NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.
A TOOL IN THE LAB:
A COMPUTER PROGRAMMING LANGUAGE
THAT OPERATES ON CONTEXT-FREE
SENTENCES

by

Andrew Marcus Lear Kurn
B.A., Reed College, 1970
M.Sc., Simon Fraser University, 1977

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

© Andrew Marcus Lear Kurn 1991
SIMON FRASER UNIVERSITY
December 1991

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.
The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L’auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L’auteur conserve la propriété du droit d’auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

APPROVAL

Name: Andrew Marcus Lear Kurn
Degree: Doctor of Philosophy
Title of Thesis: A Tool in the Lab: A Computer Programming Language That Operates on Context-Free Sentences
Examinining Committee: James J. Weinkam, D.Sc.
Chairman

Robert D. Cameron, Ph.D.
Associate Professor
Senior Supervisor

Louis J. Hafner, Ph.D.
Assistant Professor

Joseph G. Peters, Ph.D.
Associate Professor

Geoff McKeown, Ph.D., AFIMA
Senior Lecturer in Computing Science
University of East Anglia
External Examiner

Fred Popowich, Ph.D.
Assistant Professor
Examiner

December 10, 1991
PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

A Tool in the Lab: A Computer Programming Language that Operates on Context-Free Sentences.

Author: ____________________________

(signature)

Andrew Marcus Lear Kurn

(name)

December 10, 1991

(date)
Abstract

Context-free grammar is the most commonly used method of specifying the structure of formal languages, including mathematical expressions and computer programming languages. Although these languages may be more complex than what is implied by their context-free grammar, the method is so elegant and powerful that it has gained universal acceptance.

This work describes a new programming language, called G language, and the philosophy motivating its design. The language is designed especially for operating on sentences from context-free grammars, and thus finds application in computer language processing and symbolic algebra.

G is imperative and uses a store of cells, which is to say that its statements are usually executed serially and that variables may be changed so as to have different values at different times. The cell used here has a rather more complex structure than in conventional languages, which is the most fundamental difference between them and G language.

Several other unusual features are also present. For one, G unifies the notion of data type with grammar, with the following consequences: First, types become first-class data, which can be constructed and then used for the creation of cells of the new types. Second, any datum may be fixed during the execution of a program.

The language also includes a facility for writing literal sentences and for attaching annotations to cells.
For Reginald Banks,
who walked around the town and got sore feet.
I was working in the lab, late one night,
When my eyes beheld an eerie sight:
My monster from the slab began to rise.

“The Monster Mash”
Bobby Pickett and Leonard Capizzi
Acknowledgements

No student can hope to get far without a good advisor. Rob Cameron has been a fine one, both in the positive way and as the loyal opposition. Lou Hafer has also been generous with his time and constructive with his criticism. He also deserves thanks for proposing the initial problem: Bliss-to-C conversion. Joe Peters lent a sympathetic ear, even when he was most busy.

I must thank several people for their ideas and insights on specific problems and for learning enough about G to see what the problems were: Rob Cameron, Ken Collins, Mike Dyck, Cathy Levinson, Ed Merks, Jeff Rudd. For help regarding how and when to work I thank Joyce Cowan and Gillian Nonay. Typesetting is done with \TeX, for which I thank Leslie Lamport and Donald Knuth.

Finally, I thank my support group, my kitchen committee, Barbara Beach, Ken Collins, and Cathy Levinson and, for support, the defining occurrence, Sarah DuFresne, last as she was first.
# Contents

Abstract iii

Acknowledgements vi

Note x

## 1 Introduction

1.1 Syntagms and Software Tools .......................... 2
1.2 G Language ............................................. 5
1.3 An Example ............................................. 8

## 2 Design Philosophy

2.1 The Specification of a Formal Language ................. 13
2.2 Type and Grammar ....................................... 16
2.3 What's in a Cell? ....................................... 20
2.4 Creating New Types ..................................... 26
2.5 Other Features of the Data Space ......................... 28
2.6 Parsals ................................................. 34
2.7 On Assignment ......................................... 40
2.8 Input/Output and Parsing ............................... 45
2.9 Implementation Issues .................................. 46
3 G Language Manual

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Lexical Conventions</td>
<td>53</td>
</tr>
<tr>
<td>3.2 What's in a Cell?</td>
<td>57</td>
</tr>
<tr>
<td>3.3 Foundation Grammars</td>
<td>65</td>
</tr>
<tr>
<td>3.4 Unique Strings</td>
<td>73</td>
</tr>
<tr>
<td>3.5 Namelists</td>
<td>75</td>
</tr>
<tr>
<td>3.6 L-Values</td>
<td>79</td>
</tr>
<tr>
<td>3.7 Assignment and Coercion</td>
<td>80</td>
</tr>
<tr>
<td>3.8 Notation</td>
<td>82</td>
</tr>
<tr>
<td>3.9 Expressions</td>
<td>83</td>
</tr>
<tr>
<td>3.10 Declarations</td>
<td>103</td>
</tr>
<tr>
<td>3.11 Global Declarations</td>
<td>108</td>
</tr>
<tr>
<td>3.12 Exceptions</td>
<td>112</td>
</tr>
<tr>
<td>3.13 Statements</td>
<td>114</td>
</tr>
<tr>
<td>3.14 External Objects at Compile</td>
<td>121</td>
</tr>
</tbody>
</table>

4 Evaluation

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 The Work as Proposed</td>
<td>126</td>
</tr>
<tr>
<td>4.2 Example Programs</td>
<td>133</td>
</tr>
<tr>
<td>4.3 Related Work</td>
<td>160</td>
</tr>
<tr>
<td>4.4 Implementation</td>
<td>177</td>
</tr>
</tbody>
</table>

5 Conclusion

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Retrospect</td>
<td>181</td>
</tr>
<tr>
<td>5.2 Prospect</td>
<td>184</td>
</tr>
</tbody>
</table>

A Code for the Pointer Guard

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 An Abstract C Grammar</td>
<td>186</td>
</tr>
<tr>
<td>A.2 The Program</td>
<td>191</td>
</tr>
</tbody>
</table>
Note

It is perhaps advisable for me to prepare the reader for my use of the first person, singular in this work. It occurs quite rarely in scholarly discourse, and the reader may easily imagine it to be a sign of immodesty.

Allow me to assure him that just the opposite is my intent. While I can content myself with the usual euphemism, the first person, plural, when carrying out a more or less straightforward progression of ideas, when, however, I must acknowledge that I have made an intuitive leap which is somewhat doubtful or arbitrary, I am unwilling to assume complicity on the part of the reader. The singular marks these places.

Since this thesis is a design, there are many such places, too many, I felt, for circumlocutions such as “the present author.” Please, bear with me.

I have also used structure and access as if they were verbs, and I have sometimes used data as if it were the singular, as in data space instead of datum space. For these sins the computing science community must share the blame with me, although nothing can really relieve me of my author’s responsibility.
Chapter 1

Introduction

I'm Harry. I'm Walter.
It's wonderful to be here.
We guarantee to put a smile
On every face we see here.

"Harry and Walter Go to New York"
(dir.) Mark Rydell

Since computer programming ceased to be the sole property of a few scattered researchers, computer programs have ceased to be just the directions a man chooses to give a machine. Programming has become a joint effort; programs must fit together into larger units. Programs are passed from hand to hand over the years and need adaptation for changing circumstances and changing requirements.

The ideal program has moved from being one that best satisfies one purpose, to one that achieves a balance between several: machine-readability, machine-efficiency, man-readability, modifiability, abstractability, and maintainability. Often, to increase the suitability for a particular purpose, secondary programs, *software tools*, are written which operate on, examine or transform, the program in aid of that purpose.
CHAPTER 1. INTRODUCTION

1.1 Syntagms and Software Tools

This research proposes a new language, called G, designed specifically for expressing programs that operate on programs and program fragments. It is a language for writing software tools.

For a language designed for manipulating program fragments, the question of the internal representation of these fragments is central. We shall choose to represent them as sentences of a context-free grammar (CFG) and to call these representations syntagms, following [CI84] who emphasized their usefulness for general program transformations. We shall make a brief digression to introduce CFG.

CFG is by far the most commonly used formal specification of programming languages. Although these languages may be more complex than what is implied by their context-free grammar, the method is so elegant and powerful that it has gained universal acceptance. It is precise, yet readily understandable to practitioners who have had a brief introduction. (I often use the abbreviation CFG to mean the method or process of defining or using a context-free grammar, instead of its ordinary meaning, a given grammar.)

A CFG appears conveniently in tabular form. It is simply a list of rules, each of the form:

\[ A \leftarrow X Y Z \]

meaning that wherever \( A \) appears \( X Y Z \) may be substituted. The symbols are from two sets called terminals and non-terminals (nT). The goal is to create a list, called a sentence, composed entirely of terminals, by starting with a particular nT called the start-symbol and replacing nTs, one at a time, using rules from the CFG.

The left-hand side (lhs) of each rule must be a single nT. The right-hand side (rhs) may be any number of symbols (even zero) of both kinds. When
several rules have the same left-hand side, one is free to choose which one to use when substituting for that nT. This is how one grammar may generate many different sentences. It is called alternation. The entire set of sentences which may be generated from a CFG is called its language.

The record of the derivation of a particular sentence is neatly represented as a tree. The start-symbol labels its root. Each time a substitution is made for a nT, the tree's node with that label gets daughters labeled with the symbols from the rhs of the grammar rule being used. When the substitutions are all done, the leaves of the tree are all terminal symbols, and the derived sentence is read from these leaves.

The derivation tree, or a subtree of it, modeled in computer memory, is what we call a syntagm. Note that not every sublist from a sentence may be so represented, but only those that derive from some particular nT.

Also, we must mention that sometimes two CFG's are useful when dealing with a programming language. They are called concrete and abstract. For a given sentence (program) in the language, the concrete specifies every printed character, including the reserved words and the parentheses, semicolons, and other punctuation. The abstract omits these words and punctuation, yielding a greatly simplified derivation tree, yet retaining the essential form so that the concrete tree may be reconstructed from the abstract tree. The abstract syntagm is easier for a programmer to work with and saves machine resources, too. However, unlike the concrete tree, which may be inferred from its leaves, (that is, it is created by parsing the leaves) nothing can be recovered from just the leaves of the abstract tree. The whole tree is needed.

Although it is by far the most commonly used formal specification of programming languages, CFG is not the only one, nor is it the best in this important sense: its degree of specificity. (We shall speak about other specifications in §2.1.) However, its popularity is well deserved. It combines good expressive power with the greatest parsimony in its vocabulary of basic operations.
Basically it has just two, which are here called substitution and alternation, but are widely known by other names in other fields. In mathematics they are Cartesian product and union (or discriminated union, depending on details); in programming they are called (Pascal) record fields and variants, or (C) structures and unions. (See §2.2.)

The observation that the operations which give structure to sentences in CFG are essentially the same as those which give structure to data in the familiar programming languages has led me to a design decision which gives G a most distinctive character. CFG, now extended for the purpose, is the sole device for structuring data in G language.

It is the experience of mathematicians, engineers, and scientists that elegance is simplicity. I have attempted in this work to simplify and unify concepts, notation, and mechanisms in order to achieve an elegant language when I could do so without sacrificing its understandability or its expressive power. The combination of the data structuring mechanism with CFG, the syntagm structuring mechanism, is the first and the most important of these.

Although G is designed for program fragments, the class of inputs which are conveniently thought of as CFG sentences is the whole realm of symbolic computing, containing symbolic algebra and natural language, as well as any where trees play a prominent rôle, such as planning and game-playing.

Broadly speaking, G is an ordinary imperative language, but one which operates over a data space which is founded on CFG. Any software tool that pays attention to the structure of the programs being manipulated, e.g. a syntax-directed editor or a translator, will benefit from this kind of data space and will therefore benefit from being written in G.
1.2 G Language

G is a special-purpose programming language. Its purpose is to manipulate syntagms, especially programming language fragments. It is a collection of features, mostly from other languages, a few original. The basic approach, i.e. the syntagmatic view of computer programs, and, indeed, the inspiration for this entire work has been taken from [CI84], who advocates an imperative approach to program transformation. The C language [KR78, KR88] is prominent, both in G's statement structure and the surface syntax generally. This reflects my admiration for that language. To me it seems intuitive, clean, and brief to a degree which I cannot hope to equal.

Although my admiration is not unmitigated, I fall back on C when a feature of G does not warrant a new design. C's design is my default design, and I often let the default prevail, not only when it is the best design, but also when our familiarity with C gives us a shortcut to learning G, and when the feature is sufficiently tangential to our main purpose, the syntactic approach to language processing, to fall outside our scope of interest. Thus, several features of C have migrated into G which do not have my complete approval.

The purpose behind G's design has always been practical. It should be able to handle large programs efficiently, and it should be accessible to working programmers. These two goals, however, sometimes conflict. The desire to give good error reports increases storage and reduces execution speed. The desire to give power to the language has precluded full static type checking, although most type checking can be done statically.

We now introduce the important features of G and sketch the reason for the existence of each (in effect, a précis of chapter 2). Much of G can be seen as a straightforward development from a few crucial design decisions.

- First, as mentioned, the data types of G become CFG. No datum exists without a type and every type is a symbol in a CFG.
Next, all types are first-class citizens. By this we mean that a type may be passed as an argument or returned and that in general the operations that apply to data also apply to types. In fact there are extra operations related to types: A primitive exists to return the type of a given datum, and another exists to create a new datum given its type.

Why do we need first-class types? We wish to be able to write routines which operate on syntagms of any type. A parser and a pretty-printer are examples. Such a routine must be able to examine the relevant grammar in order to carry out its mission. If some types must be first-class, in the absence of good reasons to the contrary, we unify the language by making all types first-class.

It develops that types can be represented as syntagms according to a CFG, a grammar of types. This is the fundamental grammar of G and must exist in every G data space. Since we can thus include types cheaply, i.e. without extending the class of objects in the data space, we feel justified in this decision.

Types may be constructed dynamically. Just as we wish to explore the types of syntagms passed to subroutines, we wish to create syntagms of completely new types, say as described by a user during a terminal session. To accomplish this is straightforward, given that syntagms may be constructed and types are syntagms.

Syntagms may be fixed dynamically, i.e. made unalterable at an arbitrary time. Given that types are visible, it follows that, in the absence of a rule to the contrary, one might alter a type after syntagms of that type exist. Our rule to the contrary shall be that no syntagm may be created of a given type unless that type is fixed. This requires fixation for types. The existence of dynamic types implies dynamic fixation. We unify the language by making fixation available for all syntagms.
We should like to be able to write literal syntagms in the program text. At first glance this seems easy, since the grammar for the syntagm is bound to be available. However, there is a problem. It happens frequently that there are two grammars for a target programming language, the concrete and the abstract (§1.1). The concrete grammar is easily parsed, but the abstract, although unparsable, defines the trees which correspond nicely to our intuitive view of the syntagms.

So, now it is up to the compiler to parse according to the concrete grammar and then translate to the abstract. Moreover, a literal may be a mixture of components from different grammars; there may be more than two grammars; the compiler may be required to perform extra services while parsing, such as counting, marking, or recording nodes. This is to illustrate that no simple prescription is possible which can convey what the compiler should do with a literal.

Our solution is to allow the user to write his own parser, which the compiler calls as a subroutine. It requires the interpretation of a literal as a parser argument list, or parsal, according to an established convention so that the parser and compiler can communicate.

- To facilitate error reporting by the parser, we include in G a simple exception mechanism.

- We need a way of attaching annotations to parse trees. We call these properties and arrange for properties from different sources not to interfere with each other.

- We add flat lists. This is accomplished using a distinct kind of CFG rule so that the daughters of a node may be declared to be such a list. These lists are explicitly of infinite length, so that no lengthening operation is needed. Likewise, all properties exist, so that the set of properties
attached to each node may be thought to be infinite.

- Since we wish to handle problems of realistic size, we will bear in mind that all objects must be designed so as to have persistent representations on an external medium. It must be possible to save and restore syntagms.

- G has no primitive I/O facility, nor one for parsing or formatted printing. As in C, we take the view that such things are not properly part of the core language and so are best implemented as subroutines.

1.3 An Example

We present a complete G subroutine at this point in order to give some feeling for the language before we work through the details. Further examples and explanation can be found in chapter 4.

We will look at a subroutine which makes a simple program transformation on C statements. If the condition of a do-while statement happens to be known to be 0 (false), perhaps as the result of a previous transformation, then the whole statement may be replaced by its body, the contained statement. Schematically [DB76] we would say this:

\[
\text{do } \text{statement} \text{ while (0);} \\
\quad \Rightarrow \text{statement}
\]

In G we first build a template which contains the known parts of the statement and then we try to match it with the given statement. If this is successful a variable is set to the unknown nested statement so it can be returned. Here is the complete program:

\[
\text{[} \text{extern namelist unequal GramGram@GRule GramGram;}
\]
extern namelist unqual  
  GRule gramC;

statement

do_while_check (statement s)
{
  statement a;
  if (
    statement["do 'statement[a] while (0);"]
    /  s)
    return a;
  else
    return s;
}

Line 1 introduces the standard environment by requesting GramGram, the grammar of grammars, i.e. the type of all types. This generates secondary requests for several other grammars and miscellaneous data. In all, more than 50 identifiers become defined as a result. Every program starts with this statement.

Line 2 loads the grammar for C [KR78]. Again, several symbols become defined as a result. The only one of immediate interest is statement, which is the type of C statements. The unqual here means that statement may be used as shown. Otherwise we would have to write gramC@statement.

At this point we must mention that the grammar for C and its associated parser (mentioned below) are supplied by the user. We assume that they are well written and behave according to G's conventions.

Lines 3 and 4 are the function header, declaring a function which expects a statement and returns a statement.
After declaring a local variable, the function constructs on line \( \ell 5 \) a temporary syntagm to use as a template. The resulting syntagm is shown in fig. 1.1. Reading line \( \ell 5 \), the first \texttt{statement} gives the type of the top node of the syntagm, but it also tells who is to construct it. Since \texttt{statement} is from the C grammar, the C parser, which accompanies the grammar, is called to interpret the list between the square brackets. It constructs an \textit{abstract} syntagm for the statement, encoding the keywords and punctuation of the statement (called \textit{concrete} or \textit{sugar}) in the \textit{type} of the various nodes.

The only remaining unexplained feature is the string \texttt{\textquotesingle\texttt{statement}}[\texttt{a}]. It is an escape sequence. It directs the parser to make a node of type \texttt{statement} (fig. 1.1) with a "non-value." There is just one special value in \( G \), called
none, used whenever an empty or bottom value is required. It is used here. Last, the node is assigned to the variable a, so that a's value is the node. (We explain the meaning of the dashed lines in §2.3.)

On line 6 this template is matched with the argument. If the two match, the expression is True and the none in the template is replaced by a copy of the matching part of s. If the two do not match, say because s is not a do.statement or because the expression is not 0, the condition in the G statement evaluates to False and no change is made to storage.

Thus, the subroutine makes the desired change when applicable and returns a new syntagm. If the change does not apply, it simply returns the argument.
Chapter 2

Design Philosophy

Gozer the Traveler. He will come in one of the pre-chosen forms.

During the rectification of the Voldran the Traveler came as a large moving tor. Then, during the third reconciliation of the last of the Makhtric supplicants, they chose a new form for him—that of a giant slor. Many Shauvs and Zoos knew what it was to be roasted in the depths of the slor that day, I can tell you.

"Ghostbusters"
(dir.) Ivan Reitman

G language was inspired by the need to express software tools. Given this need, of course, we have sought to make the scope of the language as broad as possible without compromising its usefulness for its original purpose. Yet, the original purpose, the expression of programs about programs, remains our point of departure.
2.1 The Specification of a Formal Language

In the beginning was the Word,
And the Word was with God,
And the Word was God.

John 1:1

In this section we refine the notion of a valid program in some programming language and ask how we might recognize one.

Let us briefly introduce a few of the terms used in the study of formal languages. An alphabet is simply a finite set. A sentence or word is a finite sequence of elements from the alphabet. Word is often used, but we shall use sentence here since it describes our data more intuitively, given their length and structure. We often call the elements of the alphabet symbols or terminals and give each a printed character as a label, so that we can write a sentence as a sequence of characters.

Finally, a (formal) language is a set of sentences over some alphabet. It may be empty or finite or infinite. One infinite language is the set of every sentence over the alphabet.

In §1.1 we defined CFG and said that it was a popular method of specifying programming languages. There are, however, some needs which are not met by CFG and other specification methods which meet some of them.

The formal languages, of which programming languages are an important subclass, may be classified according to several different taxonomies. There are two main ones. The first is grammatical: the languages generated by grammars restricted in some given way. The second is mechanistic: the languages generated or recognized by automata restricted in some given way. The mechanistic classification is often augmented by restrictions which we might call economic, those requiring a machine of a given category to recognize a
sentence using no more than a prescribed amount of time or space. This subject area has seen a considerable amount of interest, especially since 1970. A good introduction to it may be had in [HU79].

The foundation was laid for the grammatical taxonomy in [Cho56, Cho59]. Grammars are put in classes designated 0, 1, 2, and 3 or called, respectively, unrestricted, context-sensitive, context-free, and regular.

Languages are generated from grammars 0, 1, and 3 much as they are from type 2, CFG. They differ as follows: (We will not discuss type 3.) In type 0 the restriction that the lhs must be a single nT is lifted. Any string of symbols of both kinds is allowed on the lhs providing that there is at least one. In order to apply a substitution, the entire lhs must match a substring of the sentence under derivation. The substring is then replaced by the rhs.

Type 1 grammars are just like type 0 except that the rhs of each rule is constrained to be at least as long as the lhs. The name context-sensitive comes from a normal form for these grammars.

It is known for the sets of languages 0, 1, 2, and 3 that each set properly contains the following set.

Now let us specifically consider programming languages. When we ask whether a particular program is valid, what do we mean? In this context we shall mean that the program considered as a string is a sentence of the relevant programming language as recognized by an ideal compiler, a recognizer for a formal language.

But, how does the ideal compiler behave? Is there a rigorous definition of it? We can ask whether a specific compiler accepts a program, but there are often several compilers for a language, and their behavior differs. One reason this happens is that the language is not precisely defined in the first place.

We must leave the question open, trusting to our intuition about what the ideal compiler may legitimately do. We can agree, I hope, that the compiler may in no sense execute a part of the program as a prerequisite to accepting
CHAPTER 2. DESIGN PHILOSOPHY

that part. (However, as we shall see, G feels free to execute one part in order to accept another part.) But, what the compiler must actually do to accept a program varies from language to language, and sometimes from compiler to compiler. Imprecise definition is a recognized problem, and, in fairness, it is true that more recent languages are more likely to define precisely when a program is valid. (However, we note with dismay that Ada creates some errors which cannot be detected at all, let alone by the compiler [Ada83, §6.2, array parameters; §1.6.c].)

That said, it is well known that ordinary programming languages are not type 2, i.e. not CF languages. A programming language is specified by giving a CFG and then further constraining it by eliminating some programs which are valid according to the CFG. Many common constraints, such as the rule requiring a variable to be declared before it is used, cannot be expressed by CFG. (See, for instance, [Pag81, p. 19].)

Can such a rule be captured by type 1? The answer is yes in this particular case. The type 1 languages correspond exactly to a class in the mechanistic scheme called *linear-bounded automata* (LBA) [HU79, §9.3], which are quite powerful and easily equal to the task.

However, whether any rule that a compiler might be required to check can be checked by an LBA is ineffable. The human mind is a wonderful thing, and some day it may conceive very difficult tests which it may require a compiler to execute. Moreover, this day may not be far off. The compilation of ML [Mil78, HMM86, HMT88] is already known to be intractable [Mai90] and may conceivably be beyond the power of the LBA. On the other hand, in 1979 [HU79, p. 224] the only languages known not in type 1 had been constructed by diagonalization. Thus, we have some hope that the type 1 languages still contain all known programming languages.

Although we now understand why it is almost surely legitimate to call
CHAPTER 2. DESIGN PHILOSOPHY

the constraints of any given programming language context-sensitive and understand where the recognizer for it fits into the mechanistic hierarchy, it is by no means true that CSG’s are a convenient way of presenting the formal definition of a programming language. On the contrary, while there are a few formal specifications of languages including context-sensitive constraints, there are none using CSG.

Undoubtedly, a declarative specification of these constraints is best for proving correctness in applications such as automatic compiler generation. However, I have chosen a procedural approach as the basis for G because of its superior readability and writability for programmers. CFG declarations are part of G because they are simple enough to be convenient, but context-sensitive constraints are programmed using the procedural facility because a declarative presentation would be opaque to most people.

Of course, G provides the programmer with the full power of a Turing machine. To give him ease in learning and using G, I chose to model it after an existing, popular programming language, C [KR78, KR88], not only on the theory that there was a fair chance that he would know C already, but also because the popularity of a tool is a good indicator that it is a good tool, that it is handy, convenient, and suits its purpose.

2.2 Type and Grammar

Let us explore the relationship between data type and grammar. We will find that the usual idea of data type is generally richer in structural power than CFG, but that, although the differences are important, CFG is still very powerful.

Programming languages typically start out with a few primitive base types and a few type constructors, which make new types from existing ones. For example, in C there are 5 constructors: structure, union, array, function,
and pointer. The structure constructor takes a list of types. The space of structure instances is just the Cartesian product of the spaces of the listed types. Similarly, the space of instances of the union constructor is the union of the listed types.

The array constructor yields the Cartesian product of $n$ instances of its argument type and thus may be considered a shorthand for a structure constructor.

For these three cases the mathematical abstraction and the usual machine model are in close correspondence. The model is the simplest one, the one that treats each value as a string of bits. The number of bits used to store a structure is the logarithm of the number of instances, the sum of the bits of each component. Similarly, an array is $n$ times the bits of the component type. (A union in C has one bit fewer than it ought to have because a true union is not formed. The number of values is only that of the greater of the two components.)

Of course, all programming languages do not have any given constructor. The situation varies widely. Sometimes (Lisp) there are no data types. Sometimes (Fortran) there are arrays but no structures or (ML) structures but no arrays. (ML lists are akin to structures.)

What we can say is that the two operations, structure and union, or product and sum, are very common and carry the larger part of the burden of specifying data structure where they exist. They exist in Cobol, PL/I, C, Pascal, Modula, Ada, CLU, and ML, among others.

These operations have close analogs in CFG (§1.1): substitution and alternation. To show the correspondence, consider this C type declaration:

```c
struct S {D d;
    union T {E e;
        struct U {F f;
            union V {G g;
```
Here is the storage map for this type.

\[
\begin{array}{|c|c|}
\hline
D & E \\
\hline
D & F & G \\
\hline
D & F & H \\
\hline
\end{array}
\]

The width of each box represents the amount of storage required, and the alternatives are shown on separate lines.

Despite its operational-sounding name, the storage map has theoretical significance. It shows the base types in their proper order for each alternative and lists all the alternatives. It embodies the relationship between the base types and the constructed type.

We compare it to the corresponding CFG,

\[
\begin{align*}
S & \leftarrow D \ T \\
T & \leftarrow E \\
U & \leftarrow F \ V \\
V & \leftarrow G \\
V & \leftarrow H
\end{align*}
\]
which we see generates a language with only 3 sentences: \textit{DE, DFG, and DFH}. It is easy to see that every such C declaration can be converted into a CFG which generates the same language as the "language" of the storage map.

In one sense this is obvious, but in another it is lucky, since in a program-transformation context we naturally want to think of grammatical type as data type. The type of an \textit{if}-statement should be \textit{if-statement}. Programmers and practitioners have become so used to the usages and concepts of CFG that our intuitive notion of type for a program fragment (a fragment that forms a conceptual unit) is the CFG syntactic category. This being the case, a strong effort is warranted to unify this with G language data type, both to aid the intuition and to avoid a duplication of facilities between the type structure and the grammatical structure.

We have not argued that everything which can be reasonably thought of as a data type can be described by the two constructors. It is not true. Two counter-examples will illustrate this.

Functions are a very important class of objects, which are universally given special treatment by programming languages. The reason is practical. A function can be stored in a very much smaller space in coded form than as an uninterpreted map. Of course, this is not true for most functions, but those which reflect algorithms can take advantage of the coded form of the algorithm to economize. Moreover, functions stored as maps are so enormous that a naïve function type constructor would be of no use. Instead, they are stored in blocks of undefined size with perhaps an \textit{ad hoc} type or pseudo-type.

Another weakness of the type constructors we have discussed so far is their simple view of dynamic structures. Consider a pair of stacks growing in from opposite ends of a free block of storage. There is no left-to-right way of describing an area which starts with instances of one structure, has an unallocated space of unknown length, and then ends with instances of another structure. Yet, this area is obviously highly structured. Given the
two stack pointers, there is no difficulty deciding whether the data in the area are consistent with their individual types. Yet, no one has devised a way of declaring the block so that its consistency as a whole can be verified.

2.3 What's in a Cell?

"Come away by yourselves to a lonely place"

Mark 6:31

Since I have chosen an imperative model for G language, we know immediately that its data space will contain elements which can be updated, the technical term for which is cells [Sto77, Ch. 12, p. 284]. Our first job is to design these cells, given that type is grammar.

My first impulse is to limit the data space so that it contains only cells, thus making it completely uniform. In this I am not successful. I am forced to introduce one non-cell object (that is, a pure value), called none. Furthermore, besides the standard cells, two extra categories of cell are needed. They are both restricted versions of the standard cell. The first is called ident, and the second is primitive. Idents are part of any precise description of G. Primitive cells are properly an implementation convenience, but what one might call a necessary convenience.

Cells are, of course, to be the nodes of the derivation trees or syntagms of §1.1. As such, each has other cells (or perhaps the value none) associated with it. From the denotational viewpoint, these cells (I will continue to say "cells" when I should include none, as long as the meaning is clear.) are reached using a store, the function which associates values with cells (e.g. [Pag81, §4.2.3]).

In our case the store would yield a quintuple, the five things associated with cells. However, in order to make the discussion clearer, we will call the five things attributes and explain each separately in the natural order. The
attributes are value, mother, type, fixatzon, and properties. The first three are explained here, the last two in §2.4 and §2.5.1, respectively.

The most important thing about a tree node is its place in the tree: the nodes immediately below and above it. So, the first two attributes are value and mother. Mother is a single cell. Value is a list, since a node may have several daughters. Value is the most important attribute, and we may speak of it in a variety of ways. We may speak of the cells held by a given cell, or the subtree rooted or at a given cell, always referring to its value. (Syntagm navigation operators are detailed in §3.9.5.)

Since a syntagm is intuitively a cohesive object, a thing, it is natural that we should be able to move from one part of it to any other part, independent of whether the path is strictly downward, upward, or neither. For this reason both upward and downward links are standard for every node. Our syntagms are doubly-linked trees. However, as we shall soon see, it is awkward to make this perfectly uniform. Some one-way links are useful.

Type is the next attribute. For the reasons given in §1.2 it is desirable to have types become objects in the data space. For simplicity (i.e. to avoid duplication of operators, etc.) we prefer to make them cells. It is not difficult to do so. Since we know that each type is some symbol from a CFG, it is only required that we have some sort of representation of a CFG as a syntagm. (The details are given in §3.2.2, §3.3.3.) Naturally, there will be context-sensitive constraints that we must worry about, but the difficulty of enforcing them is minor. (We sketch a way of enforcing them in §2.4. For what the constraints are, see §3.3.3.1.)

The idea that a cell’s type should be a cell does, however, give us a small problem immediately with respect to CFG. Consider these rules from a possible concrete C grammar:
Given a syntagm whose root cell was type $stmt$, we might want to know which rule governs its daughters. We would have to check for the first case by examining the first daughter and then distinguish between the second and third by testing the existence of the seventh daughter. This is unpleasant because we must examine the grammar carefully to invent the series of tests necessary to distinguish completely all the possible cases. We should have a single simple test where we now have a grammar-dependent series.

We remedy this by creating two kinds of type, called construction and alternation. A cell of a construction type has several daughters, each of some specific type. A cell of an alternation type has exactly one daughter, but it may be of any of several different types.

This forces us to write

\[
\begin{align*}
stmt & \leftarrow \text{while} \ (\ exp \ ) \ stmt \\
stmt & \leftarrow \text{if} \ (\ exp \ ) \ stmt \\
stmt & \leftarrow \text{if-else} \ stmt
\end{align*}
\]

making $stmt$ an alternation kind. Now we can learn immediately which sort of $stmt$ we have by examining the daughter of the $stmt$ cell. Although this is a restriction on CFG, we have not sacrificed any of its power. Any CFG may be converted to this form by adding some nTs. (We differ from [Cl84] who have a similar scheme. They have alternation types but never create cells of those types.)
Before we go further into type, let us look at the reason for one-way links. Conceptually, our data space must contain not only what we think of as syntagms, that is, program fragments in the process of transformation, but also connections between them. What sorts of connections? For one thing, we will want to note parts of syntagms. A note will take the form of a reference stored in a data structure, a reference to some interior cell of the syntagm. But, to store something in a data structure means to make it the value of some cell, since all our data structures are in fact syntagms. If the cell is not to be in two syntagms at once, then it must be attached only weakly to one of them; one of the links to it must be weak. To put it another way, only one of the cells holding it may be its mother, i.e. reached by the mother attribute.

We accomplish this by creating a special kind of type, which we call gate type. A cell whose type is one of the gate types is called a gate and has the desired feature, i.e. that its daughter does not have it for its mother.

Recall that our model is that we shall have a class of program syntagms and a class of other syntagms that refer to them, each class with its own grammar. Since we have it in mind that gates are to hold cells from a foreign grammar, it is natural that we improve the specification method so better to accommodate larger classes of type.

My idea is this: At present we have construction types, each of whose daughters is a specific type, and alternation types, whose daughter is one of the types from a set named explicitly in a grammar. In each case the focus is on types from a single grammar. The advent of gates broadens our consideration to multiple grammars. We will confine constructions and alternations to be composed of types from within a single grammar, but gates will see more.

I have sought to compromise between the desirability of a type which accepts a wide variety of daughters and the difficulty of implementing such a type when all the accepted types must be named explicitly. Here is the way I have chosen: There are three degrees of acceptance. A gate may accept
CHAPTER 2. DESIGN PHILOSOPHY

exactly one type, although the type may come from a foreign grammar. (In G this might be denoted \texttt{Pascal@statement}.) Or, a gate may accept any type from some (perhaps foreign) grammar (\texttt{Pascal@any}). Or, a gate may accept any type from any grammar at all (\texttt{ANY}). (There is a fourth kind of gate, those that accept function cells with a particular profile, to be discussed in §2.5.3.) This four-fold way of specifying type, especially the \texttt{any} and \texttt{ANY} way, will be called \textit{broad} types.

Basically, gates are useful in two situations: where connections must be made to syntagms of foreign grammars and where it is necessary to make graph structures which are more general than trees. Both uses are made of them in this work. The original idea comes from \cite{DKLM84} to whom I wish to pay homage by the use of their term \textit{gate}. They use gates for inter-syntagmatic connections, but not for cyclic graphs. (They do, however, form DAG's.)

We differ from \cite{Dyc90}, who points to the aforementioned two situations and suggests that the two uses of gates are orthogonal and should be separated, the non-treeness remaining with gates while the acceptance of broad types is extended to all kinds of type.

Idents, which we discuss now, also have daughters of broad types and also have one-way links to them. Idents and gates are the only citizens of our data space which have this character. One-way links are indicated in our diagrams by dashed lines, as on p. 10.

Roughly, an ident is half a gate. It has a value, but it itself is never the value of anything, not even a gate. It was created to be the denotation of \textit{G} program identifiers.

It may occur to some readers to ask why idents need exist. Are they not just orphan gates? No, there are several differences. First is the fact that they can only be altered in one specific way. In the present context, this means by writing the appropriate identifier as the target of an assignment. (But, see also §2.5.1.)
CHAPTER 2. DESIGN PHILOSOPHY

This creates a certain safety. Because of the existence of gates there can be many ways of reaching, and therefore altering, a cell. If there were several ways of reaching an ident, the cell denoted by an identifier could be altered without writing it, making certain program errors difficult to find.

The subject of the mutability of the meaning of identifiers is controversial. Some languages go so far as to fix the value of every identifier in their routines throughout each activation. We are near the opposite extreme, where many data may be denoted in a variety of ways and changed freely. This creates the danger of some very obscure program errors, and we compensate for this with two features, the first of which is idents. (The second is fixation. §2.4) Our scheme is much like that of CLU [LAB*81], where they dignify this danger with the name *sharing*.

The second difference between idents and gates, and the last we shall mention here, is a technical one regarding type. The type of a gate is a cell whose meaning is "gate to x." This type cell must exist in a grammar created for the purpose. An ident whose effective type is "gate to x" can exist, usually does exist, without any such cell or grammar. It requires only the existence of the type cell x itself. Theoretically, I suppose, this difference is no reason for the existence of idents. It only saves computer memory. But, to spend memory on grammars for objects of which there are only one or two seems wasteful, especially since there are likely to be many idents in the data space of a program of moderate complexity. So, our choice is to make idents implicitly be gates without the associated grammatical machinery.

We finish this section by introducing primitive cells and none. Cells whose type specifies that they have no daughters are called *atoms*. They are natural candidates for the leaves of trees. In such frequent use are certain atoms, the integers, the Booleans, and the printing characters, that it is a considerable optimization for an implementation to recognize them by their machine addresses. If this is done there is no need for the actual cells to exist in storage,
and so this storage may be recovered and used otherwise. In the case of the integers the saving in storage can be considerable.

We therefore designate these cells primitive, by which we mean that they shall be perfectly predictable so that they need not actually exist. With respect to the attributes we have discussed so far, it means they must be orphans, that is, have no mothers. (There is no effect on type. It is known and fixed for primitives as it is for other cells.) We shall mention the effects of primitives on the remaining two attributes as we introduce them.

Our pure value none is the ubiquitous default value. It is the value of the mother and all daughters of a newly created cell. It effectively has all types. It is not a cell, and so has no attributes of its own.

Once or twice during this work I have been tempted to create other pure values. It has, however, seemed to me that the number of special cases to be dealt with in the various operators, and therefore by programmers using G, increases with the number of pure values. So, that number has been held down to one.

2.4 Creating New Types

Now, since types are syntagms and syntagms are constructable, it follows that types are constructable. Although the enforcement of context-sensitive constraints must again worry us, the benefits surely outweigh the costs.

However, there is another cost to be counted. We lose the ability statically to check type. Actually, we never had it. (The question of type equivalence is deferred to §2.5.3.)

The ability referred to is the ability always to know the type of an expression so that the compiler may decide whether the expression may be safely assigned to a given variable. In the presence of alternation this is, of course, impossible. Among the so-called statically-checkable languages, the response
CHAPTER 2. DESIGN PHILOSOPHY

to the problem varies. Pascal, C, and Modula-2 simply fail to detect whether the expression has the purported type. Ada [Ada83, §4.1.3] performs a dynamic check.

The situation in $G$ is, however, worsened by the features we have discussed. Not only may the daughter of a given cell be one of several alternatives, but the mother may, as well. While the types allowed for a daughter are easy to see when reading the type (CFG) for the cell, those for the mother are not, and the mother may just as easily be the value of an expression as the daughter.

There are, however, many situations in $G$, as there are in Pascal, etc., where the type of an expression is statically determinable. It is the compiler’s business to recognize them and make the determinations. Also, it should be at least an option to the user that the compiler insert dynamic checks where static ones are not possible.

There is an obvious problem with types as syntagms, however, when we recall that syntagms are alterable. (We should really have mentioned it in the last section.) If a type cell is to describe a certain cell (with respect to the types of its daughters) then the type must be fixed. We know that a single type may be the description of many cells. We have already guaranteed that the type cell of each cell shall be fixed throughout the life of the cell. But, since the type is merely a cell in a representation of a grammar, what is to keep the user from changing the grammar by error after cells exist with that type?

It must be that the values of the cells in the grammar are fixed. Necessarily, fixed means not only that the value of a cell may not change, but that the value itself, other cells, must likewise be fixed recursively. This is so even for gates.

Thus, fixation is the fourth attribute of cells. Its value is Boolean, answering “Is this cell fixed?” (Fixation is dealt with by several primitive operators
§3.9.1.2.) Since I have decided to create types dynamically, all that remains to do is to rule that types may be fixed dynamically and that no cells may be created of a type unless it is fixed. This, also, is the point at which we will introduce the enforcer of the context-sensitive constraints. That is, we will allow objects of grammar type to be constructed freely, but, before a cell may be created of the new types, we will somehow ensure that the grammar is fixed and conforms to constraints.

(One feasible method of ensuring this follows: We embody a verifier for the constraints (§3.3.3.1) in a subroutine. If this subroutine agrees that a newly tendered grammar is correct then it fixes it and attaches a private property (§2.5.1) with the value True to the root. Before gcons (§3.9.1.2) creates a cell of any type, it examines the root of the type/grammar and value. If they are not there, gcons raises an exception (§3.12) without creating a cell.

(The verifier for grammars has a fixed name, GVerifyGram (§3.3.3.1), and is part of the standard environment (§3.14.3). Thus, there is no difficulty in arranging for it to share a private property with gcons.)

Naturally, we extend dynamic fixation to all cells. It is easy to imagine many uses for it. We are committed to providing the machinery for it, anyway. It costs nothing to make it generally available.

Idents may be fixed. Formally, primitive cells are already fixed, although, since they are atomic, fixation has no effect.

2.5 Other Features of the Data Space

2.5.1 Properties

Just as the gate makes it possible to reach a cell from cells outside the syntagm, we introduce a way to reach cells outside the syntagm from an interior
cell. This is convenient when we wish to attach comments or documentation to a program or to mark a subtree suitable or unsuitable for a certain transformation. These attached objects collectively form the fifth attribute, properties.

To give as much scope to properties as we have given to gates, we wish there to be no limit to the number of them and no limit on when they may be created. Yet, we don't want them to interfere with each other. Last, we wish it to be possible to require a property to conform to some grammar.

We can manage this in the following way: We keep the properties from interfering with each other by creating a property function, a map from a distinguished type of cell, the property markers, to values in the data space. There is one such function for each cell, excluding primitives and idents. The problem of creating new properties thus becomes the problem of creating new markers, which are just cells, so the problem is solved. The problem of conforming to a grammar is solved by making the marker grammar such that the value of each marker is a type and then requiring the property value to be of that type. To avoid a problem similar to the one with types, we require each marker to be fixed before it is used. Finally, each property value is an ident, and we allow broad types.

(The property operator, described in §3.9.5, enforces these two requirements. When a property is referenced, the marker is examined to see whether it is fixed (§2.4). If not, the operator terminates by raising an exception (§3.12). If so, its ident is fetched. If the ident does not exist, it is created with the type held by the marker. On every subsequent occasion, the same ident is fetched. This effectively constrains the type of the property value because it is installed using ordinary assignment (§3.7). As mentioned in §2.3, idents may hold broad types. For details see §3.2.3.)

The map as a whole is never fixed. However, individual property values may be fixed. All property values (idents) reached with a newly created
marker are initially not fixed and have value none.

The property map keeps properties from interfering so well that it gives us a good method of information hiding. A newly created marker is a newly minted key. No one can use it unless the creator provides access to it, and without the marker, the property value is inaccessible. These private properties are the crucial feature used in G when providing an abstract data type, that is, an atom of some particular type, to whose hidden structure only certain routines have access. Private properties can be used for other kinds of information hiding, as well.

The notion of property lists goes back to Lisp [MAE*62, Wei67], where they are attached only to atoms. More recently, we find grammatical properties used anywhere in syntags in [DKLM84] called annotations.

2.5.2 Flat Lists

In CFG as described so far, a list of indefinite length must be described in a rather artificial way, i.e. using recursion. For example, the grammar

\[
S \leftarrow A
\]

\[
S \leftarrow B
\]

\[
B \leftarrow S \cdot A
\]

\[
A \leftarrow
\]

does this. We have introduced an extra nT B to conform to the rule introduced in §2.3, namely that S must be classifiable as an alternation type or a construction type, but not both. A is also explicitly an atom (terminal), which is a construction type.

A syntagm from this grammar, fig. 2.1, shows what is wrong, the undesirable difference of the depth of the leaves, for the sentence AAAA. We remedy this by introducing two new kinds of type, list and plist. With construction, alternation, and gate this brings the total to five. These new types may have
any number of daughters, although they must all be of some one type, a type from the grammar to which the list/plist belongs. A plist (positive list) differs from a list in that it must have at least one daughter to keep it from being incomplete (§3.2.2).

Actually, the list/plist types all have the same number of daughters: infinity or, more precisely, \( \omega \) (fig. 2.2). When a new cell is created all daughters are none. They may then be replaced using assignment. G also provides insertion and deletion, which shift all the cells right of a given point. (§3.9.12, §3.9.13)

My introduction of flat lists is more justified by their widespread use than by any argument about the shape of syntagms. (See, for instance, [Pag81, §2.2].)
2.5.3 Function Cells

Functions are first-class citizens of the G data space. They are orphan atoms. Their semantics, their meaning, is invisible. This part of the design is straightforward. Their types are anomalous, however, and the reason for this requires a bit of explanation.

In languages with invisible types we have mentioned (§2.4) that the compiler sometimes undertakes to verify that expressions may be safely assigned to targets. This implies computing the types of the expression and the target and then deciding whether they are equivalent or, failing that, whether one may be coerced into the other. It is the question of type equivalence that concerns us here.

Broadly speaking, there are two commonly accepted notions of type equivalence [Sch86, p. 178] called structural and occurrence (or name) [Ten81, §12.2.7]. Structural equivalence calls the types of two variables equivalent if they are defined by the same constructor with the same arguments (recursively, when the arguments are themselves constructors, until base types are reached). Thus, in C a pointer to int is structurally equivalent to any other pointer to int.

Occurrence equivalence calls the type of two variables the same only when they are defined at the same place, that is, only when the type definition is exactly the same substring of the source text, the nth through mth characters, for some n, m. In most languages using occurrence equivalence it is possible to give a name to an occurrence, thus making it convenient to create variables of that type at distant places in the source text. But, this does not change the rule that the definition occurrence must be the same one for the two variables.

The rule in G is most like occurrence equivalence. Two cells must share the same type cell in order to be the same type. It doesn't matter if two grammars are structurally identical. Their types are different types.
However, functions are considered to be the same type if their profiles (signatures) are structurally equivalent. Structural equivalence of functions is also used in other languages which generally use occurrence equivalence, e.g. Pascal, Modula-2. Occurrence equivalence of functions is practically impossible, as we show by two examples.

Suppose we have two functions which accept functions as arguments and the types are structurally equivalent. If a user wishes to pass the same function to both, it requires that the type cell describing the argument be the same for both, which means there must be some sort of directory of types. That is, the author of a function which passes a function must be careful to name the type of that function with the occurrence in the directory (which must be unique) of the appropriate type, or else the user will be prevented from passing the function as we supposed because it will be required to be two different types at once. Constructing and maintaining such a directory, while possible, is most impractical.

Again, suppose we have a function invocation:

```
substring( "The quick brown fox", 10 )
```

The argument profile is `string x int`, but is it occurrence equivalent to that of the function? Surely, it is not. We may rule that such an invocation is an implicit coercion or again hypothesize a directory and canonical instances of every type, but to go to such lengths to maintain the apparent consistency of the type scheme is silly, especially since it subverts the intention of occurrence equivalence, namely, to maintain the distinctness of many different structurally identical types.

Now, having decided on structural equivalence, all we need for the type of a function cell is some sort of representation of its profile. (The details are given in §3.2.4, §3.3.4.) I have chosen to separate the grammar of profiles from the grammar of types, although this is arbitrary.
2.6 Parsals

2.6.1 Parsals as Literals

A literal is simply a string representation of a specific syntagm. 35 is a literal int.

while (a->next != NULL) {a = a->next;}

is a literal C statement.

In §1.2 I argued that it is often impractical for the compiler to create syntagms from literals, especially when it must cope with a concrete and an abstract grammar. Of course, it can and does when the literals are integers, Booleans, or strings. But, in general the compiler will recognize an appropriately formed piece of source code as a parsal (parser argument list) and invoke a subroutine to interpret it. This subroutine then has the job of parsing according to whatever rules seemed appropriate when it was written and returning a (perhaps abstract) syntagm to the compiler.

I feel free to omit this part of the design because parsing is now well understood [AU77]. A single parser with the ability to parse every LR(1) language is in G a straightforward programming problem. Once written, it would reside in G’s library of parsers. More specialized parsers, such as one to parse according to a concrete grammar and automatically convert to an implicitly related abstract one, might be written as modifications to the universal parser.

Is the parsing problem solved, then? No, this relationship between abstract and concrete grammars has not been characterized in any general way. It seems a fruitful area for future work, given that sometimes the relationship is fairly subtle. (See, for example, the description and rationale for an abstract C grammar given in §4.2.3.) But, once the two grammars are defined and their relationship is understood, there is enough programming language machinery in G to construct them.
Moreover, there is enough machinery to construct them from a convenient notation. A programming language is, after all, a system of notation. The problem for G has always been to make the notation extensible so that fragments from arbitrary languages could be embedded in it. This is what parsals do. When this was accomplished it paid a dividend in grammar notation: It’s extensible, too. Future developments in the study of grammars can be accommodated without changes in G simply by adding library parsers for the grammar of grammars.

[CI84] have a good convention for interpreting a single grammar as both a concrete and an abstract grammar. (We offer some criticism of it in §4.3.1.3. However, an examination of the merits of different conventions is beyond the scope of this work.) We leave the method of parsing undecided, so the user may choose from the library or invent his own method as needed. (Parsals and suggested conventions for their use are described in §3.9.1.1.)

2.6.2 Exceptions

This does, however, mean that the compiler will run user subroutines. These routines will need a clean way of reporting parsing failures, not to mention the possibility that they may themselves contain errors. For this reason I have included a simple exception mechanism. (Exceptions may be used by any G program, of course.)

The mechanism is based on that of CLU [LAB+81, §12], but has been simplified and brought more into the style of C. The major difference is that CLU’s exception handlers are attached to the statements they serve, putting them at a different nesting level (higher, although they appear afterwards), while G’s catchers follow the statements they serve at the same level.

G’s catchers are weaker in that each may catch only one type of exception (except that one may catch all exceptions). CLU may handle any collection
named explicitly or all not previously named within the current handler.

My purpose in changing CLU's design was to simplify while giving the same power. I believe G's catchers are easier to see because of their nesting level. They may serve several statements or a whole block without being visually removed from it. On the other hand, if one is to serve a single statement in the middle of a block, the construction is no worse in appearance than CLU's.

Any scope of service available in CLU is also available in G with a minor program transformation. It may be fair to say that in G there is more room for an exception to propagate unexpectedly far to a catcher, but even in CLU handlers may be far from the source of their exception. In fact my original complaint was that CLU handlers, even when well placed, must be rather far from their sources.

In any case G's scheme is the most natural when each catcher must only see one type of exception and several are to be caught from one source.

The last simplification is that the number of parameters an exception may pass has been limited to one. This accords with the other method a function may use to terminate, namely the return, which may pass at most one argument. (Exceptions are described in §3.12.)

2.6.3 Parsals and Pattern Matching

Pattern matching is a popular and intuitive method of recognizing a class of syntagms. It asks whether a test syntagm has a certain form. We did this in §1.3 when we wanted to recognize do-while statements.

However, there is some variation in what authors mean by "has a certain form." We might mean, taking the syntagmatic view of program text, that the test syntagm must match a template starting at its root and going some distance downward, perhaps a different distance along each branch. Let us call this tree matching. This is what we did in fig. 1.1 (p. 10) when we did
not care what the statement part was, but insisted that the expression be \( 0 \), thus specifying that branch all the way to leaf level.

Tree matching is the simplest and most intuitive match for tree structured data. It is the basis for Prolog unification [CM87] and has appeared in the context of syntagm manipulation [vdB80, §4.2.2]. G implements a primitive operator for it (§3.9.8), which we describe below.

It is easy to make the template itself a syntagm merely by leaving the parts we don’t care about off, replacing them by none. (This works as long as the test syntagm never contains none.) We will call this kind of syntagm partial, while one that does not contain none is total. (For the exact definition see §3.2.2.)

Having established the usefulness of partial syntagms and partial parsals for templates (For the relevant conventions, see §3.9.1.3.), we will presently explore some other varieties of pattern matching and propose ways of using parsals for them. First, however, let us meet the other main features of parsals.

Besides being literal, a parsal may incorporate syntagms generated by expressions, even other parsals.

For this to be useful, the expressions, and hence the parsal itself, must be evaluated dynamically. We have been assuming that parsals would be static. Now we see both kinds are needed. A small variation in the syntax tells when evaluation is to occur: statically for efficiency or dynamically when necessary.

Last, a parsal can be a multiple assignment, assigning several of the generated cells to variables.

Of course, parsals have all the freedom that comes from evaluation by a user-supplied program: It may contrive to produce abstract syntagms from concrete parsals, or from parsals containing parts from both grammars. It may contrive automatically to generate new lexemes, which it can do by examining the lexeme grammar. The full range of uses of parsals is unexplored. We hope only to give the user scope for his ingenuity.
At present there is only one operator in G for tree matching (§3.9.8). Several other possibilities exist, but we await experience with an implementation before proceeding. The present operator takes a template and a syntagm and, if they match, copies pieces of the syntagm into the template. This has the effect of setting variables in the template parsal to the unknown parts of the matched syntagm.

2.6.3.1 Advanced Pattern Matching

We conclude by sketching some methods of using parsal templates in pattern matching situations where simple tree matching is not sufficient.

We first note that the most likely place for a template to stop is after an alternation node or gate, since the type of its daughter is unknown. In contrast, the daughters of a construction node may easily be filled in by examining the grammar. For a list, the type of the daughters is known, but not the number of them.

Now suppose we want to specify something about the daughter of an alternation, for example, that the subtree there must be identical to another in some other part of the syntagm, perhaps that the two $x$'s are identical in

\[
\text{if ( expression ) } x \text{ else } x
\]

One way to do this is to use properties. We may attach a property to each of the two dummy \texttt{statement} cells, linking them together and specifying that they be the same. Then the matcher, seeing this property, acts appropriately.

Even without any new primitives it is entirely within the power of the user to use this kind of matching. He writes the parser to recognize the request in the parsal for some sort of property flag, which then goes to a user-written matcher which uses it to guide the matching algorithm. We hope that most matching problems can be solved in this way, leaving the burden of solution
of all but the most common problems on the user rather than on the language designer.

Another common matching problem is list matching. Suppose we wish to find whether a statement list contains a backward goto:

```
: up: ...
: goto up;
: ...
```

This requires us to use the above trick, but also to recognize two subtrees separated by an unspecified distance. Again, we use properties in the template. We mark the template subtrees, the two statement’s saying that each may be preceded by an unknown number of syntags in the statement list. (For several other interesting list matching problems see [HC90], who have explored transformations on assembly language code.)

The problem of matching a true none in the test syntagm may also be handled using properties.

There is one problem which is presently awkward to solve, and for which a new design seems necessary, namely delayed assignment. Suppose, as in §1.3, we match a syntagm and want variables to point to parts of it, but, instead of pointing to copies of the parts, we want them to point to the originals, most probably because of properties attached to them. We must construct the template, match it, and then return for assignment, not cells of the template syntagm, but cells from the matched one.

It can be done, at present, by having the parser carry out the required sequence of operations. For example, the parser might build the template as usual, but then encounter the syntagm to match inside the parser, guarded by a special flag. It would then carry out the match and return the matched
cells for assignment. However, a more elegant solution, especially one viewing this as one of a class of related problems, must be left for future design.

2.7 On Assignment

This section deals with some miscellaneous issues related to assignment but not properly data space issues.

2.7.1 L-Values

We shall call the assignable values, *i.e.* the elements of the data space except idents, *R-values*. We shall call places to put *R-values* *L-values*. An *L*-value is an integer \( n \) and a cell which takes \( m \) daughters, where \( 1 \leq n \leq m \). The names \( L \) and \( R \) refer to the left- and right-hand side of an assignment statement, of which the left-hand side must be an *L*-value. The right-hand side may also be an *L*-value, since it’s easy to convert a location, the \( n \)th daughter of \( x \), into an *R*-value, the cell held in that location. (*L*-values are described in §3.6.)

As mentioned in §2.3 it is only possible to denote the ident associated with a program identifier by actually writing the identifier. It is now possible to explain how I have implemented this. The meaning of an appropriately declared program identifier is an *L*-value, the daughter of an ident. By restricting the passage of *L*-values I have contrived never to give an identifier an alias.

*L*-values are never passed to subroutines. They are converted to *R*-values when they appear in argument lists. Most primitive operators also convert their arguments to *R*-values. The exceptions are assignment (and assignment-like operations: parsal assignments, list insert/remove) and *fixid*, whose job is to fix idents.

The result is that it is simply impossible to write an ident as an *R*-value, so it is never held in another cell. This strongly restricts the class of expressions
which yields idents. In fact there is only one other (non-identifier) way to do so, the property expression, where idents contribute to the security of private properties.

2.7.2 Nil and Foster Types

There are two types that are constrained to be part of every grammar. That is, two additional rules are invisibly added to each grammar, so that, for example in Pascal, the types \texttt{Pascal@nil} and \texttt{Pascal@foster} would automatically exist. (For more on this see §3.3.3.1.)

The \texttt{nil} type is used to denote the absence of an optional element in a syntagm. For example, consider this rule, which might be from an abstract C

```plaintext
if_statement
  expression
  statement
  ( )
  nil
```

Figure 2.4: Without \texttt{else}-clause.
CHAPTER 2. DESIGN PHILOSOPHY

grammar:

\[
\text{if\_statement} ::= \text{expression statement } [\text{statement}]
\]

It refers to the fact that the else part of an if-statement may or may not be present, giving rise respectively to fig. 2.3 or fig. 2.4. The nil cell is a true cell with all attributes. It exists so that the syntagm may be distinguished from a partial (i.e. partially specified) syntagm, which contains none.

The principle is, if an alternation cell may logically hold any of several types or none of them, then in practice the nil type is included as one of the allowed types, and the alternation is required to hold one.

The foster type is a cell which can be the mother of any other cell type in its grammar. It is just a large alternation. The reason for its existence can be shown by an example.

We are often more interested in the location of a syntagm than in the syntagm itself. Consider a grammar-based visual editor. There would be a cursor showing by visual indication the current syntagm, the point of interest. Suppose a user indicates that he wishes the current syntagm replaced. He expects to see the screen rewritten with the replacement syntagm in place of the other, embedded in the original context. The cursor should appear on the root of the replacement. Clearly the cursor indicated a location, an L-value.

How, then is the programmer to proceed? Can he always store the value of the mother of the cell of interest? Root cells do not have mothers. Can he make a special case for the root and create a gate to point to it? Well, no, not if there are several cursors, he cannot, not easily. That would require that all cursors would, when necessary, migrate to the same gate, which implies a lot of cross-checking between cursors.

The only natural, simple solution is for the stored value to be the apparent value, the cell of interest. The user wants the cursors, whether they are naturally L-values or R-values, to behave properly as a side-effect of assignment.
CHAPTER 2. DESIGN PHILOSOPHY

This means that it must be possible to find a context for any cell, which is why I invented foster mothers. They exist to be the mother of any cell which is logically an orphan, but whose mother must sometimes be found.

Fosters have a special status with respect to assignment. Since no cell may have two mothers, there is a conflict when a cell which already has a mother is assigned to a new one. I have chosen to fail the assignment in this case by raising an exception, rather than destroy the previous link. This also differs from [CI84] who, in this situation, would silently copy the R-value. Just in case, however, the assigned cell's mother is a foster, the link is silently broken and G proceeds with the assignment. (This conflict does not, of course, arise when the assignment is being made to a gate.) For this reason fosters become a part of the primitive machinery of G.

2.7.3 Coercion

It is often the case that several cells represent the same text. This happened in fig. 1.1 on p. 10 when no fewer than four cells (expression, primary, constant, and int) stood for the expression 0. This is unfortunate because we are used to thinking of program elements, expressions and statements, as text, and not to recalling all the possible types they may have. The difficulty is alternations and gates, i.e. cells that have a single daughter, so that mother and daughter represent the same text.

In an assignment it sometimes happens that one has a cell and a target L-value type that are not identical but are related in this way. What corresponds to type coercion in other languages is tree navigation in G, and the same issue arises: whether to coerce a cell into the required type when it is not that type already—or, rather, how much trouble to take doing the coercion. From the standpoint of mathematical simplicity it is cleanest to take none. The user can certainly write coercions explicitly. But, from the standpoint of software
engineering, especially when trying to give the best readability to programs, the judicious use of automatic coercion seems indicated.

There are three kinds of coercion in $G$: into alternations and into and out of gates. (§3.7) The first case moves only upward in the tree from the value to the target type. Although I considered the downward coercion, I omitted it on the theory that it would often fail. The daughter of an alternation might be any alternative. I hoped the user would write an explicit test when moving downward, and if he didn't, then at least the failure would be a simple type error, rather than the more complex failure of coercion.

Two gate coercions were also included to allow the use of a list of gates as temporaries. (Again, see §3.7.)

I am not entirely satisfied with the choice of coercions; it seems arbitrary and ad hoc. My philosophy has been minimalistic: to include only those coercions for which I had a good reason and to include no others. Thus, there are no combinations of gates and alts, no double gate coercions, no coercions where the system must be very clever making its calculation.

Quite a different kind of coercion was considered for function cells, one which compares profiles. If a function is a map from $x$ to $y$, it may reside happily in a variable which holds a map from $a$ to $b$, as long as $y \subseteq b$ and $a \subseteq x$. If we say that the function's type is $p$ and the variable's $q$, we may then define $p \subseteq q$ and use this rule recursively.

This is the so-called arrow-rule [CDJ*89, §3.4], and I forewent including a coercion based on it with real regret. There was a temptation to explore the type structure it induces because of its apparent richness. However, this remains for future work.

Although I have tried to be minimal in the kinds of coercion I have allowed, I have not succeeded in making all coercions statically visible. The dynamic type structure of $G$ forbids this. The most I can say is that some coercions may be detected and in-line coded by the compiler. This also must remain a
2.8 Input/Output and Parsing

There are no primitives for I/O in G. It is to be done by library subroutines, as yet undefined. However, I have given some thought to how input characters, once obtained, will be converted to syntags.

The inspiration comes from an intuition about when two strings represent the same thing. This is a model that is quite specific to programming languages. A facility has been sketched as a library subroutine.

The intuition is, then, that two statements appearing at different places in a program are never the same thing, even if they are character-by-character identical. Their equivalence is only coincidental. No two declarations, no two expressions, no two syntags of any programming language syntactic category can be.

What can and should be identical are two lexemes. Two occurrences of the same identifier should be the same thing; in G this means the same cell. Using gates and the natural division between the language grammar and the lexeme grammar, we will link different occurrences of a lexeme, which will be different leaves of the language syntagm, to the same cell, a canonical representation of that lexeme.

To accomplish this I have designed u-strings (unique strings §3.4). They are supported by a few routines which, when given a string, return a ustring atom—the same atom for the same sequence of characters.

With this in mind we envision a lexical reader which returns, not a string, but a u-string. It must refer to a lexeme grammar to do this, of course, but u-strings are from their own grammar. It would be best for the reader to tag these atoms with their token types (using a property) so that a parser need not examine the string represented by the u-string (which is also attached as
a property).

Is this scheme versatile? If two lexeme grammars are in use simultaneously the token tags may be kept on two separate properties. Identifiers themselves actually present a similar problem. One string may represent different logical identifiers in different scopes. A solution is for each scope to own a property and use it to find the data relevant to that scope attached to the u-string. There are other approaches too, but we can at least see that the scheme is sufficiently versatile. (One could simply store u-strings in the scope data structure, or attach them as properties to the root of the scope syntagm.)

There are some things which are probably not best as u-strings. String literals from the source code are probably best presented as strings. Also, a detailed description of white space, especially comments, when this must be preserved, is no doubt best presented in some ad hoc way.

2.9 Implementation Issues

2.9.1 Early Execution

In addition to parsals, which may demand it (§2.6.3), there are two kinds of functions which may be executed early, as part of compilation: initializers and pure functions.

The decision to allow functions to be executed as part of initialization follows from the use of parsals as initializers. Before we can declare anything we must be able to name its type. Therefore, we must be able to declare and initialize a type constant. (Except for the axiomatic existence of the grammar of grammars (§3.3.3, §3.14.3) this would involve us in an infinite regress.) Although it is possible to build a grammar- constructor into the compiler, this is inelegant and unnecessary. We must already turn text into syntagms for program literals. That's what parsals do. All we need do is postulate parsals
for the grammar of grammars and we have our type syntagms.

One kind of parsal already gets early execution, but a static initializer, no matter what it is, must get early execution by definition. The only question is whether to allow unrestricted expressions as initializers, and since we have allowed parsals, we may as well do so. (See also §3.10.2.)

This choice may involve the compiler in loading, perhaps building up as extensive an environment as the final execution environment of the program. This is unusual, some might say inelegant, but the alternative is to build the grammar-constructor into the compiler, which would make it rigid. Also, if we delay the production of other initializers until just prior to execution, we deny the compiler access to any namelist symbols which might be generated.

The namelist (explained more fully in §2.9.2) is a method for introducing a set of names during initialization into the compiler’s symbol table. The compiler finds the names by searching the syntagm that results from evaluating the initializer. The namelist facility exists so that the user need not reiterate the names (i.e. type names) of a grammar both inside and outside a grammar literal. It is a general facility, however, and need not be used only with grammars.

The other kind of early execution, pure functions, is in a sense similar to the previous case: the generalization of a necessary facility.

Functions in imperative languages often have important side-effects, beyond the construction of their result, the returned value. This is in contrast to mathematical functions, which do nothing but return a result, which in fact are their result. When an imperative function is free of side-effects, we call it pure to liken it to the mathematical one.

A function definition may be marked pure to notify the compiler of this. If it is so marked and invoked in an expression, the compiler may, if the values of the arguments are known to it, evaluate it immediately as an optimization, substituting the result in the expression. The compiler is not constrained to
do so; it is only an option. (See also §3.2.4.)

This is a generalization of a standard optimization on descent (selection) operations. We often descend from a cell to its daughter, not by giving the number of the daughter, but some name, perhaps statement (a type name), when only one of the daughters of the cell is of type statement, or stmt (a given name), when there are several. This name usually comes via namelist in a static initialization, so the function which calculates the number given the name is a good candidate for early execution.

2.9.2 Namelists and Separate Compilation

The namelist is a method by which an initializer may add symbols to the compiler's symbol table. It is a property with a specific marker attached to the initializing syntagm (§3.5). Its reason for being is first to let the user avoid repeating the names of a grammar literal (§2.9.1, above) and second to let him avoid writing them at all, as follows:

Usually, a grammar will be a syntagm compiled on a previous occasion, stored in a library. Since the number of programming languages is small (that is, compared to the number of programs written to process them) and the task of writing grammars is exacting, this is the reasonable thing to do. By loading an external grammar from a library a user avoids writing type names, even avoids knowing them, those which are not relevant to his purpose. This contributes to the brevity, and therefore readability, of his program.

(We must distinguish between the compiler's symbol table and the loader's symbol table. The loader may know symbols the compiler does not, since loaded modules may define many symbols besides the one that caused the loading. The compiler drives the loading process by demanding modules needed for namelist symbols. No others are loaded.)

We can go farther in the direction of brevity if we enlarge our view a little.
If a grammar relies intimately on another grammar it could be that it is never used without the other, and also that its namelist is never used without the other's. In G there is a mechanism by which a namelist may demand the inclusion of another namelist without a declaration in the program text. This added brevity may sometimes be useful. I have taken some care to make sure that declarations have a simple structure but still give the user some control when namelists cascade like this.

One of the things under the user's control is whether namelist identifiers will be qualified. We use qualified in the sense of Modula [Wir85, §24]: the appearance of the grammar's name as the initial part of the identifier, as in Pascal@statement.

Cascaded namelists may be no less qualified than those that demanded them.

Unfortunately, the most frequent use for a secondary namelist is rather clumsy. Every programming language is built on another language, its lexeme language; we cannot use the first without the second. However, if we use the model of §2.8 all the leaves of the language syntagms will gate to u-strings, rather than lexeme syntagms. Although the lexical grammar can still be demanded, it will not be used anywhere in the programming language grammar. The true types of the leaves must be represented in some indirect way, probably with properties.

2.9.3 Memory Allocation and Separate Compilation

While running, the compiler must sometimes pause to load in previously compiled syntagms, especially grammars. (See also §3.14.1.) When the compilation is complete memory contains both the loaded syntagms and the newly created ones. Since memory is allocated as a heap it is easiest simply to let the syntagms become intermixed.
CHAPTER 2. DESIGN PHILOSOPHY

However, it should be that, as the result of compilation, the new syntagms are written out as a compact block. They must not be contaminated with the old ones or with free storage. How can this be done?

To show feasibility we will summarize one method based on a Lispesque scheme for storage management [MAE'62, §7.5]. Suppose that all memory is allocated in small blocks of one size. To store large structures like lists, several blocks must be allocated and linked together with pointers. During the ordinary course of execution these blocks are not moved. The garbage collector creates a chain of free blocks so that allocation can be done simply by taking the first block off the chain. In this method free blocks become scattered throughout memory.

In order to cope with loaded syntagms we must guarantee that they are loaded in compact blocks. To do this we must have a storage arena of variable size so that if necessary we may increment the size, thus creating a contiguous block of free storage for our new syntagm.

When we have loaded the syntagm we record its location and length along with the locations of cells named by external symbols in a CDE or contents directory entry. CDE's are stored in a special area which is used only by memory management routines.

While the compiler is running, a routine keeps track of whether any alteration is made to any loaded syntagm by monitoring assignments, etc. The slightest change made to one, even to a property, mother, or fixation, makes the CDE dirty. Dirty CDE's are treated like the created syntagms and become part of the output module.

When compilation is finished, storage is compressed. First the storage associated with clean CDE's is marked free. Then all allocated blocks are moved in memory order to the then lowest free location, forming a compact mass in low memory. It is necessary during this process to change the value of every pointer in memory. This is done using the bit map that shows which
blocks are free.

Using this map it is possible to compute the destination of any allocated block in memory, given its original location. The simple-minded use of this map would make compression an $O(n^2)$ operation, but by building an index this may be improved to $O(n)$.

Some of the pointers will, of course, point to cells in syntagms of unloaded CDE's. These must be reinstated as external references using the information in the CDE and added to the list of external references. The compressed information is made relocatable and is then ready to be written to the output file.

### 2.9.4 Idents and Separate Compilation

Some identifiers do not naturally have associated idents. For example, the type cells in a local grammar are constructed according to a grammar literal and returned as part of a cell structure with no idents, yet they have names. In this case the compiler knows that they represent R-values and treats them as such, but in separate compilation it must sometimes be instructed of this with an explicit declaration.

This takes the form of the word `rval` attached to a declaration, and its meaning is that the name declared must never be used in the current scope as an L-value, *i.e.* as if there is an ident. (See also §3.10.1.)

This declaration is useful in its own right, of course, as program documentation, and so may be attached to strictly local identifiers at the user's discretion.

The global case, the case in which several declarations for a name refer to the same object, we will call *linked* (§3.11). Linked declarations may disagree regarding `rval` provided that the definition does *not* have it. The value of such an object may be changed as long as it is changed outside the scope of
an \texttt{rval} declaration.

The compiler copes with this possibility by assuming the ident exists and by creating idents for all defined objects that have linkage. A special check must be made when separately compiled modules are loaded together to insure that \texttt{rval} definitions are linked only to \texttt{rval} declarations. Failing this, the loader must mark the result in error. (A weak alternative to this check is feasible for atoms, especially function cells. The ident may simply be fixed, and an attempt to change it is caught during execution.)

The loader must check and report several other similar errors which are unknown to conventional program loaders, disagreements between declaration and definition with respect to type and fixation and, for functions, disagreements about exceptions generated and whether \texttt{pure}. 
Chapter 3

G Language Manual

Nobody steps on a church in my town.
“Ghostbusters”
(dir.) Ivan Reitman

3.1 Lexical Conventions

3.1.1 Tokens and White Space

Blanks, tabs, newlines, and comments are white space.

If the input stream has been parsed into tokens up to a given character, the longest possible string is always taken to be the next token.

The use of white space distinguishes qualid’s from targid’s and typeid’s, both used only in parsals. (See §3.9.1.1.) Otherwise, its only purpose is to separate otherwise adjacent tokens.

Comments are delimited by /* and */ They do not nest.
3.1.2 Identifiers

An identifier, a member of the category \( id \), is a sequence of letters and digits beginning with a letter. Any such sequence is an identifier unless it is a keyword. See §3.1.3. Identifiers are of unlimited length. Case is significant. _ is a letter. A \( qualid \) has the form \[ id \@ id \] (See §3.3.1.). It may not contain white space.

3.1.3 Keywords

Table 3.1 is a list of the keywords. They are never identifiers nor are they either part of a \( qualid \).

<table>
<thead>
<tr>
<th>ANY</th>
<th>extern</th>
<th>index</th>
<th>static</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>False</td>
<td>length</td>
<td>switch</td>
</tr>
<tr>
<td>case</td>
<td>fix</td>
<td>mapcar</td>
<td>True</td>
</tr>
<tr>
<td>catch</td>
<td>fixed</td>
<td>namelist</td>
<td>type</td>
</tr>
<tr>
<td>cons</td>
<td>fixid</td>
<td>none</td>
<td>unequal</td>
</tr>
<tr>
<td>continue</td>
<td>fixidp</td>
<td>pure</td>
<td>varying</td>
</tr>
<tr>
<td>default</td>
<td>fixp</td>
<td>resignal</td>
<td>void</td>
</tr>
<tr>
<td>do</td>
<td>for</td>
<td>return</td>
<td>while</td>
</tr>
<tr>
<td>else</td>
<td>gcons</td>
<td>rval</td>
<td></td>
</tr>
<tr>
<td>elseif</td>
<td>goto</td>
<td>signal</td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>if</td>
<td>signals</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Keywords.

Certain identifiers appear in the standard environment. A list of them appears in §3.14.3. They are not syntactically distinguished from other identifiers, but their use in other than their standard sense is discouraged.

3.1.4 Literals

There are four kinds of literal: integer, character, Boolean, and string. none is a literal and a keyword.
Integers (int) are denoted in decimal notation and may have a leading -. There's an implementation-dependent largest and smallest.

Characters (char) are begun with ' a single quote. Excepting white space, any single printing character may follow. The next following character must be non-comment white space. There is no closing quote.

The following sequences also represent characters.

'\s   SP space
'\n   NL newline
'\t   HT tab
'\b   BS backspace
'\r   CR carriage return
'\f   FF form feed
'\   backslant
'\'   single quote
'\"   double quote
'\"   double quote
'\012 1, 2, 3 octal digits
'\xOA 1, 2 hex digits
'\x0a 1, 2 hex digits

In all cases, the following character must be white space.

There are two Boolean (bool) literals: True and False. They are also keywords.

3.1.4.1 String Literals

Strings are delimited by u-quotes. (This term half translates the German Umklappanführungszeichen or "flip quotation marks".) They consist of an n-open-quote and an n-close-quote, where n must be the same non-negative
integer in both cases. An \textit{n-open-quote} is

\begin{verbatim}
intosh\end{verbatim}

\textit{n slant marks}

An \textit{n-close-quote} is

\begin{verbatim}
"intosh\end{verbatim}

\textit{n slant marks}

All characters have their natural meanings within string literals (in particular, there are no comments inside strings) except those preceded by an escape sequence: either an \textit{n-backslant} or an \textit{n-single-quote}. (Again, \textit{n} must be the same integer as it was in the \textit{n-open-quote}.) An \textit{n-backslant} is

\begin{verbatim}
\intosh\end{verbatim}

\textit{n slant marks}

An \textit{n-single-quote} is

\begin{verbatim}
\'intosh\end{verbatim}

\textit{n slant marks}

After an \textit{n-backslant},

\begin{verbatim}
,mbf\end{verbatim}

have similar meanings to above. Of course, \texttt{\'"} are most easily put in strings bare when \texttt{n} > 0. An \textit{n-backslant} followed by a \texttt{NL} is ignored (does not appear in the string).

String literals are interpreted and converted to syntagms of the \texttt{string} grammar. See \S3.3.8. They are then fixed. Their length may be found with the \texttt{length} primitive. See \S3.9.1.2.

An \textit{n-single-quote} may only appear in strings inside \texttt{parsals}. (See \S3.9.1.1.) It is effectively an anti-quote, causing the following string (up to the next white space) to be removed from the quotation. Using an \textit{n-single-quote} is exactly
equivalent to writing 3 (or 2 or 1) separate things, as shown in these examples:

"XXX 'YYY ZZZ" $\Rightarrow$ "XXX " YYY " ZZZ"

"XXX 'YYY" $\Rightarrow$ "XXX " YYY"

"'XXX YYY" $\Rightarrow$ "XXX " YYY"

"'YYY" $\Rightarrow$ YYY

The white space following the single-quoted object joins the second string.

### 3.2 What's in a Cell?

The set of objects which a G-language program may manipulate is comprised of two broad classes, cells and idents, plus one special element, none. Cells have a superset of the attributes of idents. (Actually, primitive cells have 3 attributes, just like idents.) none is a single structureless point with no attributes.

An ordinary cell has 5 attributes: value (also called daughter(s) or content), type, properties, mother, and whether it is fixed.

The value of a cell is a list of cells. The length of the list is fixed, and may be zero, finite, or infinite, depending on the cell type. Any element of this list may be none, and all but a finite number of elements of an infinite list must be none.

Intuitively, the value of a cell is its most important characteristic, and we may speak of it in a variety of ways. We may speak of the cells or daughters held by a given cell, or the subtree rooted or at a given cell, always referring to its value.

The type of a cell is fixed and may never be none. The type is itself a cell. The familiar types, such as int, char, and bool, as well as user-defined types, are all cells. In fact, they are all cells of the same type, which is (as
it must be) a cell. Its name is GRule. (This is not the case for function type
cells, for which see §3.2.4.)

There is a map associated with each cell, called its properties. We define
property marker (property, speaking loosely) to be simply every cell of type
GramProp. (See §3.3.5.) The map carries each property marker to some cell,
the property value. Property values may be none, and all but a finite number
must be none. (Intuitively, properties which don't apply to a given cell are
none.)

Consider a cell. Its value is a list of cells. It itself is held as a value by
other cells. At most one of these is called its mother. The reason for this
is as follows: G language is primarily concerned with tree structures. Cells
are meant to be tree nodes, having only one mother (zero if they're the root).
From each cell, we should have the ability to move either toward the leaves (to
daughter and granddaughter) or toward the root (mother and grandmother).
This is a doubly linked tree.

However, a singly linked cell is needed. It is also useful to be able to pass to
an arbitrary cell in an existing tree from a place outside of it, from a pointer,
as it were. These pointer cells are called gates in G. The value of a gate is only
a single cell. The gate is never the mother of its target. Gates are first-class
cells, however, with all attributes including mothers of their own.

Gates also find application as the leaves of a tree. When a leaf has a
meaning which can easily be embodied in a tree-structure of its own, a gate
is used to reach it.

Motherhood (i.e. whether a cell is a gate or not) depends on type and is
therefore fixed for each cell. The mother of a cell is none if it is an orphan,
that is, if it is held as a value only by gates.

Ordinarily, the value of a cell may be altered dynamically by the user.
However, it is possible to block this by fixing a cell. Fixation applies only
to the value of the cell. If a cell is fixed, all the daughter cells are also fixed.
recursively. (If a gate is fixed its target is fixed.) Its mother and any gate cells which hold it as a value are not thereby constrained. Fixation may occur dynamically, that is, a cell may be variable at the start of program execution and be fixed subsequently. This causes recursive fixation of the value cells. Properties are not thereby affected. An attempt to vary (the value of) a fixed cell is an error. Fixation is permanent. Properties are independently and individually fixable. (See §3.2.5.)

3.2.1 Idents

There is a second class of (second-class) objects called idents. Some program identifiers denote idents, hence the name. They have only the attributes value, type, and fixation. They have no properties nor mothers. They are effectively gates. They are never values.

Moreover, gates and idents may be permissive about the types of their values. Instead of naming specific types, they may use broad types (§3.2.3).

In order to fix the denotation of a program identifier completely, the ident must be fixed.

Each cell-property pair has an associated ident. In order to fix a property value completely, this ident must be fixed.

Program identifiers which are declared (but not declared static) have idents in automatic storage, which means they are allocated on a stack. Ordinary cells have dynamic (heap) allocation.

3.2.2 Type and Grammar

The type of a cell defines a template that the cells of its value, its daughter cells, must conform to. More precisely, the types of the daughters are defined by the template.

A cell's type is itself a cell. Except for function cells (§3.2.4) the type cell
is embedded in a structure called a *grammar*. (For example, the type of a type is always *GRule*, a particular cell in the grammar of grammars, *GramGram*.) The template for each type is recorded in the grammar. Several cells may have the same type. Related types are grouped together in a single grammar, and several different grammars are typically used in a program.

A grammar is just another cell structure, built according to the grammar of grammars. It embodies the complete description of the types it defines. It is common practice for a user to build a new grammar using *GramGram* and then immediately start creating cells of the new types. This is an important reason for the existence of visible types and gives G its particular dynamic character.

(Note, however, that before the user may use a grammar to create new cells, it must be fixed and verified. Fixation is described above. Verification is done by a library routine which enforces the rules set out in §3.3.3.1.)

There are 5 kinds of type: construction, alternation, list, positive list, and gate. They are abbreviated, respectively, *cons*, *alt*, *list*, *plst*, *gate*.

Except for gates, types take their component types exclusively from their own grammar.

Recall that *none* may be the daughter of any cell in any position. Thus, it effectively has all types. The presence of *none* may make a cell *incomplete*. We define this term as we go, for each of the 5 kinds.

A construction (*cons*) cell has a finite fixed number of components (perhaps zero), each possibly of a different type, but each of some one particular type.

A cell is called *atomic* if it is a zero-length *cons* cell.

An alternation cell has exactly one component, but it may be of one of several (at least one) different types.

An alt or *cons* or gate cell is called *incomplete* if it has *none* for a daughter. A cell (or rather the subtree rooted at that cell) is called *partial* if it is incomplete or if any of its daughters is partial.
A list has an infinite number of components, each of the same one type. A plist has an infinite number of components, each of the same one type. It is distinguished from a list only in the slightly different definition of incomplete which applies to it.

Lists and plists have none in many positions and only a finite number of cell daughters. If all the positions up to the last cell are cells, the list/plist is complete. Otherwise it is incomplete. In addition, a plist is incomplete if the first daughter is not a cell.

The above definition of partial will be used again in our discussion of matching (see §3.9.8) where we shall consider structures containing daughters, granddaughters, great-granddaughters, etc., so that we shall naturally think of them as trees. When we wish to emphasize that the trees are constructed according to a grammar (syntax), we will call them syntagms. If we use the word syntagm when speaking of an interior cell, we emphasize that the cell is the root of a subtree.

Last, a gate cell has exactly one daughter. It is never a mother (against the others, which are always). Its value's type need not be from its own grammar. Its value's type may be broad.

### 3.2.3 Broad Types

We have more power to specify the set of types which gates and idents may hold than we have for other objects. There are four ways of specifying this set, broad and narrow, and these ways apply always and only to gates and idents. (We agree that it is an abuse of the language to say a gate’s type is broad, meaning only that its value’s type is specified in one of the four ways.)

First, this set may consist of a single type, a single GRule. This type is never restricted to any one grammar; it is one particular type from any grammar.
CHAPTER 3. G LANGUAGE MANUAL

Second, it may be all types from some grammar. Again, which grammar it is is not restricted; it may be any grammar. Such a specification uses the word any, as in Pascal\texttt{any}.

Third, it may be all types. This specification is written \texttt{ANY}.

Fourth, it may be some particular function type ($\S 3.2.4$).

(Although it makes sense, there is no type inclusion for function gates. Function cells must be held either in the third or fourth way.)

A representation of broad types is used in grammars and property markers. It is described in $\S 3.3.3$.

3.2.4 Function Cells

A G-language program, when compiled, becomes a cell. (Of course, it is attached to a large hidden structure.) Such a cell is an orphan atom, and because it is atomic it is effectively fixed. (G-language primitive operators, however, are not cells.)

The type of a function cell is a syntagm of the \texttt{functype} syntax. (See $\S 3.3.4$.) This syntagm is the function's \textit{profile}, and thus describes the parameter types and the returned type of the function.

As far as I know, there is no agreement on what the proper words are for expressions passed and received by functions. So, I define \texttt{parameter} to be the formal item in the function declaration and \texttt{argument} to be the actual value passed in.

The types of the parameters and the returned value may be broad. In addition, the function may be declared \texttt{void}, in which case nothing is returned at all.

At function invocation, arguments are processed left to right and assigned to newly created idents of the denoted types. A parameter list may be specified varying, in which case there must be zero or more (simple) parameters.
followed by a single parameter which is a list of gates. In a varying invocation, the arguments corresponding to the simple parameters are processed as before. Any remaining arguments are each assigned to a gate of the list and the list is assigned to the last variable. Thus, a fixed length invocation must have the same number of arguments as parameters. However, a varying invocation with \( n \) parameters may have \( n - 1 \) or more arguments.

Except for a function with a \texttt{void} return, the profile may indicate that a function is \texttt{pure}. The value returned by a \texttt{pure} function depends only on the arguments.

The \texttt{pure} declaration exists just to allow an optimizing compiler extra leeway. A function declared \texttt{pure} may be executed by the compiler during compilation if it can determine the values of the arguments. The result will be substituted for the function call. (However, functions used in initializations need not be \texttt{pure}.)

For two value cells to be of the same type, their type must actually be the same cell (of type \texttt{GRule}). However, function cells are considered to be the same type if their \texttt{functype} syntagms are structurally identical. These two rules are used recursively on the leaves of \texttt{functype} syntagms (the parameter and returned types).

### 3.2.5 Properties

Associated with each cell is a map from every \textit{property marker} to some \textit{property value}. A marker is simply a cell of type \texttt{GramProp} which has been assigned an appropriate value and fixed. (§3.3.5) The marker value denotes a broad type. (§3.2.3, §3.3.3) The \textit{property value} must be of the type denoted by the marker. When first created, properties have value \texttt{none}.

As implied, properties may be created by the user dynamically, just as cells of other types may be. A property so created is distinct from all others.
There is no way of examining such a cell if it is not returned or somehow passed by the creating routine. Such properties are called *private properties*. They give the creator the power to hide information on cells of arbitrary type. The private property is the sole method of information hiding available in G. It will be seen to be equivalent in power to the methods of more conventional languages.

### 3.2.6 Primitive Cells

Restrictions apply to certain primitive cells. All the cells mentioned in this section are atomic.

True cells of type `char`, `int`, and `bool` may never have properties or mothers.

(In the current implementation, it is possible to create new (false) cells of these types, which then may have properties, these cells are not useful since they are not recognized by the operators that manipulate these types. The operators raise exceptions instead.)

Cells of type `ustring` may have properties (and mothers). However, again the true cells are only gotten from the associated operation `custring`. (See §3.4.)

`none` is not a cell at all. It may not have properties, type, value, fixation, or mother.

Function cells may have properties. In the current implementation only the compiler may create function cells.

`void` is not a type or a cell. It is the word used in function type declarations to denote functions which return nothing (§3.9.1).
3.3 Foundation Grammars

In this section are collected the grammars which form the foundation of the G data space. Most of them have already been mentioned in §3.2. The grammar for parsals is properly a foundation grammar but is given in §3.9.1.1.

3.3.1 Names in Grammar

Since we will need to name cells in this section, we give a brief explanation of the sense in which a cell can have a name. The most important is the existence of the name in the compiler's symbol table, so that the use of the name in a program denotes the cell or ident.

The compiler may learn the name of a thing in two different ways, either from a declaration or from an initialization. The situation is a little more complex than is given here. (See §3.10.) However, for our purposes, it's reasonable to summarize the situation as follows:

When the compiler learns a name in a declaration, it learns a simple identifier, a string of letters and digits starting with a letter. This denotes an ident. However, an initialization, of which a grammar is one kind, creates an extended structure of cells, many of which may have names. These names are qualified identifiers of the form $x@y$, where $x$ and $y$ are simple identifiers and $x$ is fixed for any particular initialization. This gives rise to names like $\text{Pascal@statement}$.

The forms $x@y$ and $y$ are called qualified and unqualified, respectively. If we use $x$ by itself we will call it qualifying name or q-name. Qualified and unqualified identifiers are treated as tokens by the lexical analyzer. (§3.1.2)

The compiler recognizes all qualified names which are visible in the current scope. The compiler may be instructed to recognize the unqualified form also. A q-name $x$ appearing by itself is interpreted to mean $x@x$, as in $\text{functype@functype}$.

See also §3.5. Every grammar given in this section
has a cell whose name is the same as its q-name. Sometimes it is an alias.

For each grammar, there is a conventional use of its internal names, either qualified or unqualified. We state it as we go. The names appearing here all have global scope. (§3.11)

There is a second sense in which a cell may have a name, which may occasionally give rise to confusion. A cell may be thought to have a name permanently, even though, as we shall see in §3.11, the compiler may temporarily suspend a cell's name in order to use that name for another cell. Throughout this work I use foundation type names (string, int, GRule, etc.) in this permanent sense, referring to the unique cell defining this type. A user who suspends these names (This is very bad coding practice.) may encounter abstruse and arcane errors in his compilation when two cells with the same name are confounded.

Many of these names begin with a G. Users of G-language are expected never to use names beginning with capital G, so as to avoid conflict with existing and future system names.

3.3.2 The Meaning of a Grammar

In common usage a grammar's reason for being is to be the method of assigning syntactic categories to words and phrases in a string, in short, for parsing. In G its main purpose is type declaration. A G grammar may also be useful for parsing, but it must be designed especially for the purpose.

Consider this example of a simple type declaration grammar:

\[
\begin{align*}
  s & ::= \text{list} \mid \text{atom} \\
  \text{list} & ::= \text{s}^* \\
  \text{atom} & ::= \\
\end{align*}
\]

It names 3 type cells, s, list, and atom. (We use the unqualified names.) Cells of type s are alt kind, and each may have one daughter, either of type
list or type atom. Similarly, list's are list kind and atom's are atomic cons kind. (See §3.2.2.)

Figure 3.1 is a syntagm of type s. Each circle represents a distinct cell. The annotation near it is its type, i.e. the name of its type cell.

When writing ordinary BNF, it is common to create syntactic categories with no name, as in

\[ s ::= s* \mid \text{atom} \]

We have been careful to avoid this practice in this section, although it is not forbidden in G.

Our example uses a bit of notation for type declaration without defining it. This is with malice aforethought. There is in fact no syntax for type
declaration native to G. Type declaration is done using the syntagm returned by an initializer (§3.10.2, §3.11), specifically a parser (§3.9.1.1) for grammars (§3.3.3), which is a library subroutine.

There is no convention yet for the language accepted by the parser, although the following is a perfectly reasonable extrapolation, based on popular variants of BNF.

The rule which defines each named type is put on one line. We will be careful that each rule shall be of a single kind and have no anonymous types. The ones with no special marks are cons kind. * indicates list kind. + indicates plist kind. | indicates alt kind. Finally, -> indicates gate.

We will use brackets [ ] around the alternatives in an alt rule to indicate that nil is an extra, implicit alternative. See §3.3.3.1. We will use == to give an alias for a type name.

Now it only remains for us to write a declaration for grammars, so that it becomes clear how a grammar is represented as a cell structure.

### 3.3.3 The Grammar of Grammars

The q-name of this grammar is GramGram. All names appearing here are ordinarily unqualified.

```plaintext
GT ::= GRules GMothers
GRules ::= GRule+
GRule ::= Galt | Gcons | Glist | Gplist | Ggate
Galt ::= Glink+
Gcons ::= Glink*
Glist ::= Glink
Gplist ::= Glink
Ggate ::= Glink | Gglink
Glink -> GRule
```
As indicated, grammars (i.e. syntagms of the GramGram grammar) have a list of rules of type GRule. These are the type cells mentioned throughout this chapter. In our example there would be one named s, one named list, and one named atom. (We will see in the next section that they would be rules 4, 5, and 6.) All the rules of the grammar appear in this one flat list. By moving down the subtree at each rule we see what kind each one is and what the component rules are. The component rules are reached via gates, since they already have a mother. (Of course, the real reason that component rules are reached via gates is so that one may be a component of several different rules.)

Yet, it is still desirable to know the set of rules in which each rule appears. This is recorded by the GMother nodes. For each rule, say the fifth, one may find all the rules in which it appears by examining the fifth GMother in the list. It has links to all the desired rules.

The type of a gate's target need not be in the same grammar as the gate. In this case the gate does not appear in the list of mothers of the target rule. By special dispensation, however, if the target is in the same grammar, the gate does appear.

Broad types (§3.2.3) are represented as follows: If a gate's target is some any-type, the gate links to the node of that name in the appropriate grammar (as described below). If a gate's target is ANY, the daughter of the Ggate cell is a Glink whose daughter is none.

Last, a gate which is to hold a function cell is indicated by a link to its profile syntagm.
3.3.3.1 Constraints

There are several constraints on valid G grammars which cannot be embodied in BNF. First, as mentioned, alt, cons, list, and plist may only draw their components from their own grammar. That is, a GRule of those 4 alternatives must gate to a GRule which has the same mother. A gate is not so constrained. However, the target of a gate type must be in a valid, fixed grammar.

Next, we make the obvious requirement that the length of the GRules and GMothers lists be equal. The information in the mothers list must be consistent with the information in the rules list.

Each grammar begins with 3 rules, which are invisible in the text version: any, nil, and foster. They are, respectively, daughter number 1, 2, and 3 of the cell of type GRules (that is, the first, second, and third GRule). Their names are always qualified (See §3.5.). The any type is a pseudo-type. No cells may be created with this type. It exists so that grammar gates may hold it to indicate the set of types containing every type in this grammar. In the rare case that there is a gate to it within its own grammar, the corresponding mother list reflects this. It may not be a daughter of any other kind of rule. The value of the GRule cell named any is always none.

The nil type is the distinguished atomic type for this grammar. It is usually thought to be the empty string, or the case where an optional item is omitted. Thus, it commonly appears as an alternative daughter of an alt type. Its list of mothers reflects these types.

The foster type is a large alt type. It contains every visible rule in the grammar plus the nil type. Thus, a cell of this type may hold any cell of any type of this grammar, excepting only the foster type itself. The cell so held is the foster’s daughter. This type exists so that (foster) mothers may be created for cells which are logically orphans. (An instructive example appears in §2.7.2.) The foster type may not be used as a component of any type in
its own grammar. It has no mothers. Moreover, it does not appear among the mothers of any node of its grammar, although it logically should do. The function GVerifyGram accepts a GT and verifies that all the above properties hold. It may make some alterations in aid of this, such as construction of the GMothers. In any case, at normal termination the grammar will have been fixed and marked in some (perhaps hidden) authenticable or imitable way. (One way is described in §2.4.) If GVerifyGram finds fault with the grammar, it signals an exception. It does not guarantee to leave a faulty grammar unaltered.

3.3.4 The Grammar of Function Profiles

The q-name of this grammar is functype. All names appearing here are ordinarily qualified (except GRule).

functype ::= TP L
TP ::= [ T | P ]
L ::= TF | TV
TF ::= T*
TV ::= T+
T ::= Y | Z
P ::= Y | Z
Y -> GRule
Z -> functype

TP gives the type of the returned object and also tells whether the function is pure (P) or impure (T). If the returned object is absent (i.e., void) the function must be impure. Otherwise, TP leads to a gate either to a type or a profile. L is the parameter list, which is either of fixed length (TF) or varying (TV). Broad types are represented as in §3.3.3.
3.3.5 The Grammar of Property Markers

The q-name of this grammar is GramProp. All names appearing here are ordinarily qualified (except functype and GRule).

\[
\text{GramProp ::= ptype | pfunc}
\]

\[
\text{ptype} \rightarrow \text{GRule}
\]

\[
\text{pfunc} \rightarrow \text{functype}
\]

A property marker holds a type. This is so that each property may have its own associated grammar. The property value must be a syntagm of that grammar/type. Broad types are represented as in §3.3.3.

3.3.6 The Grammar of Primitives

The q-name of this grammar is Gprim. All names appearing here are unqualified and global in scope.

\[
\text{int ::=}
\]

\[
\text{char ::=}
\]

\[
\text{bool ::=}
\]

\[
\text{Gprim == int}
\]

These types are atomic. In the current implementation cells of these types are limited (§3.2.6) in the following sense: They may not have properties or (foster) mothers.

3.3.7 The Grammar of Unique Strings

The q-name of this grammar is ustring. All names appearing here are unqualified and global in scope.

\[
\text{ustring ::=}
\]

\[
\text{tustring ::=}
\]
These types are atomic. See §3.4.

### 3.3.8 The Grammar of Strings

The q-name of this grammar is string. The name bead here is ordinarily qualified.

\[
\text{string} \ ::= \ \text{bead}* \\
\text{bead} \ ightarrow \ \text{char}
\]

### 3.3.9 The Grammar of Vectors

The q-name of this grammar is vect. The name bead here is ordinarily qualified.

\[
\text{vect} \ ::= \ \text{bead}* \\
\text{bead} \ ightarrow \ \text{int}
\]

### 3.3.10 The Grammar of Lists of Gates

The q-name of this grammar is Galist. The name bead here is ordinarily qualified.

\[
\text{Galist} \ ::= \ \text{bead}* \\
\text{bead} \ ightarrow \ \text{ANY}
\]

### 3.4 Unique Strings

In G a string is a list of characters. To test whether two are equal requires a time depending on their length. In order that this time may be saved, a
grammar and some associated routines are provided which define \texttt{ustring}'s, an abstract data type. According to the grammar (See §3.3.7), \texttt{ustring}'s are atoms. Each is intended to stand for a distinct string. Two strings which are equal are represented by the same \texttt{ustring}.

They may have properties, and, in fact, a typical lexical processor makes heavy use of their properties. The string representation of a \texttt{ustring} is available under the property \texttt{name}.

\texttt{ustring}'s are not primitive in any sense and are implemented by library routines. Here are the routines which implement \texttt{ustring}'s:

\texttt{custring}: \hspace{1em} \texttt{string} \rightarrow \texttt{ustring}

retrieves the \texttt{ustring} for the given \texttt{string}. This is the important routine. The others exist for unusual applications.

\texttt{lustring}: \hspace{1em} \texttt{ustring} \times \texttt{ustring} \rightarrow \texttt{bool}

is the \textit{internal} order predicate. It may have nothing to do with the lexical order of the represented strings. It returns \texttt{True} if the first argument is strictly less than the second.

For convenience or necessity (\textit{e.g.} in separate compilations) the user may have several tables of \texttt{ustring}'s. Of course, uniqueness is not guaranteed across tables, only within a table.

\texttt{Gtustring}: \hspace{1em} \texttt{ustring}
The `ustring` string table is hidden on an atom of type `ustring`. The table currently in use is the one stored in the global variable `Gtustring`. If, when it is needed, the value of `Gtustring` is `none` a new empty table will be created. The user may save and restore tables freely.

The `ustring` function:

```plaintext
ustring: ustring →
```

enters the `ustring`, which has been obtained from a different table, into the current table. It is an error if the argument collides with an existing entry for the same string.

### 3.5 Namelists

The property name `name` is the standard for giving a name to a cell. Its value must be a `string`. The property value should be fixed at the time of cell creation.

As we saw in §3.3.1, it is necessary for the grammar initializer (a user-written routine) to pass a set of named cells to the compiler, along with the grammar it creates. In order to do this, it creates a structure called a `namelist`. Other initializers may also pass namelists. For the semantic meaning of namelist names see §3.9.1.

The namelist is a property (i.e. a property value) of the top (GT) node of a grammar. The property marker is `GPropNameList`. When a namelist is passed by any other initializer, it should be a property (with the same property marker) of the top (root) node of the object passed.

The value returned by an initializer is a cell. The compiler searches upward to find the cell's highest (non-foster) mother and examines it for the property `GPropNameList`. Absence of a namelist is not an error.
CHAPTER 3. G LANGUAGE MANUAL

Sometimes, especially in the case of grammar, it is necessary for the name of a cell to be known at run time. For example, if one wishes to select the statement component of an object of type ANY, one must in general wait until run time to discover the actual type of the object and then search for a node with the name statement among the components of that type. By convention, only the names in the namelist are searched.

The q-name of this grammar is GramNameList. All names appearing here are ordinarily qualified (except string).

\[
\begin{align*}
nl & ::= gn \ ns \ ls \\
gn & \to \ string \\
g & \to \ ANY \\
q & \to \ ANY \\
ns & ::= n* \\
n & ::= gn \ r \\
r & ::= g \mid q \mid gs \\
gs & ::= g* \\
ls & ::= l* \\
l & ::= ex \ g \\
ex & ::= exn1 \mid exq \mid exsym \\
exq & ::= gn \ gn \\
exn1 & ::= gn \ gn \\
exsym & ::= gn \ gn \\
GramNameList & == nl
\end{align*}
\]

The first daughter of nl is the qualifying name (q-name), the name of the whole body, e.g. Pascal. The second is the namelist itself. The third is a list of external references, explained in §3.5.1.

The names appear in lexical order. The r part of each namelist entry is
distinguished into 3 categories. For grammars, the first alternative \texttt{g} always
gates to a \texttt{GRule}; its meaning is that one type. The second is the same as
the first, except that the name of the referenced node is always qualified (see
below). The third alternative is a list, all of whose elements gate to \texttt{Glink}'s,
granddaughters of \texttt{GRule}'s.

In each case, the target of the gate should have a (fixed) \texttt{name} (property).
It is contemplated that the user may wish to create aliases by having several
namelist entries for the same cell. One should be identical to the cell's \texttt{name}
property.

The \textit{always qualified} is a request to the compiler only to recognize this
name in its qualified form, despite the fact that other names in this namelist
are unqualified. This takes precedence over the user's request. (The user may
still defeat this by going out of his way. He may create an ident with the
desired name.)

The reason for the existence of the third alternative is this: Consider that
in a context-free grammar each rule necessarily has a distinct name. Thus,
when constructing a new object, this name may be given to denote its type.
However, when examining (selecting) an object at hand, the type is already
known. Instead, the name needed must denote the part of it to which attention
will now turn. This changes things in two ways. First, the rule name is not
sufficient, since the same rule may be two (or more) different constituents
of the known type. A node must be designated which uniquely defines the
constituent, hence we use the \texttt{Glink} which links the rule at hand to the correct
constituent.

Second, since the object at hand is known, the constituent name need not
be unique within the whole grammar, but only within the context of this one
type. \cite{Cl84} make a case for sharing constituent names, and there is no reason
to preclude this. Thus, we arrive at a list of \texttt{Glink}'s, all grouped under one
name.
3.5.1 External References and the Compiler

Since types (grammars) are visible objects, it is necessary for the compiler to see them during the course of compilation. It may obtain a (non-foundation) type either from a library or by invoking a user-written routine, an initializer (§3.14.1, §3.9.1.1). Each type is usually accompanied by a namelist, which defines some of its cells to the compiler's symbol table.

Types, however, may also refer to other types, which the compiler must then likewise obtain. When the user declares the first type, he specifies whether he wants the associated namelist to enter the symbol table and, if so, whether qualified. This namelist in turn specifies which other types to obtain and how their namelists are to enter the symbol table (unqualified, qualified, or not at all). The 3 options are called, respectively, \texttt{exn1}, \texttt{exq}, \texttt{exsym}. Each namelist creates a bound on the namelists it refers to in this sense: These will each be handled with at least as much qualification as it is.

Example: If the user declares a grammar, \texttt{Pascal}, qualified, and \texttt{Pascal} refers to \texttt{PascLexeme} unqualified, the \texttt{PascLexeme} grammar still only enters the symbol table qualified.

When a namelist refers to an object but does not request its namelist (\texttt{Pascal} refers to \texttt{PascLexeme@ident} using \texttt{exsym}), still, that one name enters the symbol table.

With this preamble, it should be easy to understand the \texttt{ls} subtree of the namelist. Each \texttt{ls} is a reference outside the current object for which there is a gate in the current object. The intention is that the compiler will find the outside object using the name provided (§3.14.1) and alter the gate to point to it. The alternative used in the \texttt{ex} production tells how to update the symbol table.

The parser need not provide both names, in which case, conventionally, the second \texttt{gn} is none. The compiler uses its symbol table to try to resolve these
names. Thus, a local declaration masks an external name, except that a single \texttt{exq} name may not resolve to the second part of a qualified name. A forward reference is permitted to a grammar initialized in the same G statement, as described in §3.10.

Finally, it is permitted that \texttt{g} be \texttt{none}, in order to force loading an object without setting a gate.

### 3.6 L-Values

An R-value is simply a cell (or \texttt{none}). An L-value is a reference to a \textit{position}, a daughter of some cell or \texttt{idcnt}. The name comes from the assignment expression \( E_1 = E_2 \) in which the left operand \( E_1 \) must be an L-value expression. Conceptually, an L-value is no different from a pair of data: the holding cell or \texttt{idcnt} and a positive integer, the daughter number. The assignment demands that that position specified by \( E_1 \) shall hold \( E_2 \).

A declared program identifier denotes an L-value, namely, some particular \texttt{idcnt} and the number 1. If \( E_2 \) is also a program identifier, the assignment specifies that the \( E_1 \) \texttt{idcnt} shall have as its daughter the daughter of the \( E_2 \) \texttt{idcnt}. Since \texttt{idcnt}s do not have motherhood, there is no conflict in their sharing a daughter. The original daughter of \( E_1 \) is unchanged. It is merely disconnected from \( E_1 \).

When an L-value is supplied and an R-value is required, an automatic conversion is made from the given position to the cell \textit{occupying} that position. In this example, although \( E_2 \) was an L-value when supplied, only an R-value was required, so it was converted into one, a cell, discarding the information that it was held by the \texttt{idcnt} \( E_2 \).
3.7 Assignment and Coercion

In G, in addition to the simple assignment operation, function activation and list insertion and deletion are all assignments and may use coercion. Type coercion has an unusual meaning in G. It means moving from the value cell being assigned to another cell linked to it. This move is made because the new cell is compatible with the assignment target and the original was not. The new cell may be a descendant or an ancestor of the original. It may exist or be newly created by the coercion.

The three types of coercion described below are available with any assignment, although each is most useful in a particular context. Coercing one of the alternatives into the containing alt type, e.g. if_statement into statement, is most like mathematical injection and is useful in simple assignments. (No projection corresponding to this is provided. We require the user to test and project explicitly.)

The remaining two kinds of coercion, projection from a gate and injection into a gate, are more useful in list operations, where creating an explicit temporary ident would be awkward (See §3.9 for a description of list operators.) These coercions allow a list of gates to be used as a temporary.

We do not provide all the coercions which make sense, e.g. through singleton cons cells or combinations of gate and alt. We have compromised between utility and mnemonic ease.

3.7.1 Assignment

To succeed, an assignment must pass three tests: a type test, a motherhood test, and a fixation test. If any test fails, the data space is unchanged and the assignment raises an exception. (See §3.12.) The tests are made independently. Thus, for example, motherhood does not influence the choice of coercion.
The type test simply checks that the value being assigned can be coerced into the type required by the assignment target. This is described in the next section.

The motherhood test checks to see whether the target has motherhood (See §3.2.) and, if so, whether the value is an orphan. If it is not, the test fails unless the value's mother is only a foster (§3.3.3.1). In this case the foster's daughter becomes none.

The fixation test fails if the target is fixed. In case of motherhood, it also fails if the value has a foster mother which is fixed.

3.7.2 Coercion

In the following discussion we will need to distinguish between the assignment target (a cell) and the type(s) it requires. We shall call the set of types which are allowed by the target (at the given position) \( R \).

If the value being assigned is in \( R \), no coercion is made. This is trivially true if the value is none.

There are three ways that the value may be coerced.

If \( R \) is a single type and it is an alt type, we attempt to find a unique alt path to the value. Consider the property that a type appears as a component of only one type (its mother type) and that that type is an alt. If the value does not have this property, then the test fails. If the value has this property and the mother type is the \( R \) type, the test succeeds. Else, if the mother type does not have this property the test fails. Else, if the mother type in turn has this property and its mother type is the \( R \) type, the test succeeds. Else, ... and so forth. (It is not considered sporting to throw the compiler into a loop.)

If the test succeeds then the path to \( R \) is traversed upward from the value as far as it exists. Where it does not exist, new cells are created and assigned until \( R \) is reached, at which point the assignment proceeds.
If the above coercion fails, one of two further tests is made. We will call them A and B. If the value being assigned is a gate and the type it holds could be in $R$ (according to the gate’s type) then A applies, otherwise B applies.

The test A checks, either statically or dynamically, that the gate’s value actually is in $R$. If so, the assignment proceeds.

The test B requires $R$ to be a single type, a gate type. If the value being assigned can be held by the gate, this gate cell is created and assigned and the assignment proceeds.

### 3.8 Notation

We introduce the following notation for the grammar of G itself (as opposed to grammars declared in G): Non-terminal (nT) symbols are in *italics*. Token categories, such as *id* and *qualid*, are also printed in italics (§3.1). Literal words and characters are in *typewriter* type. A rule begins with a symbol and $\leftarrow$ an arrow. If it runs onto several lines the later lines will be indented. Alternatives are indicated by $|$ a vertical bar. Optional elements are enclosed in $[ ]$ double brackets. A list of things is indicated by a superscript $^*$ asterisk, as in *thing* $^*$. A $^+$ plus may be used instead to indicate a list of at least one element. The $^*$ and $^+$ bind more tightly than simple juxtaposition; the $|$ binds less tightly. Where this must be overridden, we use $( )$ double parentheses. White space in the rules does not represent white space in G. This is removed at the lexical level.

The grammar of G given here is intended more as an aid to comprehension than as an exact statement of the language.
3.9 Expressions

The precedence of expression operators is the same as the order of the subsections of this section, highest precedence first. Left- or right-associativity is specified in each subsection.

The order of evaluation is guaranteed to be left to right for subexpressions, i.e. postorder for expression trees. When an exception is signaled, evaluation stops immediately and control passes to a catcher for that exception. (§3.12)

The type of an expression is not in general known statically. Thus, type errors are sometimes detected as exceptions during execution, instead of being flagged at compilation.

3.9.1 Primary Expressions

Primary expressions involving function calls group left to right.

\[
\text{primary-expression ::=}
\]
\[
\text{literal}
\]
\[
\text{qualid}
\]
\[
(\text{expression})
\]
\[
\text{parsal}
\]
\[
\text{primary-expression ( [ expression-list ] )}
\]

\[
\text{expression-list ::=}
\]
\[
\text{expression}
\]
\[
\text{expression , expression-list}
\]

Literals are described in §3.1.4. They become cells of type int, char, bool, or string. (§3.3.6 and §3.3.8) They are fixed. They are not L-values. Two cells of type int, char, or bool are equal (==) whenever they represent
the same thing. However, two string's representing the same character string may not be equal. None is considered a literal.

A *qualid* may be a simple (*id*) or a qualified (*id@id*) identifier. (§3.1.2 and §3.3.1) A declared identifier (always simple) is an L-value unless it is declared rval. The L-value is (first) daughter of some ident. Idents may have broad types (§3.2.3).

A namelist identifier, whether simple or qualified, is an R-value. Its meaning depends on the subtree appearing in the namelist which defines it. Referring to §3.5, each name is defined by an n subtree. Its r component has a daughter, which may be a single gate (*g* or *q* alternative) or a list (*gs* alternative). If it is single, the meaning of the identifier is the value of the gate. (In the case of a grammar, this is a GRule.) If it is a list the meaning of the identifier is the list itself. That is, it has type *GramNameList@gs*.

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an L-value.

A parser argument list (*parsal*) is an implicit function call. Its value is the cell returned by the function. It may have as a side-effect several assignments. It is described fully in §3.9.1.1.

A function invocation consists of a primary expression, whose R-value must be a function cell, and a (possibly empty) argument list. The expressions in the argument list are evaluated left to right, and, if this happens without an exception (§3.12), the cells so obtained are assigned to newly created idents (§3.7) of the types given in the function definition and control passes to the function. In a function with a varying parameter list, the last parameter is a list of gates, and the final arguments are each assigned to one of the gates. (§3.2.4) An error in the length of the argument list which is not detected by the compiler signals an exception. The exception may occur before, during, or after evaluation of the arguments.
A function which declares a returned type is an R-value. The function must execute a return statement containing an expression. The value of the expression is coerced, as if by assignment, into the returned type, which then becomes the value of the function invocation primary expression.

A void function (sometimes called procedure) may terminate by executing a return statement not containing an expression or by passing control beyond the last statement of the function. Such a function may appear only where a void expression is allowed (§3.13.1, §3.9.11, §3.9.15).

### 3.9.1.1 Parser Argument Lists

Parser argument lists, parsals, are syntagm templates. The template may be wholly literal, thus specifying the syntagm completely, or may contain variables or expressions, causing unknown pieces to be incorporated into it. The syntagm thus generated may be partial (§3.2.2) or total.

The value of the parsal as an expression need not be the root cell, but may be any cell of the syntagm. Moreover, as a side-effect, several other cells of the syntagm may be assigned to targets.

Before we proceed, I am forced to recant some of what was just said. The behavior described is partly conventional. The parser argument list is truly a representation of an argument list passed to a parser. The parser is sometimes written by the user and thus, by intention or by error, it may not treat the list and construct the syntagm in the conventional way. In this preliminary discussion we assume a perfectly pedestrian parser and remedy our omission in the next section.

Let us consider some examples using C language.

```c
statement[ "while( x==0 ) x=getchar();"]
```

This is a completely literal syntagm. The value is the cell of type statement at the root.
statement[t "while( 'x==0 ) x=getchar();"]

This is the same syntagm, but the value returned represents x==0 and has type equality_expression. Simultaneously the statement cell is assigned to the variable t.

statement[ "while( 't )" statement[-] ]

This constructs a syntagm using the value of t as the expression and putting none as the value of the statement cell which is the daughter of the while_statement cell. This gives us some idea of the scope and appearance of parsals. Now we shall discuss them in detail.

The parsal is converted to a syntagm of the GramParseLit grammar. This is passed to a parser, which must be a function of one argument of this type. The value returned must be a Galist (§3.3.10), which holds the values needed for assignment to the various targets.

The parsal itself has this structure:

\[
\begin{align*}
parsal & \leftarrow [ \text{typeid} \ (( [ | ] [ | ] ) [ \text{target} ] \text{plarg}^* ] \\
\text{target} & \leftarrow \text{targid} | ( \text{targ-expression} ) | - \\
\text{plarg} & \leftarrow \\
\ & \text{string} \\
\ & | \text{none} \\
\ & | \text{qualid} \\
\ & | ( \text{expression} ) \\
\ & | [ \text{typeid} ] [ [ \text{target} ] \text{plarg}^* ] \\
\ & | [ \text{typeid} ] $ [ \text{target} ]
\end{align*}
\]
CHAPTER 3. G LANGUAGE MANUAL

\[ \text{typeid } \leftarrow \text{qualid} \]

\[ \text{targs } \leftarrow \text{qualid} \]

\[ \text{targ-expression } \leftarrow \text{expression} \]

White space is used to disambiguate several possibilities inside parsals. A\n\text{typeid} must not be separated from the next token by white space. A\ntargs must not be separated from the preceding token by white space. The paren-
thesis starting a \text{targ-expression} must not be separated from the preceding
token by white space. These rules distinguish these nTs from \text{plarg}'s. White
space is also significant at one point when constructing the argument list.

Strings, as usual, are delimited by \text{u-quotes}. In strings which are \text{plarg}'s
single quotes may be used as anti-quotes, as mentioned in §3.1.4.1. If an initial
substring is anti-quoted out of a string, it never forms a \text{target}.

3.9.1.1.1 The Parsal as an Argument List  A parsal is converted by
the compiler into an argument list, a syntagm of a grammar. The q-name
of this grammar is \text{GramParseLit}. All names appearing here are ordinarily
qualified.

\text{GramParseLit} ::= \text{bead}^+  
\text{bead} ::= \text{lb} | \text{cb} | \text{d} | \text{rb} | \text{datum}  
\text{lb} \rightarrow \text{GRule}  
\text{cb} \rightarrow \text{GRule}  
\text{d} \rightarrow \text{GRule}  
\text{rb} ::=  
\text{datum} \rightarrow \text{ANY}
Ordinarily, the argument list is constructed when control passes to the parser in the ordinary course of execution, after which the parser is invoked, usually returning a newly constructed syntagm. If the parsal begins with the \[ (as opposed to \] symbol, however, it is an init-parsal. An init-parsal is a static initializer (§3.10.2) and is thus converted to an argument list during compilation. The parser is called then and the result is immediately fixed. However, assignments and the evaluation of assignment targets are deferred until control passes to the parsal during execution. Exceptions signaled during init-parsal evaluation cause the compiler to flag errors.

The items in the parsal are examined left to right and a GramParseLit is constructed as follows:

\[
\begin{align*}
[ \text{typeid} ] [ [ \text{target} ] ] & \Rightarrow \text{lb} \\
[ \text{typeid} ] [ [ \text{target} ] ] & \Rightarrow \text{cb} \\
[ \text{typeid} ] $ [ [ \text{target} ] ] & \Rightarrow \text{d} \\
\{ \text{typeid} \} \{ \text{target} \} & \Rightarrow \text{rb} \\
\text{string} | \text{none} | \text{qualid} | ( \text{expression} ) & \Rightarrow \text{datum}
\end{align*}
\]

A left-bracket combination becomes an lb which gates to the type cell denoted by the typeid. If it is absent it gates to none. If the target is a targ-expression it is then evaluated and the result is noted. It must be an L-value. An init-parsal targ-expression is evaluated during compilation.

If the target is followed by a right bracket without any intervening white space, or if, the target being absent, there is no white space between the left and right bracket, then, instead of an lb, the construct is a closed-bracket combination and is converted to a cb. The typeid and target are treated as above.

A dollar-sign combination becomes a d. The typeid and target are treated as above.
CHAPTER 3. G LANGUAGE MANUAL

A right bracket which is not part of a closed-bracket combination becomes an rb atom.

A string which contains anti-quotes is broken into pieces, and the anti-quoted parts are treated as if they were not in quotes. (We may in future prohibit parenthesized expressions from being anti-quoted.) Thus, we have only to deal with the pieces, each of which is known not to contain anti-quotes.

Each string is converted to a string (§3.3.8) by the usual rules. At that place a datum is created which gates to it.

A none becomes a datum which gates to none.

A qualid becomes a datum which gates to it.

An expression is evaluated. The resulting cell becomes a datum which gates to it.

Note that the argument list so created is partly flattened. The nesting reflected by the brackets has disappeared. Only expressions protected by parentheses are processed in the usual tree order.

3.9.1.1.2 Parser Invocation and the Treatment of Results

The parser is found using the typeid on the first element of the parsal. The compiler examines the GT node of the grammar containing the GRule denoted by that typeid. If it finds the property GPropParse, its value is the parser. If there is no parser there, or if the typeid is absent, the value of GDefParse in the current scope is used. (GDefParse is defined and fixed in the global scope.)

The value returned by the parser must be a Galist whose elements correspond to the lb’s, cb’s, and d’s in the GramParseLit. The values of the Galist@bead’s are processed left to right as follows: If the target of the lb, cb, or d is a targid or a targ-expression, then the value of the Galist@bead is assigned to it. (It must be an L-value.) If the target is a - minus mark the
value is discarded. In at most one case the target may be omitted. The value at that position becomes the value of the whole parsal as an expression. If no target is omitted the parsal is a void expression.

3.9.1.1.3 The Conventional Meaning of the Parsal In this section I shall use the conditional tense, "a cup would hold water," to speak about the conventional use of parsals. Wherever this tense appears, we may automatically insert the clause, "If the convention were followed," to give the sentence its proper meaning.

The type which precedes a matched pair of brackets would give the type of the result of parsing everything between the brackets. The parser would interpret it as that type, and, failing that, would signal an exception.

The type of each object assigned or returned would be the type (if any) named in the combination denoting the target.

If the type is omitted from a bracketed list, the type implied would be the most specific type that applies to the enclosed list.

A dollar-sign combination would mean the longest possible following sublist (possibly null) which has the type denoted by the typeid, or, if it is absent, the longest possible following sublist (possibly null) which can be given a type, in which case the cell returned has the most specific type which could apply to it. In the case above where the sublist is null, the daughters of the cell returned would all be none.

Here, the cell with the most specific type is that cell of the syntagm which is lowest or closest to the leaves. This is equivalent to requiring that the returned cell never have exactly one daughter. We would suspend this rule for gates, however, since the intention is that a parser should stop at leaf level.

There is, however, no requirement that the parsal contents actually have a direct relationship to the type named in a combination. For example, it will happen fairly often that the parser will treat strings as fragments from
an associated grammar, a concrete grammar, and afterwards convert them to
the named (abstract) grammar for return to the user. The parser might even
contain pieces from several different grammars.

There is no requirement that a string be a whole nT. However, a string
would contain only whole lexemes. That is, fragments from grammars at this
level, concrete or abstract, are expected, but those from grammars below (the
leaf grammar mentioned in §3.2 in connection with gates) are not.

A dollar-sign combination appearing just before a right bracket and a left
and right bracket with only white space between both would mean the same
thing, an empty string allowed in the grammar. Its type is usually xnil. A
closed-bracket combination, on the other hand, would mean none, i.e. a cell
of the type named whose daughters are all none. If the type is absent there is
no conventional meaning. To indicate none where, for example, a statement
is required (as opposed to a statement cell whose daughter is none), we would
write statement[-none] (as opposed to statement[-]).

When an item appearing in a parsal can be used directly (i.e. without
copying it) in the construction of the syntagm, the parser would do so.

3.9.1.2 Built-In Functions

These functions appear in every scope. Their definitions cannot be suspended.
They are not loaded from a library. They are primitive and, thus, do not exist
in the cell space.

gcons: GRule \rightarrow \text{ANY}

creates a new cell of the given type. The cell is not equal (\(=\)) to
any other. It is an orphan. Its daughter(s) is none. Its properties
are none. It is not fixed. gcons signals an exception if the type
is not part of a fixed grammar or if the grammar is not verified.
§3.3.3.1
cons: \[ GRule \times ANY \times \cdots \times ANY \rightarrow ANY \]

creates a new cell of the given type, as gcons does, except that the second through nth arguments are assigned as its daughters. Assignment exceptions may be signaled. The cell returned has a foster mother.

length: \[ ANY \rightarrow int \]

for lists and plists returns the number of the last non-none daughter, which may be 0. For the remaining types, the length does not depend on the cell's value. For gates and alts it returns 1. For cons it returns the number of daughters of a complete cell.

index: \[ ANY \rightarrow int \]

returns the daughter number of this cell as seen from its mother. It returns none if the cell is an orphan.

type: \[ ANY \rightarrow GRule + functype \]

returns the type of this cell. It returns none if called with none.

fixp: \[ ANY \rightarrow bool \]

returns whether the cell is fixed. If called with none it returns True.
fix: \text{ANY} \rightarrow \text{bool}

returns whether the cell was fixed when \text{fix} was called. If called
with \text{none} it returns \text{True}. As a side-effect, it fixes its argument
and all its descendants recursively.

\text{fixidp}: \text{ident-L} \rightarrow \text{bool}

returns whether the ident is fixed. The argument must be an ident
L-value.

\text{fixid}: \text{ident-L} \rightarrow \text{bool}

as with \text{fixidp}. As a side-effect, it fixes the ident and all its
descendants recursively.

\text{mapcar}: \text{functype} \times \text{ANY} \times \cdots \times \text{ANY} \rightarrow

repeatedly calls the given function, perhaps while assembling a list.
There must be at least 3 arguments. We shall call the arguments
\text{f}, \text{r}, \text{T}_1, \text{T}_2, \ldots, \text{T}_n. Only \text{r} may be \text{none}. Let

\[ L = \max\{\text{length}(\text{T}_i)\} \]

Then, \text{mapcar} may be rendered thus:
for (i=1 ; i <= L ; i=i+1 ){
    f( T_1#i, T_2#i, ..., T_n#i );
    catch Mapcar_assign (ANY a){
        r#i = a;
    }
}
resignal ANY;

If r is none and f signals Mapcar_assign, mapcar aborts. The arguments are intuitively lists, but need not actually be. L may be zero.

### 3.9.2 Unary Minus

The unary minus is a prefix operator. Formally it groups right to left.

\[
\text{minus-expression \leftarrow - expression}
\]

The operand must be an integer. The result is an integer. The result is the arithmetic negative of the operand.

### 3.9.3 Multiplicative Expressions

The multiplicative operators are infix. They group left to right. The operands must be integers. The result is an integer.

\[
\text{multiplicative-expression \leftarrow}
\]

\[
\text{expression \ast expression}
\]

\[
\text{expression \mathbin{/} expression}
\]

\[
\text{expression \mathbin{\%} expression}
\]

The \( \ast \) indicates multiplication. If the result overflows an integer it signals an exception.
The / indicates division. When positive integers are divided truncation is toward 0. If either operand is negative the truncation is undefined. If the divisor is 0 it signals an exception.

The % yields the remainder from the division of the first operand by the second. If either operand is negative the sign of the result is undefined. If the divisor is 0 it signals an exception.

### 3.9.4 Additive Expressions

The additive operators are infix. They group left to right. The operands must be integers. The result is an integer.

\[
additive-expression ::= \\
expression + expression \\
| expression - expression
\]

If the result overflows an integer it signals an exception. The + indicates addition. The - indicates subtraction.

### 3.9.5 Navigators

There are one- and two-operand navigators, which are, respectively, postfix and infix. Navigators group left to right. The result is usually an L-value. The argument(s) are R-values.

\[
navigator ::= \\
expression -> expression \\
| expression * \\
| selector
\]
The expression $E_1 \rightarrow E_2$ retrieves a property L-value. The value of $E_1$ may be any cell except those of $\text{Gprim}$ (§3.3.6). $E_2$ must be a property marker, a cell whose type is $\text{GramProp}$ (§3.3.5, §3.2.5) and whose value is a type, which may be broad (§3.2.3). The marker must be fixed.

The property L-value retrieved is the daughter of an ident, which holds the type represented by the marker. For each pair of cell and marker there is a unique, distinct ident. Its initial value is $\text{none}$.

The expression $E_1^-$ retrieves the mother of $E_1$. The argument and result are R-values. The result is $\text{none}$ if $E_1$ is an orphan.

### 3.9.5.1 Selectors

A selector creates an L-value from arguments describing a position. The argument(s) are R-values. Except as noted, the position is one of the daughters of the first argument. The type may be broad if the first argument is a gate.

```
selector ←
  expression # expression
  | expression #
  | expression ^
  | expression #^+
  | expression #++
  | expression . expression
```

There is an apparent ambiguity in the expression $E_1#(E_2)$ since it might be an infix selector or a postfix selector and function call. This ambiguity is always resolved in favor of the former. To force the latter interpretation, we must write $(E_1#)(E_2)$. There is also an apparent ambiguity in $E_1#^+E_2$ and in $E_1#^-E_2$. However, this is resolved at the lexical level, since $#^+$ and $#^{++}$ and $^-#$ are all distinct tokens (§3.1.1).
In the expression $E_1#E_2$ the second argument $E_2$ must evaluate to an integer (int). The L-value returned is that daughter of $E_1$ which is number $E_2$.

The expression $E_1#$ is simply an abbreviation for $E_1#1$.

The expression $E_1#^\ast$ is an abbreviation for $(E_1^\ast)#index(E_1)$. However, $E_1$ is only evaluated once. Note that in this case the L-value returned is not a daughter of $E_1$.

The expression $E_1#^+$ is an abbreviation for $E_1#length(E_1)$. However, $E_1$ is only evaluated once.

The expression $E_1#^{++}$ is an abbreviation for $E_1#(length(E_1)+1)$. However, $E_1$ is only evaluated once.

The expression $E_1.E_2$ means $E_1#hidden(type(E_1),E_2)$ except that $E_1$ is only evaluated once. $E_2$ is typically a namelist name. $hidden$ may be rendered thus:

```c
if (type(E_2) == GramNameList@gs){
    for (i=1; i <= length(E2); i=i+1){
        if ((b = (a = E2 # i #)^\ast) == type(E1)){
            if (type(b) != Gcons) signal . . . ;
            return index(a);}
    } else {
        a = type(E1)# ;
        if (type(a) != Gcons) signal . . . ;
        for (i=1; i <= length(a); i=i+1){
            if(a # i # == E2)
                return i;}
    . . . /* exception generator */
}
```

where, as usual, $E_1$ and $E_2$ are only evaluated once (at function invocation, as it were). As indicated, the operation fails unless $E_1$ is a construction
type. The exceptions signaled by this and other errors are to be defined in future. Moreover, the compiler may detect them and flag them as errors before execution.

3.9.6 Unary Expressions

The unary operators are prefix. They group right to left.

\[
\text{unary-expression} \leftarrow \\
\quad ! \text{ expression} \\
\quad & \text{ expression}
\]

The ! indicates Boolean negation. The operand must be Boolean. The result is the other Boolean value.

The & indicates replication. The operand may be any cell except a function cell, none, or a primitive (§3.3.6). The result is a copy of the syntagm rooted at that cell. Copying proceeds recursively through the daughters of the cell but stops at gates. Each gate cell is copied, but the copy's daughter is the same cell as the gate's. Properties are not copied. The result is not fixed. The cell returned, the copy of the operand cell, has no mother. The type of each cell of the copy is the same as that of the corresponding cell of the original.

3.9.7 Relational Expressions

The relational operators are infix. They group left to right, although this fact is not useful; \(a < b < c\) is a type error. The operands must be integers. The result is Boolean.
relational-expression [\[\[
  \text{expression} \ < \ \text{expression} \\
  \text{expression} \ <= \ \text{expression} \\
  \text{expression} \ >= \ \text{expression} \\
  \text{expression} \ > \ \text{expression}
\]]

The operators \(<\ <= \ >= \ >\) indicate, respectively, less than, less than or equal to, greater than or equal to, greater than. The result tells whether the first operand stands in the given relationship to the second.

### 3.9.8 Equality Expressions

The equality operators are infix. They group left to right. The result is Boolean. The operands may be any cell or none.

equality-expression [\[\[
  \text{expression} \ == \ \text{expression} \\
  \text{expression} \ != \ \text{expression} \\
  \text{expression} \ /\ / \ \text{expression}
\]]

The \(==\) indicates identical equality. The result is \texttt{True} only if the two operands are the same cell.

The \(!=\) is the negation of the \(==\) operator. The result is \texttt{False} only if the two operands are the same cell.

The \(\|\) indicates half-matching. The result is \texttt{True} if the first operand matches the second. In this case, the first operand may be altered as a side-effect.

We say that \(a\) matches \(b\) when \(a\) is structurally identical with \(b\), except, perhaps, where one has the value \texttt{none}. By \textit{structurally identical} we mean that \(a\) has the same type as \(b\) and that each of its daughters is structurally identical to the same daughter of \(b\), with this exception: If \(a\) is a gate then
it is only structurally identical with \( b \) if they have the same type and their daughters are equal (\( == \)).

Whenever \texttt{none} appears in \( a \) or \( b \) it is taken to be identical to the corresponding piece of the other. If \( a \) and \( b \) are found to be structurally identical then each \texttt{none} in \( a \) is replaced by a copy (\$) of the corresponding piece of \( b \).

(We note that this does not duplicate the power of the most general unifier of Prolog, which may also require the identity of two distinct subtrees of a syntagm. This it does essentially by giving names to its none’s.)

If \( a \) and \( b \) are primitives (§3.3.6) then they match if they have the same type. If they are function cells they match if their types are compatible. If they are \texttt{none} they match. If only \( a \) is \texttt{none} the operator signals \texttt{Mapcar_assign}.

It probably would be a good idea to extend this section with a non-side-effecting matching-op and a struct-ident equal which gives \texttt{none} no special status.

### 3.9.9 And-Expressions

The \&\& operator is infix. It groups left to right. The operands and result are Boolean.

\[
\text{and-expression} \leftarrow \text{expression} \&\& \text{expression}
\]

The result is \texttt{True} if both the operands are \texttt{True}. Otherwise it is \texttt{False}. If the first operand is \texttt{False} the second is not evaluated.

### 3.9.10 Or-Expressions

The || operator is infix. It groups left to right. The operands and result are Boolean.

\[
\text{or-expression} \leftarrow \text{expression} \mid \mid \text{expression}
\]
The result is True if either of the operands is True. Otherwise it is False. If the first operand is True the second is not evaluated.

### 3.9.11 Conditional Expressions

This operator is ternary. It groups left to right. The first operand is Boolean. The others may be any R-value or void, but if one of them is void then the whole expression is void.

\[
\text{conditional-expression} \leftarrow \\
\text{expression} \ ? \ \text{expression} \ : \ \text{expression}
\]

The result is the second expression if the first expression is True. Otherwise, it is the third expression. Only one of the second and third expressions is evaluated.

### 3.9.12 Removal Expressions

The removal operators are infix and postfix. They group left to right. The operands and result are L-values. Moreover, the first operand must be a position in a list or plist.

\[
\text{removal-expression} \leftarrow \\
\text{expression} \ : > \ \text{expression} \\
| \ \text{expression} \ : >
\]

As in §3.9.5.1, E1 :>(E2) looks like an ambiguity and is resolved in the same way, i.e. in favor of the infix. The result of the expression is the first operand. As a side-effect, the R-value in that position is removed from the list, causing all higher-numbered daughters to shift left by one. If the second operand is present, the R-value is assigned to it. Thus, the result, although it is the same L-value, resolves to a new R-value.
3.9.13 Insertion Expressions

The insertion operators are infix. They group right to left. The first operand is an R-value. The second operand and result are L-values. Moreover, the second operand must be a position in a list or plist.

\[
\text{insertion-expression} \leftarrow \\
\quad \text{expression} : \text{expression} \\
| \quad \text{expression} : >: \text{expression}
\]

The result of the expression is the second operand. As a side-effect, several insertions are made into the list, causing the daughter at that position and all higher-numbered daughters to shift right. The result L-value now resolves to a new R-value, one of the inserted cells.

In the first form a single new element, the one given by the first operand, is coerced, if necessary, and inserted. In the second form the first operand is treated as a list. All its daughters, from the first position to the length position, are each coerced, if necessary, and inserted into the second operand. In both cases, when necessary, the existing mother of a cell to be inserted will be given the value none so that the assignment can proceed. Should such a mother be fixed or should coercion fail, the insertion signals an exception, in which case both operands may have been altered in an undefined way.

3.9.14 Assignment Expressions

The assignment operator is infix. It groups right to left. The first operand is an L-value. The result is the same as the second operand. Moreover, if the second operand is an L-value, the result is likewise.

\[
\text{assignment-expression} \leftarrow \text{expression} = \text{expression}
\]
The effect of the assignment is that the second operand R-value becomes the cell in the position specified by the first operand L-value. The details of assignment are discussed in §3.7. The data space is changed only if the assignment passes the type, motherhood, and fixation tests. Otherwise, assignment signals an exception. Some exceptions may be statically detected and flagged by the compiler.

### 3.9.15 Comma Expressions

The comma operator is infix. It groups left to right. The operands are R-values. The result is the second operand. Either operand may be void. If the second operand is void the whole expression is void.

```
comma-expression ← expression, expression
```

The first operand is evaluated first, with all side effects. However, the result is simply discarded. The value of the expression is the value of the second operand. In contexts where a comma is ambiguous, e.g. in a function call, it never is the comma operator. To achieve that interpretation it must be protected by parentheses, as in \( f(a, (t=3, t+2), c) \).

### 3.10 Declarations

Declarations are used to specify the interpretation which \(G\) gives to each identifier; they do not necessarily reserve the storage associated with the identifier. Only local declarations (those inside functions) are discussed in this section. For global declarations, see §3.11. Declarations have this form:

```
declaration ←
   dcl-specifier* ( ( var-dcls | func-dcl ) ) ;
```
The *var-dcl* or *func-dcl* contains the type and the identifier(s) being declared, along with initializers.

### 3.10.1 Declaration Specifiers

The declaration specifiers give the storage class and fixation of the identifier, as well as telling whether there is a namelist (§3.5) and an ident (§3.2.1).

\[
dcl-specifier \leftarrow \\
\quad \text{extern} \\
\quad \mid \text{static} \\
\quad \mid \text{rval} \\
\quad \mid \text{fixed} \\
\quad \mid \text{namelist} \\
\quad \mid \text{namelist unequal}
\]

An *extern* declaration reserves no storage. Instead, there must be an external definition (§3.11), whose type matches the declaration, somewhere outside the current function. Since an external (as opposed to *local*) declaration is not a definition, it is never initialized. A *static* declaration is a definition in that it causes storage to be reserved. The storage is statically allocated and is not reclaimed or altered between activations. *extern* and *static* may not both appear. If neither appears then the declaration is a definition which reserves automatic storage.

If *rval* appears the identifier may only be used as an R-value. Its use as an L-value is an error. If the declaration is *extern* then the object referred to may be an ident and each reference to it is to its current content, which may be altered by code outside the scope of this declaration. A local declaration is effectively a named cell (or *none*), either automatic or static. If it is initialized, it is the cell computed by the initializer. Otherwise, it is a newly created cell of the named type, in which case broad types are not allowed.
If \texttt{ rval} is absent the object is an ident. If the declaration is \texttt{ extern} the external object must be an ident. Its value is as defined by the defining declaration. If it is local its value is the cell computed by the initializer if present. If not, then if the type is a broad type or a function type, its value is \texttt{none}. Otherwise, its value is a newly created cell of the named type.

If \texttt{ fixed} appears then \texttt{ rval} is implied, even if it is absent. If the declaration is \texttt{ extern} the object must actually be fixed. (If \texttt{ fixed} is absent an external object may be fixed anyway.) If the object is local then it is initialized as described above and then fixed. In either case the object referred to is in static storage. If neither \texttt{ static} nor \texttt{ extern} appears, \texttt{ static} is assumed.

If \texttt{ namelist} or \texttt{ namelist unequal} appears then \texttt{ fixed} is implied, even if it is absent. The root of the syntagm in which the initializing cell is embedded is examined for a namelist, which may cause other identifiers to be included in the compiler’s symbol table, as described in §3.5.

### 3.10.2 Identifier Declarations

Several identifiers of the same type (except for function types, §3.10.3) may be declared in the same statement.

\[
\text{var-dcls} \leftarrow ( Qualid | \text{ANY} ) \text{id-list-init}
\]

\[
\text{id-list-init} \leftarrow
\]

\[
\text{id-init}
\]

\[
| \text{id-init} \ , \ \text{id-list-init}
\]

\[
\text{id-init} \leftarrow \text{id} \ [ \ = \ \text{expression} \ ]
\]
The \textit{qualid} must be a known type name referring to a \texttt{Gaulle} cell which has been fixed and verified (§3.3.3.1). The \texttt{id} is the identifier being introduced. It may have an initializer. The type and specifiers apply to each identifier introduced.

In automatic storage identifiers are initialized each time the compound statement (§3.13.2) containing them is entered before the statements are executed. In static storage they are initialized when their definitions are read by the compiler, before the identifier(s) in the current declaration become known. In this case the identifiers in the initialization expression must be known from previous declarations, usually from surrounding scopes, although they may be as near as the preceding declaration. They may not be from the current declaration. External identifiers may not be initialized.

However, forward reference of types is necessary to define certain recursive data structures. To this end, the resolution of namelist external references (§3.5.1) is delayed in each declaration until after all of the individual identifiers are initialized and namelist names are noted. Fixation occurs after this resolution.

Some extension to the language may be required in the area of recursive type structures. Could we arrange it that two mutually recursive types were compiled separately? It would require use of the loader to resolve external references and the delay of fixation.

One elision is permitted. If an initialization expression is a parsal, the first \texttt{typeid} may be omitted, in which case it will be taken to be the type named in the declaration, unless it is \texttt{ANY}.

3.10.3 Function Declarations

Only one identifier may be introduced in each function declaration.
\[
\text{func-decl} \leftarrow \\
\quad \text{func-returns id ( \{ ab-func-var-list \} )} \\
\quad \{ \text{exceptions} \} \{ \text{= expression} \} \\
\]

\[
\text{func-returns} \leftarrow \\
\quad \text{void} \\
\quad \{ \text{pure} \} \text{ qualid} \\
\quad \{ \text{pure} \} \text{ ANY} \\
\quad \{ \text{pure} \} \text{ ab-func} \\
\]

\[
\text{ab-func-var-list} \leftarrow \\
\quad \text{varying} \\
\quad \text{ab-func-var} \\
\quad \text{ab-func-var, ab-func-var-list} \\
\]

\[
\text{ab-func-var} \leftarrow \text{ab-func | qualid | ANY} \\
\]

\[
\text{ab-func} \leftarrow \\
\quad < \text{ab-func-returns ( \{ ab-func-var-list \} )} > \\
\]

\[
\text{ab-func-returns} \leftarrow \\
\quad \text{void} \\
\quad \text{qualid} \\
\quad \text{ANY} \\
\quad \text{ab-func} \\
\]
All of the types named explicitly must be known. The id is the identifier being introduced. Its type is a newly constructed syntagm of the \texttt{func\texttt{type}} grammar (§3.2.4, §3.3.4). The description of exceptions may usually be omitted, as explained in §3.12. The grammar for exceptions is given in §3.11.2. If there is no initializer expression the initializer is taken to be \texttt{none}. Name lists and forward reference are not currently supported for functions. Otherwise, the description of initialization in §3.10.2 applies. Although it makes sense, initialization by literal functions is not currently allowed.

The word \texttt{pure} may appear with any declaration. However, the compiler can only attempt to optimize calls to pure functions when they are also declared \texttt{fixed}. If a function which accepts or returns functions is declared \texttt{pure} then the passed functions will also be marked \texttt{pure}.

The semantics of function invocation, including \texttt{void} and \texttt{varying} functions, is given in §3.9.1.

\section*{3.11 Global Declarations}

In G, declarations may be nested inside other declarations. (More precisely, a declaration may appear in a scope which is nested inside another scope which also contains declarations.) A declaration so nested is called \textit{local}. A declaration (whose scope is) not inside any other is called \textit{global}. A global declaration is in force starting with the following declaration and persisting to the end of the file in which it appears.

However, a declaration of a name which appears in a nested scope \texttt{suspends} any declaration for the same name which may be active in the enclosing scope. At the end of the nested scope the enclosing declaration resumes. New (nested) scopes are introduced by function literals (§3.11.2) and compound statements (§3.13.2).

A program consists of a sequence of global declarations. The syntax of a
global declaration is the same as that of a local declaration (§3.10), except that only at the global level may the code for functions be given.

\[
\text{global-declaration} \leftarrow \\
\text{declaration} \quad \text{func-defn}
\]

### 3.11.1 Declaration Specifiers

Every declaration of a given name \((id)\) in a given global scope (file) refers to the same object, a cell or ident. This is called linkage. A name declared to have \textit{inside} linkage has linkage in just one file, the file it is declared in. When several files form a program, a name with \textit{outside} linkage refers to the same object in \textit{every} file in which it is so declared. However, a name with merely inside linkage in certain files refers to a different object in each file, so the scope of these names, although called global, is to some degree local.

The declaration specifiers \texttt{extern} and \texttt{static} determine what linkage a name has in each scope. In a local scope (§3.10.1), a name declared with \texttt{static} or with neither has no linkage; each declaration is a different object. A name declared \texttt{extern} has inside or outside linkage depending on whether that name has inside or outside linkage in the global scope. If there is no active global declaration then the name has outside linkage.

In a global scope the \textit{first} declaration for a given name determines whether it has inside or outside linkage: If the declaration is \texttt{static} it is inside. If it is \texttt{extern} or neither it is outside. Every declaration at this level has linkage.

As before, an \texttt{extern} declaration is not a definition and relies on a definition somewhere else, which may be, however, in the same global scope. A \texttt{static} declaration causes a definition in static storage, either by itself or in cooperation with other declarations in the same scope. In a global scope a
declaration which is neither static nor extern likewise causes a definition in static storage.

As before, an rval attached to an extern may refer to an iden, but without extern must refer to a cell. In either case, its use as an L-value within the scope of this declaration is an error, and the meaning of rval is otherwise as it is in a local declaration.

The meaning of the absence of rval is the same as in a local declaration.

The meaning of the presence or absence of fixed is as it is in a local declaration except that if neither static nor extern appears, neither is assumed, and the declaration is a definition for an object with outside linkage.

The meaning of the namelist declaration is the same as in local declarations.

Since all global declarations refer to static storage, all initializations are static and are undertaken by the compiler, which detects any attendant errors.

### 3.11.2 Function Declarations

In a global scope function names may be introduced just as they are in local scopes. However, only in a global scope may a function literal appear and be named.

```plaintext
func-defn ←
    [ static ]
    func-returns id ( [ func-var-list ] )
    [ exceptions ] compound-stmt

func-var-list ←
    func-var
    | func-var , func-var-cdr
```
A function literal constitutes a distinct local scope. The identifiers in the parameter list and the declarations in the compound statement reside in this scope. The parameter list is implicitly declared with no declaration specifiers (i.e. automatic, variable L-values). It is initialized as described in §3.9.1.

A function literal may never be extern, although it may be static. No other specifier may appear, although every function is implicitly a fixed atomic cell.

All exceptions signaled within the function must be named in the exception list. §3.12 discusses exceptions.
3.12 Exceptions

This section describes the nature of the exception facility in G. The response of the computer to an exception as an event is described here, but the associated statements are described in §3.13.

An exception is a transfer of control to a point outside the current function. The current function signals the exception by giving the name of the exception and possibly an argument. This action terminates the signaler.

Control passes to the caller, the function which invoked the signaler, but not usually at the point of invocation. Instead the caller must catch the exception with a catch statement. The catch statement contains a statement, its body, which is then executed. If control falls through the body, execution continues with the statement following the catch. (resignal may also catch exceptions; see §3.13.6.)

The catch statement which receives control must match the named exception, as defined below, and must be the nearest such catch which is beyond the invocation and no more deeply nested. More precisely, the catch is reached in this way: Set the current statement to be the smallest syntactic unit (syntagm) of type statement containing the invocation. If the current statement is one of the elements of a compound statement, then every subsequent element of the compound which is a catch statement is a candidate. If no match is made, the entire compound statement becomes the current statement and the search continues. If the current statement is not an element of a compound statement, it must be the body of some other type of statement (while, if, etc.) or a function body. In the first case, the containing statement immediately becomes the current statement and the search continues. In the second case, the search fails.

This means, for instance, that a catch in one arm of an if statement may
not catch exceptions from the other arm: that a catch may not catch exceptions generated in its own body; that a catch in a compound statement may not catch exceptions from before the beginning of that compound statement.

Exceptions may also be raised outside of statements, i.e. in initializations. In global initializations they are fatal compilation errors. Exceptions raised during static initializations inside functions are also fatal errors. (In both cases, it is not an error if exceptions arise inside an initializing function and are caught by it.) Exceptions raised by automatic initializers may not be caught by the statements of the function body (that is, by a catch which is a statement in the compound of which the declaration is a part), but cause the function to terminate, raising the exception in the caller.

A catch may only match an exception in two ways: Either the catch statement contains the literal name of the exception or it is a catch ANY. A matching literal catch which does not conform to the exception named with respect to the presence of a parameter or the type of the parameter is in error. The compiler or loader will flag these errors.

Every exception signaled by a function must be declared in the signals clause of the header of that function, along with the type of the argument passed. Every such declaration for a given exception name must match throughout the scope of that declaration. The scope of an exception declaration is the same as that of the function’s name; i.e. it always has linkage, either inside or outside (§3.11). A function may never declare FAILURE, but every function may signal FAILURE. Its argument has type string.

(Declarations for function identifiers may contain the signals clause. This is encouraged, although it is only necessary in the forward declaration of static functions.)

If the search for a catcher fails, the exception is converted to an exception named FAILURE, which the invoker then signals in turn to its own invoker. If the original exception is itself a FAILURE, it is passed unaltered. Otherwise, its
argument is discarded and a string is constructed from the original exception’s name:

"uncaught exception: " :> name#

This then becomes the argument of the FAILURE exception.

There are two more statements which deal with exceptions, resignal and exit. They are described in §3.13.6 and §3.13.5.

3.13 Statements

Except as indicated, statements are executed in sequence.

3.13.1 Expression Statement

The expression statement has the form:

\[
expression\text{-}stmt \leftarrow [ \text{expression} ] ;
\]

The expression, if present, is evaluated (§3.9). However, its value is simply discarded. The expression may be void. Thus, the expression may be a procedure call.

3.13.2 Compound Statement

The compound statement (also called block) has this form:

\[
compound\text{-}stmt \leftarrow \{ \text{declaration}^* \text{ statement}^* \}
\]

It introduces a new scope, thus the declarations which begin it are only in force during the compound statement. Any previous declarations for the same identifier(s) are suspended. Conceptually, any automatic initializations among the declarations are done before the first statement, each time the compound
statement is entered, with this exception: If it is entered via `goto` to an interior statement, the initializations are not done.

Initializations of objects in static storage are done only once, before the program begins execution.

### 3.13.3 Signal Statement

The signal statement has this form:

\[
\textit{signal-stmt} \leftarrow \text{signal } \textit{id} \ [\textit{expression}] ;
\]

The \textit{id} is the literal exception name (§3.12). It must have been declared in the function header, and the expression must be assignment compatible with the type declared there. If the expression is absent the exception must have been declared \textit{void}.

This statement terminates the current activation. Control passes to a catch or \textit{resignal} statement in the invoker, or, failing that, signals an exception in the invoker (§3.12).

### 3.13.4 Catch Statement

The catch statement has two forms:

\[
\textit{catch-stmt} \leftarrow
\]

\[
\text{catch } \textit{id} \ [\text{ab-func-var } \textit{id}] \ [\text{compound-stmt}]
\]

\[
| \text{catch ANY ( string } \textit{id}) \text{ compound-stmt}
\]

In the first form the first \textit{id} is the literal exception name. The presence and type of the argument declared in the signaler must match the type named in the \textit{catch} statement. If present, the second \textit{id} is bound to the argument of the exception. The scope of this binding is only the compound statement.
In the second form the *id* is bound to the name of the exception represented as a *string*. The scope of this binding is only the compound statement.

As explained in §3.12, an exception signaled in a called function causes a search forward from the call for a catcher. (See also §3.13.5.) The first form will catch one whose name matches; the second will catch any exception. If present, the *id* is bound, as explained above, and execution resumes with the compound statement. The exception is extinguished at this point.

If a catch statement receives control in the ordinary course of execution, control passes on without executing the compound statement. If control passes into the compound statement via *goto* the exception parameter is not initialized.

### 3.13.5 Exit Statement

The *exit* statement has the form:

```-exit-stmt ← exit id [ expression ] ;```

This statement has the effect of raising an exception named *id* in the current function, that is, as if it were signaled in a function called by this one. The search for a catcher proceeds forward from the *exit* statement itself. It must be caught by a first-form catch statement. It is never converted to a FAILURE. Moreover, no *resignal* or *catch ANY* may intervene between the *exit* and the matching catch. The exception name need not appear in the function header, and if it does not, the name has no linkage (§3.11) but is only known throughout the current function.

There must be agreement between the presence of an expression in the *exit* and in the matching catch. The expression, if present, is evaluated and assigned to the parameter in the matching catch.
3.13.6 Resignal Statement

The resignal statement has two forms:

\[
\text{resignal-stmt} \leftarrow \\
\text{resignal id ;} \\
\mid \text{resignal ANY ;}
\]

It is a combination of catch and signal. A resignal appearing in the search path of an exception will catch it in just the same circumstances as a catch will, i.e. either by matching its name (first form) or by catching all exceptions (second form). In either case the same exception with the same argument is immediately signaled in this function, thus terminating it and starting a search in this function's invoker.

The resignal statement requires an exception declaration in the function header. In the first case it is the same as if an explicit signal had been used, i.e. it names the argument type. In the second case the ANY exception declaration must be used. This is the sole use for this declaration. It never stands for exceptions named explicitly in signal or resignal statements.

If a resignal statement receives control in the ordinary course of execution, the statement has no effect; control simply passes on.

3.13.7 If Statement

The if statement has the form:

\[
\text{if-stmt} \leftarrow \\
\text{if ( expression ) statement} \\
\quad (\text{elseif ( expression ) statement})^* \\
\quad [\text{else statement}]
\]
The first expression is evaluated. If the result is True then the first statement is executed. If the result is False then the second expression (if present) is tried, and so forth, until one expression is True and the associated statement is executed. If all the expressions are False and the else clause is present then the associated statement is executed. At most one statement is executed.

It is expected that the expressions are Boolean. However, error detection is not guaranteed unless and until a given expression is evaluated. The response to an error may be a flag by the compiler (the usual) or an exception during execution.

As usual, each else or elseif clause is parsed so that it belongs to the nearest if statement to which it could possibly belong.

### 3.13.8 Iteration Statements

The iteration statements are as follows:

\[
\text{while-stmt} \leftarrow \text{while (expression) statement}
\]

\[
\text{do-stmt} \leftarrow \text{do statement while (expression) ;}
\]

\[
\text{for-stmt} \leftarrow
\begin{align*}
\text{for} \\
& \text{( [expression] ; [expression] ; [expression] )} \\
& \text{statement}
\end{align*}
\]

In the while and do statements, the contained statement is executed repeatedly as long as the value of the expression is True. The expression must be Boolean, as in §3.13.7. In the while case the test, including all side effects of the expression, occurs before each execution of the statement; with do, the test follows the statement.
In the `for` statement, the first expression is evaluated first and only once. It may be any type or `void`. The second expression must be Boolean. It is evaluated before each execution of the contained statement. If it is `False` the `for` statement terminates and control passes to the following statement. If it is absent it is taken to be `True`. The third expression is evaluated after the contained statement in each iteration. It may be any type or `void`. All side effects of expression evaluation complete before the next expression or statement is evaluated.

### 3.13.9 Switch Statement

The `switch` statement has the form:

```
switch-stmt ← switch ( expression ) compound-stmt
```

It causes control to be transferred to one of the statements in the compound statement depending on the value of the expression. Statements of the compound may have switch-labels as shown here:

```
switch-labeled-stmt ←
    case expression : statement
    | default : statement
```

The `case` expression is a static initializer, as described in §3.10.2. The transfer is made, as if by `goto` (§3.13.10), to the statement whose `case` label is equal (==) to the value of the `switch` expression.

A statement may have several switch-labels. A statement may have a switch-label even if it is deeply nested inside a `switch`; the label is associated with the nearest containing `switch`. No two of the `case` labels associated with the same `switch` may have the same value. There may be at most one `default`. If no `case` label matches the `switch` expression, and there is a `default`, control passes to the associated statement. If no `case` label
matches and there is no default then the exception FAILURE "switch: no matching case" is signaled.

The switch-labels do not in themselves alter the flow of control, which treats them as ordinary statement labels (§3.13.10). Because of this, the break statement (§3.13.11) is often used to exit from switch statements.

As mentioned in §3.13.2, automatic initializations are omitted when a goto is used. Therefore, they are useless in the switch compound statement.

### 3.13.10 Goto Statement

The goto statement has the form:

```
goto-stmt ← goto id ;
```

The *id* is a literal, a label which is somewhere attached to a statement in the following way:

```
labeled-stmt ← id : statement
```

A statement may have several labels, but each must be unique. The scope of a label is the entire function in which it is defined. Labels have a separate name space, so that the use of an identifier as a label does not collide with any other use of it.

The effect of the goto is to transfer control to the statement marked with the matching label. As mentioned in §3.13.2, when a block is entered with a goto, automatic initializations are omitted.

Attaching a label to a statement does not in itself alter its semantics or the flow of control.

### 3.13.11 Break Statement

The break statement has the form:
break-stmt ← break ;

It terminates the smallest enclosing do, while, for, or switch statement. Execution continues with the statement following the terminated statement.

### 3.13.12 Continue Statement

The continue statement has the form:

```
continue-stmt ← continue ;
```

It causes control to pass to the loop-continuation part of the smallest enclosing do, while, or for statement. It is as if control had flowed normally out of the statement enclosed in the loop statement.

### 3.13.13 Return Statement

The return statement has the form:

```
return-stmt ← return [ expression ] ;
```

It causes a function to return to its caller. The value of the expression, if present, is the value returned to the caller. The expression is converted, as if by assignment, to the type returned by the function in which it appears. The absence of the expression must agree with the function being void. (§3.9.1)

If control flows off the end of a function, it is equivalent to a return with no expression.

### 3.14 External Objects at Compilation

During a compilation the compiler may load objects from external libraries. They may be examined or modified. If not modified they will not go into
the object module which results from the compilation. There is a standard environment, a set of objects which is always loaded at the beginning of compilation. They are examined but not usually modified.

3.14.1 When the Compiler Loads Objects

In conventional languages, when an external object is mentioned in a program, the compiler gives the problem of gaining access to the object to another program, the linkage editor, which is run afterward. This is not only convenient for the compiler's writer, but desirable for its own sake, since it increases the potential for modularity. It allows program modules to be written and compiled in any order, regardless of their references to objects in other modules.

In G, however, certain external objects must be examined by the compiler. For example, the compiler must load a referenced grammar into memory so that it can learn its namelist names and the cells to which they refer. For the same reason, when it must interpret a literal grammar, it must load a parser for the grammar and run it. This is not to say that the compiler may load referenced objects freely. For the reason given, it must keep this loading to a minimum.

Specifically, two situations cause objects to be loaded. First, an external namelist object must be loaded.

Second, a static initializer may cause loading. Static initializers appear in declarations (§3.10) for static storage, of course, but also in case labels (§3.13.9) and init-parsals (§3.9.1.1). (Although a pure function application may be executed statically, it does not cause loading.)

Which static initializers cause loading? The compiler avoids loading external objects whenever it can, but there is essentially only one situation when it can, namely, when the external identifier is the whole initialization expression. Certain slightly more complex situations are equivalent, for instance, when an
CHAPTER 3. G LANGUAGE MANUAL

ident whose value is an external cell is named as the initializer. Any operation on an external object, however, either explicit or implicit, causes loading.

These operations are, specifically, any examination or alteration of the object's value, properties, fixation, type, or mother. An implicit change to its mother results if the object is assigned other than to a gate. An implicit type check is required if the compiler cannot determine from the declarations whether the object and its target are assignment compatible (§3.7). An implicit fixation occurs if an external cell which is not declared fixed is assigned to a fixed identifier.

When an initializer is a function invocation, all the arguments and the function itself are (completely) loaded and executed.

Loading an external object causes the file it is in to be read into memory. If future external references may be resolved to objects in the file, they are so resolved, in preference to a search of external libraries.

External references which are generated in turn by loading an external object may be handled in several different ways, depending on control commands to the compiler. At least these two options will be available: The minimum loading option will treat each newly generated reference as if it appeared in the source code, only loading it when it must do. The maximum loading option will always load the object referred to.

3.14.2 When the Compiler Unloads Objects

At the end of a successful compilation the compiler will have loaded a number of files to satisfy external references. Some will have been altered during compilation, some merely examined. In this context, an alteration is, as above, to value, properties, fixation, or mother. If a file is not altered in any of these ways then, before writing the object module, the file is removed from the memory image, and all references to it are restored to external references.
This allows separately compiled modules to have common references.

### 3.14.3 At the Top Level

Every program begins with the same text, formally a declaration.

```plaintext
program ←
    extern namelist unqual
    GramGram@GRule GramGram ;
genral-declaration`
```

It cannot actually be a declaration since it violates the rule that the type must previously be known. Nonetheless, the first `global-declaration` is compiled in the standard environment, the environment which in some sense results from loading `GramGram`. The names known therein are listed in Table 3.2. They

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>Galist</td>
</tr>
<tr>
<td>char</td>
<td>Galt</td>
</tr>
<tr>
<td>cestring</td>
<td>Gcons</td>
</tr>
<tr>
<td>functype</td>
<td>GDefParse</td>
</tr>
<tr>
<td>int</td>
<td>Gflink</td>
</tr>
<tr>
<td>iestring</td>
<td>Ggate</td>
</tr>
<tr>
<td>lestring</td>
<td>Glink</td>
</tr>
<tr>
<td>name</td>
<td>Glist</td>
</tr>
<tr>
<td>string</td>
<td>GMother</td>
</tr>
<tr>
<td>testring</td>
<td>GMothers</td>
</tr>
<tr>
<td>ustring</td>
<td>Gplist</td>
</tr>
<tr>
<td>vect</td>
<td>Gprim</td>
</tr>
</tbody>
</table>

Table 3.2: Names in the standard environment.

have all been discussed earlier in this chapter. (Names which are only known in qualified form are omitted.)

Two exceptions, `FAILURE` and `Mapcar_assign`, have been mentioned by name.
Execution begins with the function named \texttt{main}, whose type may be

\texttt{ANY main (varying)}

However, transmission of arguments and return of result are undefined for \texttt{main}. The treatment of exceptions signaled by \texttt{main} is undefined.
Chapter 4

Evaluation

*The Whole Business of Man is the Arts,*
*and All Things Common.*

William Blake

This chapter gives several ways of evaluating the work reported here. First are the originally proposed criteria, then some examples of G programs with criticism, then a survey of related work and a comparison of it with the present work. Finally, there is a report on the implementation.

4.1 The Work as Proposed

*Engineering is an art form that makes use of scientific principles,*
*and this marriage confuses a lot of people. We tend to think of engineering itself as a science, but it is nothing more than advanced carpentry. The practitioners learn by doing, and the craft is constantly in a state of evolution. When the Three Little Pigs built their houses, they all had acceptable designs—*
acceptable to them—but two of the piggies underestimated the wind loads.

The Warning
Mike Gray and Ira Rosen

I begin the evaluation of G by reviewing my own criteria, stated in my research proposal of 21 October 1986, quoted below.

**Language formality and elegance**

G should have a tight formal definition, as brief as possible. This is why I anticipate failing to define integer arithmetic. If it can be defined in terms of a package of subroutines, then there's no need to clutter up the formal definition with it.

In this I failed. G is neither small enough nor mathematical enough to be called elegant. I have not given it a formal definition, partly because other goals were more important, partly because such a definition is difficult, even for experts.

In [Sch86] we find evidence for this. On p. 178 type-checking for occurrence equivalence (which is like G, but simpler) is noted to be the subject of active research, from which we may conclude that it is not yet well understood. On p. 183 a similar conclusion is reached regarding polymorphism. A type structure like G's, that is, where types themselves are subject to dynamic creation, yet must be checked statically when they are static, is simply not discussed. It is too unusual.

G should borrow as much as possible from some existing languages, one if possible. (I favor C as the basis.) This will make the language easy to learn for at least one set of programmers. Also its nearness to some particular known language will make G easier
to evaluate. If it is an existing language overhauled for a new purpose, the set of features in need of evaluation is smaller and more interrelated than when the design is completely new. Evaluating an entire new language would be a daunting prospect.

Also, I would like to stick to a single design philosophy. A hodgepodge of design choices will make the language hard to learn. The use of more than one basis language militates against uniformity.

I did in fact use C as a basis, but features did enter from other languages. I rate my performance only fair.

**Procedural nature**

G should emphasize procedural programming style (as does C). I find applicative (functional) languages harder to understand.

This I did well.

**Modularity**

I reluctantly believe I should try to improve on the modularity of C. It seems arbitrary that one should be forced to use separate compilation as the most important method of hiding data. Also, there is no compiler-enforced data abstraction in C. For it to be possible (even if I don’t do it) to define integer arithmetic as if integers were an abstract data type, a data abstraction mechanism is required.

I mistakenly linked modularity and type abstraction here. When I realized that private properties provided a secure type abstraction mechanism in a
way that harmonized well with the structure of the existing data space, I abandoned the effort to improve on C modularity. Therefore, in this I rate myself both very good and very bad.

The CFG as type

G should be well-adapted to its *raison d'être*, the CFG as type. Type declaration should be done using all the common extensions to BNF (mentioned above). The data will be CF words whose structure will reflect their parse trees. Data manipulation operators will exist which are natural for these data.

This is well done. A qualification must be added to the sentence about declaration, since declaration is done using a parser, a user-written subroutine, which must interpret the aforementioned common extensions.

Storage management

Storage management should be invisible. There should be no need explicitly to allocate or free storage.

This is well done.

Dynamic types

In certain situations the CFG being used is not known at compile time, for example, in a grammar-based editor which is not specific to any language. For G to be useful there, it must be able to construct new types (and perhaps variables of those types) dynamically. We would like these types/variables to have much the same character as static ones.
We must be careful here. Our experience with typed languages is that types are declared to the compiler and type-compatibility is verified by it. In existing languages it is easy to see that these are quick calculations. What will we do if a programmer will not state type immediately, but defers it until execution-time or declares type implicitly as the result of some computation? Can we be sure that these calculations are quick?

I have no reason to believe that these calculations are quick, so I adopt the following approach: Unlike existing languages, G types will be visible at execution-time. The user will be able to construct new types as he requires them, explicitly using simple construction primitives. Thus the burden of computation of new types will be upon the user. A non-halting computation will be an extended loop (that is, several statements) in his program.

Execution-time type checking will be available, but I shall require that the user construct types which I can check quickly.

Likewise, any implicit type declaration I may allow at compile time, I will carefully restrict so that the compiler itself can be guaranteed to run quickly.

This also I say I have done successfully. Clearly, the problem discussed here has influenced nearly every part of G's design. It distinguishes G from other languages. Because of this, it is difficult to say whether this idea advances the state of the art. But, it satisfies the criterion as originally proposed.

As to the question of the last paragraph, whether the compiler will run quickly, I note that this cannot be guaranteed because the compiler is required to execute user subroutines.
CHAPTER 4. EVALUATION

Literals and initialization

It should be easy to express a particular datum (of arbitrary type) in the text of a G program. This case will make the use of literals pleasant and perspicuous, both in expressions and initializers.

This is well done, given the qualification that it is accomplished through the intervention of a user subroutine.

Partial syntagms

It is often the case that one wishes to deal with a syntagm whose value is not completely known. Parts of its syntactic structure are fixed; the rest are not. We would like a way of expressing a datum as a literal which has some constant parts and some variable parts, and we want to refer to the variable parts with variables—our usual programming language notion of variables (also called metavariables in certain contexts)—for convenience in coding.

As an example, consider

\[
\text{while (i == 0) } \$V1 \\
\]

which is a form for a while-loop. For the moment, I’m using the $-sign to introduce a variable. Notice that the value of the variable is a syntagmatic type (a statement), not just an arbitrary character string. If I wanted to vary the 0, I would have to introduce a second variable:

\[
\text{while (i == } \$V2 \text{) } \$V1 \\
\]

It should be possible to assign to these variables directly, thus instantiating a CF syntagm. However, it should also be possible to
assign implicitly by unifying two syntagms. The result of unifying two partial syntagms will in general also be partial. As a consequence a variable will sometimes be assigned a partial syntagm as the result of unification.

Unifying

\[
\text{while (} i == 0 \text{) } \$V1
\]

with

\[
\text{while (} \$V2 \text{) } i = \$V3
\]

gives

\[
\text{while (} i == 0 \text{) } i = \$V3
\]

In this case, \( \$V1 \) has the value

\[
i = \$V3
\]

This complicates the notion of evaluation. When will the result of evaluating a variable be a variable?

This idea of partial value is very rare in programming languages. It is like saying, for an integer variable, "Its value is 1 modulo 3, and it's a prime number, but I don't know what its exact value is."

(Partial values, as illustrated in the last paragraph, are used in constraint logic programming languages. [Coh90, Col90])

The reader will note my attraction to Prolog-style unification at the time of the proposal. I was later forced to refine my concept of syntagm matching in the light of the design of these things: the data space (specifically, my decision to use cells as the foundation of the data space), L-values, none, and
idents, as well as function invocation (the decision not to pass L-values, which makes idents unavailable outside their lexical scope).

The problem is, at bottom, that two cells can never be unified, as two values can. They are forever separate in the store. I designed half-match to try to reconcile the intuitive idea of unification with the store.

In the parsal this reconciliation forced me to distinguish between variables that were being assigned to and assigned from. This was accomplished by an elaboration of the syntax. At the same time syntax was added to give better control (via conventions) of the exact placement of none.

Despite the fact that, with the extended syntax, partial parsals seem a bit cluttered and awkward, I rate this part of the design fairly good, because I still cannot think of any other design which is a clear improvement on this one.

### 4.2 Example Programs

Here are some examples of G doing the sorts of things it was designed for. Each example includes a brief discussion and evaluation of the design of G as it applies to it.

#### 4.2.1 Insertion of Trace Statements

This example adds simple tracing code to an input Pascal program. It transforms every subprogram in the input. The heart of the transformation is as shown:
However, a and b are parameters. b is the name of the subprogram, and a must be a newly generated variable name. It is also necessary to declare and initialize all these generated variables in the global scope.

To understand the example, we need to see a fragment of the grammar for Pascal used by this program. It is really two grammars, a concrete and an abstract. The concrete one contains all the fixed tokens necessary to guide the parser, as shown by this sample:

Program ::= "program" Ident [ "(" Idents ")" ] ";" Block "."  

Block ::= [ "label" Labels ";" ]  
[ "const" ConstDefs ";" ]  
[ "type" TypeDefs ";" ]  
[ "var" VariableDecls ";" ]  
[ SubprogramDecls ";" ]  
"begin" Statements "end"

But, these tokens are just extra baggage once the syntagm is parsed, so they are eliminated and a syntagm of the abstract grammar results from a simple mapping.
CHAPTER 4. EVALUATION

Program ::= Ident Idents Block

Block ::= Labels
  ConstDefs
  TypeDefs
  VariableDefs
  SubprogramDecs
  Statements

SubprogramDecs ::= SubprogramDecl*

SubprogramDecl ::= ProcedureDecl
| ProcedureSpec
| FunctionSpec
| FullFunctionDecl
| FunctionBodyDecl

ProcedureDecl ::= PHHeading Block
FullFunctionDecl ::= FHeading Block
FunctionBodyDecl ::= Ident Block
ProcedureSpec ::= PHHeading
FunctionSpec ::= FHeading
PHHeading ::= Ident Parameters
FHeading ::= Ident Parameters RIdent
Statements ::= Statement*
Idents ::= Ident*
VariableDecs ::= VariableDecl*
String -> string
Ident -> ustring
Most of the abstract grammar for Pascal has been omitted. We show only what is needed here.

(Note that the optional phrases in the concrete grammar (label, const, etc.) have been represented as ordinary lists in the abstract. This is a simplification. In the concrete case, if we write the word label, we must write at least one label. The syntax must account for this. However, in the abstract there is no reason to differentiate between zero and one. Zero may be treated as the other numbers are.)

We present the program in three segments.

/* Trace program in G */

extern namelist unequal GramGram@GRule GramGram;
extern namelist unequal GRule PascalGram;

int gen = 0; /* Generated variable counter */
Ident varlist; /* List of generated variables */
Statements initlist; /* Initializes generated variables*/
Program pgm;

/*nb: All G identifiers with well defined types are initialized with a cell of that type. */

void main()
{
    Block t;

    pgm = parseinput(Program);
    tracepgm(pgm);
The abstract grammar mentioned above is the one actually named in the second declaration. The concrete grammar accompanies it and is linked to it as a property.

The top routine (always named main) uses a universal parser which, by reference to the two grammars, parses the input and returns an abstract syntax. It calls tracepgm which inserts the tracing statements and sets the variables varlist (the list of generated variables) and initlist (statements to initialize the generated variables).

Here we must remind ourselves of \# the selection operator (§3.9.5.1), which finds the daughter of the current cell, and of :>: and :> list insertion (§3.9.13) and of :> list removal (§3.9.12). Since initlist is already a statement list it is inserted immediately into the body of the Pascal program. Note the use of #1 instead of #. This documents the fact that the position indicated is one of several possibilities. We prefer to use # when there can only be one daughter.

On the next line it is necessary to use a parsal to convert an Ident list into a declaration. The parsal invokes a parser which has the ability to deal with both abstract and concrete fragments. The varlist is already abstract, but "integer" is only a string, which must be recognized as a Pascal type.

Of course it is possible to write the necessary operations explicitly, but the program is more readable and maintainable as it is.

```c
void tracepgm(PascalGramAny p)
```
{ 
int i;
PascalGram@any s;
Block t;

switch (type(p)){ /*Raises exception on missing case */
case (Program):
case (FunctionBodyDecl):
    s=p; break;
case (ProcedureDecl):
    s = p.PHeading; break;
case (FullFunctionDecl):
    s = p.FHeading; break;
case (FunctionSpec):
case (ProcedureSpec):

    return;
}
t = p.Block;
patchlist (t.Statements, gensym(), s.Ident# -> name );

s = t.SubprogramDecIs;
for (i=1 ; i <= length(s) ; i = i+1 )
   tracepgm (s#i#);
}

The function of tracepgm is to examine its argument and find the enclosed subprogram declarations. With these it recurs to itself so that more deeply nested ones are also found. In each case it locates the statement list part of the program or subprogram and passes it to patchlist for the necessary change. (Note the use of the name property in the argument list to patchlist.
It retrieves the name of the identifier (a ustring) in the form of a string. (§3.1, §3.2.5)

```cpp
void patchlist(Statements s, Ident a, string b)
{
    Statement[ "'a := 'a + 1"]  >:
        Statement["if 'a <= 5 writeln ("PascalGram@String["Entering 'b"] /* nb: No Pascal*/
            ")"]  >:  /* quote marks */
            s#1;
    s#++  = Statement["if 'a <= 5 writeln
            ('Leaving '/b ')/];  /*Pascal quotes must be */
            /* used in this approach*/
}

Ident gensym()
{
    int    j;
    string a;
    Ident  t;

    j = gen = gen + 1;
    a = i2str(j);
    &"GEN"  :> a#1;
    t = Ident[ a ];

    varlist#++  = t;
    initlist#++  = Statement[ t ":= 0" ];
    return t;
}
```
patchlist makes sophisticated use of the parsal facility. The first parsal simply synthesizes an assignment statement. However, it uses the identifier in a as the variable name. The a is anti-quoted out of the literal string, which might equally well be written

[ a " := " a " + 1" ]

Again the parser copes with fragments from different grammars.

The next statement synthesized has a more subtle problem. It contains a Pascal literal string with a variable part. This is conceptually different because the string is a Pascal lexeme and so might pose a communication problem; the parser must instruct the lexer that it must treat several passed values as a single string. However, if this is impossible the user has several options available. He may invoke the lexer explicitly by writing a lexer parsal, or, if that fails, he may simply construct the string explicitly in a subexpression.

The explicit use of the type name PascalGram@String tells the parser that the whole nested parsal becomes a string. This method avoids the need for writing any Pascal quote marks and documents the parsal. The statement shown next uses the other method: It embeds Pascal quotes in the parsal using the u-quote syntax (§3.1.4.1).

Note the s#++. It names the L-value following the last statement. Since the location is known to be empty, it is filled using assignment rather than list insertion.

gensym generates a new identifier by incrementing the global counter and converting the value to a string. It is valid to worry that the identifier created by gensym might collide with an existing identifier. We do not treat the problem here. However, the difficult part of this problem, detecting the collision, is easily dealt with by the machinery described in §4.2.3.

When constructing the new identifier, we presume i2str constructs a new string, but we know the literal "GEN" is fixed by the compiler, so we must
CHAPTER 4. EVALUATION

copy it before transferring it to a. This is necessary because list insertion will
disassemble it during the transfer.

To finish up, the string is converted (using a parseal) to a Pascal identifier,
which is then added to the list of identifiers. An assignment statement is
constructed which initializes it.

4.2.1.1 A Schematic Approach

Let us explore briefly a more schematic approach to this problem.

```c
void main()
{
  Statements s;
  Program p;
  VariableDecls v;

  pgm = parseinput(Program);

  [p "program ' $ - ( ' $ - ) ;
      'Labels' -
      'ConstDefs' -
      'TypeDefs' -
      'VariableDecls' v
      'SubprogramDecls' -

      begin ' $ s end."

  ]

  // tracepgm(pgm);

  initlist :> s#1;
```
This version uses a pattern parsal as an argument to the half-match operator. When the match is complete \( v \) holds the declaration list and \( s \) the statement list for the new variables. It is much more graphic than the version shown above and allows the reader more easily to see what parts of the syntagm are being denoted by \( v \) and \( s \).

However, it is also more verbose and less precise. The fact that the Statements cell is two levels below the Program cell is obscured in this version. It might as easily be one or three. Likewise, the extra verbiage is confusing to the eye, say, if one is scanning the program for something else.

Coding style is a matter of judgement. A novice user tends to choose the verbose form for its mnemonic value. The experienced user often chooses the succinct form because he prefers the precise statement of the action and knows the details of the grammar fairly well without a reminder. G has a broad enough spectrum to accommodate both coding styles.

There are other possible forms for the parsal. The same syntagm could be generated by

```plaintext
["program "#- ( "$- ") ;
    label "$- const "$- type "$- var "$v "
    begin "$s end. "]
```

or even

```plaintext
["program" "$- "$-
    "label" "$- "$- "$- "$v "$-
    "$s "]
```
These forms give up some of the verbosity of the original. The first one is
the more intuitive and perhaps the better compromise, but it loses clarity by
not explicitly naming any types. The last is the most compact, but relies on
a good knowledge of parsal conventions (§3.9.1.3) and the particular parser
involved.

(The reader may notice that the syntagm is copied in this version, rather
than being manipulated directly, thus doing what may be a considerable
amount of work. This is not entirely a real problem, but is more an arti-
fact of the scope of this work. We have mentioned (§2.6.3) that parsals may
do assignments, thus avoiding the necessity of expensive copies. However,
since the detailed design of parsal parsers is for future work, I have not made
use of the feature in this example.)

4.2.1.2 Evaluation

How has the design of G (ch. 2) affected its success in this example?

Persistent data were used at the outset when we loaded the grammar of
Pascal. It would be cumbersome to see the entire grammar written into the
program body. Our attention to this concern (§2.9, §3.14.1) has resulted in a
much more readable program.

The use of visible types makes the use of a single input routine, parseinput,
reasonable. It relies on a single parser, a universal parser, which parses ac-
cording to the grammar it is passed. If we suppose the contrary, that there is
no visible grammar, we would undoubtedly use a parser generator. We would
previously have fed it some arbitrarily defined representation of the grammar,
and it would have generated and stored a table for its own future use. For
LR(1) parsers we cannot get away from the table, but in G the representation
is no longer arbitrary and, so, is available for other tools. The standardized
grammar is valuable, not only for the information content, but also as a place
to store tables, etc. In fact, the parser itself might easily be stored there.
The use of \textit{cells} (in distinction to a referentially transparent language) seems to simplify a few things. In \texttt{main}, for example, we would not have explicit I/O routines, since they rely on side-effects. The input would be bound to a parameter, instead. Supposing we have an imperative language, we would likely make one or two steps and bind results to intermediate variables, \texttt{pgm1}, \texttt{pgm2}, \ldots, so as to avoid the visual confusion of reading deeply nested function calls inside out. Even so, it is useless to define something like the variable \texttt{t}, a position below the root of the syntagm. Where, in the original program, we had

\begin{verbatim}
  t = pgm.Block;
  initlist ::= t.Statements#1;
  VariableDecl[ varlist "integer"] ::= t.VariableDecls#1;
\end{verbatim}

now, the position must be repeatedly renamed:

\begin{verbatim}
  pgm2 = ReplaceStatementsOfBlockOf( pgm1, 
                                       initlist(pgm1) ::= pgm1.Block.Statements#1 );
  pgm3 = ReplaceVariableDeclsOfBlockOf( pgm2, 
                                       VariableDecl[ varlist(pgm1) "integer"] 
                                       ::= pgm2.Block.VariableDecls#1 );
\end{verbatim}

While this notational complexity does matter, the underlying conceptual complexity is more important. To me it seems clear that we want to think of a syntagm as a thing, a persistent thing, like an apple tree. We may come along and saw off a branch and graft on a branch from a peach tree (Type rules forbid us from using a branch from an orange tree.), but the tree keeps its continuing identity, even though we concentrate our attention on the point of the graft. This is really a kind of modularity.

The use of \textit{parsals} is shown, both where I used them naturally in \texttt{patchlist} and as an option for a more explicit style of coding in \texttt{main}. In \texttt{patchlist}
I consider it indispensable. The syntagm being constructed is mostly literal. If strings or type constructors are the only options for making syntagms, the program text needed to construct this one would be more obscure.

I cannot say that parsals are as readable and intuitive as I had hoped they would be. Some possible changes have occurred to me that might improve their cluttered appearance. One example is to make quotes implicit inside the brackets, thus eliminating the initial and final quote mark. However, this tinkering with the very sweetest layer of the syntactic sugar seemed rather premature. A language design needs feedback from users on deeper issues than this before it may be foist on an unsuspecting world. The current syntax seems reasonable as an initial offering. We will improve the language as users comment on it.

The use of flat lists and associated primitive operators is also evident in main. The advantage they give is in simple readability. Modification of the alternative structure (shown on p. 31), while it can be done (as a general syntagm modification), is not obviously a list insertion. The use of a bit of special syntax makes it obvious. I feel list operations are important enough and common enough to warrant this special treatment.

4.2.2 A Universal Parser

To parse is to convert a string into a syntagm. Parsers usually parse according to a specific grammar, but in a language where there is a visible representation of the grammar, there is no reason why one parser cannot parse all grammars. This removes a step from the usual approach of, for example, YACC [Joh75]. YACC is a generator, so, in a separate step, one feeds it a (string) representation of a grammar and receives a program, the parser.

The parser presented here is a recursive descent parser, and so parses L.L.(1) languages, but this is only to keep the example simple. We present first the
code and an explanation, then we will discuss some alternatives to the design used.

/* Universal recursive descent parser */

extern namelist unqual GramGram@GRule GramGram;
GramProp TokenClass;
ustring NextToken;
extern void AdvanceToken();

ANY Uparse( GRule t )
    signals string ill_formed_grammar,
            string parse_failure {
    ANY r = gcons(t);
    fixed GramGram@any k = t#;
    int i;

    switch( type(k) ) {
    case Galt:
        for (i=1 ; i <= length(k) ; i=i+1 ) {
            r# = Uparse( k#i# ) ;
            return r;
            catch parse_failure (string s) {}}
    }
    signal parse_failure "Alternatives all fail" ;
case Gcons:
    for (i=1; i <= length(k); i=i+1) {
        r#i = Uparse( k#i#);
        resignal parse_failure;
    }
    return r;

case Gplist:
    case Glist:
        i=1;
        while (True) {
            r#i = Uparse( k##);
            i = i + 1;
        }
        catch parse_failure (string s) {}
    if( type(k) == Glist || i > 1 )
        return r;
    else
        signal parse_failure
            "Plist does not find at least one element";
case Ggate:

    if ( k## != ustring )
        signal ill_formed_grammar
            "Leaves are not all ustrings";
    if( k# -> TokenClass == NextToken -> TokenClass ) {
        r# = NextToken;
        AdvanceToken();
    } else {
        signal parse_failure /"Tokens don’t match"/ ;
    }

}
}

We assume that the string to be parsed is already broken into tokens by a lexical analyzer, which converts them into ustrings (atoms that are the unique representative of a spelling) and tags them with a token class, e.g. identifier, plus-operator. The tag is by means of a property, the marker for which has been stored in the variable TokenClass previously. The first unused token is always found in NextToken. To move ahead to the following one, we call AdvanceToken.

The parsing routine has one argument, the rule governing the string to be parsed. It is a GRule, a G type. The parser advances over tokens from the stream and returns a syntagm of this type.

The body of the routine is the switch-statement, which handles each of the possible kinds of rule: alt, cons, list, plist, or gate. The possibilities are distinguished by reference to the internal structure of the grammar.

Fig. 4.1 shows four of the possibilities. (For details see §3.3.3.) An alt or cons rule gates to several rules. In the actual syntagm, the daughter of an alt cell may be any of them, while the daughters of a cons cell must be all of
CHAPTER 4. EVALUATION

Figure 4.1: Some kinds of rule.

them in the order given. A list may have any number of daughters of the one type given. A plist (not shown) must have at least one. A gate must have one of the type given.

In the figure, the words are the types of the cells shown. In G a type is itself a cell, a cell whose name is this word, in this case, a cell in the grammar of grammars. This cell is reached by the type operator (§3.9.1.2).

(Placing the target rules at the bottom of the figure is deceptive. The rules of a grammar are all in a single flat list, the daughters of the single GRules cell in each grammar. So, the target rules in the figure are in fact sisters of the rules at the top, except in the gate's case. A gate may take its target from any grammar. In this case all the gates hold usstrings (§3.4), which have their own grammar.)

Inside the switch-statement the parser examines the grammar and discovers the types needed and recurs to itself to parse them. The subtrees so obtained are then made daughters of the cell in r, which is the resulting tree. Notification of failure to parse is given by the exception facility (§3.12), so no explicit status is returned. However, allowed exceptions must be intercepted.
4.2.2.1 Alternatives to the Parser

The first and most important design decision of this example was what sort of parser to write. It is tempting to use Earley's algorithm [Ear70] and say that the parser truly is universal, or to give an LR parser and say it is realistic. However, it seemed that one "table-driven" parser ought to be as good as another, and so I let simplicity prevail.

The reader might notice that there is a difference in tables. This parser is driven directly from the grammar to be parsed. An LR parser must construct a special table, not an easy table to construct, either. Is this a significant problem? It is not, because the table need only be constructed once. It may be attached to the grammar as a private property for use on all future calls to the parser.

The interface to the lexical analyzer is straightforward. However, it has its own problems coping with several grammars. The only one that appears in the interface is the TokenClass variable. Since there is only one ustring for any spelling, in situations where there are two languages it may be necessary for the same spelling to have different token classes, one for each language. This is accomplished using different property markers, hence the passing of a marker to the parser.

Finally, it was not necessary to implement lexemes as ustrings uniformly. The reader may have noticed that some of the value of the type system is lost if every leaf is a ustring. Ustrings are most important as identifiers, giving convenient access to symbol information. For punctuation, arithmetic operators, etc., perhaps atoms, rather than gates to ustrings, make the most sense. There is no need for a leaf to have a value if its value is always a +,-sign. That information can be conveyed adequately by its type.

We have not mentioned the conversion of the concrete grammar to an abstract one. If a concrete grammar is designed for the purpose, it can be
CHAPTER 4. EVALUATION

converted into an abstract one automatically, simply by removing the punctuation and leaving the identifiers. The missing information is then implicit in the types of the interior nodes of the syntags. Such a conversion is facilitated by the above convention for gates and atoms. It indicates what to keep.

4.2.2.2 Evaluation

This example is, of course, a tribute to the value of visible types or, more precisely, visible grammar-as-type. In G writing this program is easy.

In another language, the main problem would be to design a representation of the grammar for the parser to operate on and then to operate on it using general-purpose facilities. At every step the code would be more cumbersome and obscure. The value of G's philosophy that grammar is type is that the syntagm returned by the parser is guaranteed correct by G itself, not merely by the implementation of the parser. The construction of the syntagm is checked for type correctness at each stage automatically.

The use of properties is greater here than in the last example, but they are most thoroughly exercised in the next example. Here they are used to convey the token class of each lexeme with respect to the current grammar. This is something that we believe is strongly associated with a lexeme, but is not legitimately part of its inherent structure, since it is attributed to it by a particular grammar, a particular lexical analyzer. This compartmentalization of data by use or function is the purpose and justification of properties.

We also see some use of exceptions here. The code is clearer and simpler as a result. They are used to avoid the passing of an explicit return status from the parser. The code to handle exceptional returns is still near the source of the error. But, it need not immediately follow the source, as it would without exceptions, but can stand at the point where control resumes, a visual marker.
4.2.3 The Pointer Guard

In this section we develop an example which solves a problem of commercial size and difficulty. It analyzes and transforms a correct (ANSI) C language program. The transformation puts a guard at every place in the program where a pointer is used to address data. The guard is a call to a subroutine (not shown) which examines the pointer in an implementation-dependent way to verify correctness. Schematically, the transformation may be written thus:

\[ * \ expr =\to * (\ type-name ) \ pointer\_guard(\ void * ) \ expr \]

It makes use of the guarantee that any pointer may be cast to a void pointer and back without loss of information. The main difficulty is, clearly, to find the type of an arbitrary expression, a highly context-dependent calculation.

For this example we will need a complete grammar for C language and more code than will conveniently fit in this section. Both appear in appendix A.

The grammar appearing is abstract. It corresponds to the concrete grammar shown in [KR88, §A13]. For the purpose of program transformation the abstract grammar is an improvement over the concrete one in several ways. Besides the usual omission of key-words and punctuation, it has been simplified. The grammar of expressions has been flattened to a single large alternation, Expr. Use has been made of plists and lists where appropriate. The grammar for “old-style” functions has been omitted.

Some use has been made of anonymous types in this grammar. For instance, the first daughter of Function_def is an anonymous list, a list of Decl_spec. Likewise, an optional element, such as the Ident in Struct_spec is implicitly an anonymous alternation, with nil as the implied alternative.

Several small changes have been made to the grammar for simplicity. Functions of a fixed and a variable number of arguments are distinguished earlier in the grammar. The production for Struct_spec, etc., now allows the empty string, which corresponds to an illegal concrete program.
Struct_declarator, likewise, allows an empty declarator.

The grammar for Declarator has been linearized so that, as for expressions, the structure of the syntagm now has a closer relationship to the meaning and a more distant relationship to the textual version of the program. In this case I have chosen to represent the Abs_declarator, a list of Type_modifier, in order from most tightly to least tightly bound. Thus,

\[
\text{int (* a[15]) (int)}
\]

becomes as shown in fig. 4.2, that is, "array of pointers to functions."

There are four type modifiers: Array, Pointer, Func, and V_func, that is, "array of," "pointer to," "function of fixed number of arguments returning," and "function of variable number of arguments returning."

The transformation program itself is in two parts: maketypeinfo goes through the program and discovers the types of all identifiers. The second part is guardpointers, which actually performs the transformation.

We will see that the first part is quite general and powerful, containing almost all of the machinery necessary to discover the answer to any question about type that can be posed about program identifiers. The major omissions are statement labels, which are ignored, and forward references to structure.
definitions, which have been assumed away. Both are easily remedied and are
omitted here for the sake of clarity.

The type information is kept in two kinds of record, which \texttt{maketypeinfo}
creates and attaches to the syntagm in properties. The record types \texttt{DclType}
and \texttt{ScopeDictionary} are both defined in literal grammars on p. 191.

Following this are declarations for all the routines called by this program.
Except for the last two, they are all defined in the appendix and appear in
the order shown.

They implement the following idea: The identifiers in the C program will
become gates to \texttt{ustring} in the abstract syntagm. As hinted in §3.4, we will
keep a pointer to every defining occurrence for a given identifier on its \texttt{ustring}
in properties. That is, each scope will have a distinct property marker. At-
tached as a property to each \texttt{ustring} known in that scope is the defining
occurrence (a gate to the \texttt{ustring}). The attachment is made using the scope
property marker.

On each defining occurrence gate we will keep an abstract representation of
its declared type (in a property) in a \texttt{DclType} record. On each scope subtree
we will keep its distinct marker (in a property) as part of a \texttt{ScopeDictionary}
record.

The \texttt{ScopeDictionary} contains some other data as well. Since in C, struc-
ture, union, enumeration tags are in a separate name space from ordinary
identifiers it is possible for the same \texttt{ustring} to have both meanings in a
given scope. Therefore, there is a separate marker for tags in each scope so
that the proper defining cell may be found. The \texttt{ScopeDictionary} also con-
tains a gate to the \texttt{ScopeDictionary} of the enclosing scope and lists of the
identifiers and tags known in this scope; these data are created but not used
in this application.

After the global scope there are three ways of introducing new scopes.
Each compound statement introduces a new scope. A function definition has
CHAPTER 4. EVALUATION

an extra scope for its parameter declarations. A structure or union definition opens a new scope for identifiers but not for tags.

A DclType record is in two parts, the BaseType and the modifiers. If the BaseType is a structure, it contains a gate to the subtree forming the defining occurrence. Primitive BaseType's are atoms. Type modifiers are used directly from the host syntagnm. They are in the same order and have the same meaning as they do there.

typedef names are not BaseType's. Instead the type structure represented by the name is fetched and any new modifiers appearing in the defining declarator are appended to the existing ones. This accords with the C philosophy, which says that typedef's do not introduce new types but only aliases.

4.2.3.1 The Functions

We now look briefly at each function to see what it does. We discuss them in their order of appearance in the appendix.

maketypeinfo is called when a new scope is encountered. It creates a new ScopeDictionary and attaches it to the appropriate cell. It then calls proc_decls to process the declarations in the new scope. When the new scope is a function definition, it is known to contain another new scope, a compound statement, so maketypeinfo recurs to itself. When the new scope is a compound statement it may contain further scopes, so it must be searched. proc_stmts does this.

proc_decls calls proc_spec to read the type specifier. This is usually just the BaseType mentioned above, in which case a DclType syntagnm is returned which contains no modifiers. However, when the specifier is a typedef name, modifiers may appear.

Next, proc_declarators is called to read the declarators and complete the DclType for each identifier declared. When the declaration is a function
definition, a new scope must be created.

**proc_declarators** exists mainly to find the declarators in the syntagn it is given. Because of the irregularity of the C grammar it has several cases it distinguishes. When the argument is an enumeration literal, no declarator is there at all, so **proc_declarators** must create one.

**single_declarator** does the actual work. It copies the type modifiers into the DclType syntagn. It adds the new identifier to the ScopeDictionary if it wasn’t there before. It adorns the ustring with the defining occurrence using the current scope’s property and stores the DclType on that occurrence using the global property **markDclType**.

**proc_spec**, like **proc_declarators**, first has to cope with the C grammar to find the specifiers. Then it steps through them one by one using a switch statement to note what it has seen. Next it constructs a new DclType syntagn. (Note the use of a parsal with no literal elements.) Finally, one more big switch statement completes the DclType based on the information noted above.

If the base type is simple (**float**, **char**, **int**), **BaseType** is set to the appropriate value, and the job is done. However, a structure or union type requires more work. If this declaration contains the definition, then the tag (if it exists) must be added to the ScopeDictionary (if it wasn’t there) and the tag’s ustring must be adorned with a property gating to this defining syntagn.

Finally, the new definition must be explored for new declarations. A new scope dictionary is constructed here in line, since it has a notably different structure from the dictionaries constructed by **maketypeinfo**. The difference is that only those identifiers declared in the structure are known there. However, tags persist from the enclosing scope. In fact there is no new scope for tags. New structures declared inside are immediately exported to the enclosing scope.
This new ScopeDictionary has been constructed "by hand," i.e., using only G built-in functions. We note the more cumbersome appearance when contrasted with the DclType made via parsll, mentioned in proc_spec.

An enumeration is treated much as a structure is, except that it does not have its own scope. The literals are implicitly declared as integers in the enclosing scope. A new DclType is made in line for this purpose.

Finally, an identifier is recognized as a typedef name. The DclType is copied from its own defining occurrence, which is easily found using the current scope dictionary and its ustring.

proc_stmts and single_stmt simply search a statement list for nested scopes and recur to maketypeinfo when each is found.

We have now seen the meat of this example, the use of properties and markers to adorn a syntagm with context-sensitive type information so that the type and defining occurrence of every identifier may be quickly discovered from every use of it anywhere in a C program. We now go on to use this information to construct pointer guards.

guard_pointers must find every place a pointer is used as an address and make the appropriate change to it. Unfortunately, these places are not always simple references (*e). A pointer reference may be disguised in one of three ways: as an array reference (a[2]), as an indirect structure reference (a->n), or as a function invocation (f(x)) when f has been declared to be a function pointer (perhaps implicitly, by being passed as an argument).

So, guard_pointers converts each of these cases to an explicit reference before it works the pointer transformation. It finishes by adding a global declaration for pointer_guard, the C program which does the actual check.

Its method for searching and modifying the syntagm is notable. It uses a general routine, find_type_in_tree, which, by taking advantage of the visibility of grammar, can search an arbitrary syntagm for a given type and invoke a passed function when it is found. This is a particularly compact
way of writing what might otherwise be a fairly complex control structure. Compare it with proc_stmts and single_stmt on p. 201, which also search the syntagm for a particular type. Instead of naming types explicitly in appropriate pieces of the control structure, the search is conducted uniformly and simply.

We pay a price for this simplicity, namely, the loss of some control over what parts of the syntagm are searched and when. However, our concern here is not with the best trade-off, but with the fact that there is enough scope in G for making the trade-off.

find_type_in_tree needs little further comment except to mention that it is one of several possibilities. It could be top-down or bottom-up. It need not stop at gates. It might take a list of types and functions and call the appropriate one when each type is found. It might backtrack or implement other more complex control strategies. Almost everything done in this example program might have been done with the aid of a tree-searching function.

q_array, q_follow, and q_fun simply rewrite their expressions into explicit pointer expressions, as mentioned above.

q_ptr obtains the type of the expression being referenced from expr_type. It comes in the form of a DclType, our internal representation of type, so it must be converted back to a C syntagm, a Type_name, so it can be incorporated into a cast. First the BaseType is converted, using a big switch, then the type modifiers are copied. With the Type_name in hand it is easy to rewrite the syntagm into the desired form.

expr_type is responsible for discovering the type of an arbitrary C expression. It does not appear here in full generality. Several operators which can never return pointers (e.g. divide, xor, shift, modulo) are all taken to return int. We rely on the presumed correctness of the input program to exclude expressions like *(a + 1.3).

There are many cases which still must concern us. We will comment on
just a few and leave the others for interested readers to explore.

A function call begins with an expression which must be a function returning \( t \) or a pointer to a function returning \( t \). There is no need to examine the argument list. We simply remove the "function" and "pointer" modifiers from the modifier list, returning \( t \).

A cast requires us to locate the active scope dictionary, since we may have to look up a structure tag in it. Since we took care in \texttt{guard pointets} (p. 202) only to search function definitions, we can be sure that the nearest scope dictionary is on a \texttt{Compound stat} cell. We simply climb the tree until we find it.

Having found it, \texttt{proc spec} interprets the specifiers for us and we copy the modifiers in line.

To understand an identifier also requires a scope dictionary, which we obtain just as we did above. From this we immediately have the \texttt{DclType} from its defining occurrence. Usually this is enough. But, just in case the defining occurrence is in a function parameter list and its declared type is "function returning," we must correct it to "pointer to function returning," its true type. This mistake is officially allowed by the standard [KR88, p. 226], so we must be prepared for it to appear in our input program.

\subsection{4.2.3.2 Evaluation}

This example is most notable for its heavy use of \textit{properties}. The defining occurrence(s) of every identifier may be found (on its representation as a \texttt{ustring}) in the appropriate property. In turn, each defining occurrence has an abstract representation of its declared type attached in a property. Finally, each scope of definition for identifiers has a property containing some useful information, especially the marker needed to retrieve defining occurrences in that scope.

All this information makes answering context-sensitive questions about
identifiers easy, yet it doesn't interfere with the original syntagm (cf. §4.3.2), which is available for examination and transformation by other routines as if the information weren't there.

The use of type as CFG is shown here in our definition of auxiliary data structures, ScopeDictionary and DclType. Syntagms of these types are never parsed or generated. There is no concrete syntax. Yet, the grammar is given in the same way, using the same notation, as the grammar of C, the target language of this example. Surely, this is a convenience for users of G language.

A little more use is made of parsals, which were discussed as they were encountered in the previous section.

4.3 Related Work

FS: Talk, talk, talk, talk,
    Bicker, bicker, bicker,
    You can talk all you want to,
    But it's different than it was.
SS: No it ain't. No it ain't.
    But you gotta know the territory.

"The Music Man"
(dir.) Morton DaCosta

In this section we touch on a few of the many contributions made to the study of syntagm manipulation. No one seems to have set out to write an imperative programming language for the simple purpose of general syntagm manipulation, but there are many systems for more specific purposes.

The largest class is, of course, compilers. There are so many that automatic compiler generators are themselves now fairly common, for instance YACC [Joh75] and others mentioned in [AU77, chapter 1].
Program transformation, testing, and evaluation have also received attention. An extensive bibliography on program transformation systems appears in [PS83]. An overview of commercial test and evaluation software may be had in [DMMP87].

In what follows here, we present a sampler of manipulation approaches. There is no attempt to be complete. We have simply chosen a few to represent each approach. We first look at a few systems that use imperative languages as the means to their ends, then two that use purely declarative languages, then a few that use a mixture or hybrid language. Finally, we visit the world of general-purpose programming languages. There has been some recent success in applying object-oriented languages to our field. We mention two such cases.

4.3.1 Imperative Languages

4.3.1.1 Mentor

Mentor is an early effort to implement the syntactic view of computer languages [DHKL84, DKL83]. It started with a single-language syntax-based editor and became multi-lingual in two senses: It may edit any language by reading the grammar it needs from a table, and it may edit objects containing syntags from several grammars that are linked together in a structured way.

The method of linkage was of special interest to me [DKLM84]. I took two of the central ideas of G from this source, namely properties and gates (their annotations and gates). The differences are these: In Mentor a gate does not have (or does not simply have) a value. Instead it is isomorphic to it. (The language of [DKLM84] is vague, perhaps understandably, since the gate is only proposed in this work.) In G a gate has a value, much as other cells do, but with an enhancement: The types available for the value may be specified in more powerful ways, called broad types (§3.2.3). In Mentor a property is found using a named object, its frame. Knowing the name of a frame allows
one to see the property. In G markers may have no names, so that a property becomes invisible to all but a single function, if necessary. Again, property types may be narrow or broad.

They [DKL'83, §2.2.ii] note the usefulness of frames with secret names, a close relative of private properties, but while a frame’s name may be revealed, compromising the secret, a private property is permanently private.

The precise nature of grammar is different in Mentor and G. Mentor (like most systems) does not create alternation cells. Although both alternations and constructions (their terms: phyla and operators) must exist in the grammar, no alternations ever exist in syntagms. In G I felt the necessity to create these cells for two uses: properties and partial syntagms.

First, it seemed important to be able to place properties on alternations because they might intrinsically belong there. If a statement is to perform a sort, a comment property to that effect should be attached to the statement cell, no matter what particular type of statement is attached to implement the sort. Then, the implementation can be inserted or replaced without disturbing the property.

Second, even without properties, partial syntagms may profitably use alternations in this case: where one alternative is itself an alternation. In this case the appearance of an alternation in a partial syntagm conveys information about the possible types of its daughters which could not possibly be inferred if its appearance were forbidden.

Mentor is a collection of tools in an integrated environment. Included are Metal, which specifies the grammars, Typol, a context-sensitive constraint interpreter, and Mentol, the syntagm manipulator. G’s closest counterpart is Mentol, but Mentol is a weaker language, a command language with a simple pattern-matching facility.

We are told that tree traversal may involve metavariable search and instantiation, but many of the details are elided. Since we are told [DKLM84]
that Mentol can search for

```latex
if x=0 then $\text{MV1}$ else $\text{MV2}$
```

we can guess that the search command can proceed without explicitly being
told the root operator of the tree being searched for. But, there is the problem
of ambiguity.

Suppose the search schema is a bare $\text{MV1}$. This could match the whole
program tree or any subtree. What is Mentor to do? Mentol addresses this
problem to a certain extent by providing a facility for qualifying a metavariable
with a phylum name.

But, there is more difference between Mentol and G than the rules for
writing search patterns. Mentol is meant to be one tool in a set of tools. It
may be used whenever it happens to suit one’s purpose. G is meant as the only
user interface for syntagm manipulation. As such it contains interfaces in turn
to other tools. Parsals allow the user to interface to a separate tool for pattern
creation and matching (cf. §2.6.3.1). Parsals, again, are used to reach the tool
for grammar declaration, the analog of Metal (cf. §2.6.1). No implicit interface
verifies context-sensitive constraints (corresponding to Typol). When this is
needed, routines must be invoked explicitly, as in §4.2.3.

4.3.1.2 Centaur

Centaur [BCD*88] grew from Mentor under the same group at INRIA. Al-
though most don’t concern us, many changes were made: addition of persistent
storage and a graphical user interface, change of the implementation language
to Lisp with good access to a Prolog workspace, employment of object-oriented
methodology, availability of iterators. Mentol was removed completely as a
language and an abstract machine took over its function: the Virtual Tree
Processor.
CHAPTER 4. EVALUATION

Reaffirmed are the concepts: visible grammar, (untyped?) gates, and annotations (properties). Annotations may be created dynamically and attached to anything, as in G and Smalltalk. They do not seem to have hit on the idea that a grammar might be represented as a syntagm, nor of dynamic fixation for general syntagms, nor of enhanced types for gates.

They have an interesting idea about annotations, namely that a particular marker (their term: decor) may have not only a grammar associated with its value but a grammar to which it may be attached. If I were to include this in G, I would use broad types here as well, so that the marker might be grammar-wide or attached to just one type.

They have begun to formalize the idea of L-value (their term: context). Contexts are more general than in G: A context may be a sublist of daughters of a list cell. Moreover, it is to be extended to arbitrary sets of individual daughters.

They remark that they have as yet no persistent representation of contexts. This may be no more difficult to achieve than it is in G. However, if a context is to behave in highly sophisticated ways (e.g. if a sublist context is to grow when an insertion is made within the sublist), specification of these objects may be a thorny problem. I do not favor increasing the complexity of L-values in G.

One remark particularly has caught my interest (p. 16):

But in fact, the choice of a proper representation for structured objects depends on what manipulations have to be performed on them and we believe that the need for multiple representations is inherent to symbolic processing.

Although I can see the need to which they refer, I cannot imagine a way conveniently to re-represent a syntagm so that its properties adhere to it. If a syntagm is mapped to a non-syntagm, altered, and mapped back, properties,
as well as gates that hold its cells, must surely be lost. It seems an extreme statement for them to make.

Yet, I cannot dismiss it completely, either. Their years of experience with Mentor and now Centaur give weight to opinions they express on such practical matters.

4.3.1.3 Gramps

Gramps [Cl84] is the immediate predecessor of G and a strong source of inspiration. It is a generator for a subroutine library to be called during execution of a host language. The subroutines work on a space of syntagms composed of cells, with operations similar to G's: constructors, selectors, type, parent, sublist, insert, delete.

However, it is the generator that has the real knowledge of the grammar. Many single-purpose subroutines, e.g. MakeIfStatement, are needed because of the lack of a visible grammar. Steps were taken later [Cam86] to make the grammar visible in the host language, but dynamic construction of grammar remained unavailable.

The advantage of using a subroutine library in an existing language is teachability. We do not have to learn an entire new language (assuming that we know the host language already) but only the structure of Gramps syntagms and the operations provided.

The disadvantage of this approach is the distance between the compiler and the types, the grammar of the target language. Suppose we wish to write a literal syntagm (a parsal in G). We may write a string literal and have it parsed at execution, but the compiler can never parse it or report errors in it. Implicit assignments are impossible unless the host language has multiple assignment, and then they are still awkward.

Partial syntagms, necessary for pattern matching, are against the Gramps design philosophy, in which all syntagms must be total.
Gramps extracts the abstract grammar from the concrete grammar table. This causes some inflexibility in the abstract grammar, since the concrete one must unambiguously parse the language. There are cases where extra nodes appear in the abstract tree and where there are what we might call errors. These arise when the language being parsed is truly ambiguous with respect to its CFG. There are examples of this in Pascal, C, Fortran, and probably others. Here is one example. In a Pascal expression it is impossible to distinguish between a simple variable and a zero-argument function call without referring to the context. Gramps must permanently treat all such cases as variables. There is no way of correcting the type of the syntagm once the true nature of the identifier is known.

There are no properties in Gramps. There is no fixation. There are no true gates, but leaf cells may have values in the host language (integers, strings, etc.). Program variables from the host language are like idents, since they have values and are never mothers.

4.3.2 Declarative Languages

If the world of general-purpose programming languages may be divided into imperative and declarative languages, it is no surprise that syntagm processors may be also. In this section we look at two programs, TAMPR and the Cornell Program Synthesizer, which use the declarative style. Neither of them is meant for all types of applications. TAMPR is specifically designed for batch program transformations, while Cornell is an editor, accepting and modifying source code during an interactive session.

4.3.2.1 TAMPR

TAMPR's language, therefore specifies program transformations. When the problem at hand is to work simple transformations within a single language
(e.g. Fortran), a declarative language becomes particularly attractive.

The reason is that the transformation may often be expressed very simply in a schema, using only a cursory knowledge of the grammar underlying the target language. (We have taken advantage of schemata to illuminate transformations several times in this monograph, most recently on p. 152.) Nor should we overlook the problem of demonstrating correctness, which is easier in the more confined realm of the schema.

TAMPR [BM84, Boy89] is an important example of the declarative style since it was written early and is still in use. It has probably seen the most useful service of all the transformation programs.

Recall the example of ch. 1:

\[
\text{do statement while (0)}; \\
\implies \text{statement}
\]

TAMPR expresses this transformation simply:

\[
\langle \text{statement} \rangle \\
\text{.sd.} \\
\text{do \langle statement\rangle"i" while (0);} \\
\implies \\
\text{\langle statement\rangle"1"} \\
\text{.sc.}
\]

Of course, TAMPR, must apply this transformation as often as necessary to an input syntagm until it is completely transformed. This is the place at which TAMPR's simplicity runs out.

TAMPR has quite a variety of different orders in which it may apply the available transformations. It makes several bottom-up (post-order) traversals of the input syntagm. A particular transformation may be applied as a member of a set, so that it applies whenever it is applicable, or from a list, so that
it applies only if no previous element applies, or from a sequence, so that it applies after the previous elements and before the subsequent ones, whether they were applicable or not. It may apply as often as possible or once, in the context of another transformation [Boy70]. Although this implicit control structure gives TAMPR the power of a Turing machine, difficult transformations, it seems to me, are more clearly expressed using explicit control structures in an imperative language.

One would expect TAMPR to have the most difficulty in context-sensitive situations. For purposes of demonstration, Boyle tackles the problem of discovering identifier denotations in the context of nested declarations (cf. §4.2.3). The recommended solution [Boy70, §3.1] is simply to transform a program so that every appearance of each identifier is adorned with its denotation. The grammar of the target language must be extended for this, both because there are no properties in TAMPR and because each appearance must be marked as the transformation is applied. The transformation applies only to unmarked identifiers, so marking guarantees termination.

When challenged [Cl84] to show a method for another context-sensitive calculation, namely, to show that a function is free of side-effects, TAMPR [BM84, §IV.D] also proposes the same approach, i.e. a series of transformations to adorn syntagms with various embedded marks.

In both of the above cases the use of properties or a property-like facility would have simplified the computation. Besides the fact that marks and attached descriptions are more perspicuous as properties, the use of properties avoids extensions to the grammar and eliminates the clean-up phase of a TAMPR transformation, the removal of the various embedded marks so that the finished program can be emitted.

In G I have tried to take the best of the imperative and declarative worlds, parsals and matching but with explicit control structures. In G we might write
if (  
    statement["do 'statement[a] while (0);"]  
  // s)  
  s#^ = a;

to work the example transformation. TAMPR is evidently more clear and compact, while G must explicitly invoke the match and assignment operations. Moreover, not shown is the machinery for tree traversal, which must be written in G, but not in TAMPR.

4.3.2.2 The Cornell Synthesizer Generator

Cornell [RT89] offers a generator that will produce an editor for any language defined by an attribute grammar [Kn88]. This editor is called a Program Synthesizer because it allows a program to be created by specifying its syntagmatic structure (as opposed to its concrete syntax).

However, the user's method of directing the editor is not of interest here. What is is the nature of attribute grammar itself. It attaches to each node in a syntagm a set of attributes, which are much like properties. However, all and only the attributes allowed on a node of a given type must be present. Rules are given which define the values of attributes in terms of those on neighboring nodes. The main task of the Synthesizer is to maintain attribute consistency while the syntagm is constructed or altered.

For the Synthesizer the choice of a declarative specification language, in particular attribute grammar, is a good one. (A recent improvement to this choice is given in [TC90].) However, the rigid behavior of the Synthesizer that it always works with a pre-defined grammar, a pre-defined set of attributes, a pre-defined method for evaluating them, is what makes this choice good. Although it too has the power of a Turing machine, it is not suitable for general programming; it is an editor. Rather than competing with G, it is an
ideal candidate for an application written in G.

4.3.3 Mixed Languages

It has seemed to some people that neither a purely declarative nor a purely imperative language is best. This happened especially to those writing compiler generators, who saw that the framework for the compiler would be a CFG, which would be declared, but that semantic actions, code generation and so forth, would be more conveniently expressed imperatively.

4.3.3.1 YACC

The compiler generator YACC [Joh75] is an example of this. The compiler it produces is driven by a parser, which is derived from the input CFG. This parser, upon recognizing the various nTs represented by the input string, invokes the semantic actions which had been given to YACC in the form of imperative (C) code fragments.

Since YACC is widely known, it is no surprise that it is an obvious choice for program transformation problems as well. How well does it do?

Well, that depends on the nature of the problem. Because a fragment only gets control when its associated nT is found, the problems that are handled best are the ones in which a localized view of the input is sufficient or almost sufficient.

Consider an ordinary compiler for an Algol-like language. It must build up and retain an environment of declarations for the interpretation of the identifiers that appear in the statements it compiles. Nested blocks form a tree structure, so the environments do too. But, as the compiler walks through the code, it needs to see only a path, the path from the most recent block, up through the tree to the root, the global block, or rather the environments thereof. Retaining these data is nicely handled with a stack, and thus the local
view of the input is fairly easy to maintain. Undoubtedly, ease of compilation strongly influenced the design of these languages.

When a purely local or a path view of the code is good enough a pseudo-compiler is a good transformation tool. It works fairly well, surprisingly well, for the examples of §4.2. For the insertion of Trace Statements the program mostly goes forward through the code, transforming as it goes. Only at the very end, where it adds declarations for the new variables to the global declaration list, does it do something which must necessarily be done out of order. Yet, this is enough to make YACC awkward for this problem. If we must use YACC, we could proceed in one of two ways: We could use two passes, storing intermediate data in disk files. The variable names would be in a separate file so that they would be available to the second pass at the beginning, rather than the end. Or, we could build syntagms, retaining the whole program in memory until the variables could be accumulated and inserted at the proper place.

In the case of the Pointer Guard (§4.2.3) this analysis makes YACC look deceptively good. It can guard the pointers easily in one pass. However, the type records created in this example are of a kind which is generally useful, both in this transformation and in the kind that must go back and forth over the syntagm in order to compute a transformation (e.g. in-line coding a subroutine). Type records for the whole tree are present during the whole time, available whenever needed. So, if we evaluate this example in perspective we get a stronger sense of the weakness of the pseudo-compiler.

4.3.3.2 Others

YACC builds a table for a fast table-driven parser and attaches semantic actions in C.

By contrast Rigal [Aug90] combines pattern-based parsing and semantic actions into a single language (or perhaps two compatible sub-languages).
The dominant control structure is the pattern-matching process. A complete program consists of many small patterns, and since each has its own name and scope, it tends to resemble a Prolog program in structure.

Parsing is necessarily by recursive descent. Imperative code may be interleaved with matching as the pattern is interpreted. Little emphasis is placed on the syntagmatic view of the code. Although syntags may be built as intermediate structures, input and output are of character strings.

The data structures available are (named) atoms and lists and an unordered record, that is, a collection of objects, each tagged with a field name. Each pattern matches at only one level, a single list or record. To match nested levels one must use nested patterns. There are no types. There are no properties. Since patterns are control structures, they are second-class citizens, not visible in the data space.

The language of \[HC90\] also relies mainly on pattern-matching. However, their purpose is program transformation, especially peep-hole optimization. The matching language is stronger than Rigal, containing matching of repeated variable elements and true backtracking. However, in place of the imperative part is an escape into Prolog.

Here the view of code is purely linear. No syntags are ever built up. There are no types or properties. Although the application of individual transformations is described in this article, no mechanism is described for higher-level control: selecting and ordering transformations.

4.3.4 Object-Oriented Languages

In this section we have been examining special-purpose languages, specifically those whose purpose is syntagm manipulation. Often, however, a user who has this purpose will turn to a general tool, a familiar tool, rather than go to the trouble of learning to use a special-purpose language. A frequent choice
is one of the object-oriented languages, which we discuss here. In §4.3.3 we looked at the compiler generator as it serves this purpose.

4.3.4.1 M3AST

[Jon90] discusses the use of Modula-3 for syntagm manipulation, but it focuses attention on a particular representation of the syntagm, a discipline called M3AST (abstract syntax tree), which we contrast with the G data space.

The M3AST discipline defines object classes and subclasses based on the target (Modula-3) grammar. Alternation objects do not exist. Instead, each alternative (if-statement) is a subclass of the class (statement) required for a given position. The absence of alternations places the usual restrictions on partial syntagms and the placement of properties. Construction objects contain their component objects in typed attributes (properties). The type specifies either an alternation (class) or construction (class) or a basic type (those reached through gates in G).

In order to achieve modularity, sets of attributes are grouped together into views. As usual, there is no motherhood and no fixation. The only distinction between value and properties is in whether a given attribute is in the value view.

Attributes not in view are invisible. Attributes and views may be added freely without affecting the existing ones.

This achieves very much the same effect as G properties. A property is invisible except to the owner of its marker, yet it resides with the syntagm and requires no effort by the owner to insure its continued independent existence. Static typing is available for properties.

What M3AST has that G lacks is a formal interface specification listing all the attributes (markers) that belong to a particular view (application).
4.3.4.2 RPDE

RPDE is a multi-lingual development environment that allows us to edit and display objects of different languages simultaneously without visibly switching modes. [OH90] is a particularly stimulating exploration of extensions to the object-oriented model, those extensions that facilitate program development in the large, and multi-lingual program development. The article also reports the experience of integrating isolated development efforts back into a single version, an experience that never fails to be educational.

Two of the ideas appealed to me especially as possibly useful for incorporation into G: *structure-bound messages* and *roles*.

Structure-bound messages, translated into G jargon, are actions given to the current cell for execution at a distant cell. By examining the current cell and the message the mechanism can tell whether the current cell is the intended target, and, if not, where to look next. There is motherhood in RPDE, so the search for the intended target may go up or down the tree. An implementation in G would at present make use of visible types and properties, but if this idea proves important special operators could be designed to make it convenient.

Roles are generalized types. A cell may have only one type, but it may have several roles. These tell whether various operations apply to it, whether the cell can play a certain role. In G this seems most useful in abstract situations, that is, where an atom with private properties is to be passed to some function. Of course, the ultimate usefulness of this idea is unknown, but some loosening of the notion of data type has been beneficial in RPDE, so it is good to think about.
4.3.5 Specific Features

Here I wish to mention a few articles whose main thrust was not syntactic manipulation (although it was often there in the background) but some specific feature of the machinery.

4.3.5.1 On Parsals

[APS88] have proposed an extension to the language ML [HMT88]. They wish to define grammars (target languages) by correspondence with ML data structures (in the same spirit as §2.2) and to be able to write literals in the new languages in their programs. To do this, they have invented something very like the parsal, even to using square brackets and anti-quotes. The authors acknowledge their debt to the original (LCF) ML [GMW79], which allows literals (with anti-quotes) from an embedded target language (PPLAMBDA) using a different syntax.

Since ML is a more conventional language than G, their parsals are more limited in power, i.e. because they must be parsable by the compiler. This forces on them several decisions which they can’t properly make. At the lexical level, for example, they must choose a way to handle white space. (Multiple blanks in a parsal are always condensed to a single blank; zero- and one-blanks are never equivalent.) While this is a pretty good way, any hard-wired restriction on the visual format of a parsal is regrettable. Another choice they make is one-character lexemes.

As a solution to their lexical problems they actually consider user-written subroutines to the compiler.

They are also bothered by the difficulty of using one grammar for both concrete and abstract purposes. Like C184, they have an unnecessary singleton
production when parentheses are found in an expression and unnecessary productions defining operator precedence and associativity. Their proposed solution is a collection of more-or-less *ad hoc* modifiers to the abstract grammar, which implicitly describes the concrete grammar without stating it. While this is not a bad solution, given an existing language with an existing philosophy, it is worth considering, now that the door is open to user subroutines (at least a crack), the use of one for parsing as well.

### 4.3.5.2 On Fixation

Several designers of grammar-based editors have chosen to use visible or almost visible (gray?) grammars. In situations like language prototyping, grammar fixation is fairly undesirable, so an exploration of alternatives is worthwhile.

SbyS [Min90] allows a syntagm to be visible to one activation of the editor while the grammar is visible to another. A change to the grammar, naturally, may invalidate many syntagms. At present SbyS requires manual intervention to repair such syntagms.

TransformGen [GKS87] is a tool in the Gandalf [Not85] environment. It starts with an editor for grammars with a special facility which monitors and records changes made to a grammar. As simple changes are made it updates a table describing the cumulative change. For changes beyond a certain difficulty it asks the user for help, which may take the form of manual changes to the table or an imperative routine. The result, consisting of the final table and any hand-written routines, becomes the syntagm transformer between the associated versions of the grammar. Currently, they apply the transformer only as out-of-date syntagms are encountered, using version stamps on the syntagms and grammar to determine this.
4.3.5.3 On CFG as Type

Q language [Cua83] has explored the possibility of using CFG as the sole method of type declaration. Q is, however, a general-purpose programming language, and the use of Q for syntagm processing is not examined. Type is not visible in the data space.

Imperative code may be part of a type declaration in order to restrict the set of allowed values of that type. However, this is not useful for enforcing context-sensitive constraints since the code is only executed when the entire value is assigned. That is, it has the ability to verify a whole program for type correctness but not the ability to verify (easily) an incremental change.

4.4 Implementation

_Screw the implementation!_

Adin Falkoff

To verify the implementability of the G data space, an implementation was written in Lisp [MAE'62]. Lisp was chosen because it is a symbolic computing language and because it has implicit memory allocation and garbage collection, which were then inherited by the G environment without explicit coding.

An interpreter for an intermediate language was written in order to simulate the data space and the operations over it. All the possible evaluations are available, including exceptions, but no attempt was made to simulate separate compilation and symbol resolution.

Writing this program refined my ideas about G and its eventual implementation, especially the nature of assignment coercion and the role of idents. However, the program proved unsuitable for a practical evaluation because it ran slowly. It was impossible to get a "feel" for a running program. Factorial
6 was calculated as a bench-mark, using a recursive algorithm. As written, each activation involved 10 operations (if, equal, select, else, return, times, call, subtract, select, select), except for the base case, which was 4.

The total time (as reported by a Lisp monitor function) for the 54 operations was 227 ms on an IBM 3081. 1870 Lisp cons cells were used. This gives an average of 4.2 ms per operation.

Even though interesting calculations can be done at this low speed, it is impossible to use this program for language design, for this reason: The purpose of an implementation is to stimulate programming in people who have problems to solve. Their experience as perceived forms the basis for a true human factors analysis of the language, which heretofore has been biased by my prejudices. An implementation that runs very slowly would skew users’ perceptions of the language, giving rise to false evaluations. That is, there would be a strong inclination to use only those features of the language that run fastest because of the time penalty attached to the others. This phenomenon was well known in the APL community, where the popular implementation had an overhead associated with the execution of each line. Users went to great lengths to compress the entire program onto one line, at the expense of readability.

We will now see the basic cell structure and a sketch of the intermediate language which was interpreted. A detailed description can be found in appendix B. In what follows, a working knowledge of Lisp is assumed.

Each cell is a separate Lisp atom. Most are created by GENSYM, although a few have ordinary names on the OBLIST. The attributes are stored in a list as a property. The list has 6 elements.

\begin{align*}
\text{cell} / \text{ident} & \text{ One bit flag. If this is an ident the list is only 4 elements.} \\
\text{var} / \text{fixed} & \text{ One bit flag. Is this cell fixed?}
\end{align*}
value

This is always a list, usually a list of G cells (Lisp atoms), but, in case this is a gate to int, char, bool, the single element is a Lisp integer, character atom, or Boolean atom.

type

A single G cell.

mother

A single G cell.

properties

A list, alternating markers (G cells) and property values (G idents).

If the object is an ident, then type x should be understood to mean "gate to x." If this is a G list cell then the value is a list of length at least length (§3.9.1.2). Any of its elements may be none, which is represented by a distinguished Lisp atom u-none. If an element is inserted past the current end of the list, it is extended with elements of value none. No effort is made to trim trailing none's from a list. The value of cells which are not of list kind is kept at its proper length at all times.

Properties are also created as needed. They are never destroyed.

Idents which are program identifiers are stored on atoms with the same name as the identifier. The (hypothetical) compiler incorporates a scope name into the identifier name so that identifiers from different scopes do not collide. There is no explicit execution stack. Instead, new activations of a given identifier are simply pushed on the property list of the Lisp atom.

Expressions are modeled as lists with the operator first and the operands following. The operands may be nested lists. The result of an expression may be an L-value, an R-value, or an exception. Care was taken at every stage to stop evaluation as soon as an exception was generated. The exception becomes the result of the statement containing the expression.

The statements of a function are flattened into a single list, and the flow
of control is altered by the equivalent of conditional go-to.

Provision must also be made in statements for exceptions. In any statement which may contain a nested statement, an extra go-to label is included. If a search for an exception catcher is in progress and this statement is encountered, none of the statements in the nested statement is eligible to be the catcher. Therefore, the search follows the go-to rather than proceeding sequentially. This is always a forward go-to, so the search will never loop.
Chapter 5

Conclusion

As coroner, I must aver
I've thoroughly examined her.
Not only is she really dead.
She's really most sincerely dead.

"The Wizard of Oz"
(dir.) Victor Fleming

This chapter begins with a retrospective view of the design of G. We mention only the most important decisions and their impact on the design as a whole. Then we look at the prospect for future development of the language.

5.1 Retrospect

The main contribution of this work to the art of computation is its exploration of the idea that its data space might be a space of assignable cells. I have labored to make this space uniform, and my failures have been just as instructive as my successes.

The idea that each cell must have a type led to the need for none, a pure value, an object with no type, to be the value of a newly created cell. I might
have created other values, but I tried always to do only the minimum that was needed. This gave rise to infinite lists, since the value found “beyond the end” of a list must be the same as the one in an “empty” interior location. This seems a good choice, since it eliminates the need for lengthening primitives; all list operations can be expressed purely in terms of assignment.

The idea of an ident evolved rather slowly. It came only after I had decided what the denotation of an L-value must be (a cell with an integer) and realized that a program identifier customarily was one. It is not an x but a holder for an x. Two things confirmed the idea of a distinct society of idents: that properties (property holders) must also be idents and that the meaning of the type of an ident was different from that of a cell.

In trying to decide whether to link cells doubly or singly, it soon became clear that both choices were needed. Motherhood as a concept was formulated, and gates were specifically excepted from it. Really, this only confirms the work of [DKLM84].

They also first thought to use properties with cells. However, I have improved their idea with Lispesque property markers. These allow information to be hidden on cells, thus creating a mechanism for true data abstraction. Both property markers and gates are native cells in the data space. They are distinguished from other cells only by their type.

Solely for convenience in the implementation, I have restricted certain cells (integers, characters, Booleans) to having no properties.

The treatment of type is unique in G. Types are always visible and constructable. They, too, are native elements of the data space, distinguished from others only by their type.

The wish to guarantee the consistency of the data space led me to create dynamic fixation. When extended to all data, dynamic fixation weakens the objections many have to cell spaces. Now, when desired, the value of a cell or ident may be fixed, either for part or all of its activation. This is useful for
proofs of correctness, as documentation, and as a debugging tool.

When applied to types during compilation, fixation allows the compiler to apply the same checks to expressions and assignments as are applied in strongly typed languages. In these cases an important optimization is possible, since no checks are required in the compiled code.

I found, speaking very roughly, that I could stop the design here. There was enough machinery to produce all the effects required. For instance, the viability of a type syntagm for producing valid cells could be guaranteed by a program (i.e. not a primitive) which would examine the syntagm and mark it with a private property. Several interesting models for lexemes were possible, most notably u-strings, which could be implemented by a program, using data abstraction again via private property. A relationship between two grammars (abstract and concrete) could be established and elaborated using properties, so that a parser, a program, could translate between them. Having found that the remaining problems were programming problems in G, the design of G could stop.

The decision to move one particular thing out of G deserves special mention: parsals. Without exception, at least in truly compiled languages, the compiler evaluates literals. If required to do this in G, I would have had to define a notation for grammar (no great problem) and for the relationship between abstract and concrete grammars, an area where no known method seems to be clearly superior. (Compare, for example, [CI84, §2], [KLMM83, §1.5, 1.6] and [RT85, §2.7].) Since no good method is known, I am forced to omit it. Yet, I cannot omit from the language the power to declare types. Thus, I am forced to run a library (i.e. user-supplied) subroutine during compilation. By this one decision I gain the power to take advantage of, make immediate use of, any advances in grammar notation that should occur, and the power to parse parsals having any form whatsoever, any mixture of abstract and concrete syntax and other items, such as flags and markers relevant
to the various kinds of pattern-matching. This unusual step seems justified by the extent of the power gained and the benefits that derive from it.

The danger in running a subroutine during compilation is that the compiler's correctness and modularity may be compromised. By using well tested subroutines, this danger may be reduced, but not eliminated.

5.2 Prospect

This work sets out to develop a new special-purpose programming language, the need for which is inferred from the myriad programs for syntagm processing that are written in general-purpose languages. Although I set out to develop G, I have by no means perfected it. This is a beginning, a good beginning, I hope, but only a beginning.

The course of development of a language, at least this language, may perhaps be divided into six stages, only one of which is complete and set forth in this thesis:

1. Initial language design, with data space, expressions, control structures

2. Pilot implementation of the compiler and loader

3. Subroutine library, with a small selection of ordinary parsers, parsal parsers, and pattern matchers

4. Field trial, with evaluation and criticism from programmers

5. Refinement of language and library

6. Optimization of compiler and generated code

The thrust of the initial design is to define the scope of the language. One philosophy motivates this: that G should be capable of expressing the
most general kind of syntagm manipulations. This brings about dynamically constructed types and, in my opinion, an imperative language with separate compilation. The design of the data space is the central part of the design, so it was particularly gratifying to be able to include dynamic types in the space as a restriction (rather than a generalization) of ordinary syntagms.

In stage 2, in addition to the pedestrian work of writing the compiler and loader, we must refine the design. The method of storage management must be chosen. I incline toward the system common in Lisp implementations: the store divided into blocks of a uniform size which are not moved during execution. But, whether this is practical remains to be seen.

Also, this is the point at which the reporting of errors must be decided and specified. Most type errors can and should be reported by the compiler. Likewise, wrong namelist selectors ("dot-selectors" §3.9.5.1) can be reported. The set of errors which a standard compiler may and must actually detect will be precisely defined in this stage.

In stage 3 we will have some very interesting design problems. Devising a convention for denoting the relationship between an arbitrary abstract and concrete grammar is the one I look forward to most, both because it will be useful outside this context and because it requires a balance between precision, completeness, and convenient, perspicuous notation. As mentioned in §2.6.3.1 there are also some nice problems in the design of parsals for pattern matching.

Stages 4, 5, and 6 are to be repeated freely until the design is validated and the users are satisfied. All the design activity of the first 3 stages is continued and completed as a result of user demand and experience.
Appendix A

Code for the Pointer Guard

A.1 An Abstract C Grammar

Translation_unit ::= External_decl+
External_decl ::= Function_def | Declaration
Function_def ::= Decl_spec* Declarator Compound_stmt
Declaration ::= Decl_spec+ Init_declarator*
Decl_spec ::= Sc_spec
 | Type_spec
 | Type_qual
Sc_spec ::= K_auto | K_register
 | K_static | K_extern | K_typedef
K_auto ::= :
K_register ::= :
K_static ::= /* These atoms represent keywords */
K_extern ::= :
K_typedef ::= :
Type_spec ::= K_void | K_char
 | K_short | K_int | K_long | K_float
 | K_double | K_signed | K_unsigned
 | Struct_spec
 | Union_spec
 | Enum_spec
 | Ident /* Typedef'd name */
K_void ::= :
K_char ::= :
K_short ::= :
APPENDIX A.  CODE FOR THE POINTER GUARD

\[
\begin{align*}
K_{\text{int}} & ::= \\
K_{\text{long}} & ::= \\
K_{\text{float}} & ::= \\
K_{\text{double}} & ::= \\
K_{\text{signed}} & ::= \\
K_{\text{unsigned}} & ::= \\
\text{Type\_qual} & ::= K_{\text{const}} \mid K_{\text{volatile}} \\
K_{\text{const}} & ::= \\
K_{\text{volatile}} & ::= \\
\text{Struct\_spec} & ::= [\text{Ident}] \text{ Struct\_decl*} \\
\text{Union\_spec} & ::= [\text{Ident}] \text{ Struct\_decl*} \\
\text{Init\_declarator} & ::= \text{Declarator} [\text{Initializer}] \\
\text{Struct\_decl} & ::= \text{Spec\_qual} + \text{Struct\_declarator*} \\
\text{Spec\_qual} & ::= \text{Type\_spec} \mid \text{Type\_qual} \\
\text{Struct\_declarator} & ::= [\text{Declarator}] [\text{Const\_expr}] /* Const => bit field */ \\
\text{Enum\_spec} & ::= [\text{Ident}] \text{ Enumerator*} \\
\text{Enumerator} & ::= \text{Ident} [\text{Const\_expr}] \\
/* \text{Choice: int \*a[15]} \\
\text{stored as (Array 15) (Pointer)} \\
i.e. \text{ array of pointers to integers} */ \\
\text{Declarator} & ::= \text{Ident} \text{ Abs\_declarator} \\
\text{Abs\_declarator} & ::= \text{Type\_modifier*} \\
\text{Type\_modifier} & ::= \text{Pointer} \mid \text{Array} \mid \text{Func} \mid \text{V\_func} \\
\text{Pointer} & ::= \text{Type\_qual*} \\
\text{Array} & ::= [\text{Const\_expr}] \\
\text{Func} & ::= \text{Parm\_decl*} \\
\text{V\_func} & ::= \text{Parm\_decl+} \\
\text{Parm\_decl} & ::= \text{Parm\_full\_decl} \mid \text{Parm\_abs\_decl} \\
\text{Parm\_full\_decl} & ::= \text{Decl\_spec} + \text{Declarator} \\
\text{Parm\_abs\_decl} & ::= \text{Decl\_spec} + \text{Abs\_declarator} \\
\text{Initializer} & ::= \text{Expr} \mid \text{Initializer*} \\
\text{Type\_name} & ::= \text{Spec\_qual} + \text{Abs\_declarator} \\
\text{Const\_expr} & ::= \text{Expr} \\
\text{Stmt} & ::= \text{Labeled\_stmt} \\
\text{Optiona\_expr} \\
\text{Compound\_stmt} \\
\text{If\_stmt} \\
\text{If\_else\_stmt}
\end{align*}
\]
| Switch_stmt |
| While_stmt |
| Do_stmt |
| For_stmt |
| Goto_stmt |
| Continue_stmt |
| Break_stmt |
| Return_stmt |

Optional_expr ::= [Expr]
Labeled_stmt ::= Ident Stmt
                | Const_expr Stmt /* case x: */
                | Default_stmt /* default: */
Default_stmt ::= Stmt
Compound_stmt ::= Declaration* Stmts
Stmts ::= Stmt*
If_stmt ::= Expr Stmt
If_else_stmt ::= Expr Stmt Stmt
Switch_stmt ::= Expr Stmt
While_stmt ::= Expr Stmt
Do_stmt ::= Stmt Expr
For_stmt ::= Optional_expr Optional_expr Optional_expr Stmt
Goto_stmt ::= Ident
Continue_stmt ::= :
Break_stmt ::= :
Return_stmt ::= Optional_expr

Expr ::= Comma_e /* , */
        | Asg_e /* = */
        | T_asg_e /* *= */
        | D_asg_e /* /= */
        | M_asg_e /* %= */
        | A_asg_e /* += */
        | S_asg_e /* -= */
        | L_asg_e /* <<= */
        | R_asg_e /* >>= */
        | N_asg_e /* & */
        | X_asg_e /* *= */
        | O_asg_e /* |= */
        | Cond_e /* ? : */
        | Or_e /* || */
        | And_e /* && */
        | O_e /* | */
        | X_e /* ^ */
        | N_e /* & */
        | Eq_e /* == */
APPENDIX A. CODE FOR THE POINTER GUARD

```plaintext
| Ne_e     | /* != */ |
| Lt_e     | /* < */  |
| Gt_e     | /* > */  |
| Le_e     | /* <= */ |
| Ge_e     | /* >= */ |
| L_e      | /* << */ |
| R_e      | /* >> */ |
| A_e      | /* + */  |
| S_e      | /* - */  |
| T_e      | /* * */  |
| D_e      | /* / */  |
| M_e      | /* % */  |
| Cast_e   | /* (t) */|
| Inc_e    | /* ++e */|
| Pinc_e   | /* e++ */|
| Dec_e    | /* --e */|
| Pdec_e   | /* e-- */|
| Adr_e    | /* &e */ |
| Ptr_e    | /* *e */ |
| Up_e     | /* +e */ |
| Ua_e     | /* -e */ |
| Ux_e     | /* ^e */ |
| Not_e    | /* !e */ |
| Size_e   | /* sizeof(e) */ |
| Size_t   | /* sizeof(type) */ |
| Array_e  | /* a[b] */ |
| Fun_e    | /* f(e) */ |
| Field_e  | /* e.n */ |
| Follow_e | /* e->n */ |
| Ident    |
| String   |
| Int      |
| Char     |
| Float    |

Comma_e ::= Expr Expr
Asg_e ::= Expr Expr
T_asg_e ::= Expr Expr
D_asg_e ::= Expr Expr
M_asg_e ::= Expr Expr
A_asg_e ::= Expr Expr
S_asg_e ::= Expr Expr
L_asg_e ::= Expr Expr
R_asg_e ::= Expr Expr
M_asg_e ::= Expr Expr
```
APPENDIX A. CODE FOR THE POINTER GUARD

\[
\begin{align*}
  X_{\text{asg}} & \quad ::= \text{Expr} \ \text{Expr} \\
  0_{\text{asg}} & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{Cond} & \quad ::= \text{Expr} \ \text{Expr} \ \text{Expr} \\
  \text{Or} & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{And} & \quad ::= \text{Expr} \ \text{Expr} \\
  0 & \quad ::= \text{Expr} \ \text{Expr} \\
  X & \quad ::= \text{Expr} \ \text{Expr} \\
  N & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{Eq} & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{Ne} & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{Lt} & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{Gt} & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{Le} & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{Ge} & \quad ::= \text{Expr} \ \text{Expr} \\
  L & \quad ::= \text{Expr} \ \text{Expr} \\
  R & \quad ::= \text{Expr} \ \text{Expr} \\
  A & \quad ::= \text{Expr} \ \text{Expr} \\
  S & \quad ::= \text{Expr} \ \text{Expr} \\
  T & \quad ::= \text{Expr} \ \text{Expr} \\
  D & \quad ::= \text{Expr} \ \text{Expr} \\
  M & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{Cast} & \quad ::= \text{Type\_name} \ \text{Expr} \\
  \text{Inc} & \quad ::= \text{Expr} \\
  \text{Pinc} & \quad ::= \text{Expr} \\
  \text{Dec} & \quad ::= \text{Expr} \\
  \text{Pdec} & \quad ::= \text{Expr} \\
  \text{Adr} & \quad ::= \text{Expr} \\
  \text{Ptr} & \quad ::= \text{Expr} \\
  \text{Up} & \quad ::= \text{Expr} \\
  \text{Um} & \quad ::= \text{Expr} \\
  \text{Ux} & \quad ::= \text{Expr} \\
  \text{Not} & \quad ::= \text{Expr} \\
  \text{Size} & \quad ::= \text{Expr} \\
  \text{Size\_t} & \quad ::= \text{Type\_name} \\
  \text{Array} & \quad ::= \text{Expr} \ \text{Expr} \\
  \text{Fun} & \quad ::= \text{Expr} \ \text{Expr}\* \\
  \text{Field} & \quad ::= \text{Expr} \ \text{Ident} \\
  \text{Follow} & \quad ::= \text{Expr} \ \text{Ident} \\
  \text{Ident} & \quad \rightarrow \ \text{ustring} \\
  \text{String} & \quad \rightarrow \ \text{string} \\
  \text{Int} & \quad \rightarrow \ \text{int} \\
  \text{Char} & \quad \rightarrow \ \text{char} \\
  \text{Float} & \quad \rightarrow \ \text{FloatGram}
\end{align*}
\]
A.2 The Program

extern namelist unqual GramGram@GRule GramGram;
extern namelist unqual GRule CGram;

namelist unqual

GRule ScopeDictionary = [

ScopeDictionary ::= Idm: SMarker
    Tag: SMarker
    SMother
    Ids: SEntries
    Tags: SEntries

SEntries ::= SEntry*
SMarker -> GramProp
SMother -> ScopeDictionary
SEntry -> ustring

"

GRule DclType = [

DclType ::= BaseType B_mods
BaseType ::= 

| B_float |
| B_double |
| B_long_double |
| B_char |
| B_sign_char |
| B_uns_char |
| B_void |
| B_struct |
| B_union |
| B_enum |

B_float ::= 
B_double ::= 
B_int ::= 
B_long_double ::= 
B_long_int ::= 
B_char ::= 
B_short_int ::= 
B_sign_char ::= 
B_uns_int ::= 
B_uns_char ::= 
B_uns_long_int ::= 
B_void ::= 
B_uns_short_int ::= 

B_struct -> CGram@Struct_spec
B_union -> CGram@Union_spec
B_enum -> CGram@Enum_spec
B_mods ::= B_mod*
B_mod -> CGram@Type_modifier
"

fixed GramProp markDclType = cons( GramProp@ptype, DclType);
fixed GramProp markScopeDictionary = cons(GramProp@ptype, ScopeDictionary);
Routine Directory

Here are declarations for all routines used here, in the order in which they appear in the file (except the last 2, which don't appear).

```c
extern void maketypeinfo(CGramQany, ScopeDictionary);
extern void proc_decls(CGramQany decls, ScopeDictionary dict);
extern void proc_declarators(CGramQany declarators, DclType dcltype, ScopeDictionary dict);
extern void single_declarator(Declarator dtor, DclType dcltype, ScopeDictionary dict);
extern DclType proc_spec(CGramQany decl, ScopeDictionary dict);
extern void proc_stmts(Stmts stmts, ScopeDictionary dict);
extern void single_stmt(Stmt stmt, ScopeDictionary dict);
extern void guardpointers(Translation_unit);
extern void find_type_in_tree(ANY, GRule, ANY);
extern void q_array(Array_e);
extern void q_follow(Follow_e);
extern void q_fun(Fun_e);
extern void q_ptr(Ptr_e);
extern DclType expr_type(Expr e);
extern ANY parseinput(GRule);
extern void genoutput(ANY);
```

```c
void main()
{
    Translation_unit tran;

    tran = parseinput(Translation_unit);
    maketypeinfo(tran, none);
    guardpointers(tran);
    genoutput(tran);
}
```
void maketypeinfo( CGramQany scope, ScopeDictionary over_dict)
{
    rval ScopeDictionary dict;
    GramProp marker = cons( GramPropQtype, CGramQIdent),
        tag_marker = cons( GramPropQtype, CGramQType_spec);

    int i;
    ustring uid;
    fix(marker);
    fix(tag_marker);

    /* Make a new scope dictionary, copying
    the names known in the enclosing scope */

    dict.Idm = marker;
    dict.Tagm = tag_marker;
    dict.SMother = over_dict;

    if( over_dict == none ) {
        dict.Ids = gcons(SEntries);    dict.Tags = gcons(SEntries);
    } else {
        dict.Ids = & over_dict.Ids;
        dict.Tags = & over_dict.Tags;
        for( i=1 ; i <= length(dict.Ids); i=i+1 ) {
            uid = dict.Ids#i#;
            uid->marker = uid->(over_dict.Idm#);
        }
        for( i=1 ; i <= length(dict.Tags); i=i+1 ) {
            uid = dict.Tags#i#;
            uid->tag_marker = uid->(over_dict.Tagm#);
        }
    }

    scope->markScopeDictionary = dict;
    switch( type(scope) ) {
    case Translation_unit:
        proc_decls( scope, dict );
        break;
    case Function_def:
        proc_decls( scope.Declarator.Abs_declarator#1#, dict );
            /* The function name is already in over_dict.
        Only the parameter names go into the new dictionary */
        maketypeinfo( scope.Compond_stmt, dict);
            /* Recur for nested scope */
        break;
    }
case Compound_stmt:
    proc_decls( scope#1, dict );
    proc_stmts( scope.Stmts, dict );

void proc_decls( CGramQany decls, ScopeDictionary dict )
{
    DclType dcltype,
    CGramQany decl;
    int i;

    for( i=1 ; i <= length(decls) ; i=i+1 ) {
        decl = decls#i;
        dcltype = proc_spec( decl, dict );
        proc_declarators( decl#2, dcltype, dict);
        if( type(decl) == Function_def )
            maketypeinfo( decl, dict); /* recur */
    }
}

void proc_declarators( CGramQany dtor, DclType dcltype,
                        ScopeDictionary dict)
{

    /* This routine processes declarators, constructing the
     * type-modifiers list for each one. It may be entered
     * in one of 4 situations.
     *
     * 1. Parm dtor may itself be a Declarator cell: the
     *    function name from a function definition or a parameter
     *    name from a function definition.
     * 2. Parm dtor may be a list of Init_declarators
     *    from an ordinary declaration.
     * 3. It may be a list of Struct_declarators from
     *    a structure definition.
     * 4. It may be a list of Enumerators from an Enum
     *    definition.
     */
int i;

if( Declarator == type(dtor) )
    single_declarator( dtor, dcltype, dict );
else
    for( i=1 ; i <= length(dtor) ; i=i+1 )
        switch (type(dtor#i)) {
            case Init_declarator:
                single_declarator( dtor#i#1, dcltype, dict );
                break;
            case Struct_declarator:
                if( Declarator == type(dtor#i#1#))
                    single_declarator( dtor#i#1#, dcltype, dict);
                break;
            case Enumerator:
                single_declarator( cons( Declarator, dtor#i#1,
                                           gcons(Abs.declarator)),
                                dcltype,
                                dict);
                break;
        }

void single_declarator( Declarator dtor, DclType dcltype, ScopeDictionary dict )
{
    GramProp markId = dict.Idm#;
    DclType dt = & dcltype;
    ustring uid = dtor.Ident#;

    dtor.Abs_declarator :> dt.B_modsitl; /* Completes the dcltype for ident */

    if( none == uid->markId)
        dict.Ids#++ = uid;
    uid->markId = dtor.Ident;
    dtor.Ident->markDclType = dt;
    /* No need to process func declarations since variables
go out of scope immediately */
}
DclType proc_spec( CGram@any decl, ScopeDictionary dict )
{
DclType dcltype;
int j;
CGram@any spec, base_type_cell, tcell;
bool long, short, signed, unsigned, typedef;
GRule type_spec;
string uid;
ScopeDictionary suscope;

    /* Note: NO forward structure declarations */
    /* NO old-style functions */

if( type(decl) != Declaration
    && type(decl) != Struct_decl /* Here are all the possible types */
    && type(decl) != Type_name )
{
    decl = decl#;
    if( type(decl) != Declaration
        && type(decl) != Function_def
        && type(decl) != Parm_full_decl )
        signal FAILURE "C syntagm incorrect";
}
typedef = long = short = unsigned = signed = False;
type_spec = K_int; /* integer is the default type */
base_type_cell = none;
for( j=1 ; j <= length(decl#1) ; j=j+1 )
{
    if( Type_spec != type(spec = decl#1#j#) )
    {
        continue;
    }
}
switch (type(spec#))
{
case K_void:
case K_char:
case K_int:
case K_float:
case K_double:
type_spec = type(spec#);
    continue;
case K_short:
    short = True;
    continue;
case K_long:
    long = True;
    continue;
case K_signed:
    signed = True;
continue;
case K_unsigned:
    unsigned = True;
    continue;
case Struct_spec:
case Enum_spec:
case Ident:
    /* i.e. use of typedef'd name */

    type_spec = type(spec#);
    base_type_cell = spec#;
    continue;
}
/* end of reading type-specifiers */
dcltype = DclType[ BaseType[-1] ]; /* construct a prototype DclType record */

switch(type_spec) {
case K_void:
    dcltype.BaseType# = gcons(B_void); break;
case K_float:
    dcltype.BaseType# = gcons(B_float); break;
case K_double:
    dcltype.BaseType# = gcons(long? B_long_double : B_double);
    break;
case K_char:
    dcltype.BaseType# = gcons(signed? B_sign_char :
      unsigned? unsigned? B_uns_char : B_char );
    break;
case K_int:
    dcltype.BaseType# = gcons( unsigned? short? B_uns_short_int :
      long? B_uns_long_int :
      B_uns_int :
      short? B_short_int :
      long? B_long_int :
      B_int );
    break;
case Union_spec:
case Struct_spec:
    dcltype.BaseType# = gcons( type_spec == Struct_spec ?
      B_struct : B_union );

    if( 0 != length(base_type_cell#2) ) {
        /* Found a structure definition */
dcltype.BaseType## = base_type_cell;
if( Ident == type(tcell = base_type_cell##) ) {
    if( none == tcell##->(dict.Tagm#) )
        dict.Tags#++ = tcell#;
    tcell##->(dict.Tagm#) = base_type_cell~;
}

/* Make up a blank scope dictionary */
suscope = gcons(ScopeDictionary);
suscope.Idm = cons(GramProp, cons(GramPropQptype,
    CGramQIdent));
    /* new scope for names */
suscope.Tagm = dict.Tagm#;
    /* old scope for tags */
suscope.SMother = dict;
suscope.Ids = gcons(SEntries); /* Only names and */
suscope.Tags = gcons(SEntries); /* tags from this */
    /* scope are here. No accumulation */
base_type_cell->markScopeDictionary = suscope;

    /* recur */
proc_decls( base_type_cell##2, suscope );
}

} else { /* No content: struct is defined elsewhere */
tcell = base_type_cell#1## -> (dict.Tagm#);
dcltype.BaseType## = tcell#;
}
break;

case Enum_spec:
dcltype.BaseType# = gcons( B_enum );
if( 0 != length(base_type_cell##2) ) {
    dcltype.BaseType## = base_type_cell; /* Definition */
    if( Ident == type(tcell = base_type_cell##1#) ) {
        if( none == tcell##->(dict.Tagm#) )
            dict.Tags#++ = tcell#;
        tcell##->(dict.Tagm#) = base_type_cell~;
    }
}

/* The names in an enumeration become declared as integers */
proc_declarators( base_type_cell#2,
    cons( DclType, gcons(8_int),
          gcons(8_mods)),
    dict);
}

} else {    /* No content: enum is defined elsewhere */
tcell = base_type_cell#1# -> (dict.Tagm#);
dcltype.BaseType## = tcell#;
}
break;

case Ident:    /* This is a reference to a typedef */
dcltype = & base_type_cell# ->(dict.Idm#)->markDclType;
break;    /* It is impossible to choose unambiguously
            the appropriate typedef name, so we settle
            for the underlying type structure. */
}
return dcltype;
}
void proc_stmts( Stmts stmts, ScopeDictionary dict )
{
    int i;

    for( i=1 ; i <= length(stmts) ; i=i+1 )
        single_stmt( stmts#i , dict );
}

void single_stmt( Stmt stmt, ScopeDictionary dict )
{
    switch( type(stmt#) ) {
    case Labeled_stmt:
        single_stmt( stmt#.Stmt, dict );
        break;
    case Optional_expr:
    case Goto_stmt:
    case Continue_stmt:
    case Break_stmt:
    case Return_stmt:
        break;
    case If_stmt:
    case Switch_stmt:
    case While_stmt:
    case Do_stmt:
    case For_stmt:
        single_stmt( stmt#.Stmt, dict );
        break;
    case If_else_stmt:
        single_stmt( stmt#2, dict );
        single_stmt( stmt#3, dict );
        break;
    case Compound_stmt:
        if( length(stmt#1) == 0 ) {
            /* no declarations */
            int i;
            Stmts stmts = stmt#.Stmts;

            for( i=0 ; i <= length(stmts) ; i=i+1 )
                single_stmt( stmts#i , dict );
        } else
            maketypeinfo( stmt#, dict );
        break;
    }
}
void guard_pointers( Translation_unit t )
{
CGram@any tcell;
int i;

for( i=1 ; i <= length(t) ; i=i+1 )
  if( Function_def == type(tcell=t#i#) ) {
    find_type_in_tree( tcell, Array_e, q_array );
    find_type_in_tree( tcell, Follow_e, q_follow );
    find_type_in_tree( tcell, Fun_e, q_fun );
    find_type_in_tree( tcell, Ptr_e, q_ptr );
  }

External_decl[ "void * pointer_guard( void *);" ]
  >: t#1;
}

void find_type_in_tree( ANY t, Grule g, ANY fun )
{
  /* This is post-order/bottom-up traversal. 
     It doesn't get confused by alterations to the 
     tree because after calling fun on a 
     subtree it never looks at that subtree again 
     */

  int i;

  if( t == none )
    return;
  if( Ggate != type(type(t)#) )
    for( i=1 ; i <= length(t) ; i=i+1 )
      find_type_in_tree( t#i, g, fun );
  if( type(t) == g )
    fun(t);
}

void q_array( Array_e e )
{
Expr e1 = e#1,       /* Rewrite A[B] */
e2 = e#2;           /* to *(A+B) */
APPENDIX A. CODE FOR THE POINTER GUARD

```c
void q_follow( Follow_e e )
{
Expr el = e.Expr;       /* Rewrite A->B to (*A).B */
Ident i2 = e.Ident;

e1#* = none;             /* el, e2 become orphans */
e2#* = none;

e#* = Ptr_e[ A_e[- el e2 ]];
}

void q_fun( Fun_e e )
{
    /* A function ptr used as a function name
    is implicitly followed.
    Here we convert this to explicit.
    
    I infer: The only place a func dcl is
    implicitly converted to ptr-to-func
    is in a parameter list. Local dcls
    implicitly go extern. Struct dcls
    are errors.
    */

DclType dt;
Expr   e1 = e#1;

dt = expr_type(e1);
if( type(dt.B.mods#1##) == Pointer ) { 
    e1#* = none;
e1#1 = Ptr_e[ e1 ];
}
}
```
void q_ptr( Ptr_e e )
{
DclType dt;
Type_name type_name;
CGramAny tcell;
GRule g;
Expr e1;
int i;

dt = expr_type(e#);
switch (g = type(dt.BaseType#)) {
    case B_float:
        type_name = Type_name[ "float" ];
        break;
    case B_double:
        type_name = Type_name[ "double" ];
        break;
    case B_long_double:
        type_name = Type_name[ "long double" ];
        break;
    case B_char:
        type_name = Type_name[ "char" ];
        break;
    case B_sign_char:
        type_name = Type_name[ "signed char" ];
        break;
    case B_unsigned_char:
        type_name = Type_name[ "unsigned char" ];
        break;
    case B_unsigned_int:
        type_name = Type_name[ "unsigned int" ];
        break;
    case B_uns_char:
        type_name = Type_name[ "unsigned short int" ];
        break;
    case B_1ong_int:
        type_name = Type_name[ "unsigned long int" ];
        break;
    case B_struct:
    case B_union:
    case B_enum:
        if( Ident != type(tcell = dt.BaseType##i#) )
            return;
        /* If the structure has no tag,
           there's no way to write a cast for it. */
        type_name = Type_name( ( g==B_struct ? "struct" :
                               g==B_union ? "union" :
                                      "enum" )
                   tcell);
        break;
    }
    /* Copy type modifiers */
    for( i=1 ; i <= length(dt.B_mods#i#) ; i=i+1 )
        type_name.Abs_declarator#i = & dt.B_mods#i#;
    (e1 = e#1) = none;
e#1 = Expr[ type_name "pointer_guard( (void *) e1 )" ];
    /* Rewrite *x => * (original-type) pointer_guard((void *) x) */
}
DclType expr_type( Expr e )
{
    /* Specialized for pointer-guard example.
Returns int for all arithmetic types.
*/

DclType dt, dtt;
B_mods b;

switch( type(e#) ) {
default:
    return DclType[ B_int[- ]];
    case Fun_e:
        dt = & expr_type(e##1);
        b = dt.B_mods;
            if( type(b#1##) == Pointer )
                b#1 := ;    /* Throw away "pointer to" if present */
                b#1 := ;    /* Throw away front modifier:
* must be Func or V_func. */
        return dt;
        case Field_e:
            /* This is a structure member reference */
            dt = expr_type( e#.Expr );    /* Must be structure or union */
            return e#.Ident# -> (dt.BaseType##->markScopeDictionary.Idm#)
                ->markDclType;
        case Inc_e:
        case Pinc_e:
        case Dec_e:
            /* Might be pointer type or integer */
        case Pdec_e:
            return expr_type(e##);
        case Ptr_e:
            dt = & expr_type(e##);
            dt.B_mods1 := ;    /* Throw away "pointer to" modifier */
            return dt;
        case Adr_e:
            dt = & expr_type(e##);
            B_mod[ Pointer[-]] := dt.B_mods1;    /* Add "pointer to" */
            return dt;
        case Cast_e:
            /* We have found a brand-new type
expression. Construct dcltype for it,
since the maketypeinfo machinery has
never seen it.
*/
            {
                ScopeDictionary dict;

C.msg any tcell;

for( tcell = e ;
    !( type(tcell) == Compound_stmt 
      && none != (dict = tcell->markScopeDictionary) ) ;
    tcell = tcell^ )
    ; /* Find the active scope dictionary */
dt = proc_spec( e#.Type_name, dict );
e#.Type_name.Abs_declarator
    ::: dt.B_mods#1 ;
}
return dt;

case A_e:  /* Addition expression */
     /* Legal expressions may have one operand
     a pointer or both arithmetic types. */
if( Pointer == type( (dt = expr_type(e##1)).B_mods#1## ) )
    return dt;
else
    return expr_type(e##2);

case S_e:  /* Subtraction */
dt = expr_type(e##1);
dtt= expr_type(e##2);
if( Pointer == type(dt. B_mods#1##)
     && Pointer == type(dtt.B_mods#1##))
    return DclType[ B_int[- ]];
else
    return dt;

case Cond_e:
dt = expr_type(e##2);
dtt = expr_type(e##3);
if( B_void == type( dt.BaseTy# ) )
    return dt; /*Either (void) or (void *) */
if( B_void == type(dtt.BaseTy# ) )
    return dtt; /* ditto */
if( Pointer == type( dt.B_mods#1## ) )
    return dt;
else
    return dtt;

case Comma_e:
    return expr_type( e##2 );

case Asg_e:

case A_asg_e:

case S_asg_e:
    return expr_type( e##1 );

case ident:
{  /* Fetch scope dictionary */
    ScopeDictionary dict;
    CGram@any tcell, ucell;

    for( tcell = e ;
        !(type(tcell) == Compound_stmt
             && none != (dict = tcell->markScopeDictionary)) ;
        tcell = tcell->next )
    ;
    dt = & (tcell = e->(dict.Idm)->markDclType;
    if( type(tcell) == Parm_full_decl
        && ( type(uCell = dt.B_mods#1) == Func
             || type(uCell) == V_func))
        /* This is a function parameter declaration of function type which we now adjust to "pointer to" */
        B_mod[ Pointer[-] ] : dt.B_mods#1;
    return dt;
}
Appendix B

The Interpreter

To demonstrate the implementability of the G data space, an interpreter was written. Rather than write a compiler, which is unnecessary for this limited purpose, I designed an intermediate language to which G could be compiled and wrote an interpreter for it in the Lisp language. It was tested using code hand-compiled from G source code. A sketch of it appears in §4.4.

B.1 The Intermediate Language

The intermediate code is divided into functions, reflecting the original G functions. However, the internal statement structure of a function, which usually reflects the control structure, is flattened to a list of statements. Execution of this list is serial, except for certain statements which act as go-to's. Although these statements have a variety of names (if, if-close, while, etc.), their function is either conditional or unconditional go-to, as described below.

Each function is a list of two elements, the type and the body. The type is used to construct a G function atom. This is saved, as are all G cells, in a property Pcell on a Lisp atom. The body goes into another property Pfunc of the same atom. The body also has two elements, the automatic variables...
and the statements.

When the function is activated, new idents are created from this variable list and pushed onto the Lisp property lists of the appropriate atoms. The profile is examined to see which are parameters, and those are initialized from the arguments. (Initialization of the others is by ordinary statements.)

The statement list contains statements, which are Lisp lists, and labels, which are atoms. Statements contain expressions, which are evaluated, and labels, to which control is sometimes transferred. A statement may be interpreted in one of two modes, which we will call n-mode and e-mode, for normal and exceptional.

When the interpreter is in e-mode, there is an associated exception (§3.12), which may have a value. E-mode is a search for a catch statement which can handle this exception. When it is found the interpreter reverts to n-mode and the value is bound to an identifier. As explained in §4.4, most statements transfer forward in e-mode, skipping statements corresponding to those in nested scopes in the G code.

Statement: if

(if t-close exp t-fail1)
  :
(if-close t-close)
  t-fail1
(else-if t-close exp t-fail2)
  :
(if-close t-close)
  t-fail2
(else t-close)
  :
A G if-statement is translated using I-code if, if-close, else-if, and else. When if is encountered in n-mode the expression is evaluated. If it raises an exception, e-mode begins at t-close. If it is False a branch is made to t-fail. Otherwise, the next sequential statement is executed. In e-mode the branch to t-close is always made.

if-close always branches to t-close in either mode.
else-if is the same as if.
else never branches. It cannot be reached in e-mode.

Statement: while

A G while-statement is translated using I-code while and while-close. When while is encountered in n-mode the expression is evaluated. If it raises an exception, e-mode begins at t-fail. If it is False a branch is made to t-fail. Otherwise, the next sequential statement is executed. In e-mode the branch to t-fail is always made.
In n-mode while-close always branches to t-close. In e-mode the next sequential statement is searched.

G break and continue-statements become I-code break and cont. They are the same as while-close.

Statement: for

(for-init exp1)
   t-close
   (for t-fail exp2 t-fail)
       :
   (break t-fail)
       :
   (cont t-continue)
       :
   t-continue
   (for-inc exp3)
   (for-close t-close)
   t-fail

A G for-statement is translated using I-code for, for-close, for-init, and for-inc. When encountered in n-mode for-init evaluates exp1. Control then passes to the next sequential statement. If an exception is raised, e-mode begins with the next sequential statement. In e-mode the search continues with the next sequential statement. for-inc is the same as for-init.

for is the same as while. for-close is the same as while-close. break and cont were described with while above.

Statement: do-while
APPENDIX B. THE INTERPRETER

\[ \text{t-close} \]
\[ \text{(do t-fail)} \]
\[ \vdots \]
\[ \text{(break t-fail)} \]
\[ \vdots \]
\[ \text{(cont t-continue)} \]
\[ \vdots \]
\[ \text{t-continue} \]
\[ \text{(do-while t-fail exp t-close)} \]
\[ \text{t-fail} \]

A G do-statement is translated using t-code do and do-while. When do is encountered in n-mode control passes to the next sequential statement. In e-mode the search continues at t-fail. do-while is like while except that the branch is taken when the expression evaluates to True.

**Statement: switch**

\[ \text{(switch t-fail exp (cell1 t-1 cell2 t-2 \ldots ) t-default)} \]
\[ \vdots \]
\[ \text{t-1} \]
\[ \vdots \]
\[ \text{t-2} \]
\[ \vdots \]
\[ \text{(break t-fail)} \]
\[ \vdots \]
\[ \text{(switch-close t-fail)} \]
\[ \text{t-fail} \]
A G switch-statement is translated using I-code switch and switch-close. When switch is encountered in n-mode the expression is evaluated. If it raises an exception, e-mode begins at t-fail. Otherwise, the value is compared to each of the cell1 ... celln. The first one which is found to be the same cell as the value causes a branch to the corresponding t. If none are the same, control passes to t-default.

(Note: If no default clause is specified in the original G code, then two statements are inserted immediately after the switch.

(switch ... )

(t-default)

(signal ... )

This raises the required exception.)

switch-close is the same as do.

**Statement: catch**

(catch t-fail atom [ (ident . 1) ])

—or—

(catch t-fail NIL (ident . 1) )

: 

(catch-close t-fail)

(t-fail)

A G catch-statement is translated using I-code catch and catch-close. When catch is encountered in n-mode control passes immediately to t-fail. In e-mode the exception being searched for (represented by a Lisp atom) is compared with atom. If they are the same, or if NIL is found, n-mode begins with the next sequential statement. Otherwise, the search continues at t-fail.
If the transition is to be made to n-mode, the presence or absence of a value with the exception must match that of the ident. The value, if present, is assigned to the ident before n-mode begins.

catch-close is the same as switch-close.

**Statement: compound**

\[(\text{compound } t\text{-close})\]

\[
: \\
\text{(cp-close } t\text{-close})
\]

\[t\text{-close}\]

A G compound-statement is translated using l-code compound and cp-close. Both are the same as do.

**Miscellaneous Statements**

\[(\text{exit } \text{atom } [\text{type-cell } \text{exp}])\]

If encountered in n-mode, transition is made to e-mode. The exception searched for is the one named by atom. If present, the expression is coerced as if by assignment into the type of type-cell, which then becomes the value of the exception. The search begins at the next sequential statement.

If encountered in e-mode the search continues with the next sequential statement.

\[(\text{signal } \text{atom } [\text{type-cell } \text{exp}])\]

If encountered in n-mode the effect of this statement is just the same as exit except for the place where the search begins. Here the search begins in the function which called this one. The expression which names this function returns the exception. The activation of this function terminates.
If encountered in e-mode the search continues with the next sequential statement.

(resignal atom)

This statement combines the function of catch and signal. As with catch, atom may be NIL.

If encountered in n-mode, n-mode continues with the next sequential statement.

(return [type-cell exp])

If encountered in n-mode the expression, if present, is evaluated. If this raises an exception, e-mode begins with the next sequential statement. Otherwise, the value of exp is coerced as if by assignment into the type of type-cell. This activation terminates and the value becomes the value of this function at the point of invocation.

If encountered in e-mode the search continues with the next sequential statement.

(goto t)

If encountered in n-mode, n-mode continues at the statement after t.

If encountered in e-mode the search continues with the next sequential statement.

This is the same as if-close.

(exp exp)

If encountered in n-mode the expression is evaluated. If this raises an exception, e-mode begins with the next sequential statement. Otherwise, the value, which may or may not be present, is simply discarded. Then n-mode continues at the next sequential statement.

If encountered in e-mode the search continues with the next sequential statement.
B.2 The Functions

There are four important Lisp functions that implement the interpreter: **Iapply**, **Ieval**, **Interpreter**, and **Stex**.

**Iapply** is the main entry for evaluation of function applications. It evaluates all primitive functions itself, and passes the task of evaluating defined functions on to **Ieval**. Thus, it is the only place where arguments may be L-values.

**Ieval** creates the new environment by pushing new instances of all the automatic variables. Then, it calls **Interpreter** to execute the function body. Finally, it restores the old environment by popping the variables.

**Interpreter** simply executes the function body as described above.

**Stex** is the general expression evaluator. It evaluates lists element by element (stopping immediately if an exception is raised) and, finally, applies the first element to the rest using **Iapply**. Atoms are returned unchanged.

A few operators are given special treatment here, since their argument lists cannot be evaluated in the usual way. **And** (&&), **or** (||), and **conditional** (?::) are evaluated in the order given in their definitions. An extra operator (**lisp**) exists so that Lisp code can be executed directly.

Many auxiliary functions to the four above were written, the largest of which is **AssignCoerce**, which implements the complex rules of assignment coercion.

Some machinery was also needed to construct the standard environment, the grammar of grammars, of profiles, etc., and some primitive operators which are conveniently defined as functions in I-code: arithmetic, logical, structural equality, copy, etc. These operators often make use of embedded Lisp.

The entire program is about 1500 lines of Lisp and 500 of I-code.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


BIBLIOGRAPHY


