COMPILING FROM THE
GYPSY VERIFICATION ENVIRONMENT

Lawrence Mark Smith

ICSCA-CMP-20 August, 1980

Institute for Computing Science and Computer Applications
Certifiable Minicomputer Project
The University of Texas at Austin
Austin, Texas 78712
# TABLE OF CONTENTS

## SECTION 1 Introduction

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Background</td>
<td>5</td>
</tr>
<tr>
<td>1.1.1 Programming Languages</td>
<td>6</td>
</tr>
<tr>
<td>1.1.2 Compiling</td>
<td>7</td>
</tr>
<tr>
<td>1.1.3 Code Generation</td>
<td>8</td>
</tr>
<tr>
<td>1.1.4 Compiling into Another High Level Language</td>
<td>8</td>
</tr>
<tr>
<td>1.1.5 Attribute Grammars</td>
<td>9</td>
</tr>
<tr>
<td>1.2 The Gypsy Verification Environment</td>
<td>10</td>
</tr>
<tr>
<td>1.3 The New Gypsy Code Generator</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Acknowledgements</td>
<td>13</td>
</tr>
</tbody>
</table>

## SECTION 2 Gypsy, Prefix, and Bliss

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 An Overview of the Gypsy Programming Language</td>
<td>14</td>
</tr>
<tr>
<td>2.1.1 Flow of Control</td>
<td>14</td>
</tr>
<tr>
<td>2.1.2 Data Structures</td>
<td>16</td>
</tr>
<tr>
<td>2.1.3 Condition Handling</td>
<td>17</td>
</tr>
<tr>
<td>2.1.4 Concurrency</td>
<td>18</td>
</tr>
<tr>
<td>2.1.5 Programming Style</td>
<td>20</td>
</tr>
<tr>
<td>2.2 Prefix Vs. Abstract Syntax</td>
<td>20</td>
</tr>
<tr>
<td>2.3 An Overview of the Bliss Programming Language</td>
<td>21</td>
</tr>
<tr>
<td>2.3.1 Flow of Control</td>
<td>23</td>
</tr>
<tr>
<td>2.3.2 Data Structures</td>
<td>23</td>
</tr>
<tr>
<td>2.3.3 Language Extension</td>
<td>24</td>
</tr>
<tr>
<td>2.3.4 Condition Handling in Bliss</td>
<td>24</td>
</tr>
<tr>
<td>2.3.5 Bliss Coroutines</td>
<td>26</td>
</tr>
<tr>
<td>2.4 Bliss as Target</td>
<td>26</td>
</tr>
</tbody>
</table>

## SECTION 3 The Bliss Image of Gypsy Constructs

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Basic Control Structures</td>
<td>28</td>
</tr>
<tr>
<td>3.2 Condition Handling</td>
<td>28</td>
</tr>
<tr>
<td>3.2.1 Bliss Image of Gypsy Condition Handling</td>
<td>30</td>
</tr>
<tr>
<td>3.2.2 Initial Implementation of When Parts</td>
<td>30</td>
</tr>
<tr>
<td>3.2.3 Handling Gypsy Cond Parameters in a Bliss Image</td>
<td>31</td>
</tr>
<tr>
<td>3.2.4 Initial Implementation of Condition Parameters</td>
<td>33</td>
</tr>
<tr>
<td>3.3 Gypsy Data Declarations in Bliss</td>
<td>33</td>
</tr>
<tr>
<td>3.3.1 Allocation of Gypsy Locals</td>
<td>33</td>
</tr>
<tr>
<td>3.3.2 Gypsy Parameters</td>
<td>34</td>
</tr>
<tr>
<td>3.3.3 Descriptors for Variables</td>
<td>35</td>
</tr>
</tbody>
</table>
ABSTRACT

Gypsy is a programming language designed to support formal specification and verification of software. The Gypsy Verification Environment brings together a set of tools to aid the programmer to incrementally specify, design, verify and implement programs.

The subject of this report is the design and initial implementation of a Gypsy translator. The translator is a LISP module integrated into the Gypsy Verification Environment. It will translate the internal form of a Gypsy program into Bliss.

The Bliss images of Gypsy language features are outlined, and the table-driven translation algorithm is discussed.
CHAPTER 1
INTRODUCTION

Gipsy [22] is a programming language based on Pascal [38] which was designed to support program specification and verification. The "units" of a Gipsy program define constants, types, procedures, and functions, as in most Pascal-like languages, and lemmas, which are used in proofs.

A Gipsy programmer annotates the program with specifications about the state of program variables and parameters. A program is verified by proving mathematically that the executable code meets the specifications, and by validating the specifications at run time. Thus verified programs are more trustworthy than traditionally developed programs.

The purpose of this project is to implement a Gipsy compiler by translating Gipsy into Bliss. This translator is being integrated into the Gipsy verification Environment. The translator generates a file containing the Bliss image of a Gipsy program, which is then processed by a Bliss compiler.

The translator is being developed in several phases. This report describes two aspects of the total project. The first is the design of runtime structures as implemented in Bliss. Emphasis is placed on the subset of Gipsy that is included in the first phase of translator implementation. The second aspect described here is the table-driven algorithm on which the translator is based.

This introduction provides background information on compiling, a description of the Gipsy Verification Environment, and an overview of the translation process from Gipsy to Bliss. The second chapter presents an overview of Gipsy and its internal form, and Bliss. Chapter three discusses individual features of Gipsy and their translation into Bliss. For each feature, problems and choices are discussed, then possible implementations are outlined. For those features in the subset included in the initial version of the translator, the chosen implementation is described. The fourth chapter discusses features of Gipsy not included in the initial implementation, that will be added later. The major areas discussed in chapters three and four are control flow, condition handling, data storage, dynamic memory, and concurrency. Chapter five describes the table-driven algorithm on which the translator is based.
1.1 Background

1.1.1 Programming Languages

Each different brand and model of computer has a different set of primitive capabilities wired into it. This set of instructions is termed the "machine language" for that particular computer. Typical machine instructions include fetching data from memory, adding or subtracting two numbers, and skipping other instructions if a data value is zero. The first computers were programmed by entering the instructions into a large array of switches - on for a binary 1, off for a zero. It was a very time consuming and error prone procedure.

The next step in the development of programming languages was the development of "assembly languages." The assembly language programmer is still required to know the primitive instruction set of the computer, but needs not remember the binary codes that make the machine do the operation. For example, to make the computer add two numbers together, the programmer can say

ADD x,y

instead of

110 011 111 011 111.

A computer program called an assembler translates the assembly language to machine language. The input data for the assembler program is the text of another program written in assembly language. The output of the assembler is machine language.

Most modern programs are written in "high level" programming languages. Each instruction in a high level language may cause up to thousands of machine instructions to be executed. Programming languages can be classified in terms of "level" roughly according to how much machine language corresponds to a statement in the language. LISP is a very high level language, Gypsy is a high level language, and SLS is a low level compiled language.

The process of translating the human-readable form of a high level language into machine language is termed "compiling," and the program which compiles is a compiler. If a compiler running on one computer produces the machine language for a different computer, it is called a cross-compiler.

These high level languages are machine independent; that is, they are designed for an application area, rather than for a
particular machine. For example, FORTRAN is used for scientific programming involving complex numerical computations, COBOL and RPG are used in business, SMPOL is used in text processing, LISP deals with lists and hierarchies of data, and Bliss is used in "systems programming," such as writing assemblers and compilers.

1.1.2 Compiling

A compiler is a large, complex program. There is a very large body of literature discussing issues in compilation, from the original language design to the runtime structures of the compiled program. Pratt [39] compares language features and describes how they are implemented.

There are many books describing the mechanics of compiling. Some are Gries [21], Cocke and Schwartz [16], and Ano and Ullman [2]. For the purposes of this document, the most important is Hult et al. [1], which describes in detail the internals of the Bliss compiler.

Various authors have broken up the compiling process into from two to six stages. The Gypsy Verification Environment contains five: syntactic analysis, semantic analysis, compilability checking, optimization, and translation.

The first stage is syntactic analysis of the source text. This corresponds to the diagramming of sentences many people were forced to do in grammar school: deciding what is the subject and verb, and dividing the sentences into phrases. In this area, computer science and linguistics share a body of literature. Hopper and Ullman [27] is a representative text. The next step is semantic analysis, or deciding the meaning of the text. As an example, the sentence "A rock is alive" is syntactically correct but semantically incorrect, and "It jumped greenly" is semantically meaningless.

The syntactic and semantic analysis are done in the Gypsy Verification Environment by the Parser module. The output from the parser is a data base containing a manipulatable version of the "meaning" of the program. In the Gypsy Verification Environment, this "intermediate language" of the program is known as Prefix, because it is a tree structure in which the operators precede the operands. Other possible internal forms include Polish notation, triples, quadruples, and graph structures ( [21] pages 245-259). Recently, more emphasis has been placed on the structure of the intermediate language, because its design can affect the quality of the resulting machine code. See Harrison [24, 25] for a formal treatment of an intermediate language.
1.1.3 Code Generation

The final step in the compiling process is code generation, the translation from the internal form of the program into the final target language version. The target language may be machine language, or it may be assembly language or a different high level language. In compiling Gypsy, the target language is Bliss.

A big problem with programming in high level languages is efficiency. It is easier to write programs in high level languages, but the resulting machine language often is not as efficient as that obtainable by an assembly language programmer. For this reason, most of the work of compiling is in various optimization phases in the code generator. This is another area where there is a large body of literature, because the problems are complex. Fieler [20] formalizes the "quality" of generated code, traditionally an ad hoc comparison. Allen [3] gives a bibliography of papers in the area, and in Austin [4], he catalogues optimizing transformations.

People have realized that many optimizations are independent of the final target language, or, for that matter, of the source language. An example is the deletion of code that is never executed. These transformations can be made by manipulation of the intermediate language independent from code generation.

In the G.V.F., code generation takes place in three stages: compilability checking, optimizing, and translating into Bliss. These stages are discussed in section 1.3.

1.1.4 Compiling into Another High Level Language

Generating code by translating from one high level programming language to another is usually considered a "quick and dirty" way to implement a compiler for a language.

Some programming language processors do not include a code generation step: The output from the "compiler" is the intermediate language. An intermediate program simulates a machine whose machine language is the intermediate language. LTSP and SNPGUL programs are typically run in this way. There is a Gypsy Prefix interpreter in the Gypsy Verification Environment.

Some languages have been designed specifically for compilation into another high level language. RATFOR [40] is an important example. Many people consider FORTRAN to be antiquated and outdated, but its use is firmly entrenched. RATFOR (RATIONAL FORTRAN) adds "modern" programming structures to FORTRAN, and is intended to be translated into FORTRAN by a simple process.

Compilers can be written in the high level language they are meant to compile, by the techniques of bootstrapping, source-to-source translation, and cross compiling. All three
techniques were used in the development of the Common Bliss family of compilers \[14\].

Machine independent optimizations make these high level translations more important. Scheck \[42\] describes a FORTRAN-to-FORTRAN optimizing compiler. Schwartz \[43\] describes a compiler for SEIL, a very high level language, which translates SEIL into a high level language "of roughly PL/1 level." Loveman \[33\] discusses "levels of optimizations" corresponding to levels of languages. A compiler for a very high level language can do very high level optimizations, then translate into a high level language. This language may be optimized in turn, then translated into a lower level language, say Bliss, which does low-level optimizations and generates machine code. This is the strategy used in the Gypsy compiler.

1.1.5 Attribute Grammars

Attribute grammars have influenced the implementation of the Gypsy to Bliss translator.

Most of the literature on formal languages (section 1.1.2) concentrates on the syntax of languages. For compiling languages, it is important to connect semantics with the language definition. Normally, the tokens in the input stream are assigned semantic attributes. These attributes are combined as the tokens are grouped into phrases, until the "meaning" of the program emerges from the root of the program tree.

Donald Knuth introduced a new subfield of study with his paper "Semantics of Context-Free Languages" \[30\]. The important new idea in this paper is that a production in the grammar can inherit semantic attributes from its ancestors as well as synthesize them from the attributes of its descendants.

Consider, for example, a grammar describing a decimal number as a string of digits. Each digit has a value as an attribute. In a bottom-up syntactic parse, the value of a digit is the number it represents, between zero and nine. When two digits are combined to make a number, the synthesized value of the number is value(first digit) * 10 + value(second digit).

With attribute grammars, each digit can inherit its "place" from its parent, and its value attribute can depend on the place. Thus the "value" of the digit "2" in "2105" can be two thousand instead of two. This is a more natural interpretation.

Here is a formal definition of decimal numbers using synthesized and inherited attributes. The synthesized attributes are initial and value, and the inherited attribute is place.
Syntactic Rule

\[ \langle \text{dig} \rangle ::= ^0 \ldots ^9 \]

Semantic Rules

\[ \text{Dval}(\langle \text{dig} \rangle) = 0 \ldots 9 \]
\[ \text{value}(\langle \text{dig} \rangle) = \text{Dval}(\langle \text{dig} \rangle) \]
\[ \text{place}(\langle \text{dig} \rangle) = \text{place}(\langle \text{dig} \rangle) \]

\[ \langle \text{str} \rangle ::= \langle \text{dig} \rangle \]
\[ \text{value}(\langle \text{str} \rangle) = \text{value}(\langle \text{dig} \rangle) \]
\[ \text{place}(\langle \text{dig} \rangle) = \text{place}(\langle \text{str} \rangle) \]

\[ \langle \text{str}\text{L} \rangle ::= \langle \text{str}\text{R} \rangle \langle \text{dig} \rangle \]
\[ \text{value}(\langle \text{str}\text{L} \rangle) = \text{value}(\langle \text{str}\text{R} \rangle) + \text{value}(\langle \text{dig} \rangle) \]
\[ \text{place}(\langle \text{str}\text{R} \rangle) = \text{place}(\langle \text{str}\text{L} \rangle) \times 10 \]
\[ \text{place}(\langle \text{dig} \rangle) = \text{place}(\langle \text{str}\text{L} \rangle) \]

\[ \langle \text{num} \rangle ::= \langle \text{str} \rangle \]
\[ \text{value}(\langle \text{num} \rangle) = \text{value}(\langle \text{str} \rangle) \]
\[ \text{place}(\langle \text{str} \rangle) = 1 \]

Knuth's paper has generated a lot of interest over the years. A recent bibliography [5] lists over 150 references in English, German and French. A representative few will be discussed here.

Attribute grammars have a problem. It is possible to define an attribute that cannot be calculated because it depends on itself. Furthermore, Jazayeri [28] proves that to check for such a circular definition can take exponential time. On the positive side, Rochmann [12] gives an algorithm which, assuming the grammar is not circular, will tell how many passes are required to calculate all attributes. It also specifies conditions under which the attributes can all be computed in a single pass.

Rochmann and Ward [13] describe a compiler generator which uses attribute grammars. Given an attribute grammar, it generates a recursive descent compiler in Pascal to recognize that language. The semantic routines have CONST parameters for inherited attributes and VAR parameters for synthesized attributes.

Attribute grammars have been used to describe a number of program optimizations. Babich and Jazayeri [6] use them to analyze dead variables and available expressions.

1.2 The Gypsy Verification Environment

The Gypsy Verification Environment (G.V.E.) is a large LISP system which brings together a set of tools to aid the programmer in designing, specifying, implementing, and verifying Gypsy programs. The following description of the G.V.E. is based on the discussion in Dwight Pare's report [23].
The source text of the Gypsy program is read into the G.V.E. by a parser based on Wilhelm Burger's ROGS parser generator [15]. The program is maintained internally to the system as a LISP S-expression [34] known as Prefix. A program can be changed by either re-parsing or by editing with a structure editor [23]. The editor alters the Prefix form in response to the user's changes to the displayed Gypsy segment. The Prefix is un-parsed for display by a pretty print package called Inprint. The parser, the editor, and Inprint were all implemented by Dwight Hare.

Global effects of program changes are analysed by the "top level" component of the system. As incremental program development proceeds, changing one unit may require semantic checks to be applied to other units, or proofs to be redone. The top level was implemented in large part by Mark Moriconi [35].

Verification of a unit begins with generation of verification conditions. A verification condition is a theorem representing the relationship between the unit's specifications and code. If it can be proved, then the specifications and code are consistent. The VCCGen component was written by Richard Cohen and Judith Verrié.

The verification conditions are proved by a powerful interactive natural deduction theorem prover by W. W. Bledsoe [7, P]. Prover was integrated into the G.V.E. by Mabry Tyson and Peter Hudell.

Judith Verrié has written an interpreter for a large subset of Gypsy. Given a routine and a set of parameters, it can either simply return the result of the call, or print a detailed trace of the interpreting process.

Don Good, Donald Lynn, and Richard Alterman wrote a symbolic expression evaluator called Xeval. For constant arguments, it returns the constant result, otherwise it returns a canonicalized symbolic form.

The compiler which the subject of this document is to replace was implemented by Lawrence Hunter, this author, Charles Hoch, and Ann Sierbert. The compile command in the G.V.E. produces a text file containing the Prefix of a program. The Prefix is transformed into CMU, the Capability Machine Language for a hypothetical stack-oriented capability computer designed by Charles Hoch [26]. The CMU is then translated into PDP-11 assembly language. The assembly language is assembled with a runtime support package for execution either stand-alone on an LSI-11 or under the Unix operating system.
1.3 The New Gypsy Code Generator

The new compiler discussed in this document consists of three stages in the G.V.E.: a compilability checker, optimizer, and the Bliss translator. This "compiler" performs the function of the code generator module in a traditionally organized compiler.

Not all parts of Gypsy are intended for implementation. Some, such as rational numbers, are intended primarily for specification. The compilability checker insures that the program is completely defined and that it uses machine-dependent features properly. It also checks that all features used in the program have been implemented.

The second phase of the new compiler is an interactive optimizer being implemented by John McNam [35]. The optimizer will map the internal form of a Gypsy unit into a compatible, enhanced form. It will concentrate on issues unique to a verifiable language and a programming environment, such as suppression of runtime exception checks via proof methods, in line expansion of routines, and optimization of user data structures. The output of the optimizer is a new program tree, with annotations to guide the Bliss translator.

The Gypsy to Bliss translator will take the program tree, optimized or unoptimized, and produce a file containing the Bliss image of the program. The Bliss file will be compiled with one of the Bliss compilers and run.

This translation scheme has several advantages. Integration of the translator into the rest of the Gypsy Environment opens possibilities otherwise impractical, with access to the full database, very global optimizations are possible. Incremental compilation can be implemented by mechanisms parallel to the present incremental verification. The user can direct the compilation towards different machine environments or different entry points. The incremental compilation feature can be added to the top level in a controlled manner parallel to the current structure. The resulting status information can be displayed to the user similarly to the present information.

The productivity of the compiler development effort will be increased by the integration of the translator into the Gypsy Environment. Support for LISP includes online symbolic debuggers, program-oriented editors, and dynamic program modification. Access functions for the Gypsy database will be available to the translator among other predefined support operations.

There are disadvantages and unknown factors in the implementation of the Bliss translator. Integration of the translator into the Gypsy Environment may be difficult, considering that the system is already experiencing space...
problems. If necessary, a separate "Gypsy Compilation Environment" could be constructed, consisting of the Exec top level, the database handlers, and other modules directly relevant to compilation, leaving out verification-related modules. A user would then use the system save-restore feature to switch from one compatible environment to another.

For the Gypsy programmer, the Gypsy to Bliss translator will be more time-consuming to use than a compiler which outputs assembly language or object modules. The programmer must parse the Gypsy program into the Verification Environment, run the translator, then run the Bliss compiler on the resulting bliss image. The programmer must be satisfied in knowing that the compiler is operational much sooner than it would be without the intermediate Bliss.

1.4 Acknowledgments

Don Good first suggested translating Gypsy Prefix to Bliss.

This author did a preliminary analysis of the feasibility of using Bliss for the target of Gypsy's compiler. That analysis became a "straw man" for a series of discussions among this author, John McHugh, Don Good, and Richard Cohen. This document is the essence of those discussions. Lawrence Hunter has made additional comments. All of these people, especially Don Good, have made invaluable suggestions on the design and contents of this document.

Several design notes by John McHugh inspired useful discussions. In particular, material from those notes are included in the sections on dynamic data types and stack management.

Jennifer Boes, Nancy Wagner, and Scribe [44] participated in the typing and formatting of the document. Richard Cohen and John McHugh helped set up the Scribe macros to produce these results.
CHAPTER 2
GYPSY, PREFIX, AND BLISS

This chapter describes the language Gypsy, its internal form, Prefix, and the language Bliss.

2.1 An Overview of The Gypsy Programming Language

The Gypsy programming language has a number of features in common with the other members of the family of Pascal-like languages. The programmer declares a set of data types built up from primitive types, then can declare objects of that type. The language enforces that only certain operations are allowed on each type.

The programmer defines a set of functional and procedural subroutines that can be called elsewhere in the program. The formal parameters to the subroutines are declared either CONST, if read-only; or VAR if read-write. A call to a subroutine supplies the actual parameters referred to by the formals in the routine. The language enforces that the types of the formals and actuals are compatible.

A program is made up of a series of statements, such as procedure calls, assignment statements, and control statements. Unlike other Pascal-like languages, Gypsy has a flat name space; that is, the unit definitions are not nested (with the exception of unnamed "anonymous" types, which are allowed in some contexts; and local constants declared in routines). Gypsy units are divided into related groups by "scopes," which help control a proliferation of unit names, and provide access control. All local variables are allocated on entry to a routine and destroyed on exit.

2.1.1 Flow of Control

Gypsy's control statements perform the familiar testing and iteration of other languages, but the syntax is slightly different. In positions where Pascal requires a compound statement, Gypsy uses a list of statements terminated by a keyword.

The Gypsy IF has the syntax
IF <boolean expression>
THEN <statement list>
ELIF <boolean expression>
THEN <statement list>
...
ELSE <statement list>
END;

The ELIF's and ELSE are optional. An arbitrary number of statements can appear between the keywords ELSE and END, for example, because the keywords bracket the list. The keyword ELIF expands into ELSE IF, eliminating the need for a long list of END's to match each IF.

Gypsy has only one form of iteration: the LOOP statement. It is equivalent to a Pascal "while True do...." The LEAVE statement exits the innermost enclosing LOOP. Depending on where the LEAVE appears, the loop acts like a Pascal WHILE, REPEAT, or more general form of iteration.

Most high level languages provide a statement for multi-way branching. Gypsy's CASE statement has the form

CASE <expression>
IS <constant> ... : <statement list>
IS ...
ELSE: <statement list>;
END;

Manageable program verification requires that the effects of a statement be localized and carefully controlled. It should come as no surprise to those familiar with Dijkstra's "Go To Statement Considered Harmful" [19] and the subsequent debate, that Gypsy does not have a GOTO statement. The statements Gypsy does have which interrupt control flow, SIGNAL and LEAVE, have controlled behaviors. This enables the compiler to perform flow analysis of the program enabling the application of numerous optimizations.

Gypsy assertion statements are intermixed with the executable code. An assertion is a boolean expression on the states of program variables. One may ASSUME or PROVE an assertion at proof time, and validate or ignore assertions at run time.

An important feature of Gypsy impacting its compilation is the careful specification of expression evaluation and implicit conditions. It was initially feared that the language definitions prohibited optimizations by restricting the flow of control within expressions. To illustrate, consider the expression
\[ a + (b - c) \]

The *Gypsy* manual ([22], pg 66) maps the expression to an explicit functional composition

\[
\text{add}(a, \\
\text{subtract}(b, c) \text{ unless (subtracterror)} \\
) \text{ unless (adderror)}.
\]

*Gypsy* function arguments are evaluated before the function proceeds ([22] pg 93). Because machine arithmetic on a subrange of integers is not necessarily associative, the order of evaluation may determine which condition is signalled. It has been proposed that the strict interpretation of the functional form be relaxed; since a program verification is a partial proof, all reorderings of the expression remain valid as long as the overflow is detected. This is not entirely satisfactory from the programmer's point of view. In the example, if all three variables are known to be large, then the programmer would want to force the subtraction to be executed first. This issue is currently under discussion.

### 2.1.2 Data Structures

Access to data is very restricted in *Gypsy*. *Gypsy* does not allow global variables, functions may have only `CONST` parameters, and there are no `VAR` variables. Eliminating `GOTO`'s simplifies control flow analysis, and eliminating globals simplifies data flow analysis. However, it causes *Gypsy* programs to have long parameter lists, which can cause context switches to be more expensive.

Explicit pointers cause problems similar to those of global variables, and they, too, were omitted from *Gypsy*. To give the programmer the convenience of pointers in a more controlled manner, the *Gypsy* language designers included dynamic types: sequences, sets, and mappings. The implementation of the dynamic types and their underlying memory management system has many options, which will be discussed later.

*Gypsy* has a rich set of primitive types. Simple types are integer, character, and boolean. *Gypsy* provides rational numbers for specifications only. Floating point numbers are excluded because machine implementations do not follow the mathematical axioms of arithmetic. Structures can be built up of records, arrays, sequences, sets, mappings, and message buffers. Each structure can be made up of elements of the other types. (The language definition does not prohibit types from being recursive,
but the current implementation of the parser does not allow it. Sequence types could be recursive, with empty sequences at the leaves.)

Gyosy sequence types have a rich set of operations. An existing sequence element is changed with the same notation as for arrays. A programmer may insert elements anywhere in a sequence with BEFORE/BEHIND seq[i] := ..., delete elements with REMOVE seq[i]; and access or assign to subsequences with the notation seq[low..hi]. Other operations are append, adjoin, first and nonfirst, last and nonlast, size, in, initial, null, assignment, eq, neq and sub.

Set types correspond directly to the equivalent mathematical concept. Operations on sets are eq, ne, intersect, difference, in, adjoin, omit, sub, with, size, and assignment.

A mapping type is conceptually a set of ordered pairs, the first of the pair from the DOMAIN type and the second from the RANGE type. A mapping access is of the form \( m(x) \), where \( x \) is in the domain of \( m \). If there is an ordered pair with \( x \) as the first element, the value of the access is the paired range element. Otherwise, IndexError is signalled. Other operations on mappings are domain, range, eq, union, intersection, difference, submapping, size, and element omit and assign.

2.1.3 Condition Handling

Gyosy has statements to generate and handle runtime conditions. Gyosy condition handling pervades its implementation, because most constructs in Gyosy have implicit error checking.

Conditions can be signalled implicitly by Gyosy predefined operators or explicitly by the programmer. Predefined conditions signalled implicitly include AddError and MultiplyError for arithmetic overflow, IndexError for array index bounds, or ValueError when an actual parameter is outside the type of the corresponding formal at binding time. Any predefined or programmer declared condition can also be signalled explicitly.

Any control structure can end in a \(<\text{when clause}>\), which is syntactically similar to a CASE statement where the "case labels" are condition names. When a condition is raised, program control transfers immediately to the arm of the innermost \(<\text{when clause}>\) which handles the condition, without changing the program state. After the statements in the \(<\text{WHEN}>\) arm are executed, control passes out of the containing control statement.

If there is no \(<\text{when clause}>\) to handle a condition in a routine, it is passed to the calling routine via a condition parameter. If the condition raised is not a formal condition parameter, it is mapped by default to the implicit condition parameter RoutineError. The programmer declares formal cond
parameters by this syntax:

PROCEDURE ifflle(CONST d1:t1; VAR d2:t2)
   UNLESS (COND c1,c2,...) = ...

Whenever a routine's formal condition is signalled, the corresponding actual condition is signalled in the calling environment (unless some <when clause> in the called routine catches it first). If the actual condition parameters are omitted at the call site, the formal names are used by default as the actuals. (Thus x+y is Function T#add(x,y:T):T unless (AddError).

Because of its direct transfer of control, a SIGNAL is like a GOTO. However, unlike Bliss' LEAVE and C's [41] BREAK, a signal is a well-regulated GOTO amenable to more straightforward program analysis. Condition parameters, on the other hand, GOTO a location outside the current routine, essentially providing multiple return addresses, like Algol label parameters.

2.1.4 Concurrency

2.1.4-A COBEGIN Statement

A family of concurrent processes is started in Gypsy with a COBEGIN statement, which has this form:

Cobegin
   p1(v11,v12,...) unless (c11,c12...);
   p2(v21,v22,...) unless (c21,c22...);
   ...
   when
   is ...
   else...
end;

The procedure calls Pi are invoked concurrently, and the process which executed the COBEGIN statement is suspended until all the arms terminate. It, after all of the arms terminate, one of the arms has signalled a condition, it is handled in the usual way. The language definition does not say what happens if more than one arm terminates with a condition.

The arms of a COBEGIN cannot share any VAR parameters except message buffers. Since global variables do not exist in Gypsy, the buffers are the only possible form of interprocess communication. It is impossible for an external action to cause an arm to terminate; one can only request through a buffer that the process terminate itself. Likewise, there is no way for an arm of a COBEGIN to know when a sibling terminates, unless the
sibling explicitly sends the information through a buffer.

Note that there is no restriction on the procedures called as the arms of a \texttt{CUBEGIN}. They may themselves contain \texttt{CUBEGIN}'s. A procedure containing a \texttt{CUBEGIN} can even be recursive.

2.1.4-B Message Buffers

A Gypsy message buffer is a predefined structure allowing interprocess communication. A buffer is essentially a first-come first-served queue. A buffer type can be declared to contain messages of any (non-buffer) type, from integers to complicated structures. A Gypsy buffer type has a specified length which is the maximum number of messages it can contain. The operations defined on buffers are \texttt{SEND}, \texttt{RECEIVE}, and \texttt{GIVE}; \texttt{FULL}, \texttt{EMPTY}, and \texttt{CONTENTS}; buffer history extraction functions; and \texttt{AWAIT}.

\texttt{SEND} and \texttt{RECEIVE} are the basic message transfer statements. \texttt{SEND} and its variant, \texttt{GIVE}, place a message into a buffer. Any process that has that buffer as a parameter can then \texttt{RECEIVE} it. If there is no room in the buffer, the sender blocks until a receive from the buffer by some other process frees up space. Similarly a \texttt{RECEIVE} can block until some other process sends to the buffer.

\texttt{FULL} and \texttt{EMPTY} are boolean functions that allow a process to find the status of a buffer. \texttt{CONTENTS} is a function that extracts a copy of the current buffer queue as a sequence. These functions provide instantaneous snapshots of the buffer.

A buffer history is a copy of every message that passed through the buffer, with annotations who sent the message when. They are necessary for proofs of concurrent programs. Implementation of buffer histories is not planned.

2.1.4-C AWAIT Statement

Gypsy's \texttt{AWAIT} statement provides a process the ability to execute one of a set of buffer statements. The syntax is

\begin{verbatim}
AWAIT
  ON <SEND/RECEIVE/GIVE stmt>1 THEN <stmtlist>1;
  ON <SEND/RECEIVE/GIVE stmt>2 THEN <stmtlist>2;
  ...
  WHEN
  END;
\end{verbatim}

The awaiting process blocks until not all of the buffer statements are blocked, then nondeterministically executes one of the receptive buffer operations and the corresponding statement list. After the single arm is executed, the \texttt{AWAIT} is finished.
2.1.5 Programming Style

Other than its effect on the language features, verification also affects programming style. Data abstraction and proof modularity cause a proliferation of very small subroutines. This means that flow analysis, no matter how sophisticated, cannot significantly optimize the code if limited to one routine at a time. The Gypsy compiler must optimize an entire program.

2.2 Prefix Vs. Abstract Syntax

The Gypsy Verification Environment stores programs in an internal form known as Prefix. Each unit is represented as a tree structure (a Lisp S-expression) with the operator in the in the first (or CAR) position at each internal node of the tree.

The G.V.E. is a very large, evolutionary system implemented over the years mostly by a group of transient graduate students. As a result, the grammar for Prefix has developed some problems. The most important problem is inconsistency. There are two dialects of Prefix in the system that are accepted by different modules. The dialects differ in small but significant ways. For example, in "Blackboxed" Prefix, a statement list has the operator CAR. In "Blueboxed" Prefix, the operator has been omitted, and the statement list is simply a bare list of statements. The operator was omitted because the various Prefix processors can (and now must) determine the implicit operator by context. There are also overloaded keywords. For example, the keyword $S$ can be either a sequence type definition, an element or subsequence extraction, or an insertion. Again, context, both from the parent and the children of the node, is necessary to disambiguate. As mentioned, there are also subtrees in the structure that do not have operators. NIL appears frequently as a place holder for optional arguments.

When the Prefix is processed explicitly by recursive descent, these anomalies do not cause serious problems. However, a simple keyword driven, context free processor cannot handle it. Knowledge of the grammar of the internal language must be present, either implicitly in the code, as in the recursive descent processor, or explicitly in some data structure. This kind of overhead should not be necessary.

The name Prefix shows another problem with the internal form. System components know the concrete representation of the program and are allowed to directly access into the structure. For example, components know that the fourth element of the node representing a CASE statement is the statement list for the ELSE part.

In the early history of the G.V.E., Prefix nodes were considered to have descriptors giving additional information about the operator and the subtree arguments. For example, the
Prefix for an array access could be annotated to indicate that
IndexError checking has been proven unnecessary for this node.
Extraction functions were provided for the operator, descriptor,
and argument lists of Prefix nodes. The descriptors have fallen
into disuse, however, because several components could not be
readily modified to handle them.

As a part of a study of the abstract syntax and semantics of
Gypsy, Rich Cohen has designed an abstract syntax for Gypsy. It
is similar to Prefix, except that the problems discussed above
have been removed.

Rich Cohen, Johnchurch, and this author have developed a
set of abstraction functions to implement abstract syntax. There
are extraction functions for the operator and descriptors of a
node, and a large number of argument extractors. Instead of
pulling out the fourth element of a CASE statement, a programmer
will invoke a macro AAS-CASE-FLSH. Similar functions exist for
every nonterminal argument. Symbol tables in the Prefix
structure are no longer directly accessible to the program. In
the abstract syntax, they have been hidden behind a separate set
of extraction functions.

As modules of the G.V.E. are rewritten in the future, they
will be retrofitted to use the new abstract syntax. For the near
future, the abstract syntax will have to be yet another dialect
of the internal language coexisting in the system. There is an
important difference between the abstract syntax and the other
dialects. Because the abstract syntax is an abstraction, it need
not exist in concrete form. The abstraction routines are written
to accept any of the dialects of Prefix. Each converts the
Prefix form given it to the abstract form before returning the
extracted argument. The conversion need only recurse until a
keyword is found that is not sensitive to outer context. The
concrete representation will be converted to the abstract syntax
at the top level, yet remain in the old forms at lower levels.
This has the interesting effect of enforcing use of the
abstraction functions!

2.3 An Overview of The Bliss Programming Language

Bliss [11] is a programming language for DEC computers. It
was specifically designed for the implementation of "System
Software" such as operating systems and compilers. The Bliss
language designers chose a set of objectives to meet their
requirements for a "good" system language ([17], pg 1):

1. highly optimizable object code,
2. simple and consistent facilities for operating on
   addresses.
3. Control structures which encourage well structured source code.

4. Representing and accessing user-defined data structures.

5. Optional access to specific features of the target-system hardware or operating system.

6. Facilities for defining the linkage convention used in calling routines.

Bliss has five dialects. bliss-10 [9] is the original Bliss, implemented for the DEC-10. The compiler is available from DEC. Bliss-11 [10] is a close relative to bliss-10 for execution on the PDP-11. Its compiler is available from DECUS. It is a cross-compiler which runs on the DEC-10 or DEC-20 and produces object code for the PDP-11.

Common Bliss [17] is a redesign of Bliss done by DEC, consisting of three dialects and five compilers:

1. bliss-36 compiles and runs on the DEC-10 or DEC-20.

2. bliss-32 runs on the VAX, and has two compilers: a cross compiler on the DEC-10, and a native mode compiler on the VAX.

3. bliss-16 runs on the PDP-11 from cross compilers on the DEC-10 or VAX. There is no native mode bliss-16 compiler.

Common Bliss constitutes the transportable language base, and the dialects and features allowing for dissimilarities among the target systems. If a program is written in Common Bliss, it can be compiled and run on any of the three machines.

The Bliss compilers are famous for their optimizations. Each makes a multitude of passes over its internal form folding expressions, reusing intermediate results, and removing redundant or unreachable code. Registers are allocated to variables according to the variable's usage in each basic program block. After all these higher level optimizations are completed, it runs a peephole optimizer over the resulting assembly language image of the program.
2.3.1 Flow of Control

Bliss is similar to other high level languages in several ways. In Bliss one can define routines with formal parameters, and call them from other routines, instantiating the actual parameters. It has a block structure, with variables allocated on an automatic stack. Expressions are constructed in "normal" infix algebraic notation. Its features for iteration and branching provide the kinds of control flow that support structured programming.

Bliss is unique in several ways. As the manual says, "the reader should strive to master the implications and meaning of the statement:

"Bliss is an expression language."

There are no statements in Bliss in the traditional sense. All routines are functions. If the value returned is not used, it is discarded. All of the other forms traditionally considered statements return a value in Bliss, so there are control expressions like WHILE, IF, and CASE; compound expressions, BEGIN-END and parenthesized expressions; and simple expressions such as "100" and "a+b."

2.3.2 Data Structures

The basic data unit in Bliss is a pointer. A name declaring a data segment is interpreted as the address of the segment instead of as the value of the segment as in other languages. The language includes an explicit fetch operator ",", which denotes the contents of a location. Thus if "a" is declared a Bliss array, then "a" is a pointer to the array, ",a" is the contents of the first element of a, "a+2" is a pointer to the third byte of a, and "...(a+2)" treats the second word of a as a pointer, and fetches the object it points to. The expression ",10" will fetch the contents of absolute memory location 10.

Accessing parameters in Bliss is not syntactically consistent with accessing locals. Consider a routine call

\[ \text{rout}(A, B) \]

which corresponds to a routine declaration

\[ \text{routine rout}(A_1, A_2) = \quad \]

The first actual parameter is a pointer to the data location A,
and the second is the contents of location B. The formal name F1 is a pointer to the first parameter. So for the call above, ".F1" is a pointer to A, and ".F1" is the contents of location A. ".F2" is the contents of location B. In this example, A was passed call by reference, and B was passed call by value.

Identifiers can be bound at compile time to numbers:

BIND CON=10;

In this example the expression "CON" gives the number 10, and "CON," gives the contents of memory location 10.

It can be seen that depending on whether an identifier is a constant, local data, or a parameter, references to it may require zero, one, or two fetch operators. The expression on the left hand side of an assignment expression requires one less fetch operation than the right hand side. Thus "a = .a" is a no-op, and ".(rout(a,n)) + 100 = 1" treats the result of the routine call PNUM as an address of an array, and assigns 1 to the 100th element of the array. Since Bliss is an expression language and every number is a pointer, the right hand side of an assignment expression can be any arbitrary expression including routine calls, control expressions, or other assignment expressions.

A programmer can use the Bliss structure declaration to define an abstract data type. The structure declaration contains an arbitrary expression to access data from a variable of that type.

2.3.3 Language Extension

Bliss has two other features that make it different from many high-level languages. The first is its extremely general macro facility. Thus, the language is extensible. At the other end of the scale, Bliss allows a programmer the ability to invite the compiler to generate arbitrary assembly language in its output stream. The term "invite" is used because there is no guarantee the Bliss optimizer will leave the assembly language as specified by the programmer.

2.3.4 Condition Handling in Bliss

Condition handling is inconsistent among the dialects of Bliss. Bliss-10 does not support condition handling at all. The mechanisms in Common Bliss and Bliss-11 have nothing in common except the keywords ENABLE and SIGNAL.
2.3.4-A Common Bliss

The Common Bliss condition handling is an artifact of the VAX/VMS condition handling mechanisms. On entry to a routine (not a block), an ENABLE declaration can specify another routine to be the handler for all signals generated during the execution of this routine. This includes the signals generated in called routines which do not themselves have an ENABLE declaration. The signals can be generated explicitly by the Bliss program or be generated indirectly by the hardware or operating system, for example by arithmetic overflow or address violation.

The handler routine has three parameters, each of which is a pointer to a vector. The first is the "Signal Vector," which contains the signal value and other information provided at the point of the signal. The second is the "Mechanism Vector," which contains information provided by the "Condition Handling Facility," including a pointer to the value to be returned by the handler. The third is the "Enable Vector," which is a vector of parameters provided in the ENABLE declaration.

The handler can modify the program state by changing values of parameters passed it in its Signal vector or Enable vector, or by changing global variables. It can then request one of three actions to be performed after the handler returns: continue the signalling routine, resendal, or unwind. If the handler can deal with the condition and patch the program state, then it can continue execution after the SIGNAL expression. The handler can specify a value to be returned by the SIGNAL expression. Resignalling is the appropriate response if the handler cannot process a certain signal value. The Condition Handling Facility invokes the handler for the next enclosing Enable declaration. Unwinding causes the signalling routine, and other routines in the dynamic calling sequence, to be terminated. The handler can specify the returned value from the outermost enclosing routine that is terminated.

The programmer specifies which of these three actions is to be taken by a combination of setting variables, calling parameterless routines with side effects, and returning 1 or 0 from the handler. In addition, there are numerous restrictions on the forms of the handler routines and their data. These details will not be discussed here.

2.3.4-B Bliss-11

The Bliss-11 mechanism for signalling and handling conditions is much simpler. Any block can have an ENABLE declaration. The syntax of the ENABLE is exactly the body of a SELECT expression, which is one of Bliss' two statements for multi-way branching. The conditions are signalled only explicitly by the SIGNAL expression. The SIGNAL expression has one argument, a word value. That value is used to select an arm of the most recent enclosing ENABLE declaration on the run time
stack. The scopes of ENABLE declarations are determined by the run time calling sequence and ignore routine boundaries. If no arm of the ENABLE declaration handles the condition value, the next enclosing ENABLE is automatically searched, and so on. After the arm of the ENABLE declaration is executed, the block which declared it is terminated, yielding the value of the executed ENABLE arm. If no ENABLE handles the condition, the program crashes.

2.3.5 Bliss Coroutines

Bliss' process management consists of two expressions which allow coroutines. The different dialects of Bliss are not exactly compatible; the discussion here is on Bliss-11.

STB = CREATE e0(e1,e2,...) AT e3 LENGTH e4 THEN e5;
OTHERSEXP = EXCHJ(STB,EXP);

CREATE initializes a Bliss process. E0 is the routine to be called with parameter list (e1,e2,...). E3 is the address of a piece of memory e4 words long, which is to be used for the new process' stack. E5 is the name of a parameterless routine that is to be executed if the new process e0 ever tries to return. E5 itself must terminate with an EXCHJ. If e5 returns, the program will crash without warning. The value of CREATE, STB, is the "stack base" of the new process. CREATE does not actually invoke e0, it only initializes the stack.

EXCHJ causes a context switch to the process whose stack base is the parameter STB. The first time a process is invoked, execution begins at the beginning of routine e0. The new process gives up the CPU only by executing another EXCHJ on some other process' stack base. When a process returns from an EXCHJ, it is because some other process has in turn executed an EXCHJ to that process. The returned value OTHERSEXP is the value EXP from the process yielding control.

Message buffers or semaphores are not provided as primitives in Bliss.

2.4 Bliss as Target

Choosing Bliss as a target language for the translation has a number of advantages and a few disadvantages. The most important advantage is programmer efficiency. Writing a compiler is a very old task. The code generator for the previous compiler, the module which corresponds to the Bliss translator, took about two programmer years to complete. By translating into Bliss a Gypsy compiler can be implemented with considerably less effort. The Bliss compiler can take care of the details of machine code generation, allowing the translator to concentrate on the issues
unique to Gypsy.

By translating into Bliss, Gypsy can make use of the Bliss optimizers. The Gypsy Prefix to Prefix optimizer will handle high level issues unique to Gypsy. The Bliss optimizers are famous for their handling of low level issues. There is no need to duplicate this effort.

Considering the difference in the philosophies of Gypsy and Bliss, the languages are quite compatible. If Gypsy had Pascal's nested routine structure with dynamic scoping of global variables, then the translation into Bliss would have been much more difficult. Gypsy has a flat structure to facilitate verification, Bliss has it for efficient runtime execution. Similarly, Gotos were omitted in both languages, because they complicate flow analysis. Again, Gypsy omitted them to facilitate proof time analysis, and Bliss omitted them for optimization time analysis. The basic sequential control structures in Gypsy can be translated into Bliss with syntactic massaging.

Bliss has compilers for the PDP-11, VAX, and DEC-10. With limited extra effort, Gypsy can run on three machines. In addition, with a set of macros or runtime support routines, the PDP-11 code output by the Bliss-11 compiler can run on the bare LSI-11 or under an operating system on a bigger machine.

Bliss has a set of debugging tools which can be used as is to debug Gypsy programs, or be adapted to know about Gypsy constructs.

There are problems with translating into Bliss, but they are outweighed by the advantages. Some language features do not map cleanly into the various dialects of Bliss. In other cases, translating into Bliss is at worst no worse than translating directly into assembly language. Bliss was designed to be a low level system implementation language; it has been called a structured assembly language. With that philosophy, the high level features must be built up from more primitive features. The most important examples are dynamic memory and concurrency.
CHAPTER 3
THE BLISS IMAGE OF GYPSY CONSTRUCTS

This chapter describes the runtime image of Gypsy structures as implemented by the Gypsy Prefix to Bliss translator. Emphasis is placed on the features included in the initial implementation of the translator. The discussion of each feature is followed by a section "Initial Implementation of ..." where the method used in the initial implementation is prescribed. These features include:

1. Sequential control structures
2. Routine calls with data and condition parameters.
3. Condition handling
4. Simple (non-structured) data types
5. Expressions of simple types
6. Specialized buffer operations for terminal interfaces.

3.1 Basic Control Structures

This section discusses Gypsy control structures without WHEN parts, and excluding the concurrent structures. Those topics will be treated individually later. All programming languages have statements for iteration and branching. The syntax of many high-level languages is similar. Despite their differences in philosophy, Bliss and Gypsy have basic control statements that are quite compatible. The control statements in Gypsy are IF, LOOP, LEAVE, and CASE, and routine calls. The corresponding expressions in Bliss are the IF; UNILWHILE UD, DO UNTILWHILE, INCR, and DECR; LEAVE and EXITLOOP; CASE and SELECT. The richness of Bliss compared with Gypsy allows the Gypsy to Bliss translator the opportunity to chose the most efficient of several possible implementations. The translations from Gypsy to Bliss control structures are merely syntactic massaging.

The Bliss image of a Gypsy IF is obvious:
IF <boolean expression>
THEN BEGIN <statement list> END
ELSE IF <boolean expression>
THEN BEGIN <statement list> END
...
ELSE BEGIN ... END;

The most straightforward Bliss implementation of a Gypsy LOOP is

WHILE 1 DO BEGIN
   <statement list>
END;

where the Gypsy LEAVE is translated into a Bliss "EXITLOOP." Having a single iteration statement is convenient for generating and proving verification conditions, but it makes the loop harder to optimize. In the future, the Gypsy optimizer should try to recognize when a Gypsy LOOP corresponds to a WHILE DO, DO WHILE, INCP, or DECR, and produce the optimal Bliss code.

Bliss' two multi-branch expressions are very closely tied to their low-level implementations. The Bliss CASE expression is implemented as a jump table with one entry for each of the possible values in the RANGE of the arms. Every possible value must be specified, and no bounds checking is done. The SELECT expression evaluates its selecting expressions at run time, and allows any number of matching arms to be selected and executed in turn. A CASE expression is very efficient in execution time, but potentially takes up a lot of space. A SELECT takes more execution time but takes less space for a sparse set of selectors. The Gypsy to Bliss translator must decide whether a Gypsy CASE statement can be more efficiently implemented as a Bliss CASE or SELECT. The criterion used to make that decision is simple. The Bliss CASE expression requires one word of storage for each value in the RANGE of the arms, including those handled by the ELSE case in Gypsy. A SELECT requires four words for each Gypsy case arm value. The method that requires the least space is chosen.

The implementation of routine calls is discussed in the sections on condition and data parameters.
3.2 Condition Handling

3.2.1 Bliss Image of Gypsy Condition Handling

The Gypsy signalling mechanism maps onto Bliss-11's almost trivially. Gypsy condition names are bound to Bliss compile time integers, so the Gypsy signals and the Bliss image are identical. The <when part>'s are syntactically massaged into an ENABLE declaration and moved from the end of the controlling structure to the beginning. Condition parameters will be discussed later.

One of the purposes of the translation from Gypsy to Bliss was for machine transportability. The Bliss-11 Image of Gypsy condition handling is easy, but it does not work on the DEC-10. For future expansion, there are several alternative ways to implement Gypsy condition signaling within a Bliss routine without resorting to dialect-specific features. Three promising ways are with nested blocks, or with macros, or as a case statement.

The easiest way is to use Bliss macros and assembly language out-of-line routines to implement Bliss-11-style ENABLE's and SIGNAL's in Bliss-10, as discussed in Leverett [32].

Unnecessary generality in Bliss-11 condition handling causes extra overhead in the Gypsy image. The value signalled in Bliss can be a runtime variable, while the Gypsy condition value is known at compile time. This causes overhead in the normal case where a control structure executes without signalling a condition. The Bliss runtime linking of ENABLE blocks is unnecessary for Gypsy because the handlers for conditions can be known at compile time. For other implementations that are more efficient, but more complicated, see the internal note on the subject [45].

3.2.2 Initial Implementation of When Parts

There is no need for this initial implementation to have full machine compatibility, so the Bliss-11 mechanisms are used. The extra runtime overhead is not considered excessive at this time.

Each predefined Gypsy condition has a value bound to it at compile time. The conditions are numbered beginning with 1, in alphabetical order. Each user-declared condition is assigned a unique value. If the same user condition name is declared in more than one routine, the different invocations have the same bound values.

Here is a Gypsy BEGIN statement and the corresponding Bliss expression:
COND indexerror, c1, c2;
begin
  signal c1;
when
  is indexerror: ...
else: ...
end;

bind indexerror=7, c1=101, c2=102;
begin
  enable
  indexerror: begin ... end;
  otherwise: begin...end
  elone;
  signal c1;
end;

3.2.3 Handling Gypsy Cond Parameters in a Bliss Image

Common Bliss's condition handling is on a per-routine basis.
One can SIGNAL a normal return from the site of the SIGNAL, or
from the current routine or some number of calls "up" in the
display. Unless every nested control structure with a
<when clause> were implemented as a separate Bliss routine,
Gypsy's condition parameters cannot be mapped directly onto this
mechanism. This is not feasible for runtime efficiency.

As discussed above, Bliss-10 does not support condition
handling.

Bliss-11's condition handling mechanism ignores routine
boundaries, so it does not support formal/actual condition
parameter bindings. Gypsy's mapping from formal to actual
condition parameters is the only feature of condition handling
without a direct analog in Bliss-11 condition handling. A Gypsy
routine with condition parameters must have extra Bliss data
parameters describing the formal/actual mapping.

It is possible in Bliss-10 or Bliss-11 to create imoures
Bliss by laying down arbitrary assembly language instructions in
line. An image of a Gypsy routine call can be anything desired.
For example, the actual COND parameter values could be the
absolute address of the code that handles the condition, plus the
stack top in the when part's context. It is not desirable to
consider this possibility at this time for compatibility and
simplicity. If it becomes necessary, large space and time savings
may be realized. (See section 4.4.2.)

The alternative implementations of condition parameters
(without resorting to assembly language) differ in how the
condition parameter values are passed in to the called routine.
The values may be passed in either as separate Bliss input
parameters or as a structure. The discussion here is again limited to the Bliss-I1 implementation with ENABLE and SIGNAL. There are methods of handling Gypsy condition parameters in Bliss that do not involve the Bliss-I1 condition mechanisms. They are discussed in an internal note [45].

There are several ways for a calling routine to pass in the condition values it wants returned for each condition parameter. One way is by providing the condition parameters explicitly in the Bliss call:

```c
ifile(a1,a2,actcond_1,act_cond_2...,);
```

Signaling a condition parameter would become SIGNAL _FOR_ _COND_1. Since locally handled condition values are known at compile time, overhead can be avoided by passing a pointer to a literal:

```c
ifile(a1,a2,olit(actcond_1,actcond_2...,);
```

Signaling the ith condition parameter would become SIGNAL _ACTCOND_1[i].

If condition names have a unique binding that is global to all routines, then an interesting optimization is possible. If every call to a routine leaves the condition parameters implicit, then there is no need for the extra Bliss parameter. The actual/formal mapping is the identity function in Bliss as well as Gypsy. In order to efficiently implement implicit condition parameters, every predefined and user defined condition name must have a unique value. This, again, is a trivial problem.

There is one very important special case of a condition parameter that must be considered. Past experience has shown that by far most signalled conditions are handled nowhere in the program, and simply bubble out of the program as "RoutineError Signaled from 'main." For runtime efficiency and for debugging, these conditions should be treated specially. If it can be shown at compile time that no condition signalled in a routine is trapped anywhere in the entire program, then that routine does not need any Bliss condition parameters. Condition parameters signalled within that routine call a global 'main RoutineError handler which either exits the program with a message or invokes a debugger. This requires extensive compile time analysis of every possible calling tree of the program. If only one caller of the routine handles its condition, then a Bliss condition parameter handling mechanism must be provided. For debugging, it is desirable to know at runtime when a given signal will bubble out of 'main so a debugger can be invoked. For the routines whose condition parameters are sometimes handled, this will require the
callers to pass in a boolean Main_Routine_Error flag. It may be desirable to treat signalling Routine_Error as an error in every case and invoke the debugger.

3.2.4 Initial Implementation of Condition Parameters

In the initial translator implementation, the Bliss-11 condition handling mechanisms are used. Each routine with condition parameters has an extra parameter in the Bliss-11 image. This parameter is a vector of actual condition values corresponding to the formals. The optimizations concerning implicit actuals and Routine_Error from Main are not done in the initial implementation.

3.3 Gypsy Data Declarations in Bliss

3.3.1 Allocation of Gypsy Locals

In Gypsy, variables can be either locals or parameters. There are no global variables, and the language is not block structured, so there is no need for any kind of runtime display (as defined in Pratt [39], pg 209). Variables are considered allocated on entry to a routine and destroyed on exit. Nothing is said in the language definition about the runtime implementation of the variables.

Bliss gives the programmer complete power and flexibility. Variables may be global, own to a routine, allocated on the stack, or in a register. (Bliss own data are statically allocated, unlike Algol 60's variable length own structures.)

Because of the architecture of the LSI-11, not all Bliss data references have the same cost. In order of increasing cost, they are

<table>
<thead>
<tr>
<th>Bliss use</th>
<th>LSI-11 address</th>
<th>Timing (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>local in register</td>
<td>R1</td>
<td>0</td>
</tr>
<tr>
<td>Parameter pointer</td>
<td></td>
<td>1.4</td>
</tr>
<tr>
<td>in register</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Global or own</td>
<td></td>
<td>3.5</td>
</tr>
<tr>
<td>(Memory access)</td>
<td></td>
<td>4.2</td>
</tr>
<tr>
<td>Stack local or Stack Consf parameter</td>
<td>X(R6)</td>
<td>4.2</td>
</tr>
<tr>
<td>VAR stack parameter</td>
<td></td>
<td>6.3</td>
</tr>
</tbody>
</table>

The Gypsy compiler can have a great deal of flexibility in
allocating and accessing data in Bliss. The Bliss compiler is
very smart about deciding when a local can be allocated in a
register. During compilation of a routine, the registers are
allocated to the variables used most in each basic program block,
so that a variable may move from stack to register during its
lifetime. (A basic block is a segment of code containing no
branch or branch target.) Likewise, if a structured object is
used often, a pointer to it will be moved into a register. The
Gypsy compiler need not duplicate this effort of analyzing and
allocating register usage.

Assuming the Gypsy translator will let the Bliss compiler
handle PDP-11 register allocation, it still can make a number of
choices about the allocation of its variables. The Gypsy
semantics say that local variables are allocated and destroyed on
routine entry and exit, but this need not be so in the Bliss
implementation.

If a routine is not recursive or concurrent, then its locals
can be statically allocated. This implies that the memory is
reserved for the entire execution of the program instead of being
allocated only while the routine is executing. This may make the
difference between a program fitting into the LSU-11 or not. It
is possible for a program's worst case space requirement to be
less than the sum of its local variables, depending on possible
calling sequences. On the other hand, data references are
marginally more efficient if the data is statically allocated,
and no code must be executed for stack management.

3.3.2 Gypsy Parameters

Bliss parameters may be in registers, on the stack, or in
fixed memory locations, as specified by a LINKAGE declaration.
If one specifies that a parameter is stored in a register instead
of on the stack, this saves overhead at the call site, but not
necessarily in the routine body. If a parameter is accessed
often, the Bliss optimizer will quickly copy it to a register.
If a register is used in a parameter linkage it cannot be used as
a temporary at that point in the calling routine, or saving and
restoring it will be just as expensive as the default stack
parameter linkage.

Gypsy does not have global variables, so parameter lists
tend to be long. It is often the case that a data structure is
declared at the beginning of the program, then is passed to
almost every routine in the program. If it can be found at
compile time that every actual corresponding to a formal is
traced back through the calling sequence to the same local
variable, that local can be statically allocated and treated as a
global. Good candidates for this optimization are the parameters
and locals in procedure Main, especially the buffers.

Bliss allows parameters to be passed either by value or by
reference, if they are simple types. All Bliss parameters must
fit in a single memory word. This means all structured Gypsy parameters must be passed call by reference. That is, a pointer to the object is passed. Gypsy CUNST parameters of simple types are passed by value (that is, the actual value of the variable is passed) to prevent unnecessary accessing overhead in the called routine and allow Bliss constants to be passed without generating a temporary.

### 3.3.3 Descriptors for Variables

The runtime implementation of a variable may contain more than its value. For debugging, it will be useful to have pointers to both the variable's type descriptor and its print name physically present with the variable.

There are two cases where the type of a variable may change. The first is during a formal/actual parameter mapping. The base types (Gyppy Manual [22], pg 49) of the formal and actual are the same, but the formal may have subrange or dynamic size restrictions. ValueError checks in the called routine must be done with the type of the formal.

The type of a local variable may change between invocations of a routine, if it is a subrange type. The bounds on a subrange may be specified at runtime by the value of a parameter, as

```plaintext
procedure foo(Lo, Hi: int; ...)=begin
  Var Li: (Lo..Hi); ....
```

This is the only kind of parameterized type in Gyppy. The runtime subrange cannot be applied to parameters, array index bounds, subfields of structures, or dynamic size restrictions. If runtime descriptors are maintained for simple types, then these ranges will have to be filled in at runtime. These parameterized subranges also require more compile time information to be maintained.

### 3.3.4 Initial Implementation for Data Allocation

All local variables are allocated as Bliss Locals. This allows Bliss to place them in registers when it can, otherwise on the stack. Gyppy CUNST parameters of simple type are passed by value, and all others are passed by reference. No runtime done vectors are provided.

The Bliss temporary for Gyppy functions returning structured types will be allocated in the calling routine. A pointer to the result will be passed into the Bliss routine as the first parameter. The pointer will be passed back out as the Bliss result, to preserve the functional notation at the Bliss call site.
3.4 Integers

3.4.1 Integer Subranges

The Gypsy language definition and proof methods allow the
use of the full infinite set of integers. The compiler
implementor is free to restrict which subranges of integer are
allowed in the implementation. For the PDP-11 subranges within
\[ -2 \ldots 2 -1 \] fit in a single machine word and are directly
supported by the arithmetic hardware. Other possible subranges
of integer are

\[ \begin{array}{l}
\texttt{posint} = \texttt{integer}(0 \ldots 2 -1) = \texttt{unsigned word}, \\
\texttt{bigint} = \texttt{integer}(-2 \ldots 2 -1) = \texttt{double word}.
\end{array} \]

3.4.2 Arithmetic Overflow

Gypsy semantics require that arithmetic overflow be
detected, but Bliss-11 does not generate the checks. It is
presently not clear how this problem will be resolved in full
generality. It is anticipated that it will be necessary to lay
down in line assembly language or use assembly language out of
line support routines.

The method shown here was suggested by Bruce Leverett. It
uses the assembly language Opcode/Oplabel facility of Bliss-11
[37]. The following macro takes three addresses: returns the
sum of the first two, and puts True or False in the third address
depending on overflow (True if it overflowed):

\begin{verbatim}
macro sumov (addend, augend, ovflag) =
begin
  local sum;
  opcode ADD, RVC;
  oplabel !;
  ovflag = False;
  sum = .addend;
  ADD (augend, sum);
  RVC (L);
  ovflag = True;
  li .sum
  end $;
\end{verbatim}

No matter how the overflow checks are generated, they are to
be avoided if possible. The optimizer can use interval
arithmetic, subrange bounds or both to be sure overflow will not
occur.
4.3 Initial Implementation of Integers

The initial implementation restricts integer subroutines to 16-bit, single-word PDP-11 integers. Overflow checks are not performed.

5 Summary

This chapter describes the Bliss image of a subset of Gypsy features which are currently implemented in the Gypsy to Bliss translator. The initial implementation was surprisingly easy, because a choice was made for the initial translator implementation, emphasis was placed on simplicity over efficiency. Very little effort was made to optimize or format the output Bliss code. For example, the ELSE arm of a Gypsy IF statement is generated as "IF TRUE THEN ... ." In most cases no further low-level optimizations will be necessary, because the Bliss compiler seems to produce good code for whatever Bliss source it compiles.

Two features will need to be considered further. Bliss' condition handling mechanism requires runtime overhead in the usual case where there is no condition signalled. The overhead is especially bad for condition parameters. This overhead may be avoided at a future time by using a method independent of the Bliss ENARLF and SIGNAL.

The other feature not currently handled acceptably is arithmetic overflow checks. Both condition handling in general and the overflow checks can make use of assembly language macros.
CHAPTER 4
RUNTIME DESIGN FOR FUTURE IMPLEMENTATION

The Gypsy language features discussed in this chapter are not included in the initial implementation, but will be added incrementally. As in the previous chapter, issues in translating these features into Bliss are discussed. While the previous chapter prescribes the the choices made in the implementation, this chapter usually only gives alternatives.

4.1 Structured Data Types

Gypsy's data structures are arrays, records, sequences, sets, mappings, and message buffers. Each structure can be made up of elements of the other kinds. (The language definition does not prohibit types from being recursive, but the current implementation of the parser does not allow it. Sequence types could be recursive, with empty sequences at the leaves.)

4.1.1 Arrays, Records

The translation of non-nested arrays and records is straightforward. Record field names can be compile-time bound to access offsets, then "rec.field" can be translated to "rect(field)" in Bliss.

Gypsy arrays can be implemented as Bliss structures with bounds checking specified in the access algorithm. If the bounds check is imbedded in a test, the Bliss compiler can optimize out the check. The bliss-11 data abstraction for the Gypsy fragments

```plaintext
TYPE vec = array( (MIN_i .. MAX_i) ) of int;
VAR V: vec;
V[i] := V[MIN_i + i];
```

would be
STRUCTURE vec[1, chk, cl =
((maxi - mini) * 2) ! allocation size
! access function
IF chk and
! constant IF optimized out
((.1) LSS mini
OR maxi LSS (.1))
THEN
image_of_signal(c)
ELSE (.vec + (.1) * 2);
LOCAL v: vec;
V[.1, True, IndexErr] = V[mini, 0, 0];

Using Bliss-11 ENABLE and SIGNAL, "image_of_signal(c)"
becomes "SIGNAL c," with the desired results. Bliss data
structures are expanded in line as macros, so no loss in
efficiency will result from using them over generating the code
directly.

1.1.2 Dynamic Types: Sequences

The implementation of the Gypsy dynamic data types is of
extreme importance to the compiler because of its use of dynamic
memory. It is possible to build the other dynamic types from
sequences: sets can be ordered sequences, mappings a pair of
sequences, and buffers can be three sequences, for queues of contents
and blocked processes. Therefore, the discussion here will be
limited to the implementation of sequences.

The sequence element accessing operations can be implemented
in Bliss with structure declarations. The other operations will
have to be expanded in line or implemented as support
subroutines.

The language designers had linked lists in mind when they
defined the sequence operations, but this need not be so in the
The previous compiler, for example, used an array of pointers in
the sequence header. An array plus counter approach has been
advanced for the current compiler to postpone the implementation
of dynamic memory. Each of these methods makes accessing easier
than with linked lists, at the expense of more difficulty in
inserting new elements.

As pointed out in Pratt [30], access to variable-length
data objects such as Gypsy sequences are usually first, next, and
last. Random accessing is used less often, so the random
accessing can be implemented less efficiently. The Gypsy
compiler should try to recognize when a sequence access in a loop
is actually a next, and generate more efficient Bliss code.

It is not necessary for all sequences to have the same
ternal structure. Depending on the use made of a sequence, various implementations will be more efficient. A Gypsy sequence can implement a user's stack, a queue, or an array, so different sequence operations may be executed less often, or never, on the structure. The optimizer can decide on a type by type basis, or in for individual variables, what implementation will be the t.

3 Nested Structures

Gypsy nested structures are slightly more complicated to implement in Bliss than single level structures. The Bliss structure access mechanism is triggered by a variable name, as "i,j,IndexError" above. This is not possible, for example, in an array of sequences. To access the i'th sequence would be "i]", but accessing the j'th element of that sequence would be [i,j][j]." There is no variable name to trigger the structure access into the sequence. Bliss allows these sessions which return structures to be accessed by using the structure name. If the sequence type above were named Seq, then the example A[i,j] would be translated into Seq[A[i,j]], .j.

Dynamic Memory Allocation and SpaceError

Gypsy does not allow the programmer to use explicit pointers indirectly access data. The Gypsy dynamic types, sequences, sets, and mappings, were designed to allow the programmer the ability to allocate and return data space dynamically without explicit use of pointers. This section discusses some of the issues in the support for dynamic memory allocation.

The most important point to be made about dynamic memory allocation is that it is not strictly necessary for many tasks. The compiler could implement Gypsy dynamic types by locating the maximum space requirement and keeping a count of current contents. Concurrent activations can be statically located, as discussed in 4.4.1. Unbounded dynamic types would have to be omitted from the implementation, but it is not reasonable to limit lengths of sequences since available memory may be limited. In order for a program to execute properly, the worst case space requirement must be available. For complete reality, though, the dynamic allocation is necessary. Also, a program whose worst case needs exceed the available memory can perform work done in the normal case.

When a Gypsy program tries to allocate space for a new element, and there is none available, the condition SpaceError is signalled. A major issue is what the runtime system should do when SpaceError occurs. Typically, it is not able to recover from SpaceError in the Gypsy source, so it is necessary to implement the support will take some action before signalling the error.
The support could handle SpaceError by suspending the process, until some other process frees up space. If this never occurs, the program will block infinitely, leaving the user wondering what has happened. The support should print a message once it suspends or after a time out.

If the support cannot recover from a SpaceError, it should at least allow the programmer to perform an autopsy. On an LSI-11, it can at least halt and invoke the microcode ODT, which primitive decommitting capabilities. Tools should be written to analyze the contents of the dynamic store, either by loading an analyzer over the Gypsy code in the LSI-11 or by dumping the memory into a machine with mass storage for processing.

The dynamic memory system should also be instrumented. A minimum instrumentation would dump the entire dynamic memory on every operation. This is easy on a smaller system as long as disk space permits. More sophisticated analysis routines can be built into the support. Knowledge of the behavior of the dynamic memory can have a major impact on the allocation strategy chosen.

3. Gypsy Concurrency in Bliss

Gypsy supports concurrency. Like the rest of the language's features, the concurrency is carefully controlled. Although concurrency was not included in the initial implementation, its inclusion had to be planned for. It will have an important impact on the design of the compiler and a Gypsy program's time structure.

3.1 Image of Cobegin in Bliss

The bliss coroutine mechanism is primitive, but Gypsy process management can be expressed with it rather elegantly, as proposed Bliss image of Gypsy concurrency will be outlined in general terms. The specifics are discussed elsewhere [18]. Much of the basic analysis for the implementation of concurrency was done by John McChuag.

The image of a COBEGIN will depend on three support routines: allocate, schedule, and suspend. An allocate is essentially a bliss CREATE with additional Gypsy-specific initializations. Schedule places an allocated activation record on the ready queue. Suspend selects a process from the ready queue and does an bliss EXCHJ. The image of a COBEGIN, then, is straightforward. The parent process allocates and schedules its children, then suspends. The last child to terminate reschedules the parent.

To do process management requires some means to select the next process to be activated. It is proposed that a first-come-first-served scheduler with a single ready queue is efficient for scheduling of Gypsy processes.
4.3.2 Buffer Implementation

A Gypsy buffer will contain two waiting queues - one of the processes blocked waiting to SEND, the second of the processes waiting to RECEIVE. In addition, the buffer will have a third queue for the objects passed through it.

The implementation of the buffer types will use the support routines schedule and suspend. If a process blocks on a SEND or RECEIVE, it suspends. When a second process unblocks the first by doing the complimentary buffer operation, then the first is unblocked by scheduling it.

There is an important runtime optimization when a process unblocks another on a buffer operation. Since the two processes have met at the buffer, the data transfer can be direct, without copying it through the buffer.

4.3.3 Await

An AWAIT is logically analogous to a BEGIN in a COBOL. The last arm of a BEGIN to terminate reschedules the parent process. The first arm of an AWAIT to begin execution after unblocking on its buffer operation must kill its siblings before any of them can begin to execute, then reinvoke the parent. If more than one AWAIT arm is scheduled by the complimentary buffer operation, the system may deadlock unless implemented with care. Because only one of the arms is really executed, a blocked process queued behind the second AWAIT arm may never be scheduled. Furthermore, if a process suspends itself on a buffer queue unnecessarily, that is when the buffer is receptive to the operation, then the process may never be awakened. For example, if the process is the only one receiving from the buffer, and suspends even though the buffer is full, no sending process can unblock. This further complicates the implementation of AWAIT's. With preemptive scheduling, the set of receptivity tests must be made in a critical region.

In reality, the arms of the AWAIT need not be allocated as separate processes. The AWAITing process can place itself on all of the buffer queues, then suspend. The routine schedule must recognize when the scheduled process is in an AWAIT and remove the process from the rest of the buffer waiting queues.

4.4 Stack Overflow

Bliss-11 never generates code to check for stack overflow. For a non-concurrent process the hardware interrupt, when the stack crosses location 400, is sufficient. This is not the case for concurrent processes, whose stacks can be anywhere in memory, but Bliss ignores the problem. For a memory mapped machine, the interrupt is sufficient, since each process can have its personal location 400. The LSI-11, however, is not a memory
The Gypsy implementation has several alternatives for dealing with stack overflow of concurrent processes.

Gypsy programs could follow Bliss' example and ignore the problem. Since Gypsy was designed for secure and reliable software, this is not appropriate. The following sections discuss other alternatives. There are primarily two alternative strategies: 1) preventing stack overflow by maximal worst case allocation, and 2) run time detection and recovery from stack overflow.

4.4.1 Static Allocation of Activations

If the stack needs could be determined at compile time, then the entire worst case need could be allocated for each process, and the code could run correctly with no runtime stack checks. It is noted that the processes don't usually all need their maximum amount of space simultaneously, so statically allocating each their maximum may prevent a runnable program from running. However, a program that cannot continue under its worst case behavior is not trustworthy. Also, allocating the entire need for the life of the process recovers memory space by doing away with code for runtime checks and dynamic space allocation routines.

With the program's calling tree available in the Gypsy Verification Environment data base, it is often possible to determine the maximum stack space a program needs. The worst case need of a process is the sum of the worst case needs of the routines in the worst case calling sequence.

The data space requirement for a routine is the worst case sum of

1. the sizes of Gypsy variables
2. temporaries generated by the Gypsy to Bliss translator
3. temporaries generated by the Bliss compiler,
4. the largest call site in the routine (space for pushed parameters, return address, etc), and
5. the worst case needs of the called routines.

Determining these values requires detailed knowledge of the machine code generated by the Bliss compiler. This requirement could be constructed as an expression of Bliss compile time identifiers, for example
\[ \text{size} = 1 \ast \text{tufersize} + 1 \ast \text{integersize} + \ldots \\
+ k \ast \text{parameter-overhead} \ldots \]

This would be convenient for translator development, because the runtime sizes of objects may change. The Bliss-II special function `allocation(varname)`, which returns the byte size of the variable, will be useful.

Of course, some calling structures are easier to analyze than others. A particularly difficult case for finding stack needs is recursive routines. The maximum calling depth must be determined at compile time. In general, determining the maximum recursion depth is equivalent to the halting problem (Hopcroft and Ullman [27], page 102), and is undecidable. The compiler can use the interactive theorem prover, user-supplied assertions, and the user, to try to show that there is a bounded, monotonically decreasing function of the parameters at each recursive call that shows that the recursion will stop. Until the sophisticated methods are built into the compiler, it could refuse to handle recursion in concurrent programs, or obtain a bound on recursion depth from the user and generate code to validate it.

4.4.2 Detecting Stack Overflow in Bliss

It is essentially impossible in Bliss to check the stack top each time new space is allocated. The Bliss compiler allocates local variables automatically on routine invocation, then creates and destroys temporary and call sites as it pleases. This causes the stack and its associated space requirement to change during the execution of a Bliss routine. Furthermore, because of low-level optimizing transformations, it may be difficult to trace a space requirement to a specific location in the Bliss source.

It is feasible to begin every routine with the Bliss code

```plaintext
if (SP = fudgetactor LSS stack_limit("this process"))
then B00M();
```

(SP is Bliss' predefined name for the stack pointer register. The tests are LSS instead of GTR because the stack grows toward smaller memory addresses.) If this test succeeds, then the stack top has grown out of its defined area into other data or code space. If a temporary is pushed on the stack, then the other data will be changed unpredictably. Such a temporary, for example, may be created by Bliss for the testing expression. In addition, if an interrupt occurs to occur, its pushed return context will undetectably change data in the same manner.

Using this method for detection of stack overflow, it is
very difficult to recover. The Bliss compiler would have to be very carefully inhibited from generating any stack temporaries for the stack test, and interrupts would have to be inhibited during every routine call context switch. A mechanism to recover from the stack overflow could not generate any temporaries on the stack, including the return address pushed by the jump to subroutine instruction. The best response in this situation is to abort the program with an error message.

In order to reasonably recover from stack overflow, the situation must be detected before the stack pointer is moved. This can be done as a part of the routine call mechanism. This requires every caller to know the callee's space requirement and generate a test at each call site

```c
if .SP - callee's_space_need
  USS stack_limit("this process")
Then recover_before_stack_overflow();
callee(P1...Pn);
```

If `stack_limit("this process")` and `recover()` are truly Bliss routines, then the test requires seven extra PDP11 instructions per call site, taking fifteen memory words.

The test can be done by an intermediate Bliss routine which has the callee as a parameter. This intermediate routine may have other functions, such as counting routine invocations, timing, or printing a trace. The new call site would be

```c
Gypsy_call_test(callee, callee_space_need);
callee(...);
```

and the routine `Gypsy_call_test` would look like

```c
routine Gypsy_call_test(callee, callee_space) = begin
  if .SP + call_tests_space = ..callee_space
    USS stack_limit("this process")
  then recover();
  instrument...
  debug...
  end;
```

The extra overhead per call site shrinks to six words, but the routine `Gypsy_call_test` is larger and slower. The routine `recover` is more complicated because the call site for `Gypsy_call_test` is on the stack.

Defining a special assembly language routine linkage is a
good possibility here. See the HYDRA linkage specified in [37].

4.4.3 Recovering from Stack Overflow - The "Cactus Stack"

If stack overflow in a concurrent process is detected after the fact, then recovery is extremely difficult, as discussed in the previous section. The only safe action to take in this case, and the easiest implementation, is to print an error message and exit, or to enter the debugger.

If the overflow is detected before the stack pointer actually moves outside its bounds, then the program can recover and continue. A new segment of memory must be allocated and linked in so that the new segment contains the stack top. When the routine returns and "underflows" the new segment, it must be unlinked and returned to the dynamic pool, and the stack context switched back to the previous segment. When a set of concurrent processes overflow their stacks in this way, the resulting mass of stack segments and link pointers is known as a "cactus stack." If there is no memory space left for the new segment, the process can at best suspend and hope some other process will free up space.

4.4.4 Implementation Choice

When concurrency is implemented in the bliss translator, it will not have full generality at first. The stack needs for each process will be estimated and statically allocated. Stack overflow will be detected at the call site by an assembly language linkage similar to Gypsy-call-test, but recovery will not be attempted.

4.5 Preemption

It may be necessary for the run time support to be able to force a process to give up control. The Gypsy manual states that all arms of the COMPETE "are executed concurrently" (Gypsy Manual [22], page 136), as if each were on its own processor. (This is often the case, as the language is intended for distributed communications processing.) For multiprocessing, this requires preventing any one process from monopolizing the processor.

Preemption necessitates the defining of critical regions where preemption is inhibited, in which the process queues and activation records are being examined and changed. Extreme caution must be taken not to allow interrupt windows to leave the queues inconsistent when an interrupt occurs.

The bliss mechanisms assume that the processes will voluntarily give up the CPU in some reasonable time. It is possible to have the timer interrupt force a process to suspend at any (non-critical) point. Knowledge of the PDP11 interrupt mechanism is required by the interrupt routine, so the code
cannot be machine-compatible, but the fudge is localized.

A jump to subroutine instruction in the PDP-11, "JSR PC, ROUT," places the return address on the top of the runtime stack. An interrupt operates on the stack as a sort of externally caused subroutine call. Within the interrupt handler, the stack top is the return address in the interrupted code. Unlike normal routine calls, though, the second word on the stack is the processor status word of the interrupted code. The timer interrupt can force a process to suspend by making the stack look like the process had executed a subroutine call to the routine "preempt" just before the interrupt occurred. Here is a before-and-after picture of the stack showing the changes made by the timer interrupt:

<table>
<thead>
<tr>
<th>BEFORE</th>
<th>AFTER</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack top</td>
<td>interrupt return</td>
</tr>
<tr>
<td>PSW</td>
<td>stuff</td>
</tr>
</tbody>
</table>

When the interrupt exits, the interrupted process will execute exactly as if a "JSR PC, preempt" were inserted in the code at the point where the interrupt occurred. This places some restrictions on the calling sequence used by the preempt routine. It may not have any parameters, or the timer interrupt must be able to manufacture them on the stack. The preempt routine must be called by the Bliss standard linkage of using the program counter in the JSR instruction. In reality, the interrupt return address will not be strictly on the stack top if the Bliss compiler uses the stack for local variable storage in the interrupt handler. Also, a Bliss-11 routine call returns its result in R0. The preemption routine must insure that the value of R0 on return is the same as on entry.

The timer interrupt will have a global variable which is the time until the next preemption. The suspend routine resets the countdown each time a new process is invoked. As mentioned, the program may fail if a process is preempted in a critical region. Resetting the time to preemption guarantees the current process the processor for a period of time. A second boolean global variable will inhibit the timer interrupt from preemption the current process.
CHAPTER 5
TRANSLATION METHOD - THE TABLE DRIVER

The Gypsy to Bliss Translator is a module in the Gypsy Verification Environment. The user specifies to the G.V.E command interpreter the name of a procedure to be the entry point for the program, and the translator analyzes the calling sequences and extracts the necessary unit definitions from the database. The program is then translated into Bliss and left in a file to be compiled by the user with the Bliss compiler.

It is a LISP program based on a keyword-driven table driver. The table driver, while written primarily for the translator, is intended to be of general utility for programming other modules of the G.V.E. The implementation of the table driver is an expansion of the method used by Xeval (section 1.2). Knuth's attribute grammars ([30, 31], and section 1.1.5) have influenced the design of the table driver in a very general way.

5.1 Arguments

The arguments to the table driver are abstractions. The table driver uses abstraction extraction and modification functions that hide the concrete representations. A call to the table driver is

(tab-drv NODE CX FUNC).

The parameter NODE is a node (a subtree) of the internal representation of a Gypsy program's abstract syntax tree, Prefix. The table driver uses the Prefix abstraction functions ABSP-OPR and ABSP-ARGLIST to decompose the parameter NODE, and no assumptions are made about the concrete representation. The table driver can process any LISP S-expression that is acceptable to the abstract syntax. As a result of the implementation of the abstract syntax, the programmer-supplied routines called by the table driver must expect Gysys abstract syntax, not parser Prefix (see section 2.2).

CX is a context abstraction provided by the programmer. It can be loosely considered to be Knuth's inherited attributes. The table driver uses programmer-supplied routines from its table to construct a new context at each level as it recurses through the argument NODE. Different uses of the table driver can have completely different context abstractions.
**FUNC** is an identifier (a LISP atom) specifying the "table-implemented function" to be applied to the program node. The most important example for this document is the translator. Other table-implemented functions are cross-reference analysis, analyzing Gypsy programming style, and optimizing.

### 5.2 Operation

A procedure applied to Prefix via the table driver is known as a table-implemented function. There are logically a set of tables, one for each of these procedures. The information in the table is initialized by the programmer, and is extracted by the table driver to direct it in processing the program. The table to be used is selected by the parameter **FUNC**. Each table specifies a distinct operation, for example translating to Bliss, optimizing, etc.

Here is a simplified high-level algorithmic description of the operation of the table driver:

```plaintext
routine tan_drvt(node, cx, func)=begin
   local row, newcx, results;
   row := get_row(abs_opr(node),
                   select_table(func));
   newcx := apply_ops(beforeops(row),
                       node, cx);
   results :=
      if recurseflag(row) then
      for each arg in abs_arglist(node):
         tan_drvt(arg, newcx, func)
      else abs_arglist(node);
   results := apply_ops(afterops(row),
                       node, cx, results);
   return(results);
end;
```

The operation of the table driver is in four basic steps. The zeroth step is to extract the row from its table for the **FUNC** specified and the operator extracted from the node. The row is indexed by (ABS-OPR NODE), a class name of which the operator is a member, or a default for **FUNC**.

The first logical column in this row is the **BEFOREOPS**. The names of the beforeops are extracted from the table, then are applied to the node and the context to create the new context for the processing of the node's arguments. Creating the new context is the first step in the table driver. This new context contains the attributes to be inherited by the arguments (children in the abstract syntax tree). Currently all the children get the same context. Information intended for only one of the children must
be encoded by the context abstraction. This mechanism may be changed if it turns out to be inadequate.

Each BEFOREOP has an argument list of the form

(beforeop NODE CX)

and returns a structure of the form

(NEWNODE NEWCX).

The significance of this structure is explained below.

The second step of the table driver is to process each of the arguments of NODE recursively. The recursive calls are passed the NEWCX produced by the beforeops. This step is under the control of the second column in the table, the RECURSEFLAG. If the flag is set, the table driver applies itself to each of the arguments (via a LISP MAPCAR). The results of the recursive call are formed into the list RESULTS. If the RECURSEFLAG is NIL, the intermediate result is set to the argument list. This may or may not be useful, depending on the table-implemented function. If the table-implemented function returns a new form of the Prefix, this gives no transformation.

For certain operators, recursing is meaningless. For these operators, the table driver ignores the user specified flag and does not recurse. Examples are variable names and symbol tables.

The last step in the table driver, and the last column in the table, is the AFTEROPS. The afterops are applied to the original node NODE, the original context CX, and the intermediate result RESULTS, to construct the final result of this call of the table driver. This result corresponds to Knuth's synthesized attributes. Again, the table driver does not know or care about its internal structure. The NEWRESULTS of the afterops is the result of the table driver.

Each AFTEROP has an argument list of the form

(afterop NODE CX RESULTS)

and returns a structure of the form

(NEWNODE NEWCX NEWRESULTS).
The beforeops and afterops are treated identically, except that the afterops have one more argument. The programmer can specify a list of routine names to be used as a node's beforeops or afterops. The list of ops are applied in sequence, with the returned structure from one used as the arguments of the next. Thus, the OPS application yields a final result by normal function composition of the specified functions. The parameter RESULTS of the first afterop is the intermediate result of the recursion step.

If any op returns a NEWNODE that is not identical to the NODE passed into it, the table driver immediately aborts and starts over at step zero with the new structures NEWNODE and NEWCX, and the same FUNC. This is especially useful for the optimizer. For example, a beforeop can decide that a CASE statement can degenerate to an IF. The table driver would take the IF and start over instead of recursing and applying inappropriate CASE afterops to the IF.

There is no prohibition on calling the table driver explicitly from within a beforeop or afterop. For example, a beforeop can call a different table-implemented function to construct an attribute to include in the children's context. An afterop can explicitly contain recursive calls on the table driver to process the arguments instead of, or in addition to, the automatic recursion specified by RECURSEFLAG. In this manner, the afterop can process the children in a specific order, pass a different context to each of the children, or pass information from part of one child's RESULTS into the context of one of its siblings.

An appendix (A) show a trace of the execution of the table driver.

5.3 Initialization

The programmer uses a set of initialization routines to fill in the table. The functions fill in the rows of the table and specify default actions.

The most important initialization call is

(tab-handles-init KEY FUNC BEFORES RFLAG AFTERS)

This call fills in one row of the table. KEY is an operator in the abstract syntax (corresponding to (ABS-OPR NODE)), and FUNC specifies one of the table-implemented functions (such as translating). RFLAG is the recurseflag, BEFORES and AFTERS are lists of handler routines to be applied by the beforeop and afterop steps discussed above. The op lists may be one of three forms:
1. If NEWCX then the identity operation is performed; for beforeops, NEWCX is CX; for afterops, NEWRESULTS is RESULTS.

2. If the argument is a list of oos, then they are applied pipeline fashion as discussed above.

3. If the argument is an atom, it is treated as if it were a list of one element.

For programmer convenience, five predefined afterops are available to be used in calls to tab-handlers-init:

1. TAB-AFT-NIL returns NIL no matter what its arguments are. This afterop is useful if the table-implemented function operates by side effects.

2. TAB-AFT-IDENTITY returns its arguments. Specifying NIL for the AFTFHS in a call to tab-handlers-init defaults to this routine.

3. TAB-AFT-ERROR generates a LISP BREAK if called. This is useful for debugging.

4. TAB-AFT-RESTORE assumes its RESULTS parameter is the argument list for a new node of a program tree. The operation of the table driver assures that RESULTS has the same number and ordering of arguments as the argument list of the original node. Tab-aft-restore uses ABS-GRPR and ABS-MK to put the original operator back onto the RESULTS argument list. Any descriptor on the original node is lost, as specified by the description of the abstract syntax.

5. TAB-AFT-DESCRIPTOR uses the abstraction functions to replace the original operator and its descriptor on the RESULTS argument list.

Tab-handlers-init initializes one row of the logical table. It is often desired to duplicate rows in the table for several operators. For example, all the <binary-op>'s could be treated identically. This can be accomplished by a call

\[ \text{(tab-opclass-init CLASSNAME FUNC KEYLIST)} \]

CLASSNAME is an identifier (an atom) which is to be the name of an operator class. It may or may not be an actual abstract syntax operator; if not, the table is extended. FUNC specifies the table-implemented function for which this operator class is to
apply. KEYLIST is a list of keywords. It may include the names of other operator classes. When the programmer calls the initialization function

(tap=handlers-init CLASSNAME FUNC ...)

It has the logical effect of duplicating the call

(tap=handlers-init KEY FUNC ...)

on each key in the list KEYLIST.

These two initialization functions check for multiple initializations. Handlers-init takes precedence over opclass-init. That is, if a call on opclass-init specifies a keyword to be in a class, then a later call to handlers-init specifies specific handlers for the keyword, the opclass definition is over written. Likewise if a call to opclass-init finds individual handlers are already specified for a key, the key is omitted from the operator class.

However, redefining the handlers for a keyword, or placing the keyword in more than one operator class, is normally considered an error, and the initialization function will enter the break package. The break command OK allows the redefinition to complete. For debugging a table-implemented function, the variable TAP-DEBUG can be set to T. This allows the redefinition to take place without an error message.

Two operator classes are defined universally as if a call on opclass-init were called automatically for every table-implemented function. These classes are BINARY-OP and UNARY-OP.

The final initialization function has the call

(tap=funcdefault-init FUNC KEY).

Logically, this call is equivalent to creating an opclass named KEY with all otherwise unhandled keywords. If this default is not specified, the table driver will enter the LISP break package if an unhandled operator is found. This is detected at step zero (discussed above in 5.2).

An appendix (8) shows a log of the use of the initialization functions.
5.4 TAB-BOX

Two table-implemented functions are predefined. These are TAB-BOX and TAB-UNBOX, which convert between G.V.E parser Prefix and abstract syntax. Tab-box is a trivial table-implemented function because the table driver uses the abstraction functions. All the work of converting is hidden behind the calls in the table driver of AAS-OPR, ABS-ARGLIST, and ABS-DESC. A call on tab-box is simply

(tab-box NODE).

Tab-unbox converts back to parser Prefix. The result of converting to abstract syntax and back is not guaranteed to be an identity. The result will be syntactically correct and semantically equivalent, but may not Infprint (section 1.2) the identical source program.

5.5 Relationship with Attribute Grammars

The table driver is not an interpreter for attribute grammars, as in Hochmann and Ward (13). All functions applied to a node, that is at each production in the grammar, must be explicitly initialized by the programmer. By the default action of beforeops, recursion, then afterops, a node's inherited attributes may depend only on its ancestors, and its synthesized attributes may depend on ancestors or children. None of a node's attributes may depend on its siblings.

However, the programmer can achieve full generality by calling the table driver explicitly from within a beforeop or afterop. The inherited context of a node may depend on its siblings or its children if its parent has explicit calls on the table driver in its afterops. By calling the table driver explicitly with each child, synthesized attributes of one child may be passed down as inherited attributes of another. If an afterop calls the table driver more than once for a given child, then that child's inherited attributes may depend on its children's synthesized attributes. Thus

this method ... is as powerful as any conceivable method could be, in the sense that the value of any attribute of any node in a derivation tree may depend in any desired way on the entire tree. [30]

If the table driver restarts when an op changes NEWNODE, then it is equivalent to adding a new production

form(oldnode) ::= form(newnode)
to the grammar with appropriate semantic actions.

It would be very interesting to look into an automatic table initializer which would process a grammar annotated with Knuthian semantics. Since Gypsy abstract syntax has a very regular, context free grammar, such a grammar would be syntactically very simple. There are no plans to pursue this research.
CHAPTER 6
CONCLUSION

The initial subset of Gypsy which has been implemented is small but important. The translator has processed a number of Gypsy programs, most of which do nothing useful, but contain a variety of syntactic and semantic language features. A few small programs have been translated, compiled, and run under the Unix operating system. An appendix (C) shows examples of the output of the translator.

The translator has not been fully integrated into the Gypsy Verification Environment. To use it, one must drop from the G.V.E. command scanner down into LISP and directly execute the LISP functions. The full integration is not predicted to be difficult.

The decision to translate into Bliss was a very good one, translating from one high-level language to another vastly simplifies the procedure. However quick this implementation is, it is not dirty. From the straightforward, simple minded Bliss structures generated by the translator, the Bliss compiler outputs amazingly compact assembly language code.

The table driven algorithm was also a good choice. Once the table driver was designed and the problems with Prefix were worked out, the work on the translator was smooth and incremental. The table driver is promising to be useful for optimization analysis and other functions in the system.
APPENDIX A
A LISP TRACE OF THE TABLE DRIVER

This appendix shows an annotated trace of the operation of the table driver. The table driver is applied to a Gypsy CASE statement to translate it into Bliss. The functions traced are:

- `tap-drv` - The table driver
- `trn-before-case` - Beforeop for CASE.
- `trn-case` - Afterop for CASE.
- `trn-case=arms` - Afterop for a set of CASE arms.
- `trn-case=arm` - Afterop for a CASE arm.

The case arms do not require beforeops.

```
Exec-> translate trace.gyp

scope trace=begind
procedure x(var a: int)=begind
case a
  is 1,2: a := 44;
end;
end;
end;
```

No syntax errors detected
No semantic errors detected

```
Exec-> lisp
Entering Lisp ... Type (RETURN) to exit.
```

```
!(sprint (setq thecase (tag-box (caadr (cadr (assoc !
  'body (cdr (gett "x":trace "fullprefix)

(*CASE=STVT (*VALUE=REF A))
(*CASE-ARMS (*CASE-ARM (*CASE-LABELS 1 2))
(*STMSTLIST
 (*ASSIGN
  (*VAR=PLACE=REF A)
  (*VALUE=REF 44))))))

(*STMSTLIST (*SIGNAL (*COND-ID CASEERROR)))

NIL

!(trace tab-drv trn-before-case trn-case
   trn-case=arm trn-case=arms)

(!; trace the table driver
   plus the beforeops and afterops for CASE.

```
tao-drvg The table driver.
trn-before-case Beforeop for CASE.
trn-case Afterop for CASE.
trn-case-arm Afterop for a case arm.
trn-case-arms Afterop for set of arms.
Case arms don't need beforeops.

特朗=print (translate the case)
Enter TAB-DRV:

NODE =
(*CASE-STMT (*VALUE=REF A)
 (*CASE-ARMS (*CASE-ARM (*CASE-LABELS 1 2))
 (*STMTLIST
  (*ASSIGN
   (*VAR-PLACE=REF A)
   (*VALUE=REF 44)))))
 (*STMTLIST (*SIGNAL (*COND-ID CASEERROR))))

CX = NIL
FUNC = TRN
Enter TRN-BEFORE-CASE:

(*CASE-STMT (*VALUE=REF A) ... )
CX = NIL
TRN-BEFORE-CASE =

;;; The value of BEFORE-CASE is a context
which flags this Gypsy CASE will be translated as a
Rliss CASE, and saves the range of the arm labels.
From now on, these CASE contexts will be
replaced in this log by "( { CASE CONTEXT } )," }

(*CASE-STMT (*VALUE=REF A) ... )
(CASE=[IS-CASE , T)
 (CASE-MAX=VALUE . 2)
 (CASE-MIN=VALUE . 1)))
Enter TAB-DRV: ;; with the case expression.

NODE =
(*VALUE=REF A)
CX = ( { CASE CONTEXT })
FUNC = TRN
TAB-DRV = ;; result = Rliss parameter access "..A"

 Enter TAB-DRV: ;; recursing on the arms list. }

NODE =
(*CASE-ARMS (*CASE-ARM ... )
CX = ( { CASE CONTEXT })
FUNC = TRN
Enter TAB-DRV: ;; recursing on the single case arm

NODE =
(*CASE-ARM (*CASE-LABELS 1 2) (*STMTLIST ... ))
CX = ( { CASE CONTEXT })
FUNC = TRN
Enter TAB-DRV: ;; the labels on the arm. }

NODE =
(*CASE-LABELS 1 2)
CX = ( { CASE CONTEXT })
FUNC = TRN
TAB-DRV = (;; result = list of arm labels.)

(1) Enter TAB-DRV: {;; the arm statement list.}
    {;; Lower level ops not traced.}

    NODE =
    (*STMTLIST (*ASSIGN (*VAR=PLACE=REF A) (*VALUE=REF 44)))
    CX = ( { CASE CONTEXT })
    FUNC = TRN
    Enter TAB-DRV: {;; Assignment statement.}

    NODE =
    (*ASSIGN (*VAR=PLACE=REF A) (*VALUE=REF 44))
    CX = ( { CASE CONTEXT })
    FUNC = TRN
    Enter TAB-DRV: {;; LHS of assignment.}

    NODE =
    (*VAR=PLACE=REF A)
    CX = ( { CASE CONTEXT })
    FUNC = TRN
    TAB-DRV = {;; LHS access.}

(4)

Enter TAB-DRV: {;; RHS expression.}

    VALUE=REF 44)
    CX = ( { CASE CONTEXT })
    FUNC = TRN
    TAB-DRV = {;; result = RHS expression}

    VALUE=REF 44)
    CX = ( { CASE CONTEXT })
    FUNC = TRN
    TAB-DRV = {;; BLISS assignment stmt.}

    VALUE=REF 44)
    CX = ( { CASE CONTEXT })
    FUNC = TRN
    TAB-DRV = {;; result = Bliss compound expr.}

    BEGIN ((/. A) = 44 ;)
    TAB-DRV =

    BEGIN ((/. A) = 44 ;) END)
    Enter TRN-CASE-ARM: {;; afterop for a case arm}

    NODE =
    (*CASE-ARM (*CASE-LABELS 1 2) (*STMTLIST ... ))
    CX = ( { CASE CONTEXT })
    RESULTS =

    BEGIN ((/. A) = 44 ;) END)
    TRN-CASE-ARM = {;; expands and orders arms}

    CASE-ARM (*CASE-LABELS 1 2) (*STMTLIST ... ))

( { CASE CONTEXT })

    BEGIN ((/. A) = 44 ;) END)

    BEGIN ((/. A) = 44 ;) END)
    Enter TRN-CASE-ARMS:

    NODE =
    (*CASE-ARMS (*CASE-ARM ... ))

    CX = ( { CASE CONTEXT })
    RESULTS =

    BEGIN ((/. A) = 44 ;) END)

    BEGIN ((/. A) = 44 ;) END)
    TRN-CASE-ARMS = {;; result is CASE CONTEXT + BLISS.}
((CASE-ARMS ... )
  { CASE CONTEXT })
((BLISS ((1 (BEGIN ((/. A) = 44 ;) END ;))
         (2 (BEGIN ((/. A) = 44 ;) END ;))))
  (CASE-IS-CASE . T)
  (CASE-MAX-VALUE . 2)
  (CASE-MIN-VALUE . 1))
! TAB-DRV =
((BLISS ((1 (BEGIN ((/. A) = 44 ;) END ;))
         (2 (BEGIN ((/. A) = 44 ;) END ;))))
  (CASE-IS-CASE . T)
  (CASE-MAX-VALUE . 2)
  (CASE-MIN-VALUE . 1))
! Enter TAB-DRV: { ;; ELSE defaults to signal. }
! ! NODE =
(*STMTLIST (*SIGNAL (*COND-ID CASEERROR)))
! ! CX = ({ CASE CONTEXT })
! ! FUNC = TPN
! ! Enter TAB-DRV:
! ! ! NODE =
(*SIGNAL (*COND-ID CASEERROR))
! ! CX = ({ CASE CONTEXT })
! ! FUNC = TPN
! ! TAB-DRV =
(SIGNAL CASEERROR ;)
! TAB-DRV =
(BEGIN (SIGNAL CASEERROR ;) END)
! Enter TPN-CASE: { ;; afterop for CASE. }
! ! NODE =
(*CASE-STMT ... )
! ! CX = NIL
! ! RESULTS = { ;; list of argument results. }
((/. /. A) ( ;; case expression. )
  ( ;; result is BLISS + { CASE CONTEXT } . )
  ((BLISS ((1 (HEGT ((/. A) = 44 ;) END ;))
           (2 (BEGIN ((/. A) = 44 ;) END ;))))
    (CASE-IS-CASE . T)
    (CASE-MAX-VALUE . 2)
    (CASE-MIN-VALUE . 1))
  (BEGIN (SIGNAL CASEERROR ;) END)) { ;; Bliss for ELSE. }
! TPN-CASE = ( ;; result = image of Gypsy CASE. )
  ( ;; bliss CASE doesn't have ELSE, so the IF checks bounds using the bounds from the { CASE CONTEXT } . )
(*CASE-STMT ... )
NIL
(TF (/ . / . A)
  LSS
  1
  OR
  2
  LSS
  (/ . / . A)
  THEN
(BEGIN (SIGNAL CASEERROR ;) END)
;
(CASE ((/. /. A) = 1) {; expression normalized to zero.}
  OF
  SET
  ((BEGIN ((/. A) = 44 ;) END ;)
   (BEGIN ((/. A) = 44 ;) END ;))
  TES
  ;))
1AB=URV = {; result of translating the CASE.}
  ((IF (/. /. A)
    LSS
    1
    OR
    2
    LSS
    (/. /. A)
    THEN
    (BEGIN (SIGNAL CASEERROR ;) END ;)
  )
(CASE ((/. /. A) = 1)
  OF
  SET
  ((BEGIN ((/. A) = 44 ;) END ;)
   (BEGIN ((/. A) = 44 ;) END ;))
  TES
  ;))
{; trn-print formats the result }
IF . . A LSS 1 OR 2 LSS
  . . A THEN
BEGIN SIGNAL CASEERROR END ;
CASE . . A = 1 OF SFI
BEGIN
  . A = 44 ; END ;
BEGIN
  . A = 44 ; END ; TES ;
NIL
APPENDIX B

AN EXAMPLE OF INITIALIZING THE TABLE DRIVER

This log has been edited for brevity. Most blank lines and
NIL or irrelevant responses from the LISP routines have been
deleted. Human typing is in lower case and is preceded by the
LISP prompt "!". Machine responses are in upper case.

The example starts with an arithmetic expression TEST,
represented in Geyser Abstract Syntax. The two initializations
immediately following define the table-implemented function
"tab-box." Then additional initializations are made to
demonstrate their effect on the function.

(overcom
VSYS_INT LOADFL
SYSTEM LOADING
Small verification system 6.4 0:21 12-SEP-79
welcome...you may begin
Exec-> lisp
Entering Lisp... Type (RETURN) to exit.
!(dskin (comment.lap)(test.lsp)(inall.lsp))
FILES-LOADED

!(sprint test)
(*PLUS (*'INUS (*VALUE-REF AVAR))
  (*DIFFERENCE (*QUOTIENT (*VALUE-REF 1)(*VALUE-REF 2))
   (*VALUE-REF FO0VAR)))

!(tab-tuncdefault-init "tab-box "tab-boxdefault)
!(tab-handlers-init "tab-boxdefault "tab-box
  nil t "tan-attr-descriptor"
)
!(dexpr (doit () (sprint (tab-box test))))

!(doit)
(*PLUS (*'INUS (*VALUE-REF AVAR))
  (*DIFFERENCE (*QUOTIENT (*VALUE-REF 1)(*VALUE-REF 2))
   (*VALUE-REF FO0VAR)))

!{;; The expression is already in
  abstract syntax, so it's a no-op. }
!{;; define special handler for plus
!(tab-handlers-init "*plus "tab-box nil t "aft-+plus"
)
!(dexpr aft-+plus (p cx result)
  (list p cx
    (abs-mk "newplus result"

!(doit)
(NEWPLUS (*'INUS (*VALUE-REF AVAR))
  (*DIFFERENCE (*QUOTIENT (*VALUE-REF 1)
    (*VALUE-REF FO0VAR)))
;;; define special handlers for binary-ops.
! plus will not be changed since
! it's handled individually
!(tab-hndlers-init "binary-or" "tab-box nil t" "aft-binary"

!(dexpr aft-binary (p cx result)
! (list p cx
! ( aos=mk "a-binary-op result"

!(doit)
(NEPLUS (*MINUS (*VALUE-REF AVAR))
 (A-BINARY-OP (A-BINARY-OP (*VALUE-REF 1)
 (*VALUE-REF 2))
 (*VALUE-REF FOUVAR)))

;;; give *quotient a handler of its own
!(tab-hndlers-init "*quotient" "tab-box nil t" "aft-quo"

!(dexpr aft-quo (p cx result)
 (list p cx
 ( ans=mk "newquo result"

!(doit)
(NEPLUS (*MINUS (*VALUE-REF AVAR))
 (A-BINARY-OP (NEQUO) (*VALUE-REF 1) (*VALUE-REF 2))
 (*VALUE-REF FOUVAR)))

;;; define an opclass that includes primitive no-recurse to
! show the specified recursion is ignored
!(tab-opclass-init "newclass" "tab-box" (numeric-val minus))
!(dexpr aft-new (p cx result)
! (list p cx
! ( aos=mk "foo result"

!(tab-hndlers-init "newclass" "tab-box nil t" "aft-new"
 (TAB-HANDLERS T NIL (AFT-NEW)))

!(doit)
(NEPLUS (FOO (*VALUE-REF AVAR))
 (A-BINARY-OP (NEQUO) (FOO 1) (FOO 2))
 (*VALUE-REF FOUVAR)))

;;; remove minus from opclass by specifying handler
(tab-hndlers-init "*minus" "tab-box nil t" "tab-ait-restore"

(doit)
(NEPLUS (*MINUS (*VALUE-REF AVAR))
 (A-BINARY-OP (NEQUO) (FOO 1) (FOO 2))
 (*VALUE-REF FOUVAR)))

;;; make newclass a subclass
!(tab-opclass-init "tab-box default" "tab-box" (newclass ))
!(doit)
(NEPLUS (*MINUS (*VALUE-REF AVAR))
 (A-BINARY-OP (NEQUO) (*VALUE-REF 1) (*VALUE-REF 2))


APPENDIX C

GYPSY PROCEDURES AND THEIR BLISS

This log shows the translation and execution of a very simple Gypsy program. The log shows the Gypsy program, the Bliss image, and the current procedure for translation and execution. The first routine shown here is completely senseless, but contains a number of features. The second, ECHO, is run on Unix, as shown here.

Transcript begin ntyA pid 14745 Fri Jul 18 01:30:50 1980

UTEXAS Pwr Unix Host
login: lsmith
password:

% tr isie
Open isie Fri Jul 10 01:31:14 1980

ISI=SYSTEM=E, TUPS=20 monitor 3A(3141)=1
System shutdown scheduled for 21-Jul-80 22:00:00,
Up again at 22-Jul-80 05:00:00
@LOG LSUMTH
Job 57 on TTY153 17-Jul-80 23:31:21
End of LOGIN.COM.13
@overcom
VSYS.INI LOADED

Report problems to GYP SY=SYSTEM@ISIE via mail

Gypsy 2.0 Verification System 6.8V 22:22 16-Jul-80
Welcome...you may begin

Exec=> translate examp.gyp

scope main = begin

procedure P (var A: INT) =
begin
  var X: INT[1..10];
  var C: character;
  var BUF: TTYBUF;
  cond VALUEERROR;
  if A > 10
    then A := 10
  end;
  if A < 10
    then A := 11
    else A := -11
  end;
  A := X;
x := A;
A := (x + 1 div 10) = 3;
loop
  A := 100;
  leave
when
  is VALUEERROR : A := 0
end;
begin
  signal VALUEERROR
end;
if A > 1
  then A := 1
  elsif A < 10
    then A := 14
end;
case A
  is 1   : A := 2
  is 3, 4 : A := 5
  else   : A := 14
end;
case A
  is 1   : A := 2
end;
case A
  is 1   : A := 1
  is 1000 : signal VALUEERROR
end;
send C to BUF;
receive C from BUF;
end;

procedure ECHO (var TTYIN, TTYOUT : TTYBUF) =
begin
  var C : CHARACTER;
  loop
    receive C from TTYIN;
    send C to TTYOUT
  end
end;
type TTYBUF = buffer (1) of CHARACTER;
end;
No syntax errors detected
No semantic errors detected

Exec-> lisp
Entering Lisp . . . Type (RETURN) to exit.

l(dskin (inall.lsr))

FILES-LOADED
l(dounit "echo::main")

ROUTINE NAME ( " ECHO::MAIN " )
( ITYIN , ITYOUT , COUND ) =
BEGIN LOCAL C ;
BEGIN WHILE 1 DO
BEGIN RECEIVE ( C , . . ITYIN ) ;
SEND ( . . C , . . ITYOUT ) ;
END END END ; END ;

NIL
l(dounit "p::main")

ROUTINE NAME ( " P::MAIN " )
( A , COUND ) =
BEGIN LOCAL X ;
LOCAL C ;
LOCAL BUF ;
BEGIN IF
t ( . . A GTR 10 ) THEN BEGIN
. . A = 10 ; END ;

IF ( . . A LSS 10 ) THEN BEGIN
. . A = 11 ; END ELSE IF TRUE THEN BEGIN
. . A =
( - 11 ) ; END ;

. A =
. . X :
IF . . A LSS 1 OR
A GTR 10 THEN SIGNAL VALUEERROR ;
X =
. . A ;
. A =
( ( . . X +
( 1 DIV 10 ) ) = 3 ) ;
BEGIN
ENABLE VALUEERROR :
BEGIN
   . A = 0 ; END ELBANE ;
BEGIN WHILE 1 DO
BEGIN
   . A = 100 ;
EXIT LOOP ; END ; END END END ;
BEGIN
BEGIN SIGNAL VALUEERROR ; END END ;
IF (
   . . A GT 1 ) THEN
BEGIN
   . A = 1 ; END ELSE
IF (
   . . A LSS 10 ) THEN
BEGIN
   . A = 14 ; END ;
IF . . A LSS 1 OR 4 LSS
   . . A THEN
BEGIN
   . A = 14 ; END ;
CASE . . A = 1 OF SET
BEGIN
   . A = 2 ; END ;
BEGIN
   . A = 14 ; END ;
BEGIN
   . A = 5 ; END ;
BEGIN
   . A = 5 ; END ; TES ;
IF . . A LSS 1 OR 1 LSS
   . . A THEN
BEGIN SIGNAL CASEERROR ; END ;
CASE . . A = 1 OF SET
BEGIN
   . A = 2 ; END ; TES ;
SELECT . . A OF *SET
1 :
BEGIN
   . A = 1 ; END ;
1000 :
BEGIN SIGNAL VALUEERROR ; END ;
OTHERWISE :
BEGIN SIGNAL CASEERROR ; END ; TESN ; END END ;
SEND ( . C , , BUF ) ;
RECEIVE ( C , , BUF ) ; END END END ; END ;
NIL
!(askout (examp .b11)(drop2 (dounit "echo::main")
   (dounit "p::main")
File-Dumped
!"C
@ed examp . b11
```plaintext
Edit: EXAMP.B11.1
*10,1
00000 module main=begin
00001 ! Surrounding module still
00002 ! added by a human.
00003 bind valueerror=1, caseerror=2,
00004 c1=1, c2=2, c3=3, c4=4, c5=5, c6=6, c7=7, c8=8, c9=9;
00005 external send, receive;
00006 bind true=1, false=0;
00007 $ *1*
08500 08600 ! main program
08700 ! calls ECHO, ignores Y.
08800 shname("MATN:ECHOSCOPE")(10,20,30);
08900 end eluton;
09000 $ *enu

[EXAMP,B11.2]
@Bline=1!
*,examp=examp

; Size: 127+0
; Run Time: 1 Second
; Core Used: 15K
; Compilation Complete

**C
@logout
killed Job 67, User LSMITH, TTY 153,

% cd bliss
% get1 examp.o11
    Connections established.
    30u USC-IS11 FTP Server 1.44.11.0
    > 330 user name accepted. Password, please.
    250 ASCII retrieve of <LSMITH>EXAMP,B11.3 started.
    > 231 BYE command received.
% asbl1 examp
    errors detected: 0
    errors detected: 0
% examp.out
I am typing this in.
I am typing this in.
It is typing it back out.
It is typing it back out.
How neat.
How neat.
```
Transcript ended oid 14745 Fri Jul 18 01:46:59 1980
REFERENCES


