level of compiled code could be handled. The top-level interpreter could be modified so that a particular kind of node pointer (say, one with tag CODE) would be directly invoked as a subroutine, after pushing the appropriate arguments onto a special run-time stack area. This run-time stack would also be used for local storage by the compiled code. When the compiled
code returned, normal interpretation could be resumed. However, the compiled code would be unable to call back to interpreted code.

We can solve this problem by using the same approach as Franz does; references to interpreted code within compiled code are linked so they call indirectly through a table. The table index is used to determine which interpreted function was meant. The system must copy parameters and return values between the two different representations of the interpreter stack area, and the explicit hardware stack.

9.5.2. Memory safety in compiled code

Compiled code must be executed atomically by the interpreter, since it only has control at the boundary between calls to the code. Because of this, there is no way for the interpreter to determine if the compiled code is accessing only the memory it actually should. In fact, without hardware support, there is no way to keep compiled code from modifying not only out-of-bounds storage in the memory arrays, but the memory used by the system itself. For this reason, the use of compiled code should be viewed with some caution. The only way to solve this problem outside of building a special compiler is to use the memory protection hardware present in most machines to trap illegal references; unfortunately, users often have no access to this hardware level.

For the same reason, the system cannot use the fence mechanism to determine if particular memory values have changed. Because of this, every variable whose value is being displayed has to be reexamined after the completion of a block of compiled code. This problem can also be addressed by data flow techniques, and is discussed in [CKT85].
CHAPTER 10

Textual Considerations

All this I speak in print, for in print I found it.

Two Gentlemen of Verona, I, ii

Despite their advantages, tree-based programming environments still live in a world where most programs are in text form. This chapter discusses two issues: the ways languages have been influenced by their textual representation, and techniques for translating trees to text and vice versa.

10.1. The influence of text on languages

Programming languages have a number of features that are based on the notion of the program being textually represented before compilation. In order to determine their impact on a non-textual system like Y9, we need to discuss how these features fit in a tree-based framework. Among such textual notions are macros, include files, and comments.

10.1.1. Macros

The extent of language macro facilities vary widely. They range from purely textual prepasses, as in the C preprocessor, to syntactic transformers tightly integrated into the evaluation process, as in Lisp.¹ A middle ground can be found in the PL/I preprocessor, which allows macros to define computation and generate

¹Lisp macros are much closer to the form of macros we could support, since they act on internal, “abstract” syntax, not text. Some Lisp systems provide ways to manipulate the textual syntax of the language as well, through a readmacro mechanism. Unfortunately, the great power of Lisp is that the macros may be written in Lisp themselves—a facility unmatched in the Algol family.
syntactic objects. Here the macros are expressed in a small subset of the full PL/I language.

As found in the C preprocessor, macros are simply a parameterized shorthand notation for generating blocks of text. Unlike Lisp macros, no manipulation of the program text is possible, because C programs are not represented as C data objects. (PL/I allows the manipulation of program text in a quite inconvenient way by viewing programs as character strings.) In fact, the C preprocessor allows the generation of incomplete syntactic objects; that is, the fully-expanded macro body might make no sense unless inserted into an existing block of text. For example, consider the following set of C macros:

```c
#define IF if
#define THEN }
#define ELSE } else {
#define ELIF } else if {
#define FI ;
```

It is not clear how to support these in a tree context; they seem too tightly wedded to the notion of the program as text.

Instead, Y  supports a simple macro facility by providing a hardwired macro constructor. Macros are named objects, and are scoped just like any other name. This is in contrast to C macros, that persist through the text file of their definition without regard to language scopes. The macro constructor is

```
macro params:name{param}-list body:tree
```

Body can be any subtree. Once defined, the macro can be invoked at a site by a macrocall node:

---

2Used by S.R. Bourne, an Algol 68 fan, in the "Bourne shell" of Unix.
macrocall params:tree-list

The expansion of a macro occurs at edit time. When the system statically checks a macro node, it expands complete macros by copying the body from the macro definition and replacing all instances of formal parameters with the actual parameter subtrees. (Note that as with the macro body, the actual parameters can be any subtree.) A macro definition cannot contain hard references to free variables; any free variables that were present in the macro body are resolved only when the macro is expanded. The body is then statically checked just as though it had been entered manually. The only part of the user specification that need concern itself with macro calls is the unparsers; static checking and evaluation methods see only the expansion.

The user may choose to see particular macro in either expanded or macro form. If a macro expansion is to be visible, the unparsing framework calls the unparse methods for the expansion directly.

10.1.2. Macros as generics

The C macro facility has often been used to provide a crude form of generic function definition [Str84]. Since the macros are based on textual inclusion, a generic function may be defined as a macro with some of the types in the macro body left as formal parameters; the macro is then invoked with actual type names. A similar although more formal notion of textual replacement forms the basis of Ada generic program units [Dod83, chapter 12].

Y^3's simple macro facility can also be used to build generic routines. A macro invocation can appear anywhere the resulting expansion is legal; therefore, procedures and data can be defined by macro invocation. The parameters of a macro can take on
any subtree value, so it is a simple matter to call a macro with a type subtree, thus defining a specific instance of the generic body.

10.1.3. Include files

An important consequence of separate text-based compilation is the requirement for two forms of textual declaration: definition and reference. For example, witness the C `extern` declaration, which allows the user to declare the type of an external variable in one file even though the definition occurs in another. An outgrowth of this type of declaration are the separate definition and implementation modules found in Modula 2; Ada packages have two parts that also correspond to the Modula constructs.

Because requiring the types of externals to be known in many different text files makes modifications to those types very difficult, include files are used for such declarations. Rather than imbedding the same definition in many files, it would be placed in a single file that would then be included textually into each file which used the declaration. A modification to an include file might necessitate the recompilation of every file that included it.

In a tree-based system that can access every declaration in the program, regardless of its location, include files become extraneous. They are only a mechanism to make up for the fact that although languages have a global scope, text-based compilers do not know the full contents of that scope. Because complete information about all scopes is always accessible in $\mathcal{F}$, a use of some name can immediately check the defining declaration. Also, since references are chained together from the definition, the minimal number of change sites can be determined when a declaration
is changed.

10.1.4. Comments

The single remaining bastion of text editing, even in a tree-based editor, is program annotation, or comments. Until a usable, expressive, and simple specification language becomes available, informal English descriptions of the program's means and ends will remain important.

A large part of the problem of comment management is already solved in $\Psi$, through the use of the name mechanism. This allows any name definition to have an associated block of text. Such text can explain the purpose of a function, the rationale for a data structure, or the uses for a variable. Since naming (and explaining what the names mean) is a large part of programming, we expect this facility to be heavily used. Such comments correspond loosely to the TeX part of a WEB document [Knu83].

The other form of comment simply imbeds text in the executable part of the program. In a text-based system, comments can appear anywhere the lexical analyzer can locate them, typically between tokens. In a tree-based system, a similar effect can be obtained by attaching comment text to any node in the AST.

In practice, however, there seems to be little value to attach comments to most substatement nodes. Unless an extremely clever node space allocation scheme is used, adding a "comment" field to every node will result in large amounts of wasted space.

Also, tree-based comments have been used to support the holophrastic display in the Cornell Program Synthesizer. It uses the notion of "statement comments"—blocks of code which perform the specific action described in the comment. The display can
be put into a mode where only the text of the statement comments is displayed, not
the underlying code.

Statement comments can be provided in $\text{Yg}$ by simply adding a new compound
statement node type, \texttt{comment}, containing a list of statements, and a piece of
comment text. The execution method for this node simply executes its body once.

10.2. Importing and exporting text

A $\text{Yg}$ system with an editor, interpreter, and compiler back end is a self-contained
program development environment. However, the programming world is hardly self-
contained. Programs will already exist in textual form, or will come from the outside
world, and programs developed under $\text{Yg}$ will be distributed to sites less enlightened. If
the compiler back end does not exist, then standard compilers, using textual input, will
have to be used. For these reasons a $\text{Yg}$ environment without the ability to convert
programs to and from their text forms will be of little use.

Parsing input text into an AST representation is an exceedingly well-understood
problem, if there is a direct mapping from the text to the AST form. Similarly,
unparsing trees to text is a simple operation; the techniques of Chapter 5 could be
used immediately, if the AST-to-text mapping were straightforward.

Unfortunately, these mappings are more complicated than those found in
traditional parsing and unparsing. We are instead faced with the problem of mapping
a number of discrete input files, possibly with redundant information, into a single tree
representation for the entire program. The reverse mapping requires us to construct a
number of separate output files from the single program tree. We call these
transformations \texttt{import} and \texttt{export}, respectively.
In addition, the AST representation is augmented with constructs—like virtual scopes, qualified names, name restrictions, and macros—that may not be present in the textual language at all, but only exist as conditions to be maintained by Y. We would like to be able to infer these conditions on import, and approximate them on export.

10.2.1. Importing

We assume that constructing the large majority of the AST from text information, collapsing the information contained in several declarations into a single tree, can be accomplished with a standard LR parser generator such as yacc. We concentrate instead on methods to infer the form of Y-provided constructs.

Building a simple AST, containing all of the scoping information explicit in the original text and no more, might be quite straightforward. We would like to be more sophisticated. Fortunately, many languages have constructs that can provide hints for a possible module structure. Some examples of these are:

Fortran common blocks

A first guess at the module structure of a Fortran program would be to put all users of a named common block into a single module, defining the common block members as data internal to that module.

Source files

The routines and data defined in a single file are good candidates for a single module, particularly in languages like C that have explicit file scopes. Variables

---

*We call any scope which has no direct analog in the language a virtual scope. For example, modules in C might be virtual scopes.*
and functions declared in C as static within files correspond to private names.

Include files

Objects defined in include files are known only to the source files that include them. This defines a second level of modularity, where each source file is a module, and all files with common include files are members of the same "supermodule." Of course, multiple include files may partition the source files in different ways.

User types

Languages with user-definable types often have no way to restrict knowledge about the type's structure to particular routines; for example, a C typedef has to be completely defined anywhere the type is used. However, limited restrictions on user-defined types could be inferred by looking at the kinds of references that such types appear in. If the fields of a record type are never used in a procedure, then that record type is a good candidate to be made limited in the procedure.

Macros

Macros that can be expressed using the $Yg$ macro facility can be built directly into the tree. Those which cannot must be expanded before the tree is constructed.

By providing the users with reasonable approximations to the intended structure of a program, we give them both a useful analysis tool for preexisting code and the means to maintain the program using the features of $Yg$. 
10.2.2. Exporting

Importing is somewhat simplified by the knowledge that the transformation from text to tree need not be exactly what the user wants, since the editor can be used to fix problems. Exporting is more critical, since the exported text must be legal according to the language definition, and the user never sees it. We call this text, unaugmented by YG constructs, vanilla text.

The main problem with generating vanilla text is the representation of virtual scopes. Every name qualified through a virtual scope must be explicitly qualified (with a prefix, say) in the vanilla text. For example, assume a program written in standard C, using modules in YG, as follows:

```c
module lists {
    typedef struct {
        node *first private;
        node *last private;
    } list;

    append(list, new) {
        ...
    }
}
```

module strings {
    typedef char *string public;

    append(s1, s2) {
        ...
    }
}

main() {
    string s1, s2;
    list l1, l2;
    ...
    append(s1, s2); /* overloaded uses of "append" */
    append(l1, l2);
}

The overloaded uses of append were implicitly qualified by the virtual module scopes
by the editor, but as there are no modules in true C, the vanilla text must resemble

typedef struct {
    node *first /*@private*/;
    node *last /*@private*/;
} list;

lists$append(list, new) {
    ...
}

typedef char *string /*@public*/;

strings$append(s1, s2) {
    ...
}

main() {
    string s1, s2;
    list l1, l2 /*@private*/;
    ...
    strings$append(s1, s2);
    lists$append(l1, l2);
}

Note the control comments indicating the Yg constraints inexpressible in the vanilla
language. These are included for documentation purposes and so Yg will be able to
reimport the source text.

One of the features missing from this exportation method is the segmentation of the program into source files. This is not difficult to do simplistically; the source files correspond to virtual scopes like modules, and information shared between source files is represented with include files or simply by duplication of text. For some languages, we can do even better. In C, for example, names declared as static are known only in the defining source file. This allows us to directly encode private restrictions in vanilla text. Using this method, the vanilla program will exactly match what the programmer might have written in many cases. If the language actually has modules, then they can be used directly.

10.3. Specification of import and export

As we have seen, despite the presence of language-independent guidelines, the actions required to do import and export with acceptable tree and text quality are strongly language-dependent. This would seem to preclude their static specification.

We adopt the view that the importer be a separate, hand-written program. Otherwise, we will be forced to consider a morass of parsing questions that we have neatly avoided in the rest of the system. It seems reasonable to use a conventional parser generator to do importation, relying on the tree constructors used by the editor to build the final AST. The static analysis tool described in Chapter 2 indicates that this job might not be overly difficult.

On the other hand, the export function can use the same framework as that used by the unparserr; the unparserr can simply begin at the top of the tree and continue on to the end of the program, ignoring screen size constraints. Rather than calling the
display methods of Chapter 5, separate export methods would be invoked instead. These methods would have to make the decisions about how to segment the program into source files, and how to construct declarations for names with \( \gamma \) restrictions.
CHAPTER 11

Conclusions

O! That a man might know
the end of this day’s business ere it come;

Julius Caesar, V, i

We have described a specification-driven approach to the problem of building an integrated programming environment. Throughout, our system is driven by the notion of program representation by a single intermediate language. This form is the program’s abstract syntax tree. The tree is constructed, displayed, and manipulated with a language-independent framework. To capture notions specific to the source language being supported, each constructor of the abstract syntax is annotated with several functions; the constructors are viewed as objects in an object-oriented system.

The most valuable points we have made are:

- Programming, even in conventional Algol-like languages, will be considerably different when done with a tree-based editor and debugger. Many of the textual features of "modern" languages like Ada may make little sense in such an environment.

- The use of a single tree representation for programs makes many parts of the programming environment much simpler, as a few basic functions can be used in a number of different contexts. Examples are tree manipulation, name management, and program display.

- Existing programming environments for Algol-like languages are too wedded to the notion that programs are textual. Many powerful program manipulation
techniques are ignored since they were difficult or impossible in a textual context.

- Conversely, existing programming languages are rife with features that only make sense in a text-based development environment. Some of these features are impossible to support in a structured system.

- Interpretation can be specified fairly simply, without an extremely complex underlying description language. The style we have borrowed from denotational semantics of defining a language by defining its constructors separately leads to a modular and hopefully readable specification.

- A small number of primitives can be used to describe the execution semantics of a large class of languages.

- The syntactic specification of a wide class of languages can be done statically with a small number of simple tree manipulation functions. There seems little need to use different primitives for specific languages.

- The "continuation-based" interpreter we describe has a simple, general top level. It can describe the semantics of many peculiar constructs that might cause trouble for a recursive interpreter, as well as maintaining its own state.

Of course, the success of a programming environment constructed from $Y$ specifications can only truly be measured by prolonged use in real programming projects. Unfortunately, a complete implementation of a $Y$ environment to be used in such efforts does not exist. We can only hope that our speculations about the nature of programming are correct, and that the features we have included in $Y$ to ease the programming task will really prove useful.
To test the practicality of the $\mathcal{Y}$ specification technique, we have written a complete specification for C and a Pascal-like language; a portion of the latter description is found in Appendix A. Again, a convincing demonstration of the generality of $\mathcal{Y}$ specifications will only come with several specification attempts; but the success of the technique with a language as traditionally difficult to specify as C is encouraging.

11.1. Further work

There are a number of issues we have failed to address in this thesis. We now discuss a subset of them.

11.1.1. Semantic restrictions

We have expressly limited the scope of applicability for our semantic specifications to the class of “Algol-like languages.” Restrictions like lexical scoping and call-by-value or call-by-reference parameter-passing mechanisms seem reasonable, as most Algol-like languages have only exactly those features.$^1$

However, there are a number of language features in members of the Algol family that cannot be expressed in our notation. These include concurrency, full modularity, and dynamic types.

11.1.2. Concurrency

The need for programming language support for concurrency was motivated by the extreme difficulty programmers had with low-level concurrency mechanisms like

---

$^1$It is ironic to note that Algol 60, with its call-by-name semantics, is therefore not “Algol-like.” Algol 68 partially makes up for this.
semaphores. This need has become even greater with the development of systems that either usurp the traditional functions of operating systems, or are imbedded in devices that have none.

The difficulty in providing primitive support for concurrency in Y9 is that no standard mechanism has arisen; instead, it seems as though every language has made a different decision. Ada has the rendezvous, while Mesa and Modula-2 use monitors, and Algol 68 simply provides semaphores as the only primitive. Many other control structures for concurrency also exist.

It would be comparatively simple to provide primitives to extend the Y9 interpreter to concurrent operation; all that is required is a notion of multiple "program counters" and separate control and storage stacks. However, it is not obvious what further set of primitives would be necessary to model a particular style of control, and whether these could be made language-independent.

11.1.3. Full modularity

We have advocated a disciplined subset of the modularity constructs present in languages like Ada, Modula 2, Euclid, and Mesa. Each language possesses idiosyncratic features in this area which we ignore; for example, the extremely complicated overloading rules of Ada. This might make it difficult to import many textual programs with their structure intact.

Because we choose to make naming a primitive of the system, allowing only minor specification, we limit ourselves to what the naming mechanism can handle. This seems to be a minor limitation; in fact, in many cases, the language features could do with some restrictions in the interest of intelligibility [CWW80].
11.1.4. Dynamic types

A few languages have direct support for operations whose exact behavior is unknown at compile-time. An example is the message-passing paradigm of Smalltalk-80 (and Y₅ specifications); the exact method invoked is a function not only of the message to be sent, but the "type" of the receiver at run-time.

A similar mechanism is present in C++. Here, it is called a virtual function, and is trivially implemented (as the methods of Y₅ might be) in terms of a single indirection through a table of functions at run-time. The construct is therefore seen to be a form of syntactic sugaring, easily supported by the Y₅ semantic method for function invocation. The more involved class inheritance of Smalltalk represents an actual form of late binding that we cannot handle, but this is outside the goals of Y₅.

11.2. Compilation

We expect that a complete programming environment will have an integrated compiler that uses the AST directly, avoiding the parsing phase completely. Having the entire program accessible during compilation allows a number of optimizations that cannot be attempted using separate compilation, as proposed by the IRⁿ system.

However, we have not investigated how effective the AST will be as a compiler intermediate form. Many compilers translate to a lower-level form, like quads, before attempting to generate code. It seems likely that a highly-optimizing compiler will attempt some optimizations at the AST level, before translating it to a quad-like form for further optimization. However, most tree forms in compilers use more compact and less structured representations for the tree; for example, the Portable C compiler uses a flat symbol table during code generation.
Note that the task of compilation may become much simpler because of the processing and checking already done by the editor and static checker; type checking and even simple storage layout will have already been accomplished by the time the compiler is run.

11.3. Access and version control

We have largely ignored issues of access and version control in the program being manipulated by the environment. Specifically, we have not built a specific mechanism for multiuser access control into the system; in fact, because of the single tree space we adopt, multiuser access is considerably complicated.

However, a more sophisticated view of storage management, such as that discussed in Section 2.6, may allow us to restrict user access by segmenting the tree into discrete parts on disk or in memory. Similarly, a “module interconnection language” could be specified as virtual scope nodes in the AST; permission lists would have to be incorporated into the static checkers for these nodes. Strategies for such languages are discussed in [Mul84].

11.4. Dataflow methods

Our interpreter run-time support for maintaining valid storage access relies on detecting all violations at run-time. A large number of potential violations may be detectable at edit-time via dataflow analysis of the program, both within and between procedures. Also, many problems such as providing data breakpoints in the presence of aliasing may be simplified by such analysis. This topic is explored in [Com84].
11.5. Undoing and reverse execution

A powerful environment can be confusing. Many systems seek to ameliorate this by providing "undo" commands that can be used to reverse the effects of mistaken actions. In a programming environment, such commands take the form of undoing edit operations on the tree, and reversing execution.

Undoing editing commands seems simple to support; most of the operations are easily reversible. However, keeping every deleted piece of tree structure may be an unreasonable memory burden. [Hoo85] discusses a scheme which can be used to store changes to a tree more efficiently.

Reverse execution is harder. In order to support it, we need to checkpoint the contents of the memory that might be changed by the next execution quantum. Without information to the contrary, this might well include the entire memory space. Again, dataflow analysis seems to provide the answer, by allowing us to determine the minimal amount of memory that might change.

11.6. Text entry

We have assumed throughout that the Ỹ editor was entirely a tree editor; with the exception of names (and the menu shortcuts of Chapter 3) all interactions are via template expansion. However, other editors, such as the CPS and IR^N, switch to a text-based scheme at the expression level. Such a facility could be provided easily by adding precedence information to the expression specifications, and using an operator-precedence parser to build trees from user-typed strings. Alternatively, careful use of mutation operators and focus repositioning might allow tree expression entry using infix-like notation; a similar approach is used in [KaK82].
12. A note on implementation

The $Yg$ system has been designed to be implemented in an Algol-like language; most of our thinking has been in the direction of C. We have made this choice for two reasons. First, while a language like Lisp may seem more appropriate for a system that uses trees heavily, this is not the case; Lisp as it currently exists is incapable of efficiently representing our AST without nullifying many of its advantages, because of the absence of back links in ordinary Lisp lists. Also, today's Lisp implementations are simply incapable of running fast enough on stock workstations to support the highly interactive flavor we demand.

But more importantly, we wish to use an Algol-like language so that the environment can eventually be supported by itself. While the best test of a programming environment is its widespread use, self-maintenance is the second best; few systems today have passed this test, and none exhibit the degree of integration that $Yg$ does. Only when there are such systems will programming environments be full-fledged members of the programming toolbox.
APPENDIX A

A Sample \$g$ Specification

Although we have written a partial \$g$ specification for C, even such a comparatively simple language has a lengthy specification.\footnote{This should not be surprising; despite the primitives available, there are many details to describe in an Algol-like language.} For this reason, we instead present here a specification for an almost-trivial Pascal-like language called Oard [Ken82].

Oard is a simple language with two data types, integer and character. Characters are single bytes; strings are also supported, but only to the extent that they can be printed with the built-in I/O function write. Function results are assigned into a pseudo-variable with the same name and type as the function. Nested procedures are not allowed.

Note that the specification is divided into six parts, for names, types, scopes, statements, expressions, and auxiliary functions.

In the specification, TREE is a reserved type indicating that the corresponding field is a raw tree pointer, which is usually used to store an address or offset. NAME is used as a reserved word to indicate that a name reference is required at that position. For compound statements, IN gives the name of the field positioned to by the in function.

As a measure of this specification’s complexity, the syntax was built in about 15 minutes, while the checking, size, and semantic functions took about 90 minutes.
/
Parameters in Oard are all passed by reference. As usual, the invocation
site has pushed the addresses of the parameters onto the stack. We must
create new locals, and put those addresses in them, then indirect on
each reference.
*/

param type:type loc:TREE
unparse; "parameter %type"

sizeof;
forward type

eval;
nil:
  if status = UNINIT
    loc = newStackOffset(sizeof>POINTER, POINTER)
    pushOffset(loc)
    move(sizeof>POINTER)
    status = INIT
    pushOffset(loc)
    contentsOff()
    return nil

var type:type loc:TREE
unparse; "var %type"

sizeof;
forward type

eval;
nil:
  if status = UNINIT
    loc = newStackOffset(sizeof>type, type)
    status = INIT
    pushOffset(loc)
    return nil

const type:constype loc:TREE
unparse; "const"

check;
  if selfname[0] == '\''
    if strlen(selfname) > 3 type = makeNode(STRING-CONST)
    else type = makeNode(CHAR)
    else type = makeNode(INT)
  
  if type == INT pushInt(atoi(selfname))
  else if type == CHAR pushChar(selfname[1])
  else pushBytes(selfname)

  loc = newStatic(sizeof\>type)
  move(sizeof\>type)

sizeof;
forward type

eval;
  pushPtr(loc)

Types

simple\>type = int char
type = int char array-of

string-const // scalar

sizeof; return strlen(selfname)

int // scalar

unparse; "integer"

sizeof; return 4 // We'll use long integers.

char // scalar

unparse; "character"

sizeof; return 1

range from:NAME{simpleconst} to:NAME{simpleconst}

unparse; "%from..%to"

array-of type:simple\>type dim:range
unparse; "array[%range] of %type"

sizeof; return dim * (atoi(dim:to) - atoi(dim:from) + 1)

Scopes

program decls:NAME{var,procedure,function}-list body:stmt-list

unparse; compound indent +1 IN body
open "program\n"
close "end\n"

eval;
nil:
    base = newScope()
    // set all variables to status of UNALLOCATED, then
    // evaluate just the variables.
    return first(decls)

decls:
    popPtr()
    while next(fromNode) && type(fromNode) != VAR fromNode = next(fromNode)
    if fromNode != nil return fromNode
    else return body

body:
    return nil // all done.

function params:NAME{param}-list body:block type:simple type loc:TREE

unparse; compound indent +1 IN body
open "function (%params): %type\n"
close "end\n"

sizeof;
forward type

eval;
nil:
    if invocation()
        base = newScope()
        loc = newStackOffset(sizeof&type)
        // set all parameters to status of UNALLOCATED, then
        // evaluate them.
        makeUnallocated(params)
        return first(params)
    else
        pushOffset(loc)
        return nil

params:
    popPtr()
    if next(fromNode) return next(fromNode)
else return body

body:
    // make temporary and save value into it
    pushOffset(loc)
    makeTemp(sizeof(type))
    move(sizeof(type))
    base = restoreScope()
    return returnPrimitive()

procedure params:NAME{param}-list body:stmt-list

unparse; compound indent +1 IN body
open "procedure (%params)
    close "end"

eval;
nil:
    base = newScope()
    // set all parameters to status of UNALLOCATED, then
    // evaluate them.
    makeUnallocated(params)
    return first(params)

params:
    if next(fromNode) return next(fromNode)
    else return body

body:
    base = restoreScope()
    return returnPrimitive()

block names:NAME{var}-list body:stmt-list

unparse; compound indent 0 IN body
open "begin"
    close "end"

eval;
nil:
    base = newScope()
    // set all locals to status of UNALLOCATED, then
    // evaluate them.
    makeUnallocated(names)
    return first(names)

params:
    if next(fromNode) return next(fromNode)
    else return body
body:
    base = restoreScope()
    return nil

Statements
stmt = assign if while case read write

assign lh:val rh:expr

unparse; simple "%lh := %rh

check;
if check>lh = check>rh return OK
else return ERROR

if arms:guard-pair

unparse; compound indent 0 IN arms open "" close ""

eval;
nil: in
in: nil

guard cond:expr body:stmt-list

unparse; compound indent +1 IN body
open
    if firstInList(self) mapf(self, "if %cond\n")
    else mapf("else\n")
close ""

eval;
nil:
cond
cond:
    if 0stack0(check>cond) then in
    else nil
out:
    return unwind // need to skip rest of guards

while cond:expr body:stmt-list

unparse; compound indent +1 IN body open "while %cond do\n" close ""

eval;
nil: cond
cond:
    if !stack0(check>cond) then in
    else nil
out:
    cond

```case expr:expr arms:case-elem-list
```

unparse; compound indent +1 IN arms
open "case %cond of\n" close "end\n"

eval;
nil:
    return expr
expr:
    return IN
nil:
    popPtr() // flush the case expression

```case-elem labels:NAME{const}-list body:stmt-list```

unparse; compound indent +1 IN body
open "%labels\n"
close ""
eval;
nil:
    // the stack has the case expression on top
    return first(labels)

labels:
    dup
    if check>fromNode = INT
        if intEq() return body
    else // has to be character
        if charEq() return body
    // this label didn't match...
    if next(fromNode) return next(fromNode) // try next label
    else return nil

body:
    return nil

// For brevity, the semantics of I/O is omitted.

```write expr:NAME{var,const,string-const}```

unparse; simple "write %expr\n"
read lval:lval

unparse; simple "read %expr\n"

Expressions

// For brevity, many operators are omitted.

eval = NAME{var} NAME{const} NAME{function} subscript plus succ invoke
lval = NAME{var} NAME{function} subscript

invoke sub:NAME{procedure, function} args:expr-list

unparse; "%sub(%args)"

eval;
nil:
// push args onto expr stack first to last
return first(args)

args:
/*
   if this was an expression (ie, in temp space) we need to make
   new local and initialize it, then pass its address.
*/
if tag(stackTop()) = TEMP
  pushPtr(newLocal(sizeof>(check>nodeInList)))
  dup
  move

// do the next arg if there is one
if next(nodeInList) != nil return next(nodeInList)

/*
   We're ready to invoke routine...
   The arguments are on the stack, last on top, and converted to
   canonical type.
*/
pushPtr(sub)
nargs = length(args)
return callPrimitive(nargs)

subscript base:NAME{var} sub:expr

unparse; "%base[%sub]"

check;
if check>base = array-of X then return X
else
  markError(base)
return UNKNOWN

eval;
nil:
    return base
base:
    return expr
expr:
    pushInt(sizeof(base)
multInt()
addLongToPointer()
    return nil

plus lh:expr rh:expr

unparse; "%lh+%rh"

check;
if check>lh != INT markError(lh)
if check>rh != INT markError(rh)
return INT

eval;
nil:
    return lh
lh:
    return rh
rh:
    addLong()
    return nil

succ expr:expr

unparse; "succ(%expr)"

check;
if check>expr = INT or check>expr = CHAR return check>expr
else return ERROR

eval;
nil:
    return expr
expr:
    if check>expr = INT
        pushLong(1)
        addLong()
    else
        pushChar(1)
        addChar()
    return nil
Auxiliary functions

stack0() {
    pushLong(0)
    return equalLong()
}

treePtr stackTop() {
    treePtr p = popPtr()
    pushPtr(p)
    return p
}
BIBLIOGRAPHY


