THE TRANSLATION OF PROGRAMMING LANGUAGES THROUGH THE USE OF A GRAPH TRANSFORMATION LANGUAGE

by

PETER NICO VAN DEN BOSCH

B.Sc., The University of British Columbia, 1972
M.Sc., The University of British Columbia, 1974

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

We accept this thesis as conforming
to the required standard.

THE UNIVERSITY OF BRITISH COLUMBIA

February, 1981

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Computer Science

The University of British Columbia
2075 Wesbrook Place
Vancouver, Canada
V6T 1W5

Date 27 April 1981

Abstract

It is shown that the automated translation of programming languages suffers from its traditional domination by context free parsing techniques, specifically in failing to deal uniformly with such translation-related concerns as language extension, optimization, error handling and reporting, and multi-stage translation, as well as in generally ad hoc treatment of the context sensitive aspects of translation, particularly those concerned with the identification of symbols.

A descriptive technique for discussing translation that takes into account not only the immense and continued success of the syntactic basis for programming language translation but also the need to deal uniformly with the above concerns is presented, and demonstrated to be implementable as a programming tool. This demonstration is effected both by means of a detailed discussion of the technique's application to the expression of translation algorithms, and by a consideration of the practical aspects of its implementation as a programming language.

The technique involves a representation of the complete syntactic structure of programs as a directed graph, and the expression of translations as local transformations of the graph representation. A wide range of translation concerns is discussed with reference to graph transformation. Practical experience with an experimental version of a graph transformation language is presented, and used as the basis for a further development in the design.

An evaluation of the completed research, and an assessment of its position within concurrent developments in the discipline of programming language translation, conclude this dissertation.

Table of Contents

Acknowledgements

This page, the last written (and the only one not supervised by anyone but the author), is the most pleasurable to write. Only here, at last, is it possible to give a measure of public thanks to those who have made the production of this thesis possible and frequently enjoyable.

Harvey Abramson, my research supervisor, who has watched over this thesis taking shape from its earliest moments; who not only understood what I was up to from the beginning, but was frequently a step ahead of me in seeing what could be done with it.

David Kirkpatrick, Alan Mackworth, Bary Pollack, Gunther Schrack, and Mabo Ito, members at various times of my guidance committee, all of whom somehow found time to read some or all of three drafts of a thesis proposal and three drafts of this thesis (each longer than its predecessor), and to bring such interest and enthusiasm to the task as to have contributed substantially to its form and content.

Peter Lawrence and John Peck, the university examiners, and A.V. Aho, the external examiner, for the time and interest they have been willing to put into their tasks. It was in John Peck's survey course on programming languages and two subsequent summers of work on his Algol 68 compiler that I first acquired my taste for programming languages.

John Baker, who supervised my master's thesis and thereby taught me how to write theses (and how to play Go).

The people of Compyromatic Systems, especially Al Fowler and Fred Wong, for a stimulating --frequently cliffhanging-- two years, during which the basic ideas presented in this thesis first emerged.

The faculty, staff, and graduate students of the Computer Science department, especially Michael Gorlick, Vince Manis, Rob Cameron, Mark Scott Johnson, Brian Jones, Ross Frazer, Graeme Hirst, Tom Rushworth, and Heather Johnson. There are too many I have had to leave out: these are the ones who most readily come to mind when I think back on these four years, the ones with whom now uncountable hours have been spent discussing everything from the nature of compiler writing to the merits of the arts in the twentieth century.

The Department of Computer Science, the National Science and Engineering Research Council of Canada, and the H.R. MacMillan Family, for financial support.

Finally, but first always in my heart, my parents. We have come through much together, and have grown to be like friends.

"Nature is delighted with transmutations"
Isaac Newton, Optics

Chapter 1: Introduction

The popular conception of a computer programmer is someone who toggles mysterious switches on a great control panel, sets marvellous dials, reads, with the unerring eye of a barn storming pilot, a multitude of flashing lights and shifting meters, and finally receives a single oracular answer from the great machine that he, and he alone, has mastered.

This picture portrays a stereotype; one that continues to flourish in such sources of contemporary mythology as the magazine gag cartoon and newspaper comic strip. Perhaps such a picture is appropriate to myths where, modern anthropologists from James Frazer on have told us, the dull truths of primitive existence are normally elevated into a heroic structure. The dull truth of computer programming is that the programmer, isolated from the actual machine (and its increasingly uninteresting control panel), writes his instructions in a more or less abstract formulation (itself mysterious to the uninitiated, but no more so than the special notations of mathematics, music, or cultural anthropology) and submits these to the computer simultaneously with the submissions of other such programs by other such programmers, receiving his answer, very occasionally as a single number, but much more often as a long, more or less useful sequence of complaints from the machine about its inability to perform, or even understand, his intentions.

The myth nevertheless contains, as myths will, a hard kernel of truth: with all their immense growth in sophistication from electro-mechanical devices to super-fast electronic miracles in one generation, computers remain a complex network of "switches", and the only language they genuinely understand is that of two-valued states and of instructions which cause well defined alterations in these states. If the machine is to "understand" the programmer's symbolic program, the program must first be translated into the machine's own language; and the only reason the programmer can afford to write his programs symbolically instead of, pace the cartoonist, laboriously toggling them on a Wurlitzer-like control panel, is that the translation process can be performed, quickly and accurately, by the machine itself, operating under instruction from another program.

Programming languages, then, are formal symbol systems for the description of computations. Such languages may range from symbolic assembly languages, which have a close, obvious correspondence to machine languages, and can be programmed only with a thorough understanding of the machine's operation, all the way to database manipulation languages, which describe complex relations between data, but do not explicitly describe the actions to be performed in storing and retrieving them, nor their internal representation. The clear trend, in the last thirty years or so, has been increasingly toward abstraction in languages; that is, increasingly the computer programmer uses a

language that is closer to the way technically or mathematically trained people would describe the processes to each other or to themselves than to the way machines represent these processes internally.

The advantage in this approach to programming is clear: the more nearly the description matches the programmer's own perception of the process, the less chance there is of his description being wrong in some subtle, machine related way; furthermore, the more nearly the description matches other technically or mathematically trained people's perception of the process, the more likely they are to read his description and consequently to be able to suggest improvements or extensions, and, should the program need to be revised by another programmer, the less trouble the other will have in understanding how it operates and in what fashion it can be changed.

The result of this approach to programming is also clear: in almost all cases a written program must be translated from its abstract representation into an equivalent representation in the machine's own terms before the machine can execute the program's instructions.[1] Such translations are at the heart of

---

[1] An exception to this rule is the situation in which the program is interpreted by another program being executed by the computer. Even here, we generally find it more convenient, although not strictly necessary, to design the interpretive program in such a way that it operates, not on the interpreted program's original representation, but on a

all programming efforts and, therefore, the problem of building suitable (efficient, helpful, accurate) translators is of central concern to the design of programming languages. Consider, by way of illustration, this account of the early history of programming languages:

> "The very first attempt to devise an algorithmic language [the Plankalkul] was undertaken in 1948 by K. Zuse... His notation was quite general, but the proposal never attained the consideration it deserved... In 1951 [Ruthishauser] tried to show that in principle a general purpose computer could translate an algorithmic language into machine code... However, the algorithmic language proposed in this paper was quite restricted; it allowed only evaluation of simple formulae and automatic loop control... In 1954 Corrado Boehm published a method to translate algebraic formulae into computer notation...Laning and Zierler presented their algorithmic language -- the first one ever actually used -- and shortly thereafter the IBM Fortran system was announced... A working group called the GAMM subcommittee for Programming Languages was set up after the [1955] Darmstadt meeting in order to design...a universal algorithmic language... This subcommittee had nearly completed its detailed work in the autumn of 1957, when its members, aware of the many algorithmic languages already in existence, concluded that, rather than present another such language, they should make an effort towards worldwide unification [the ultimate result of which was Algol 60]." (H. Ruthishauser, quoted by Naur [92], pp 16-17.)

The first programming language, then, dates back to 1948,[2] and the first compiler to 1954. By 1958 there were already so many efforts at producing translators for languages that the

---

more convenient "internal" form of the program: the internal form is, of course, derived from the original representation by a process of translation.

[2] The first in modern times; Ada Augusta, Countess of Lovelace, is credited with the first abstract notation for programming Babbage's unrealized analytical engine.

first proposal for simplifying the process resulted in the never realized universal intermediate language Uncol.[3] We may date from 1948, therefore, the desire to simplify by abstraction the process of programming a computer, and from 1958 the desire to simplify the implementation of translators for these abstractions. By the late 1960's there had already been so many efforts at simplifying the process of writing translators, that Feldman and Gries were able to publish a substantial and quite necessary survey of such efforts, and to give them the umbrella title of translator writing systems [39]. This survey, which ranges roughly from Uncol to the most current techniques of 1967, was more recently supplemented by a history of early trends and historical contributions to compiling, published by Bauer [11].

The most striking fact apparent from these surveys is that the early efforts were almost entirely concerned with the efficient recognition of the component parts of the program -- the process called parsing, by analogy with the model of natural language construction and recognition commonly taught in schools. The translation -- usually, by stretching the analogy between programming and spoken language, called the semantics --

---

[3] Uncol was intended to be implemented on all computers -- not an entirely unrealistic goal -- and to be a suitably abstract target for translations of all possible programming languages; the intent was to thus simplify the translation by bringing the target closer to the source. The failure of Uncol lay in its designers' innocent assumption that they knew roughly what they meant by "all possible programming languages."

of these "high level" languages to machine level languages remained, largely, a matter of attaching programs to the syntactic specification. This approach to the specification of translators, called syntax directed translation, has proved to be so successful that, in the years following the Feldman and Gries survey, it has been refined and extended, rediscovered and re-expressed, but never abandoned as a basis for translator programming. Translator writing systems continue to appear with regularity [57,65] whose only abstraction of the process of writing translators is to provide new variations on the automatic parser generation techniques that date back to the early 1960s, and to attach newer programming languages to the resulting syntactic recognizers for the purposes of translation. General textbooks on the subject of translation [53,3] continue to devote about half their space to presenting general, theoretically founded methods for automatically turning syntactic specifications into fast parsers, and the remainder presenting a collection of ad hoc techniques for translation.

Chapter 2 of this dissertation will present an in-depth analysis of the apparent reasons that translator abstraction has not qualitatively progressed beyond this position, attained rather early in its history, and will outline a descriptive basis for discussing translation that takes into account not only the immense and continued success of the syntactic basis for the translation of programming languages, but also the need to describe aspects of translation that do not fit cleanly into

the existing paradigm of syntax directed translation. The most important of these aspects are:

(1) The "context sensitive" nature of programming language syntax, which is not easily dealt with in the "context free" descriptions processed by automatic parsing techniques;

(2) The structuring of translators so that functionally independent processes are describable by independent procedures (the so-called "structured programming" discipline) rather than, as is almost inevitable in syntax directed translation, by a single integrated process spread out over a description of the syntax that is frequently distorted by the demands of automatic parser generators;

(3) The expression of optimization -- that is, the improvement of automatically generated machine language translations so that they approximate the quality of a hand-written machine level program -- in a clean, uniform manner;

(4) The handling and reporting of errors; and

(5) The increased automation of activities now performed more or less adequately by programmers, namely, making abstractions of programs explicit and synthesizing programs from their abstract specifications, improving highly abstract programs, editing the syntactically structured text of programs, proving the correctness of programs.

These topics will be treated both in terms of the proposed approach to the description of translations, and with reference to their classical treatment by compiler writers and programmers, in a close inspection of the aspects of translation presented in chapter 3.

Finally chapter 4 concludes the main thrust of this dissertation by presenting a proposal for the construction of a translator writing system, and actual experience with the use of an experimental system based on the considerations of chapters 2 and 3. An evaluation of the proposal, especially in the light of its further development and the effect it is intended to have on the design and implementation of programming language translators, concludes the dissertation in chapter 5.

The reader will pass through several stages in what we might describe as the evolution of an idea. From the primordial notion that we wish to treat the translation of programs from one language to another (what we will call, for the sake of brevity, the translation of programming languages, and often simply translation) as uniformly and abstractly as possible by thinking of it as a textual transformation; through the realization that text -- a word, incidentally, which derives from the Latin textus meaning "web" -- as a purely linear sequence of symbols, is inadequate for the representation of the syntactic relationships between the symbols to give any but the most primitive control over their correct transformation;

through the presentation of translation as a transformation of syntactic structures which have lost much of their relationship to the original text; finally to an abstraction of the syntactic structures which will return us to the original -- but now considerably enhanced -- concept of textual transformation.

It will be a long journey and, like most journeys, it will be circular: we will travel a long way from Ithaca only to arrive back in Ithaca. However, we will find ourselves -- and therefore Ithaca -- much changed for the journey.

Chapter 2: The relationship between translation and graph transformation

The primary intent of this dissertation is to provide a uniform basis upon which to discuss and, ultimately, implement translations. It is the function of this chapter to show that such a basis may be found by treating a translation as a transformation on a program with reference to a broadened model of the syntactic structures involved. These include the syntactic structures of the language in which the program is expressed, of the language into which it is to be translated, and of any intermediate forms through which this translation may proceed, including, in the case of program optimization, the flow structure.

That syntactic structure is vital to the correct translation of programs is established, in section 2.1, by an examination of the influence of syntactic definition on all aspects of programming languages. A graph representation for the syntax of programming languages is developed in section 2.2, using as its basis the long established relationship between context free syntax and a tree representation of programs. Finally, a basis for discussing translation in terms of transformations on these syntactic structures is developed in section 2.3. It is left to the next chapter to demonstrate how this model may be applied to a large range of considerations, both traditional and modern, in programming language translation

and program development.

2.1 The relationship of syntactic structure to translation

Chomsky introduced the notion of "phrase structure" grammar into linguistics; the name is appropriate to the aspect of grammars we are examining here because, while Chomsky was fully aware of the computational properties of his generative grammars, and their relationship to earlier computational models like Markov systems and Turing machines, a considerable part of his intent in investigating this specific formulation of grammars was that it should result in a precise specification of the relationship between elements of a phrase -- hence the term "phrase structure grammar" [24]. In fact, the cited paper gives a method for constructing a labelled parse tree of a phrase from its derivation in a "type 1" or context sensitive grammar (restricted to non-erasing rewriting rules).

The international committee at work on the specification of Algol 60 developed a form of syntactic specification based on Chomsky's work in order to give an unambiguous specification of the form of Algol 60 in their published report. This was a perfect case of two previously unrelated ideas coming together -- a phenomenon not uncommon in the history of science: Chomsky's recursive definition of syntax and the nested block structure of Algol 60 (already present in the Algol 58 version, but not previously rationalized) were made for each other. To

this day, BNF[1] is used to describe the context-free syntax of programming languages, and the free form nested structure of Algol 60 has remained the paradigm for the form of programming languages however little their syntactic entities now resemble those of Algol 60.

The convenient description of programming language syntax by BNF appears to have had the effect also of paving the way for the first translator writing systems. Rosen [106] credits Irons with the invention of the syntax directed compiler, but it is a concept that follows so naturally from BNF notation that it cannot be traced with certainty to any author but appeared in several places at once; nevertheless, Irons published the first paper using the term [63]. The earliest successful efforts in translator writing systems, META [113], COGENT [101], etc. (see Feldman and Gries [39], pp 92-99), were based on syntax directed translation schemes. In its simplest form, syntax directed translation involves writing a BNF specification of the language

---

[1] Originally, BNF stood for Backus Normal Form, but since Knuth's alternative suggestion [70] that it be called Backus Naur Form for historical reasons (see Naur [92], pp 20-21,36; note dissent by Bauer, p 41), the world has been divided into two camps. It is probably just as well to stick to the initials "BNF", since, according to a private communication from Ingerman reported by Rosen, "the earliest known use of [a technique analogous to BNF] was by Panini...Between 400 B.C. and 200 B.C. he (or his school) used a notation which includes all the metalinguistic apparatus reinvented by Backus (excepting only recursive rules, for which no need was evinced) to describe the peculiar grammatical constructions known as samdhi which occur in Sanskrit" [107]. The designation "BNF" seems preferable to "Panini Chomsky Backus Naur Form," and is by now well-established.

and augmenting these rules with "actions" written in a more or less adequate programming language; these actions are then performed whenever the rule is used successfully in a parse of the program being translated. The BNF specification is usually restricted by the pragmatics of automatic parser generating techniques: for instance, a recursive descent parser (the most popular form in the early days), must exclude left-recursive rules; the most popular techniques today, the deterministic LL and LR forms are known to be less inclusive in their descriptive power than general context free forms, but they are also far more efficient bases for recognizers.[2] Translators continue to be primarily syntax directed in their structure (the syntactic structure of Pascal was designed with LR(k) parsing in mind); and so do translator writing systems (a recent example is YACC [65] which uses a LALR(1) parser generator coupled with actions programmed in C).

The profound effect of BNF on various aspects of programming language research, including, but not limited to, translation, is reviewed in the next subsection (2.1.1). Subsequently (2.1.2), it will be argued that the success of BNF in these applications has led to several problems directly traceable to the representation, whether explicitly or implicitly, of programs under translation as a necessarily

---

[2] See the chart in Aho and Ullman [2], p 449, which partially orders known parsing techniques by the sets of languages they recognize.

context free parse tree. Various programming and definitional techniques that have already been devised to deal specifically with these problems will be surveyed, and their strengths and weaknesses in dealing with the translation problem will be discussed.

2.1.1 The impact of BNF

The central argument of this dissertation is based on the overwhelming importance of syntactic structure to the correct translation of programming languages. There will be little argument today over this importance, and we have already seen indications that much of the published material on programming language translation takes it for granted that translation is intimately joined to syntactic recognition.

This section, primarily a survey, deals with the central role played by syntactic structure in each of three areas of programming language research: language design, semantic specification, and compilation techniques. Its intent, in conjunction with the subsequent section, is not to heap unnecessary detail on established fact in order to argue the importance of syntactic structure, but rather to show the degree to which context free syntactic recognition -- at present the only viable and acceptably general form of automatic recognition -- has restricted the expression of programming language translations.

(a) Language design

The first use of BNF in the systematic definition of a programming language was by Naur, in a draft report he presented to the ACM-GAMM committee developing Algol 60 [92]. It is not clear from the historical accounts how much influence the adoption of a recursive definitional technique had on the design of Algol 60. Certainly IAL, the prototype now known as Algol 58, was strongly influenced by Fortran (which, if it had not been an IBM property, would probably have been adopted by the ACM as an American standard [97]), and, although the notion of recursive definition is present in the document defining Algol 58 [96], it is not used to great effect either in definition or design. For instance, the if statement has a Fortran flavor:

if (a>0); c:=a+b

Similarly, the notion of a local variable is missing (a variable may be declared anywhere in a procedure; its range is then the entire procedure), and the begin-end markers are purely a bracketing notation.

It is clear, however, that the form and presentation of Algol 60 had an immediate and lasting effect on subsequent language design. The definitional technique we call BNF is, because of its recursive nature, ideally suited to the definition of a language like Algol 60, whose syntactic structure is, by and large, recursive. Virtually every language design developed since Algol 60 has, at one time or another,

been communicated in BNF, and, to quote Perlis, "where important new linguistic inventions have occurred, they have been embedded usually within an Algol framework... Algol has become so ingrained in our thinking about programming that we almost automatically base investigations in abstract programming on an Algol representation" ([97], p 13). At the same time, languages such as Fortran, which was designed without benefit of BNF, Cobol[3] and especially Basic appear to us primitive and without elegance when compared to Algol 60.

Two final examples of the influence of syntactic definition on language design deserve to be mentioned. The first is PL/I, a language clearly based on Algol 60 but equally clearly designed without recourse to BNF or some other definition of syntactic structure.

PL/I displays an odd combination of Algol 60's free format design and Fortran's orientation toward card images, often to the programmer's confusion. By contrast, Algol 68, the second example, is at the other end (some would say extreme) of the design spectrum: the concept of "orthogonality," a kind of Occam's Razor of programming language design, led the designers

---

[3] The design of Cobol took place simultaeously with the revision of IAL that introduced BNF (and the name Algol). Sammet, in a historical summary of the language's development, discusses briefly the relationship of Cobol's still widely used "metalanguage" and BNF; she concludes that, though it can be done, BNF is not well suited to Cobol's style -- which suggests that the definition mechanism used has a strong influence on the form of the language ([109], p 141).

to attempt to reduce redundancy of function to a minimum. Thus, the assignment symbol, ":=", is treated like an operator, and "statements" virtually disappear, since such traditional syntactic structures as if-then-else and begin-end are value-returning expressions, while the semicolon, the Algol 68 statement separator, acts like an operator controlling the evaluation of the two expressions it separates. The recursive nature of BNF would appear to induce orthogonality: having "named" a concept by defining its syntactic structure in a production rule, one can then insert that concept in any other syntactic construct simply by using its "name" -- the symbol used on the left hand side of the defining rule. Algol 68 was specified, it is true, using a more powerful definitional technique than BNF, but in the original Report [131] the use of this technique, and the way it relates to the syntactic structure of the language, is not substantially different from the use of BNF in defining Algol 60.[4]

(b) The specification of semantics

As early as the definition of Algol 60, language designers sensed the need for a more precise definitional calculus than English sentences [92]; but, except for some formal systems like lambda calculus, Post systems, Markov systems, and Turing

---

[4] This statement is no longer true of the Revised Report, where the definitional technique is put to considerably more ambitious use. This aspect of Algol 68 will be discussed in greater detail in section 2.1.2.

machines, there was nothing immediately available to them to exploit for this purpose. Turing's and Church's work, at least, must have been known to members of the Algol 60 committee, but it was several years before lambda calculus was, with mixed success, applied to the definition of programming language semantics (see Feldman and Gries [39], pp 104-106). If any of these techniques were considered, nothing came of it in time for the publication deadline of the Algol 60 report; in retrospect, it is reasonable to assume that Algol 60 would have suffered from so rigorous a definition especially because none of these formal computational models relates well to a language like Algol 60. In any case, these techniques are no longer serious candidates for a definitional calculus of programming languages, and in presenting some of the methods that are seriously pursued at present, I shall be arguing that at least part of the reason for their poor applicability to programming language semantics is that the earlier methods, having been devised without a good paradigm of a programming language, and especially without a good paradigm of syntactic structure such as is provided by BNF, could not be expected to apply elegantly to the problem of semantic specification.

A good tutorial on some definitional techniques and their application to programming language definition is provided in a relatively recent paper by Marcotty et al. [87]. This survey, although limited in scope primarily to "operational" (as distinct from "denotational") semantics,[5] reviews several major

approaches to semantics, namely, the Vienna method, production systems, W-grammars, attribute grammars (and the functionally equivalent affix grammars), and axiomatic semantics. (The latter is an exception to the rule: it is not an operational semantics, in that it defines no underlying computational model.)

The most thorough attempt at creating a complete operational model of the semantics of programming languages is the effort of the IBM Vienna Laboratories, known as Vienna Definition Language (VDL). Besides the description in Marcotty et al., the form and application of VDL was described in detail by Wegner [139].

VDL definitions consist of transformation rules over a set of trees representing the abstract syntax of a programming language. For any particular program, then, it is possible to determine the computation it describes by tracing the effect of the transformations defined on it. As a definitional method, VDL is analogous to BNF: for any string it is possible to determine whether it is a syntactically correct entity in a language (excluding contextual requirements) by determining whether it can be generated by the BNF definition of the

---

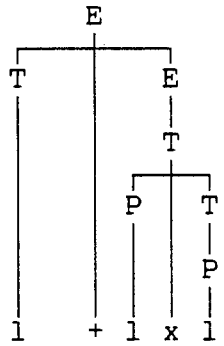
[5] An operational definition of a language provides an abstract model for computations expressed in the language and specifies a translation of programs into that model, whereas a denotational definition specifies the mathematical functions computed within the language.

language.

We will return to VDL in the next section. It is sufficient for our present discussion to note its use of syntactic structure in specifying semantics. "Abstract syntax" is a term used in analogy to the "concrete syntax" of, say, BNF. To prevent ambiguity, the concrete syntax of, for instance, expressions, must be presented, in a context free generation language like BNF, in some form similar to:

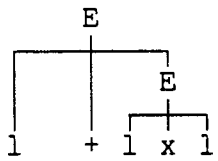
```
E ::= T
    | T "+" E
T ::= P                                          [G1]
    | P "x" T
P ::= "1" | ...
```

This grammar yields one, and only one, derivation for the string "1+1x1", namely, the one represented by the parse tree:



[T1]

However, the syntactic structure of the string is revealed just as well by a tree of the form:



[T2]

Tree T2 is derived from tree T1 by eliminating all nodes with only one branch. These, in turn, represent one-on-one productions in the grammar, like "T::=P", which exist solely to disambiguate the derivation of strings with respect to the grammar. Having reduced the number of node types that will occur in any parse tree, we will also have reduced the size of a semantic specification. Abstract syntax will appear again in section 2.2.2, where it is considered as a step toward the derivation of a syntactic representation for translation.

The other major approach to semantics is the denotational. Most of the primary sources for denotational semantics are either obscure or, in the case of several recent books, formidable; but for an easily accessible tutorial and bibliography, see Tennent [124]. We will not concern ourselves here with the application of this approach to the definition of semantics, any more than we have concerned ourselves with the details of VDL's operation. It is sufficient for our discussion of the implicit importance of syntactic structure to any consideration of programming languages to note that any denotational definition of a language will be based on a definition of the language's abstract syntax.[6] As in Hoare's

---

[6] Compare Tennent's examples: [124] p 438 (LOOP), pp 441,447 (AEXP), and p 449 (GEDANKEN).

semantics, definitions use only syntactically well-formed entities in the language, and will often depend on a recursive elaboration of those entities, based on their syntactic structure. For example, consider the definition of sequencing:

$$C[S1;S2] = C[S2] \circ C[S1]$$

Here, the function C (which gives an interpretation of the syntactic entities we shall call "commands", including loops, assignment, etc.), is defined recursively: the value of two commands separated by a semicolon is the value of the second command composed with the value of the first. C applied to a command delivers a function mapping environments to environments; hence, the composition of two such functions will yield another such function. A similar definitional mechanism is at work in axiomatic specifications, where the above construct would be defined as:

$$\frac{\{p\}S1\{q\} \text{ and } \{q\}S2\{r\}}{\{p\}S1;S2\{r\}}$$

(If S1 transforms predicate p into predicate q and S2 transforms q into r, then the construct S1;S2 transforms p into r: the predicates function as assertions defining as much of the environment as necessary for the purposes of the semantics.)

We have seen that every major approach to semantic definition devised in the last twenty years has used some notion of syntactic structure. In models like VDL, this structure is explicitly manipulated, and the value of the program is derived

from the results of the manipulation. In denotational semantics, the structure is used implicitly in the form the statement of semantics takes: the recursive definition of syntax induces an analogous recursive structure on the definition of semantics as in the above example, where the syntactic rule

SSeq ::= S ";" SSeq

is echoed in the definitional rule

C[S1;S2] = C[S2]°C[S1]

which defines the evaluation of "SSeq" by defining the evaluation of its component parts, and does so in terms of a function of its syntactic subforms. Of the major models of computation devised before syntactic definition in terms of BNF was available, Post's, Turing's, and Markov's are all abstractions of mechanical processes and are defined in terms of operations on a string of symbols, resembling a computer's memory; only lambda calculus resembles a programming language. Those of the early models that have been used as a basis for programming language definition in the last twenty years, most notably production systems, have had to be modified to include (and, in fact, be structured by) a syntactic definition, or else have been used as abstract machine models onto which may be specified a syntactically structured mapping from the language to be defined [144,77].

That this is so is indicative once again of the impact of Algol 60 and, indirectly, of the use of BNF in its definition on all aspects of programming languages. We cannot now seriously

entertain a definition of a programming language without reference to its syntactic structure any more than an astronomer can seriously entertain the notion of a geocentric universe.

(c) Compilation techniques

We have already seen, in the introduction to section 2.1, that compilation techniques were immediately affected by the introduction of BNF. An early Algol 60 compiler [63] was structured as a recursive descent parser, as were most of the early translator writing systems (see Feldman and Gries [39], pp 92-103). Recursive descent parsing restricts grammars only in that rules must not be left-recursive; it tends to be a wasteful method, in that it develops many unsuccessful partial parses in finding a correct parse. More efficient parsing techniques (simple and operator precedence, LL and LR) place greater restrictions on the grammars they can handle and therefore on the set of languages,[7] but at the same time, allow automatic determination of whether the grammar is ambiguous -- an important asset in syntax directed translation.

It is an unfortunate fact that parsing is a much better understood and, consequently, much more thoroughly documented aspect of translation than any other. This situation is clear

---

[7] However, all the aspects of existing programming languages that can be described by a general context free grammar can also be described by an LR(k) grammar; the limitations are, therefore, only of theoretical interest.

not only in the 1968 Feldman and Gries survey, but also in a 1967 survey by Bauer [11], as well as in such general texts on compiler writing as Gries [53], and Aho and Ullman [3]. Too often, therefore, "translator writing systems" turn out to consist of a parser generator and some more or less useful facility for writing "semantic" routines -- most usually, and often most usefully, a general purpose programming language, as in the cases of XPL [84], and YACC [65].

In spite of this one-sided development, the concept of syntax directed translation has been highly successful in the sense that it is the most commonly used paradigm of translation. Not only are such "classical" syntax directed compiler writers as YACC still being invented, but more sophisticated techniques such as multiple passes over the parse tree (Abramson et al. [1]), and augmented grammars (which may be, under certain conditions, implemented as multiple passes over a parse tree: see Bochmann [15]) continue to use, as their basis, the paradigm of syntax directed translation.

2.1.2 The problems with context free syntax

We have seen, in the preceding discussion, that some concept of syntactic structure is at the core of every aspect of programming languages: design, definition, and implementation. BNF, which was developed for the definition of Algol 60, the first genuinely "modern" programming language, is highly

successful as a model for the syntactic structure of programming languages and, suitably restricted, yields to well-understood and quite powerful techniques for automatically generating acceptors (parsers). Given a recursive definition of programming language syntax, it was a natural though surprisingly rapid development that syntax directed translators and translator writing systems emerged; and the success of the syntax directed approach to translation, at least in terms of the number of systems based on it, lends it great credence as a model for translation.

Nevertheless, almost from the very beginning it has been recognized that BNF is an inadequate representation for the complete syntax of programming languages such as Algol 60. Floyd demonstrated, as early as 1962, that the Algol 60 requirement that all variables used in a program be declared (a requirement followed by almost all subsequent programming languages),[8] makes it impossible to define the language completely by means of a "simple phrase structure" (context free) grammar [41].

This limitation on context free grammars, and hence on BNF, is also a limitation on syntax directed translators. Indeed, it

---

[8] A notable exception is PL/I which, as we have already remarked in the discussion on language design in section 2.1.1(a), did not entirely absorb the important innovations of Algol 60. Transforming this simple requirement into a complex set of default rules appears to have been a mistake which has brought many a PL/I programmer to grief.

is in dealing with identifiers of all kinds that syntax directed translators display their greatest weakness, either in ad hoc treatment, or even in a complete breakdown of the model in those places where identifiers occur. Context sensitive restrictions on programming languages are generally dealt with under the heading of "semantics". This merely adds further fuzziness to an already overworked term, however: as Ledgard says, in a paper describing certain extensions of the context free (BNF) model, "Relegating the context sensitive requirements to the definition of semantics obscures the issues properly considered as semantics" [76] (italics mine; see also McKeeman [85], p 14).

Because data description and the use of names has been steadily increasing in sophistication and complexity in recent years,[9] context free syntax has become increasingly more inadequate as a description of programming languages because it effectively ignores one of their most important aspects, namely the relationship between a name and the nature of the data it refers to (its declaration); it cannot be long, therefore, before translation techniques directed by context free grammars are seen as hopelessly inadequate -- not because syntactic structure will cease to be an effective basis for dealing with programming languages, but, on the contrary, because context free syntax alone will no longer describe enough of the structure of the language.[10]

---

[9] See sections 3.1.1 and 3.1.2, especially as they relate to the concept of abstraction.

Nevertheless, not only is the syntax directed translation paradigm still pursued as seriously as ever, but syntactic descriptions even more limited in scope than context free continue to be put forward as bases for general translator writing systems. In a 1967 paper describing the ML/I macro processor,[11] Brown admits the limitation of a language design using a regular expression recognizer to certain kinds of language extension [18]. But in a 1976 paper -- which is chosen here not to be singled out, but to illustrate a type -- Tanenbaum proposes the use of a macro processor, in this case ML/I, as a translator writing tool [122]; however, the language chosen to illustrate the use of ML/I has well balanced control structures, like IF-THEN-ELSE-FI and WHILE-DO-OD, which ML/I is able to parse: the proposed method would not do so well with Algol 60 or Pascal. Other general-purpose macro processors like TRAC, GPM, and Leavenworth's syntax macros, would not succeed even as well as ML/I, simply because ML/I has (and Tanenbaum makes use of) a symbol-table facility to deal with context sensitive matters.

Even nonsyntactic approaches, like the use of Markov systems as a basis for translation, continue to be suggested with little or no attention given in these dissertations to genuinely complex translation problems.[12] That such systems

---

[10] This argument is essentially similar to one put forward by Schwanke [114].

[11] ML/I will be more fully treated in the section on language extension (3.1.1)

succeed at any translations is remarkable; I can only recommend, from personal experience, writing a Ratfor preprocessor in Snobol as a sure way to be convinced that, while it can be done, it is not the way to do it, especially not for general translator writing.

Given that context free grammars are inadequate for the description of programming languages, the question naturally arises how the syntax directed approach has nevertheless managed to become so thoroughly the basis for translators. The answer lies, in almost every case, in the use of a symbol table, and a minor restriction on programming languages that declarations of names must textually precede their use.

Consider the simple language L:

```
(1 ) L ::= BLOCK
(2 ) BLOCK ::= begin DCLS; STMS end
(3a) DCLS ::= TYPE VAR
(3b)        | TYPE VAR ; DCLS
(4 ) TYPE ::= real | int | ...
(5 ) VAR ::= a | b | ...  | z
(6a) STMS ::= STM
(6b)        | STM ; STMS
(7a) STM ::= VAR
(7b)       | BLOCK
```

This context free grammar generates such programs as

begin int a ; a end [P1]

begin real b ; c end [P2]

Our intuition about the relationship between declarations and

---

[12] A recent (1979) example using Markov systems is due to Turchin [125].

uses of names tells us that P1 is a legal, and P2 an illegal program; but the context free grammar in no way distinguishes between the two, nor, as Floyd has shown, can it do so uniformly for all strings.

A parser with symbol table operates as follows for these programs: the sequence of rules in a leftmost derivation of P1 is:

```
L => BLOCK                              (1 )
  => begin DCLS ; STMS end              (2 )
  => begin TYPE VAR ; STMS end          (3a)
  => begin int VAR ; STMS end           (4b)
  => begin int a ; STMS end             (5a)
  => begin int a ; STM end              (6a)
  => begin int a ; VAR end              (7a)
  => begin int a ; a end                (5a)
```

After the first application of rule 5a, the symbol table contains the pair <a,int>, indicating that variable a has been declared as type integer. At the second application of rule 5a, the symbol table is examined for a pair <a,T>, where T may be any derivation of TYPE; since <a,int> is present, rule 5a succeeds and so does the derivation.

Notice that this sketch not only shows how the symbol table is used to restrict the context free grammar, but that it implicitly requires declarations to precede uses, since otherwise the pair <a,T> would not be present at the right moment.

The derivation of program P2 is similar, except for the two

applications of rule 5a, which generates the symbols b and c, respectively. Note what happens in the symbol table, however: the first application of rule 5a adds the pair <b,real> to the symbol table; but the second application fails to find a pair <c,T> in the symbol table, and the derivation fails.

This description of the use of symbol tables has glossed over several important details. It could afford to do so because the example was a simple one, but in general, since L is a block structured language, the symbol table would have to associate a stack of declarations with each symbol; a declaration of that symbol in a new block would push the type onto the stack, and at the end of each block, those symbols that were declared in that block would have their stacks popped to the previous declaration. In order to do this, it is necessary to put into the symbol table not only the type (which we have only used so far as a token to put on the stack) but also the block depth and, for the sake of the code generator, a count of the number of symbols already declared in the block. For example, the program

begin int x ; begin real x ; x end x end [P3]

is accepted as follows: at the first occurrence of x, the symbol table becomes:

x : <int, 1, 1>

After the second occurrence of x the symbol table changes to reflect the new declaration:

```
x : <real, 2, 1>
    <int, 1, 1>
```

The first use of x is therefore identified as <real,2,1> (the top element on the stack); after the first end, the stack is popped to return to its earlier state, and the second use of x is identified as <int,1,1>; after the second end, the stack is popped again and x ceases to be a declared symbol.

This discussion, which should be familiar in some form to anyone used to dealing with modern programming languages, is presented primarily as an introduction to more formal methods of saying the same thing. That it fails to define precisely what happens for all cases, but merely illustrates what happens in two and depends on the reader's powers of induction to generalize, illustrates the inadequacy of such informal methods, especially when one attempts to give a precise definition of the relationship between declaration and use of names in languages more realistic than L.

That more precise methods were needed was understood, as we have seen, at the time of Algol 60's definition; but such methods were not immediately forthcoming. The definition of Algol 68, however, brought matters to a head, resulting in both van Wijngaarden's two level grammars, also known as W-grammars,[13] and Koster's affix grammars [73] (equivalently,

---

[13] Both these names are dodges around the unpronouncability of "van Wijngaarden grammars" in any language but Dutch.

and contemporaneously, Knuth's attribute grammars [71, 15], which have become the more generally known form).

The original Algol 68 report [131] did not completely specify the language's context sensitive syntax in W-grammar, but depended heavily on a legalese English that has been much criticized as giving the language a reputation for obscurity; the revised report [132], issued in 1976, succeeded in describing all of the language in W-grammar form, at the cost of losing much of the relationship between the grammar and the syntactic structure of the program it describes. A complete W-grammar description of L, along with a reasonable description of how it operates (including a description of the W-grammar mechanism), is too long for the purposes of this section, especially because such a discussion is already available elsewhere (Marcotty et al. [87], pp 199-216), with reference to Asple, a small Algol-family language.

That discussion uses the metanotion TABLE to perform the function of a symbol table. The declaration "int x" in program P3 would, for example, be represented as a string of the form

loc letter x has int ... end

which is a legal derivation from TABLE (portions of Asple definitions not useful for a definition of L are represented by ellipses). This is the equivalent of adding <x,int> to the symbol table. The rule that a symbol must not be declared more than once is embodied in the hyperrule

```
LOCSETY loc TAG has MODE ... end restrictions :
  where TAG is not in LOCSETY,
    ...
    LOCSETY restrictions ;
  where LOCSETY is EMPTY .
```

This checks that the last symbol (TAG) in the portion of the table to be checked is not in the rest of the table, by use of another hyperrule which eventually produces "where TAG is not in LOCSETY" as "EMPTY" if this is so. It also checks that the same restriction holds for the rest of the table. It succeeds if the table is empty, which indicates that it has been completely checked.

The basic rule for programs is

```
program : begin,                          (0)
          dcl train of TABLE,             (1)
          TABLE restrictions,             (2)
          TABLE STMTS stm train,          (3)
          end, ...                        (4)
```

This creates TABLE in line (1), checks declaration restrictions through the above hyperrule in line (2), and then passes the symbol table on to the statement portion, to be used in context checks, in line (3). The rule

```
TABLE MODE TAG identifier :
  TAG identifier,
    where TABLE contains loc TAG has MODE,
    ...
```

(a result of certain derivations of the "TABLE STMTS stm train" portion of the preceding rule) is used to check that any symbol in the program occurs in the table and, incidentally, to establish its type, for other possible restrictions on the use of symbols.

Asple is not block structured, so the definition of TABLE is not sufficiently complex to model the stack introduced in the earlier discussion of L, and thus deal with the locality of variable declarations. However, it is easy to see that the same principles can be applied.

W-grammars of this sort do not yield to automatic generation of parsers and, as such, W-grammars are purely a definitional device. Augmented grammars on the other hand can, under automatically enforcable restrictions, be used to specify parsers [29,15]. Again, we shall depend on the discussion of attribute grammars in Marcotty et al. (pp 250-265) to shorten our discussion here to the necessary minimum.

The primary concept which permits these grammars to deal with arbitrary contextual requirements, and which gives attribute grammars their name, is that of inherited and synthesised attributes. These are effectively parameters (restricted to, respectively, input and output) to a recognition procedure in a top down parser where each nonterminal symbol represents such a procedure. Thus the context free rule

program ::= begin dcls ; stms end

is equivalent to the recursive program definition

```
define program () as
  "begin",
    dcls (),
    ";",
    stms (),
    "end".
```

In a proper context sensitive parse, however, it is necessary to check that all variables are declared, and this is done by passing parameters to, and receiving values back from, "procedures". For example,

```
define program () as
  "begin",
    dcls ({}, env),
    ";",
    stms (env),
    "end".

define dcls (value env1; result env2) as
    dcl (env1, env2)
  or
    dcl (env1, env3),
      ";",
      dcls (env3, env2) .

etc.
```

The program rule passes an empty set to the dcls rule, and expects to receive back a set (env) representing the symbols declared there, which it passes on to the stms rule. The dcls rule receives a set of declared symbols, passes this on to the dcl rule and expects to get back a new set, consisting of the old set augmented with further declarations; this it either returns or, in a recursive call, augments further before returning. The sets named "env" above, are used in a manner highly analogous to the use of TABLE in the W-grammar formulation. (See Marcotty et al., pp 250-265, for more standard notation; compare also the use of augmented grammars in section 4.1, below.)

The purpose of going through these two examples of formal

definition was to show that the problem of dealing with context sensitive aspects of programming languages has been approached in a variety of ways. Essentially equivalent formulations [1,29,77,99] need not be discussed further here, and it should be obvious from the above discussion that even two such dissimilar approaches as W-grammars and attribute grammars are applied to specific programming language definitions in essentially the same way: the context free form of the program is further restricted by comparing its use of symbols against its declarations of them in what amounts to a symbol table; they are, in the final analysis, all formal restatements of our earlier discussion on the use of such tables in compilers.

In the next section, we will develop a representation for the syntactic structure of programs that will deal with context sensitive restrictions without resorting to symbol tables.[14] This will be done so that, subsequently, a view of translation can be developed that depends on a dynamic syntax of programs (rather than the static view of syntax directed translation), and so that a uniform treatment of translation can be considered

---

[14] This is not to say that symbol tables are thereby discarded. An implementation of a particular compiler will almost certainly make use of some form of symbol table, unless there is a major departure from our current view of syntax and its role in programming language definition. The symbol table will merely, in this formulation, be pushed into the background as a primitive concept upon which more lucid conceptions of syntactic structure are based, much as the goto has been pushed into the background of modern language design issues even though it continues to be the basic control flow primitive for all computers.

in chapter 3.

2.2 Representing syntactic structure as a graph

We have seen, in section 2.1, that syntactic structure is a pervasive concept, important in all aspects of programming languages. We have seen also that context free syntax is not sufficiently expressive of the structure of programs to be able, by itself, to direct translation, and we have examined the prevalent method for overcoming this inadequacy by using some form of symbol table. In this section we will be developing the notion of syntactic structure and its representation along traditional lines, but to a new conclusion, and we will see that this eliminates -- at an important point in the translation process -- the need for such classical supports as symbol tables, and that it opens up new possiblities in the expression of translation algorithms.

This development will take place in three stages: the first (section 2.2.1) reviews the commonly accepted relationship between a tree representation of a program and its context free syntax; the next (section 2.2.2) reviews the disambiguating effect of representing syntactic structure by means of a tree, and the concept of abstract syntax; finally, section 2.2.3 develops a directed graph representation of context sensitive syntax and places this in its historical perspective.

2.2.1 Representing context free syntactic structure as a tree

The representation of syntactic structure as a tree goes back at least as far as Chomsky's work on phrase structure grammars. In a paper on formal properties [24] he describes the relationship between a derivation of a string with respect to a context sensitive grammar, and a hierarchical representation of its phrase structure (a tree).

We will concern ourselves here with context free grammars, because it is these that lend themselves to automatic parsing techniques. The following discussion is based in part on Aho and Ullman [3], pp 129-133.

Consider a grammar for arithmetic expressions:

```
E ::= E+E  |  ExE  |  (E)  |  -E  |  id                    [G2]
      (1)     (2)     (3)     (4)    (5)
```

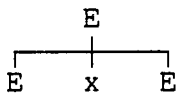
A leftmost derivation of the string "-(id+id)" is:

```
E  =>  -E  =>  -(E)  =>  -(E+E)  =>  -(id+E)  =>  -(id+id)
   (4)     (3)      (1)         (5)          (5)
```

(This derivation is leftmost because at each stage the leftmost nonterminal is replaced.) Each rule is marked with a parenthesized number (e.g., E ::= (E) is rule 3) and each step in the derivation is marked with the number of the rule that was applied.

Every rule in a grammar may be represented by a tree showing the hierarchical relationship between the left and right

hand sides. For example,



is a representation of the rule "E ::= E x E". Reading the leaves of this tree from left to right will reconstruct the right hand side of the rule, and is called a sentential form. Both of the nonterminals E that are leaves of this tree may be roots of trees themselves.

In general, in a derivation

$$E \Rightarrow A_1 \Rightarrow \ldots \Rightarrow A_n$$

where each A_i is a sentential form, and A_n is the terminal string, each step in the derivation

$$A_i \Rightarrow A_{i+1}$$

represents a rewriting of a string of (terminal and nonterminal) symbols

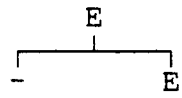
$$l_1 l_2 \ldots l_u E r_1 r_2 \ldots r_w \Rightarrow l_1 l_2 \ldots l_u m_1 m_2 \ldots m_v r_1 r_2 \ldots r_w$$

by application of a rule

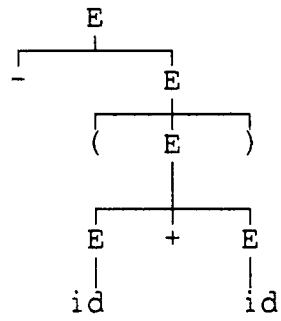
$$E ::= m_1 m_2 \ldots m_v$$

If the sentential form A_i is represented by the leaves of a partial parse tree T_i, then we can create T_{i+1}, representing sentential form A_{i+1}, by replacing the leaf representing the symbol E by the tree representing the rule. This process begins

with a single root/leaf node E; so in the above example, after application of rule 4 we have the tree



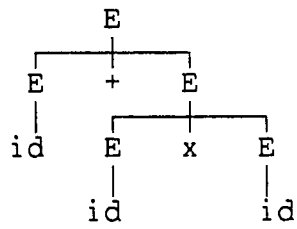
and so on, until we have the tree



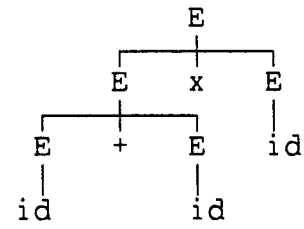
[T3]

(In a bottom-up parse the sentential form is gradually reduced to a single symbol. Obviously it is possible to form the parse tree from such a parse as well: here, instead of a single incomplete tree representing a sentential form of the string at some stage by its leaves, there is a set of trees -- usually on a stack -- with the final string at their leaves, and the sentential form reconstructable from their root nodes.)

The above example displays no ambiguity; nevertheless, the grammar is ambiguous and, for example, the string "id+idxid" has two leftmost derivations, imposing different interpretations on the order of evaluation:



(a)



(b)

[T4]

It is in examples of this sort that we can see most clearly the importance of a sense of syntactic structure. The string "id+idxid" does not in itself indicate the order of evaluation: we must impose a rule like "multiply before adding" (interpretation a), or "evaluate from left to right" (b). The tree representations, on the other hand, are unambiguous. In the next section, we will pursue this disambiguating effect of tree structure further.

2.2.2 Abstract syntax and abstract syntax trees

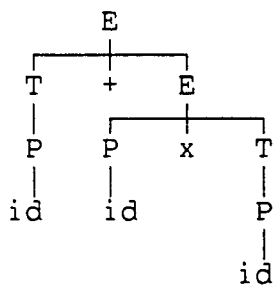
We have seen how a tree may be used to represent the syntactic structure of a string, and how a tree representation may be constructed directly from the string's derivation. For the trees in the preceding section, the converse is obviously also true: the original string may be read off from the leaves of the syntax tree, and the derivation may be reconstructed from the structure of the tree.

We have also noted that a tree represents the syntactic structure of a string unambiguously, even though the underlying grammar may be ambiguous. This is because a tree always

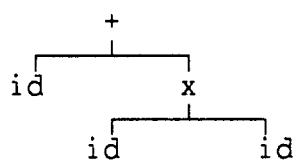
represents one or another of a number of possible derivations. This fact has been used to advantage in syntax directed translations to reduce the size of the semantic portion of the specification. For example, if the above grammar G2 were to be rewritten as an unambiguous grammar, it might take the form

```
E ::= T | T + E
T ::= P | P x T
P ::= id | -P | (E)
```

In this grammar, there is only one derivation for the string "id+idxid"; on the other hand, its parse tree now looks like

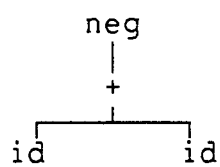


The size of the grammar has gone from five rules to seven, and the parse tree from ten nodes to twelve. In fact, the expression could be represented by the tree



[T5]

This tree contains only five symbols -- exactly as many as in the original string -- but gives unambiguous form to the string. The expression "-(id+id)" can be represented as



[T6]

Here, only the four important symbols are represented: the

parentheses, which serve only to disambiguate the string and have no value in the tree representation, are removed altogether.

Trees T5 and T6 are said to represent the abstract syntax of the strings "id+idxid" and "-(id+id)": while it is possible to give a procedure for the reconstruction of the string and its derivation from the tree, it is no longer a simple universal statement like "read the leaves from left to right," but must now be given anew for every grammar.

2.2.3 Representing the full context sensitive aspects of programming languages as graphs

So far in section 2.2 we have seen that context free syntactic structure may be represented unambiguously and, with the introduction of the idea of abstract syntax, parsimoniously in the form of trees. In section 2.1, however, we cited material which establishes that context free syntax cannot entirely describe programming languages, and therefore we would like to find a representation of the complete syntax of programming languages with the same qualities of being both parsimonious and unambiguous.

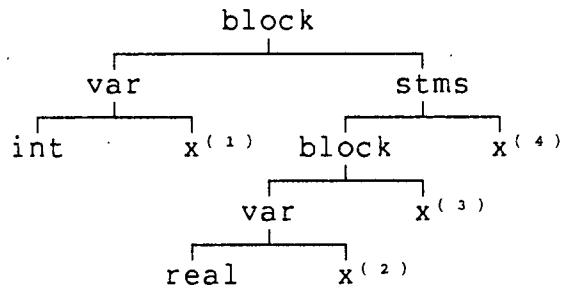
Just as grammars capable of generating programming languages exactly need be only slight extensions of context free grammars,[15] so it is possible to extend tree structure slightly

in order to represent the context sensitive syntactic structure of programs.

Consider the program illustrating block structure in section 2.1.2:

begin int $x^{(1)}$; begin real $x^{(2)}$; $x^{(3)}$ end ; $x^{(4)}$ end [P3]

An abstract context free tree representation might be:

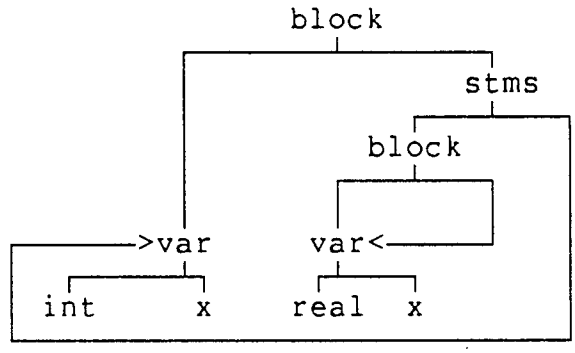


[T7]

(The actual variables in this program are all named "x"; the superscripts are purely a notational device to distinguish the four occurrences.) A symbol table, such as the one illustrated in section 2.1.2, would make an identification of symbol $x^{(3)}$ with $x^{(2)}$ and of $x^{(4)}$ with $x^{(1)}$. Such an identification can be made, entirely within the graphical representation of the program, and without resorting to a symbol table, by treating it as a more general data structure than a tree. For example, the following data structure is an abstract representation of this contextual relationship:

---

[15] See section 2.1.2, where it was argued that all the common methods for dealing with the context sensitive restrictions on programming languages essentially introduce a symbol table into the parse or derivation.



[T8]

The representation of program P3 by graph T8, above, has this advantage over representation T7, that it leaves no doubt about the types of the two symbols $x^{(3)}$, $x^{(4)}$ appearing in statements. An interpreter, making a left to right tour[16] of T8 would encounter each definition of x twice -- for a total of four encounters: exactly as many as in a tour of T7, because the hierarchical structure of the program representation remains the same.

The idea of representing context sensitive syntax structure as a directed graph is not new: I have already mentioned Chomsky's representation for derivations with respect to context sensitive grammars; both Woods [143] and Baker [8] have used similar representations for derivations with repect to type 0 grammars. In all these cases, the structure captured is that of the parse, however, not of the string being parsed. Context sensitive grammars can be used to restrict the relationships between definitions and actual uses of symbols to just those commonly accepted in programming languages; but inevitably they

---

[16] Definitions of tree, graph, and left to right tour follow in Chapter 3.

do so by a great deal of string manipulation in which nonterminal symbols are shuffled back and forth. In the directed graph representation of such derivations, the structure of the underlying string is lost.

Compare, for example, the W-grammar derivation of an Asple program in Marcotty, Ledgard and Bochmann's survey ([87], pp 199-215) with its context free derivation (p 195). The sample program should be small; say,

begin int x; x:=1 end

Draw a Chomsky-style derivation graph of the former, and a simple parse tree of the latter, if necessary; the difference should become apparent before too long -- as should the reason this exercise is left to the reader and not performed here.

It is one of the virtues of context free derivations that the grammar imposes a hierarchical structure on the language. This hierarchical structure is directly derived from the grammar, and is exploitable by syntax directed translation schemes in that it permits the recursive definition of the translation to be directly associated with the inherently recursive definition of the grammar. The graph representations defined above combine this virtue, and the economical definition of translation it makes possible, with the virtue of representing unambiguously the relationship between an actual occurrence of a symbol and its definition: a relationship that is not defined in pure context free representations except by

recourse to a symbol table.

The view of translation presented in the next section, in which translation of programs is treated as a transformation of syntactic structures, gains in clarity by being defined over syntax graphs, rather than syntax trees. In particular, although none of the ability of syntax trees to represent hierarchical structure is lost, the effect of merging the formerly separate notion of a symbol table with the syntactic representation is to simplify the definition of translation: instead of two concepts, there is now only one to deal with.

2.3 Graph transformations

Thus far in this chapter we have found that the traditional treatment of programming language syntax as context free led to a highly successful model for translation which has in recent years reached the limits of its competence. In particular, we have seen that, while the model gives a clear expression of, and workable basis for, the recursive translation of programming language control constructs, such as the begin-end block structure and the if-then-else statement, it fails to account for, and give adequate expression of, the context sensitive nature of the use of symbols in programming languages; and that, in the light of current trends in language design, such inadequacies must sooner or later result in a critical re-examination of the context free model. It is arguable that

this process is already taking place, where the currently most successful extension of the syntax directed translation model is embodied in various kinds of augmented grammars (see Koster [74], and Bochman and Ward [16], for descriptions of translator writing systems based on such grammars).

To recapitulate briefly, a syntax directed translation scheme imposes a static structure on the program under translation. This structure is related to the context free grammar defining the language whether directly, by attaching specific semantic actions to each production rule, or indirectly, by structuring the translator according to an abstract syntax derived from the grammar. The translation process may be thought of as a tree tour of this structure, whether it is implemented as such (as in Abramson et al. [1] or Crowe [29]) or, as is more common, it is merely a conceptual aid to the translator writer. During this tour, code may be emitted which represents the translation of the program.

Nevertheless, it is not uncommon for abstract discussions of translation to take a more dynamic view of the program. McKeeman ("The translation problem is...to take a given source text and produce an equivalent target text") defines a set of eight "feedback free" transformations which make up the translation process for the average programming language ([85], pp 15-19). Rosen has developed a theory of tree transformation systems analogous to the string transformation systems

represented by Chomsky-style grammars [104]; this theory was applied to translation by DeRemer [33]. Rosen himself has extended his theory to graph transformation systems, and has suggested applications in proofs of optimizer correctness and in the definition of interpreters [105]. Discussions of optimization algorithms, too, are often couched in terms of improvement transformations on graphs (Aho and Ullman [3], pp 418-438; Gries [53], pp 396-406).

It is the intent of this dissertation to take such a dynamic, transformational view of translation in all its aspects. In the next chapter it will be demonstrated that it is both possible and profitable to treat programs as syntactically structured sequences of symbols in a uniform manner, and to express various aspects of translation as transformations of this syntactic structure. It will be seen that a wide range of translation related objectives, such as language extension, abstraction, optimization, and error handling and recovery may be treated in a clear, expressive manner when presented in this form, whether in an abstract discussion or in the actual implementation of a translator.

In doing so we will be treating the syntactic structure of programs, in the fashion of section 2.2.3, as generally connected list structures here called graphs. We will see that such a treatment -- which represents those aspects of programming language syntax normally represented, in

conventional translation techniques, by a symbol table, as an integral part of the syntactic structure -- is necessary, once the static conception of syntactic structure is abandoned, because the initial relationship between the table and the internal representation of the program cannot be guaranteed to continue unchanged throughout the translation. This reduction of two concepts into one will also be seen to have the usual effect of simplifying the expression of algorithms.

Chapter 3: The expression of translation algorithms as graph transformations

This chapter is intended to examine in some detail the ways in which translation, in all its aspects, may be expressed as transformations on a graph representation of program structure. In order to do this, we shall need a clear conception of what a graph representation of a program is and precisely what we are about when we talk of transforming it. The term graph and the concept of graph transformation will be defined here; we will examine graph pattern matching in the chapter on implementation which follows.

Let N be the set of counting numbers.

$N = \{1, 2, 3, \ldots\}$

Definition (state)

A state is a pair (A,R) where

(1) A is a set of nodes, A is a finite subset of N, and

(2) R is a function, $R: A \times N \rightarrow A \cup \{0\}$, as follows:

$$\forall k \in A \; \exists n_k \in N \text{ such that } \forall n' \in N$$
$$R(k,n') \in A \text{ if } n' \leq n_k$$
$$= 0 \text{ if } n' > n_k$$

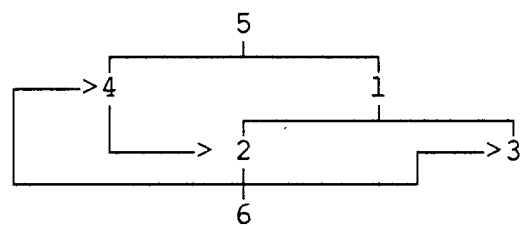
Remarks

For $x,y \in A$, and $n \in N$, $R(x,n) = y$ means that the n'th arc

out of node x is directed at node y, while R(x,n) = 0 means there is no n'th arc out of x. A state, it should be easy to see, is very close to what a graph theorist would call a graph,[1] except that the arcs leaving a node (the out arcs) are ordered by the function R. The definition of state and the following definitions which depend on it are intended not only to give an unambiguous definition of graph transformation, but also to model reasonably realistically an implementation of a graph transformation language. It is convenient to treat the state as a model for some portion of computer memory set aside to contain syntax graphs, to treat syntax graphs as connected subportions of this state, and to treat graph transformations as operations on the state rather than on individual component graphs.

Example (state)

The picture



represents a state defined by the pair (A,R) where

---

[1] Compare Aho and Ullman [2], p 41.

```
A = {1,2,3,4,5,6}

R : (5,1) -> 4        (5,2) -> 1
    (4,1) -> 2
    (1,1) -> 2        (1,2) -> 3
    (2,1) -> 4        (2,2) -> 6        (2,3) -> 3
    all others -> 0
```

Remarks

It is often convenient to label the nodes of a state with symbols that indicate their "meaning". In such cases we may think of a state as a 4-tuple (A,R,S,L) where S is a set of symbols and L is a function L: A->S .

It is entirely possible that, for q,r ∈ A, q≠r and L(q)=L(r). This will frequently be the case in states representing syntactic objects, where nodes will represent instances of syntactic classes, and there will be many such instances in a program.

The presence or absence of S,L in the state is not material to the present discussion and will in fact tend to clutter up the definitions and examples. The correspondence should be obvious and, where unspecified, we can assume that S=A and ∀a∈A, L(a)=a .

Definition (child, parent)

If S=(A,R) is a state, x,y∈A, then y is a <u>child</u> of x (with respect to S) if and only if ∃n∈N such that R(x,n) = y .

Reciprocally, x is a parent of y (with respect to S) if and only if y is a child of x (with respect to S).

Remark

It is possible that an arc is formed from a node to itself, in which case, the node will be its own child/parent. Similarly, if a pair of nodes each have an arc directed at the other, each is child/parent to the other.

Definition (descendants, ancestors)

If $S=(A,R)$ is a state, $x,y \in A$, then y is a descendant of x (with respect to S) if and only if

(a) $x = y$, or

(b) $\exists z \in A$ such that z is a child of x and y is a descendant of z.

Reciprocally, x is an ancestor of y (with respect to S) if and only if y is a descendant of x (with respect to S).

Remark

In general, there will be only one state in force at any time, and so it is obvious which state we mean when we say "y is a child of x with respect to S", etc. In such cases, it will be sufficient to say "y is a child of x", etc.

Example

In the preceding example, node 4 is a child of node 5; node 3 is a child of both node 1 and node 2. Notice that nodes 2 and 4 are each a "child" of the other.

Definition (the graph defined by a node)

If (A,R) is a state, r∈A, then (A',R',r) is the graph defined by r if and only if

(1) ∀x∈A,

x is in A' if and only if x is a descendant of r

(2) ∀x∈A, and ∀n∈N,

R'(x,n) = R(x,n) if and only if x∈A'

Remarks

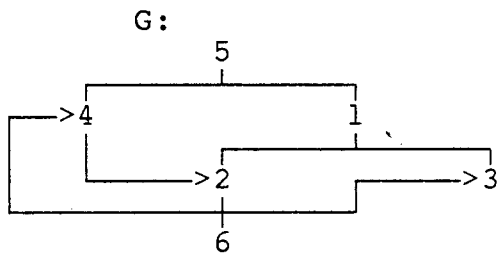
This says that any node from a state may be used to define a graph, that the graph will contain all the nodes reachable from that root node, and that it will contain all the arcs connecting nodes in the graph. In more standard terminology, the graph defined by r is the subgraph induced on the set of descendants of r.

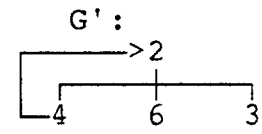
It is entirely possible that a graph will have several roots: that is, that there exist G'=(A',R',r') where r≠r'. In such cases, the graph necessarily contains a cycle, of which r and r' are part, since by definition r is a descendant of r', r' is a descendant of r, and so r is a descendant of itself.

A DAG (directed acyclic graph), therefore, is a graph in which no node is its own descendant (equivalently, ancestor). A tree is a DAG in which no node has more than one parent (to be a graph in this model, all but the root node must have one parent).

Example

G as in the previous example, G' the graph defined by node 2:





A'={2,3,4,6}
R': (2,1) -> 4 (2,2) -> 6
(2,3) -> 3. (4,1) -> 2

In state G of the above example, both nodes 2 and 4 define the graph G'; the node 3 defines a trivial graph consisting of itself.

We can now define left to right tours of graphs as follows.

Definition (left to right tour)

If G = (A,R,r) is a graph, f is a function taking arguments from A, then, given the following algorithm,

```
left-to-right (x,i):
  {if i=1 then f(x);}
  if R(x,i) ≠ 0 then
     left-to-right(R(x,i), 1);
     left-to-right(x, i+1)
  {else f(x)}
```

evaluation of left-to-right (r,1) will result in a left to right tour of G. The two optional calls of f(x), only one of which should be present, will result in a preorder and a postorder tour, respectively, of the nodes of G.

Remarks

Where r defines a tree, left-to-right(r,1) is guaranteed to visit every node in the tree exactly once. Where r defines a DAG, every arc is traversed exactly once.

In the example T8 on page 46, a node labelled x is visited once for every occurrence of identifier x in the program represented by the DAG, although each such node will be visited several times: once for its use in the definition of a distinct variable x, and once for each use referring to that definition.

If r defines a graph with cycles, the algorithm will not terminate: since graph pattern matching involves graph traversal, the question of termination in pattern matching over graphs must be addressed. This will be done in section

4.2.2, where general graph tours, for the purposes of syntactic pattern matching, are shown to be no more complex than DAG tours.

We are now ready to consider graph transformations. They are called graph transformations, because the way we will be using them in the remainder of this chapter will involve the alteration of individual graphs, where a graph is, as defined above, some node and the collection of arcs and nodes "dangling" from it. However, it will be convenient to define a graph transformation as a state transformation.[2] Individual graphs, defined by nodes in the state, will have been changed, as a result of a transformation, into differently "shaped" graphs in the resulting state.

Definition (graph transformation)

Let S = (A,R) be a state, r,r'∈A; then the operation

$$r \Rightarrow r'$$

has the effect of transforming S into a new state S' = (A,R') where

$$\forall x \in A, \text{ and } \forall n \in N,$$
$$R'(x,n) = R(x,n) \text{ if } R(x,n) \neq r$$
$$= r' \text{ if } R(x,n) = r$$

---

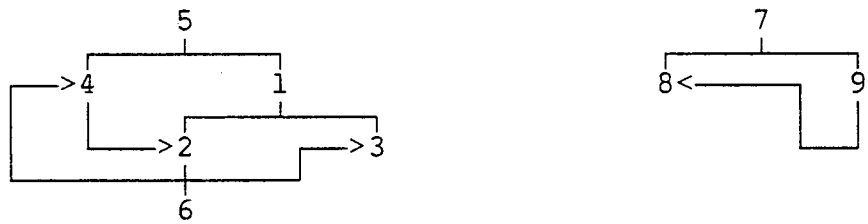
[2] This is analogous to the state transforming properties of ordinary assignment in programming languages: compare Floyd [42] and Hoare [59].

Remarks

The operation of graph transformation is defined here in terms of nodes r,r' in state S. However, since any node in a state is the root of some graph, we are really discussing the transformation of all graphs containing as subgraph the graph defined by r into new graphs wherein the graph defined by r' now occupies the (syntactic) position previously occupied by the graph defined by r.

Example

S = (A,R)



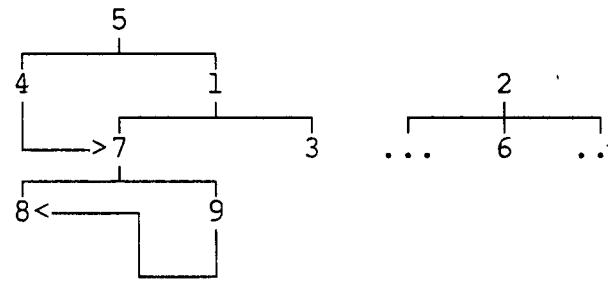
A = {1,2,3,4,5,6,7,8,9}

R : (1,1) -> 2 (1,2) -> 3
(2,1) -> 4 (2,2) -> 6 (2,3) -> 3
(4,1) -> 2
(5,1) -> 4 (5,2) -> 1
(7,1) -> 8 (7,2) -> 9
(9,1) -> 8
all others -> 0

The transformation 2 => 7 has the effect of transforming S into S' = (A,R') such that

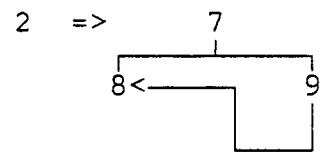
R' : (1,1) -> 7 (4,1) -> 7
all others as for R

S'=(A,R'):

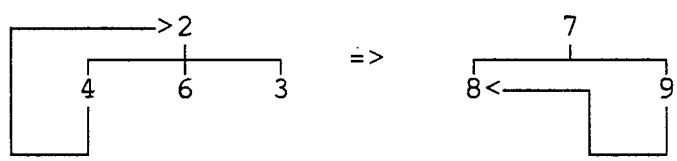


Discussion

Since the ensuing material will treat graphs that will be only slight variations of trees, and in particular graphs that will be rooted at a specific node, a transformation like "2 => 7" will be conveniently represented by the actual graphs being transformed. For example,



Since a transformation will often involve a pattern match, it is more often meaningful to represent as much of the graph being transformed -- possibly the whole subgraph defined by the node being transformed -- as is necessary to specify the pattern match that has taken place. For example,



In all representations of transformations it will be clear which are the root nodes and what the hierarchical relationship between the nodes is, either by placing children directly below parents (as 8 and 9 are placed below 7), or by

using arrows to indicate the relationship parent-->child (as 9-->8, and 4-->2).

In much of the material on graph transformation, we will want to put a graph pattern on the left of the transforming arrow, with variables at some of its leaf nodes to represent entire, but unspecified subgraphs. These variables may then reappear on the right, and it is to be understood that they represent the same subgraph on the right as on the left, much as variables represent the same value on either side of an equation. Graph patterns will allow us to designate entire classes of transformations -- instead of merely individual transformations such as in the above example; they will eventually form a part of a graph transformation programming language, in a manner similar to the use of variables in conventional programming to designate entire classes of values and manipulations of these classes, instead of merely individual values (constants).

Variables in patterns will be represented as italic (underlined) symbols, as follows:

Definition (graph patterns)

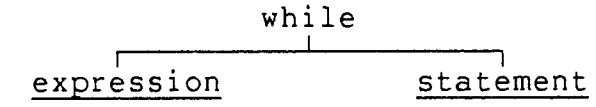
Let P = (A,R,S,L) be a state, S = V U C where V is a set of variable (italic) symbols, and C a set of constant (roman) symbols. If

L : A -> S

is a function which labels the nodes of A in such a way that ∀a∈A, L(a)∈V only if a has no children, then a graph pattern is any graph G=(A',R',r) such that r∈A and ∃a∈A' such that L(a)∈V.

Example

G:



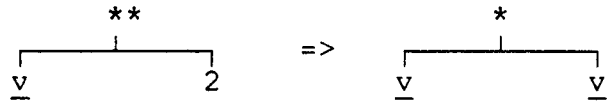
G = (A,R,1) V = {expression , statement}
A = {1,2,3} C = {while}
R : (1,1) -> 2 (1,2) -> 3
L : 1 -> while 2 -> expression 3 -> statement

notice that neither node 2 nor node 3 has children in P

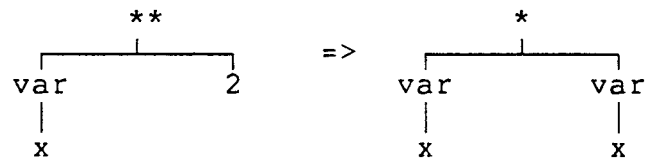
Discussion

In a programming language setting, the above example represents the abstract syntax of the class of "while" statements. In a pattern match (see section 4.2.2) G may be matched successfully against any graph G' whose root node is labelled "while" and has exactly two children. After such a match, the subgraph of G' defined by the left child of its root node will be associated with (assigned to) the variable expression, while the subgraph defined by the right child will be associated with the variable statement.

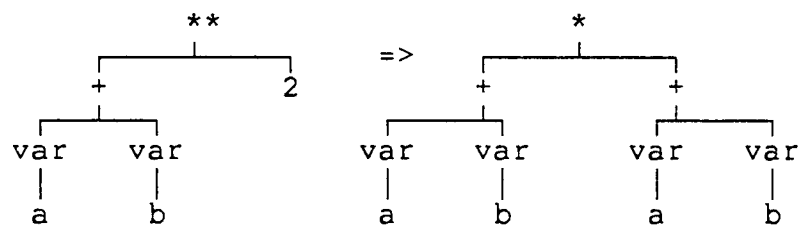
The discussion of graph transformation presented in this chapter will often describe transformations in this form:



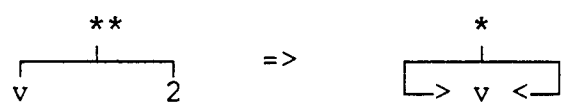
(a diagram intended to represent the transformation of any expression "v**2" into "v*v", an example of operator strength reduction). It is intended to represent a class of transformations such that, if a particular subgraph is substituted for every occurrence of a particular variable symbol (in this case, v) throughout the diagram, a particular instance of the transformation will have been generated. For example,



but also



(x**2 => x*x ; (a+b)**2 => (a+b)*(a+b)). Notice that, since no copies are made according to the definition, only substitutions, the transformation pattern



is entirely equivalent, and perhaps gives a clearer picture of the resulting structure of the graph representation of the program fragment.

This chapter is divided into four main topics: section 3.1 treats plain translation, that is, the re-expression of a program written in one language as an equivalent program in another language; section 3.2 treats optimization in a variety of forms; section 3.3 treats error handling; and section 3.4 shows that a graph transformation treatment is not out of keeping with recent developments in programming language research, so that it is not only a uniform treatment for traditional translation concerns, but promises to continue to be applicable at least into the foreseeable future.

3.1 Translation

In this section, the discussion is intended to encompass those aspects of translation that may be thought of as the re-expression of a program written in one language as an equivalent program in another. Traditionally, compilation is the most common form of translation, primarily because it is the most useful: in almost all cases, the ultimate destination of a program is to be run on a machine, and this is possible only if the program is compiled -- that is, if the program is translated into machine code, specifically in the form of a relocatable object module. But we will also consider, as special cases of this process, language extension, where a programming language includes some facility for expressing sophisticated constructs in terms of base concepts, and other translation techniques where the object "language" is not machine code but some other

programming language, or an intermediate form between programming language and machine code (an abstract language).

This section is divided, primarily for convenience, into three parts which may be thought of as treating distinct aspects of translation, but in fact will display so many cross-currents, that they form a largely unified whole. This lack of clear separation between these aspects is, indeed, a function of the treatment given them here in terms of syntactic transformations, and is one more argument in favor of a uniform treatment such as this; for where apparently distinct concepts can be shown to be related in a specific way, a simplification has occurred which allows us to save time and may, in the long run, provide a basis for other, possibly more profound insights. In fact, there will be cross-currents with other sections as well; particularly with section 3.2, where many program improvements will be seen in the same light that illuminates translations here.

3.1.1 Language extension

Simpler than compilation and related translations, and therefore an easier introduction to the topic, are those aspects of programming languages that give the programmer the ability to amplify the expressive power of the language.

The notion of a macro instruction -- a programmer defined instruction expressed in terms of a combination of more

primitive instructions -- must have appeared early and quite anonymously in several places at once. It is a simple labor saving device which permits the programmer to write a commonly used sequence of instructions once and subsequently to invoke this sequence by name. The concept was first discussed within a consistent, general framework by McIlroy in 1960, and it was specifically recommended by him as a useful concept in high-level programming languages [82]. In fact, Algol 58 (or IAL) had an inchoate macro facility in the do statement, which was defined as a textual substitution of the statement body wherever its name appears in a do ([96], p 16).

Early general purpose macro processors, standing on their own without a specific host language, were defined by Strachey [120] and Mooers [89]. These were intended as a program preprocessor (Mooers), and as an implementation language for an intermediate level abstract language (Strachey: we will have cause to examine this idea more closely in section 3.1.3), but have been used for everything from a semantics implementation language [1] to a text processing language in a word processing system [128]. Macro processors like Mooers' TRAC are perhaps too primitive to add any genuine convenience to programming, especially because they are usable only as preprocessors to a compiler. In both TRAC and Strachey's GPM, the macro is invoked as a bracketed sequence of strings; for example,

```
#(AD, #(N), 2)
```

(an example from TRAC, adding 2 to the quantity previously

defined as N and delivering the result as its value). More sophisticated macro processing languages, which attempt to emulate the Algol family model of statement form, were defined by Leavenworth [75] and Brown [18]. These fall short of a context-free syntactic form, but will deal with recursive structures of the form

```
WHILE a DO b OD
IF a THEN b ELSE c FI
```

(note closing "parentheses" OD and FI). Attempts to use languages of this form as a basis for defining compilers continue to be made (recent examples are suggestions by Tanenbaum [122] and Sassa [111]), but restrictions on the syntax of macro templates, such as the requirement that statements be closed with parenthetic markers, place additional restraints on the use of these processors, on top of the usual limitations of context free syntax in dealing with identifiers.

A recent use of a one-language oriented "macro" processor is Kernighan's Algol 60-like extension of Fortran syntax through the Ratfor preprocessor [68].[3] Macro processors have been used successfully in porting software, and have been applied systematically in this context by Waite et al. [134,135]. The success of macro processors in this kind of software tool application stems in part from its being aimed at a small group of sophisticated users, primarily in one-shot programming

---

[3] See also Hanson [56], for an evaluation of the adaptability of this concept to a rationalized SNOBOL preprocessor.

efforts such as language ports, and in part from restrictions on the range of the processor's application, primarily to a specific language. Waite's group furthermore imposed a line-oriented form on the macro statements, which saves their processor the effort of a complete context free parse.

Languages specifically designed to be extensible bear a significant resemblance to macro processors. Algol 68 permits the programmer to define new operators, but not new statement types. Garwick's GPL takes the Algol 68 concept a step further to include ML/I style "parenthesized" statement forms [46]. Fraley's "unlanguages" [44], Basili's "families" of languages [9], and the "layers" of language of Geiselbrechter et al. [47], are also formulations of the concept of extensibility. The idea of abstraction as developed by Wulf and others, primarily as a basis for stratifying proofs of program correctness, incorporates many of these ideas in a uniform manner and will be discussed, in a more appropriate context, in section 3.4.

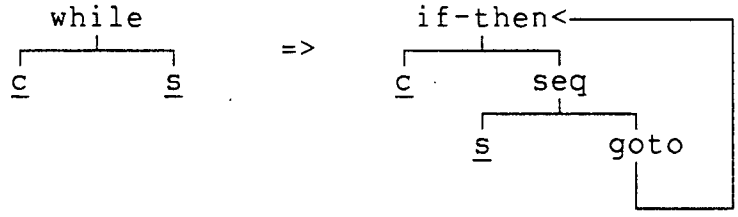
To place macro processing and related concepts in the context of translation through syntactic transformation, let us consider a syntactic macro facility for programming languages along the lines of Leavenworth's proposal. Such a macro facility, embedded in a simple programming language, would allow us to write programs of the form:

```
begin

  macro
    while a do b od                                    [P4]
  is
    L: if a then b; goto L fi
    mend;

  while x>y do x := x-n od
  end
```

We expect this program to have the same effect as the program

```
begin
  L: if x>y then x := x-n; goto L fi
  end
```

In effect, the macro definition has added to the translator a transformation rule of the form

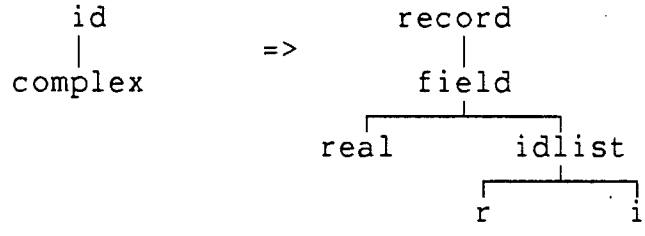


[T9]

We can think of all varieties of macro substitution in this way. To take a concrete example, Pascal's type definition facility (a useful concept first introduced systematically in Algol 68, and promised longevity by, for instance, its recent inclusion in the design of Ada) may be seen in the same light. A definition of the form

```
type complex = record r,i : real end
```

is tantamount to declaring a transformation rule like



This use of "simplifying" transformations to change a high level program into an equivalent program containing only primitive or low level concepts will be seen again in section 3.1.3, where it is shown to lie at the back of many schemes for implementing translators, and in section 3.4, where it is used to implement the concept of abstraction. The use of macro processors and their place in programming languages is described by Cole [25]; and a bibliography so structured as to emphasize the relationship between macro processing and language extension was recently published by Metzner [88].

3.1.2 Language-to-language translation

In a sense the title of this section is too inclusive: every translation is from one language to another, perhaps abstract, perhaps restricted form of the same language. In another sense it is too exclusive: if we take "language" to denote an actual programming language, then not only are abstract intermediate forms, which will be deferred until the next section, excluded, but so are the relocatable object modules generated by compilers, which are to be considered as one major aspect of compiling, in this section. However, as was already indicated, the dividing line between these topics is extremely flexible: it is difficult to decide whether Ratfor, for example, represents a language extension mechanism, a new language which is defined in terms of, and translated into, Fortran, or a basis for highly portable programs; in practice,

it is a little of all three, and ought to be mentioned in all three sections.

In this section, we will consider primarily compilation. As a simplifying assumption, we will make no distinction between translations which result in a relocatable object module (true compilation) and those which produce assembly language statements or even high level language output -- as frequently happens in the case of experimental implementations to try out the still flexible design of a new language. Just as this chapter does not concern itself with the exact method by which a string of text is turned into a graph representation of the program's syntactic structure, so it does not concern itself with the exact method whereby a translation is made to conform to the characteristics of any particular machine or operating system.

An increasingly important form of language-to-language translation is program porting. A program written in one language is to be implemented on a machine supporting only another language: where the program is a compiler, written in the language it implements, the process of porting is known as bootstrapping. Since so many languages may be described as "Algol 60-like" and indeed differ from each other only in small but syntactically subtle aspects, such a translation is essentially a straightforward one, but it is likely to be frustrated by the line, and unstructured string, orientation of

most text editors and text processing languages. Even minor differences in language can make this a major task without highly sophisticated tools: Sabin notes his experience that porting Fortran programs between different machines is made nontrivial by widespread deviation from the standard, and frequent gaps in the standard; the best tools available to him were poorly suited to this task [108]. I can add my own experience to this; and indeed, it was a consideration of the unsuitablility of available tools -- on the one hand of file editors and on the other of translator writing systems -- to this straightforward, and extremely common language processing task that triggered the research being reported here.

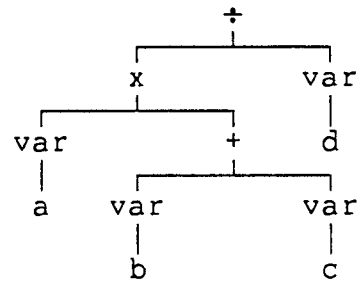
We will consider three specific problems in translation that are usually given considerable attention in textbooks: one is expression translation without optimization, one of the places where syntax directed translation is at its most applicable, another is variable translation, and the third is control flow expression translation.

(a) Simple expression translation

Consider an expression of the form

a x (b + c) ÷ d

which we will represent by a graph of the form

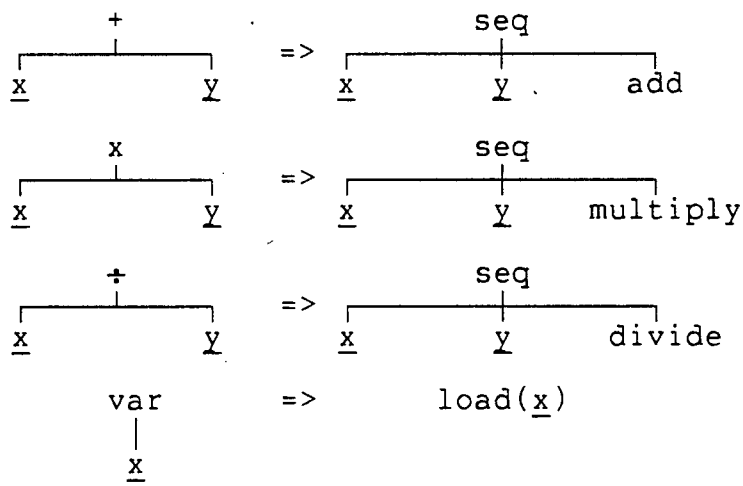


[T10]

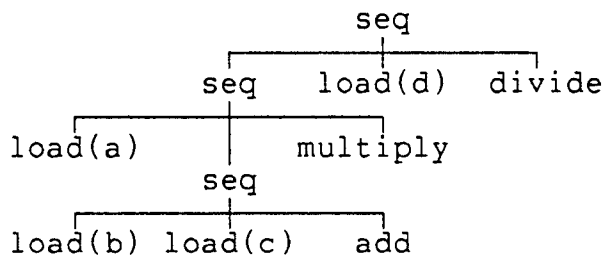
If we were translating to a stack machine, the object code, in symbolic form, would look something like this:

```
load a
load b
load c
add
multiply
load d
divide
```

Such a translation could be accomplished easily enough by a set of transformation rules of the form



Applying these rules, in any order, to T10 results in a graph of the form



A simple left to right tour over this graph, printing out the label of each leaf node, suffices to produce the assembly language form of the program.

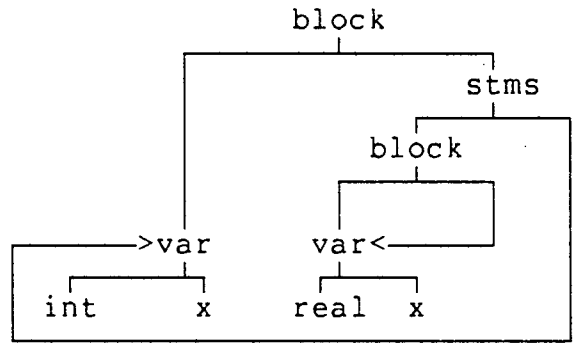
This much is pure syntax directed translation and, if the target machine does not operate on a stack model (few machines do), these instructions can be treated as macros and further amplified by the transformation mechanism demonstrated in section 3.1.1; they will not, in general, be very efficient in that form, though few compilers bother to do better, but we will leave for section 3.2.3 a discussion of how such translations can be improved.

(b) Variable translation

In the above example, we have left the variables a,b,c in their symbolic form. If the translation is to proceed to machine code form, symbols will have to become actual memory locations, or algorithms for accessing the stack. Consider the program example used to demonstrate nested definitions on page 31:

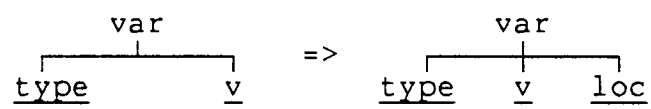
begin int x ; begin real x ; x end ; x end [P3]

This is represented as:



[T11]

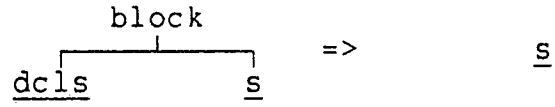
Without the complications created by procedure calls, we can choose simply to give each of these variables a unique memory location, and not bother with implementing a stack. In a tour of this structure, every time a block node is encountered, first the declarations and then the statements are processed; a global variable loc keeps track of the number of variables (memory locations) declared, and encountering a variable declaration triggers the transformation:



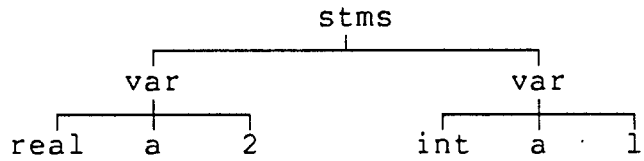
along with an updating of the value of the variable counter ("loc := loc + 1"). Notice that, in this respect, translation remains primarily syntax directed: it is convenient to translate the program in a left to right order, though not strictly necessary; in many cases, a strict ordering of some form, though not necessarily a left to right ordering, is crucial to a correct translation.

Since the block node's only function is to separate declarations from statements, we can also include the

transformation



The result of applying these transformations is the program



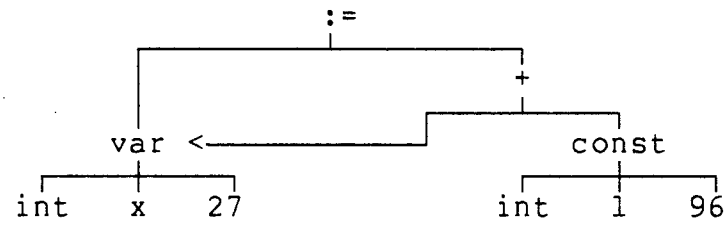
This corresponds exactly to the semantics of the original program: first evaluate the real variable (which was the second variable declared); then evaluate the int variable (which was declared first).

Notice the power of a graph representation of the program: processing the declarations, which consists in assigning each declared variable a location, had the global effect of allowing us to process the statements in the full knowledge that locations had been assigned to every variable appearing in them. This is normally accomplished by symbol table access, but here we have been able to discuss the process of variable translation without reference to any concept like symbol table, but purely in terms of graph transformations.

Consider now the translation of an assignment statement:

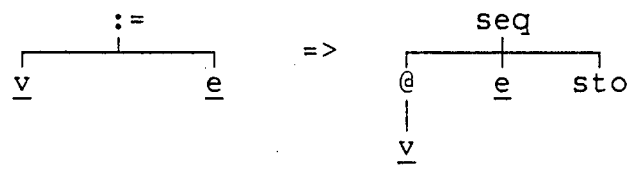
```
x := x + 1
```

which we will represent as a graph



Here, the declarations have already been processed, and the variable named x has been assigned location 27, while the constant with value 1 has been assigned location 96.[4]

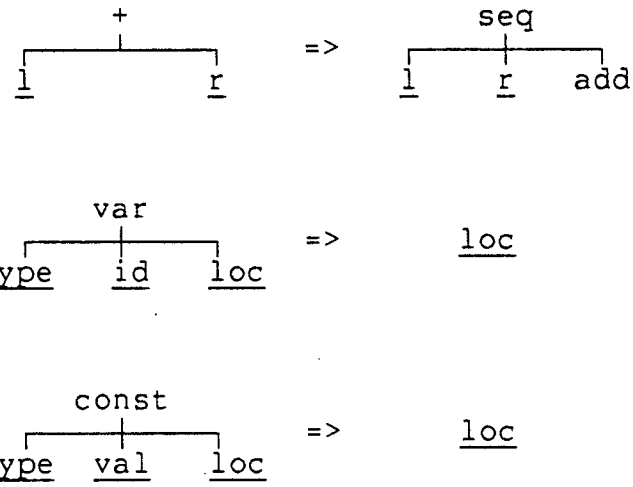
It is in the nature of the assignment operator not to treat its left and right hand sides alike: the variable on the left represents a location, whereas a variable on the right (even, as in this case, the same variable that occurred on the left) represents a value.[5] Therefore, in a tour of this graph, encountering the assignment operator will cause different rules to be applied to the left and right subtrees. We can express this as the set of rules:



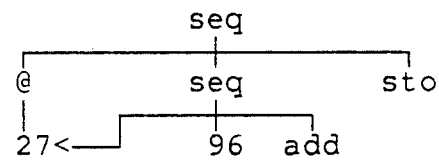
---

[4] Constants are not normally declared explicitly: because they are constants, their occurrence in the program is enough to declare implicitly their type and value. It is general practice, however, to treat all such implicit declarations like explicit declarations -- usually by recording them in a constant table analogous to a symbol table: this saves space for frequently occurring constants.

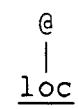
[5] Once again, it was Algol 68 which first made this distinction clear by treating assignment like an operator, and by treating operators as procedures. In the case of the integer assignment operator, the left operand is of type ref int (integer variable) whereas the right operand is of type int (integer value).



Applying these rules -- again, in any order -- transforms the original program graph into the graph



When we come to print this out in a sequential form, it is a simple matter to print:



as "lda loc", and a plain "loc" as "lod loc", which represent, respectively, loading the address loc and loading the value at address loc. A linear version of the program would then be:

```
lda 27    ; load location of x
lod 27    ; load value of x
lod 96    ; load constant 1
add       ; add x and 1
sto       ; store sum in location of x
```

(c) Statement translation

We have already seen the reduction of a statement to

simpler forms in the section on language extension. There, the while-do statement was re-expressed as an if-then-else and goto combination. This is the usual approach to translating from a high level language with structured statements whose control flow is implicit, to a low level language whose control flow is explicit in its jump commands.

It is noteworthy that, for example in the transformation on page 70, the use of a graph representation eliminates the need for an actual jump label. The advantage of this form of representation is quite obvious. Consider, for example, the expression

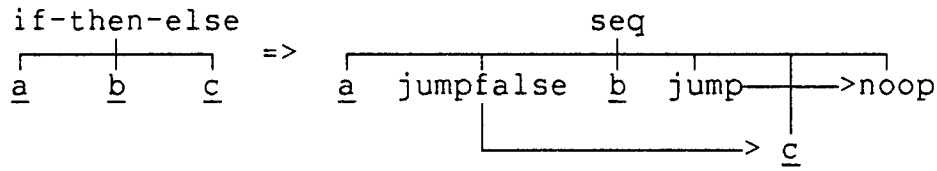
```
if a then b else c fi                                    [P5]
```

On any standard machine architecture, this must be reduced to something of the form

```
    a
    jumpfalse L1                                         [P6]
    b
    jump L2
L1:c
L2:noop
```

The sticky part of the above as a definition of a translation is the sudden appearance of the labels L1 and L2. We must answer the questions: Where do they come from? and, How can we be sure they do not occur elsewhere in the program? We would have to provide (and define) a mechanism for generating new labels that, like the use of symbol tables, is external to the translation process; and, if the translation is aimed at producing a relocatable object deck, these considerations become quite

extraneous. However, consider the transformation:



The object graph looks strikingly like a flowchart. We have answered the questions about labels L1 and L2 by removing them, and demonstrating graphically that the jumps are local and are not affected by jumps elsewhere in the program (this is a result of the definition of graph transformation presented at the beginning of this chapter). In a way, this elimination of labels is not surprising: formal treatments of program evaluation -- for example, Turing machines -- often treat programs like directed graphs whose nodes represent commands, and whose edges represent flow of control [8]; flow charts, a once popular form of program design methodology, were generally reduced to linear forms (Fortran programs, Cobol programs, etc.) by a liberal use of jump commands [28];[6] some formalizations of program control flow have depended heavily on directed graph representations to show the underlying "machine" structure.[7]

However, we have introduced a notation that is out of fashion in programming circles, and it needs an apology. For

---

[6] An interesting variant of this is the reduction of a structured Ratfor program to standard Fortran by the disciplined introduction of jumps [68].

[7] Greibach, for example, treats equivalence of program schemes (flowcharts) in terms of legal transformations turning one kind of scheme into another [52].

the duration of this chapter, we will have to put up with occasional "spaghetti" of this sort: in the next chapter we will return to the linear program form from which it derives, and will do so within a discipline that leaves well-defined the relationships between such program fragments and the larger program that contains them, and also defines the underlying structure that is being manipulated by a transformation such as that from P5 to P6. The "spaghetti" will be minimal, however, because we will be dealing with small program fragments in which jumps are localized, not, as is the case with flowcharts used in misguided program design disciplines, a complex network of jumps representing an entire program.

Let us now look at the common syntax directed translation technique used to translate to quadruples (a kind of abstract machine code used as an intermediate form for many syntax directed schemes). In such a scheme, semantic actions (programs) are attached to syntactic productions. For example, the rule defining the if-statement might look like this:

```
S ::= if B then S(1) else S(2)
        { gen( B );
          j1 := code ( jumpfalse, nil );
          gen( S(1) );
          j2 := code (jump, nil);
          l1 := gen( S(2) );
          l2 := code( noop );
          patch( j1, l1 );
          patch( j2, l2 ) }
```

Here, the procedure call "gen(X)" causes code to be generated for the syntactic unit X, and the procedure call "code(...)" generates an actual object code statement; both these procedures

return an address for the code generated. The procedure call "patch(S,L)" backpatches statement S, a jump statement, by filling it out with label L; statement S had been generated earlier with only a "nil" in place of its target label. (Compare this with Aho and Ullman [3], pp 273-280; and with Gries [53], pp 292-293.)

This process, too, creates a directed graph like the one generated by transformation T9 (p 70), but it does so by clumsily slipping in a patch at the end. For this translate-as-you-parse technique to work at all, the grammar must often be cluttered up with extra productions, or pulled out of shape (compare Aho and Ullman [3], p 277).

3.1.3 Translating in phases

Although the syntax directed translation model is primarily aimed at structuring one-pass translators, the concept of translating a programming language in several stages has a considerable history also, and is not entirely incompatible with the syntax directed approach.

The idea of an intermediate language, well suited to being at once the target language for high level language translations and a universal language for programming computers goes back to at least 1958, when initial efforts were made toward the definition of such a language, tentatively named Uncol

(Universal compiler oriented language, or, seen from the other end, Universal computer oriented language). Several years later the effort was floundering in the face of immense obstacles presented by the diversity in both computers and programming languages. Bagley, writing in 1962, suggests reasons why such a definition might be expected to fail to come into existence [7]. His analysis, seen after the fact, is accurate: Uncol, though often mentioned, and frequently "rediscovered" under some new name or formulation, has not and, it takes no prophet to see now, cannot exist in its original form.

However, the failure of Uncol is due only to its intended universality. The concept of an abstract intermediate language as a bridge support in a multi-phase translation can be found in so many and diverse translation efforts that it must be considered a successful model. We have already mentioned that a sequence of translations between such "languages" can be used as the basis for a discussion of the semi-independent portions of a translation [85].

Sklansky et al. present a formal view of program translation based on presenting the relationships of programming languages as a network, and treating translations as transductions from one node in the network to another [119]. Basili discusses a practical technique with much the same basis in presenting a family of programming languages which may be implemented by means of a bootstrap technique using source code

modification [9].

Similarly, uses for intermediate languages in specific translation efforts range from Strachey's CPL effort [120] through the Mobile Programming System of Waite et al. [135], and Richards' BCPL compiler [102,103], to the Pascal "P" compiler [94]. Most of these languages are highly machine-like, and it is interesting to note that many of them use macro processing techniques to implement the intermediate language on a target machine.

A review of experiences with intermediate languages modelling abstract machines, and the effectiveness of macro processors in transforming these into actual implementations, was presented by Newey et al. [93]; a later review by Elsworth looks at some Algol 68, Pascal, and BCPL intermediate languages [37].

Higher level intermediate languages have been a more recent development, including the contribution of JANUS from the Colorado group [26];[8] TCOL from Carnegie-Mellon's PQCC group [78,112];[9] and from Appelbe, who describes a three-stage

---

[8] See also a suggestion by Poole that program development/translation systems might use a mixture of levels or "hybrids" to greater effect than simplistic phased translations [100].

[9] See also a survey by Cattell, critically examining selected multi-pass translation efforts [21]; TCOL is not so much a single "language" as a flexible intermediate representation.

translation through a high level intermediate language, GRAIL, and a low level intermediate language, STACODE [5].

Where low level intermediate languages are aimed at a specific group of machines, usually by being themselves very low level machine models that ignore the more exotic (if useful) portions of most machines' instruction repertoire, high level intermediate languages are aimed at a specific group of languages by being an abstract (syntaxless) collection of concepts common to these languages. In both cases, attempts to be more inclusive will tend in the direction of Lisp-style "systems" by including more and more special purpose concepts that partly overlap one another, and generate unwieldy user manuals.

It is easy enough to see how staged translations may be treated as a sequence of transformations. For example, consider McKeeman's translation "steps" ([85], section 5):

```
source text => parse tree
            => abstract syntax tree          (1)
            => standard abstract tree        (2)
                 (reduction of language extensions:
                   cf. section 3.1.1)
            => attribute collected tree
                 (builds symbol table;
                   declarations pruned from tree)
            => attribute distributed tree    (3)
                 (symbol table distributed in tree)
            => sequential expression tree
                 (expressions turned into code)
            => sequential control tree       (4)
                 (control constructs turned into code)
           (=> target text)
```

This is a very fine distribution of function (as McKeeman

admits) for convenience in an abstract discussion. Undoubtedly a staged translator would implement this in four stages (marked above by bracketed numbers):

```
source text => abstract syntax tree          (1)
            => standard abstract tree        (2)
            => attribute distributed tree    (3)
            => sequential control tree       (4)
           (=> target text)
```

which we may think of as roughly equivalent to (1) context free parse, (2) language extension reduction, (3) context sensitive pass, and (4) semantic pass. The important point here is that a transformation is taking place at each stage, and it is notable that it is expressed in terms of a tree structure representing the program. To a great extent this insight is in keeping with developments in programming methodology in the last decade: multiple pass translations are more amenable to "structured" programming techniques than single pass efforts where all these functions (and McKeeman's analysis of the separability of these functions is an accurate one) must be performed at once.

We will look more closely at the amenability of a transformational view of translation to "structured" programming methodology in a concrete discussion about TWS design in section 4.2.4. It is sufficient to note here that the above transformations represent a macroscopic version of the individual construct transformations of the immediately preceding sections. Instead of transforming _while_ into _if/goto_, we are here transforming language A into language B -- yet the notion of transformation applies equally well at both levels.

We shall call these higher level transformations transductions.

3.2 Optimization

This section, like section 3.1, is divided into three parts. To some extent, this division is arbitrary: section 3.2.1 deals with improvements made at the source language level; section 3.2.2 presents the traditional representation of programs for purposes of optimization as a model consistent with the view that all translation depends on syntactic structuring; and section 3.2.3 deals with code improvements made during translation and after a target code has been produced. We can expect a certain amount of overlap between these sections, because improvement at one level inevitably will have something in common with improvement at another. At the same time, these three realms of discussion are traditionally distinct, and each has its special technical lore. That we will see a similarity between them is in no small measure due to the transformational point of view we will choose to take, and is analogous to the similarity we came to see, in section 3.1, between traditionally separate forms of translation techniques. In fact, these two sections will not have done all they were intended to do if they fail to leave the impression that there is not so great a difference after all between translation and optimization.

Optimization has never been a particularly glamorous part of the translation process, and there are several clear reasons

for this: (1) Simple optimizations seldom improve a program's running time significantly, while optimizations that do have a noticeable effect on running time often cost too much to apply to the majority of programs, and also cost more to implement than most translator development projects can afford to expend on them. (2) Programming languages were often designed, in the past especially, without a clear notion of what makes "good" programming, and this often results in programs that actually degrade or become incorrect under what had appeared to be safe optimizing transformations. (3) Machine design has always been such that considerable intuition is needed to translate high level programming concepts -- like array indexing, expression evaluation, procedure argument passing, etc. -- into efficient machine level code; and intuition is one thing when applied by an experienced system programmer working in machine code assembler, and quite another when it must be reduced to general principles by a compiler writer trying to incorporate code improvements into his program. It is the primary concern of the PQCC project [78], for example, to automate much of the optimization of languages in a general fashion, so that optimization will be as accessible in a translator writing tool as syntactic analysis is at present.

Many of these objections and their ramifications are disappearing, however, simply because of the continuing trend in programming languages of permitting increasingly abstract descriptions of processes and the data they operate on. Recent

language developments like Alphard and CLU permit the programmer himself to define an abstract domain within which program behavior may be expressed in a way that is at once clear to a human audience for purposes of communication, verification, or some combination of these, and localizes all questions of implementation in such a way that they need have no bearing on -- and therefore will not interfere with -- discussions about the program.

3.2.1 Source code optimizations

As programming languages become increasingly abstract -- that is, as they remove more and more the expression of algorithms from their execution -- automatic optimization at all levels becomes not merely a method for improving the implementation of algorithms, but crucial in making them work at all with reasonable speed.

Even relatively low level languages like Fortran have received considerable attention in the area of optimization, because the straightforward translation of array references in loops, for example, can increase the cost of a critical loop to the point where the whole computation becomes unfeasible [81]. Such early examples of languages with well developed abstraction mechanisms as Algol 68, Simula, Planner, and Smalltalk have been perceived too often as pipe-dreams primarily because their implementations were not always realistic in terms of execution

speed.

Knuth ([72], pp 280-291) discusses the improvement of programs based on abstract formulations of the algorithm by transforming high level constructs expressing recursion into lower level constructs expressing iteration. His intent is to show that, as a primitive concept, the goto lies at the back of even the most abstract control constructs, and that it can be considered, in this light, in a highly controlled fashion that takes away much of its sting. A similar formulation was under contemporaneous development by Burstall and Darlington [30,19]. More recently, these ideas have been taken up by others, concerned both with simplifying the process of proving programs correct -- either by automatic means or, more realistically, by means of the social processes that the science of algorithms inherits from mathematics -- and of writing programs at a very high level of abstraction. Wang [137], Wegbreit [138], Arsac [6], and Leverett et al. [78], are all concerned with improving the performance of abstract programs by giving well-understood transformation rules whereby they may be reduced to equivalent programs using more mechanistic or "concrete" control constructs.

In keeping with the method established in the preceding section, we will examine a simple example and its treatment in terms of syntactic transformation. We have, in fact, already seen one traditional optimization expressed as a graph

transformation: operator strength reduction (p 64).

Another improvement technique removes redundant computations. This generally requires an analysis of the flow structure of the program, as do other, more esoteric techniques. However, even on the statement level there are redundancies introduced for the sake of clarity or because of the nature of the algorithm. For instance, a computation involving arrays often produces a redundant computation:

```
A( i, j ) := A( i, j ) + C
```

translates, at a more primitive level, into ("@A" is an expression for "the address of A", while "!t" is and expression for "the location addressed by t"; "v(t)" is "the value addressed by t"):

```
@A + i*dl + j + 1 := v( @A + i*dl + j + 1 ) + C
```

which could be reduced to

```
 t := @A + i*dl + j + 1
!t := v( t ) + C
```

Naturally, the more dimensions in the array, the more complex the redundant expression becomes. Detection of redundant expressions is simplified by the use of a DAG construction (Aho and Ullman [3], pp 420-421, 425-426) which will be demonstrated in a detailed discussion later in this section.

Another valuable improvement, known as code motion, removes costly computations from the inner body of a loop, since loops, containing a small proportion of the code, generally account for

the greatest proportion of the running time. Again, this kind of optimization is traditionally performed with support from a flow graph restructuring of the program syntax, such as will be discussed in the next section. However, it is considerably simplified if we assume a relatively abstract language without gotos, but with only "structured" looping mechanisms -- in which case the flow graph and the parse graph are essentially equivalent.

An example of a loop optimization is the following (Waite [136], p 595):

```
for i:=m1 until m2 by m3 do
  ...  (k1*i + k2) ...
  end
```

The variable i (the induction variable) is used in a computation involving constants k1 and k2. This computation may be reduced to a smaller one:[10]

```
i1 := k1*m1 + k2;
i2 := k1*m3;
for i:=m1 until m2 by m3 do
  ...  ( i1 ) ...
  i1 := i1 + i2
  end
```

Besides requiring only two multiplications for an iteration of any length, and only one extra addition, on many machines i1 and i2 can be implemented in registers: the statement "i1 := i1 + i2" thus reduces to a single fast register to register

---

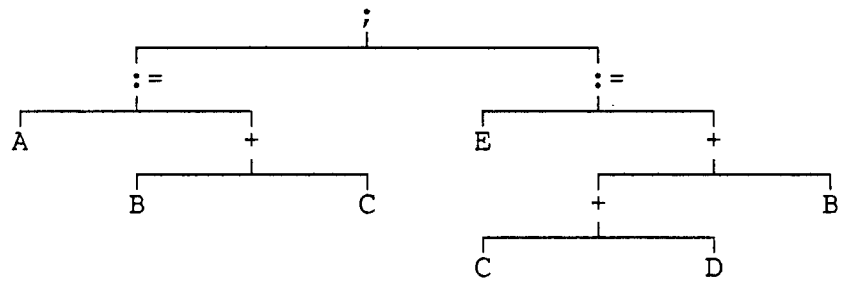
[10] The usual caveat for optimization applies in this example: if the loop is not executed at all, this version is actually more costly -- though only minimally so -- than the "unoptimized" version. Of course, in general, we can expect any loop to be executed multiple times, since that is its function.

instruction.

In general, however, expression optimization depends on a graph representation of the code (within a one-entry one-exit sequence of code called a basic block, for which, see the next section), commonly called a DAG. For example, the program sequence (Aho and Ullman, p 428):

```
A := B + C;
E := C + D + B
```

may be represented by the parse tree:

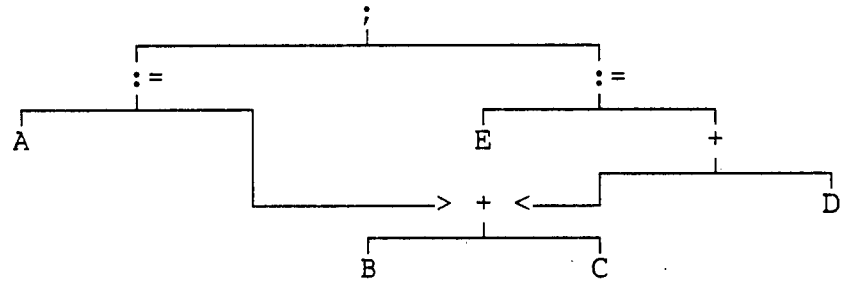


[T12]

However, if we make use of a simple DAG construction algorithm (Aho and Ullman, pp 420-423) and take advantage of the laws of associativity and commutativity that apply to addition,[11] we can construct a representation for an identical sequence in which common subexpressions have been reduced to a single subgraph.

---

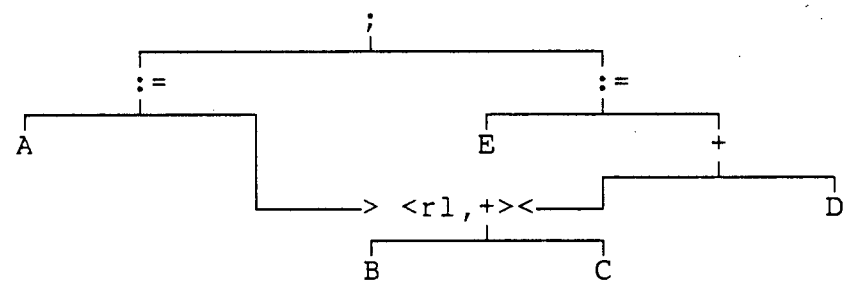
[11] They do not apply all the time on computers, where numbers are generally limited in their precision, and the order of evaluation can, and in many instances does, affect the result. Here is yet another caveat therefore: the "optimization" we are about to examine may, in fact, pervert the programmer's intent.



This could be directly translated by a tour of the DAG which, at the same time, transformed expressions into temporary locations once they had been encoded. Thus, after generating the (IBM 360/370 style) code:

```
load r1,B
add  r1,C
```

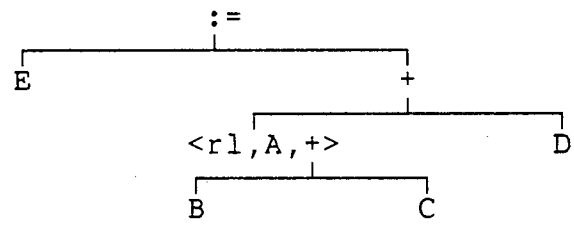
the DAG looks like



(Note that both register r1 and expression "B+C" are remembered. This is because, for complex expressions, it is possible to run out of registers; but then the subexpression can still be retrieved.) Then, the assignment operator and semicolon (sequence operator) are reduced, generating the code:

```
store r1,A
```

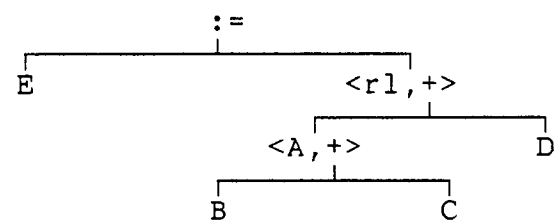
and resulting in the DAG:



(Here, r1, A, and "B+C" are all remembered as, at present, equivalent, and in order of preference for code generation.) The next code generated is for the one remaining addition:

```
add  r1,D
```

with the resulting DAG:



(The extra information is no longer interesting, but might have been useful in a larger program segment.) The final code generated is for the assignment:

```
store r1,E
```

This process has produced the code marked (a), below:

```
load  r1,B
add   r1,C
store r1,A
add   r1,D
store r1,E
```

(a)

```
load  r1,B
add   r1,C
store r1,A
load  r1,C
add   r1,D
add   r1,B
store r1,E
```

(b)

Compare this with the code generated directly from the parse tree T12, marked (b), above.

More complex forms of program improvement are also

essentially syntactic transformations, but can be discussed only in terms of fairly complex pattern matching. One simple example of this kind is recursion elimination, where a (not unusual) recursive procedure of the form

```
f(n) = if n≤1 then 1 else n*f(n-1) fi
```

is turned into an iterative one:

```
f(n) = begin t:= 1;
           L: if n>1 then
                 t := t * n;
                 n := n - 1;
                 goto L fi;
           t end
```

This kind of transformation is primarily aimed at eliminating the sadly high cost of procedure calling on most existing machine architectures, and it generally falls into the class of semi-automatic program improvement. It is intended to be a means whereby a programmer can establish the correctness of his algorithm at a high level of abstraction, because establishing correctness will tend to be a smaller task at an abstract level. Once correctness has been established the program can then be transformed, by disciplined means involving well-understood transformations, into a more efficient one. The programmer will preferably be aided in this somewhat cumbersome process by a book-keeping computer.

Semi-automatic program improvement shades over into semi-automatic program synthesis, where, beginning from a high level (abstract) description, the programmer develops the details of his program to any level of detail necessary.

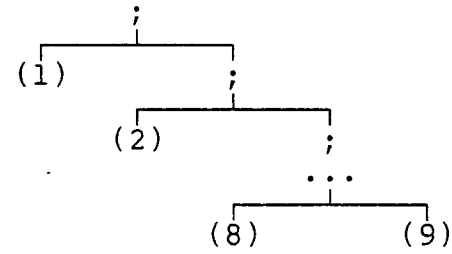
Improvements can be applied at any stage of the process of making the abstraction concrete, wherever this process has produced a less than satisfactorily efficient description. There will be a discussion of closely related material in section 3.4.

3.2.2 Flowgraphs and intermediate level optimization

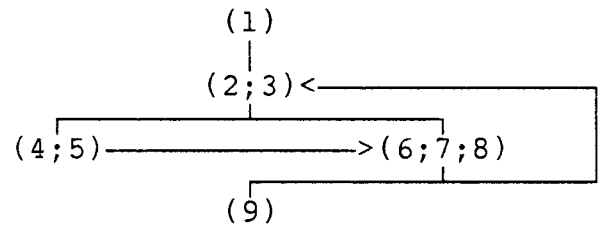
Section 3.1 (translation) based its arguments on a close correspondence between translation and syntactic structure. Even in section 3.2.1 (source code optimization) the argument was based directly on program examples and closely related, though abstract, graph representations. As we pursue optimization, however, we must sooner or later come up against serious limitations in such a correspondence. For example, if we represent the program (from Aho and Ullman [3], p 465):

```
(1) i := 1;
(2) if x<b goto (4);
(3) goto (6);
(4) i := 2;
(5) x := x + 1;
(6) y := y - 1;
(7) if y≤20 goto (9);
(8) goto (2);
(9) j := 1
```

as a graph representing the syntax discovered by a conventional parser using a BNF description, it might look like (representing statements by their line numbers):



This representation gives us no idea that the program contains a loop, let alone what variable represents the loop's induction variable, so that we cannot begin to apply any optimization strategy. On the other hand, a graph representation of the control flow:



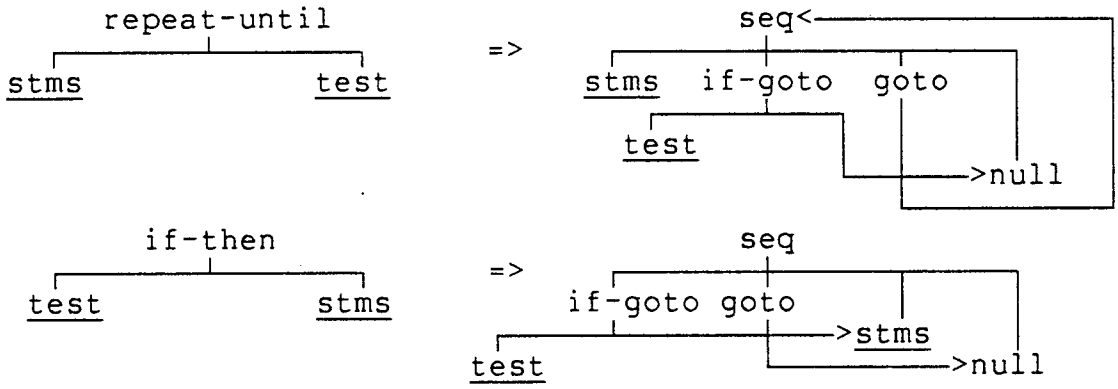
immediately indicates: that there is a loop (a cycle in the graph); which nodes participate in the loop (those that form a strongly connected set); and where to look for an induction variable (among the statements represented by the loop nodes). The nodes also represent basic blocks (blocks of code with exactly one entry and one exit point), and it is within these that we may safely apply expression optimization. Another use for this representation is in more global value-retention information (in registers, etc.), and in deciding which computations can be safely hoisted outside the loop to reduce the computations done inside.

Such a representation is in no sense out of keeping with

the general philosophy adopted in this chapter (and, ultimately, in this dissertation) that programs are usefully represented by syntactic structures for purposes of translation. Consider a more abstract formulation of the program:

```
i := 1;
repeat
  if x<b then
    i := 2;
    x := x + 1 fi;
  y := y - 1
until y≤20;
j := 1
```

which contains no gotos and therefore reveals more immediately its basic block structure than the linear version. The difference between these two programs lies in the transformations



The transformations simply reduce a program in an Algol 60-like language to an equivalent program in a Fortran-like language (compare Kernighan [68], and Hanson [56]); that is, they re-express the abstract control flow structure as more primitive, more machine-like control structures, and in this they are performing a kind of translation.

Flowgraphs are therefore a conception of a program's

syntactic structure as well. In fact, they structure basic blocks, which in turn give a sequential structure to statements and a DAG structure to expressions.[12] We can now apply the more exotic forms of optimization, which depend on the knowledge about flow of control embodied in flow graphs. The next section considers several of these optimization techniques.

3.2.3 Object code optimizations

In section 3.2.1, we saw an example of object code generation requiring only one register for optimal evaluation of the sequence

```
A := B + C;                                    [P7]
E := C + D + B;
```

In general, however, the use of multiple registers can produce better code than if the translation is restricted to one register. Consider the expression

```
A*B + C*D                                      [P8]
```

No amount of manipulation will reduce this (in a one register translation) to anything less than

```
load  r,A
mult  r,B
store r,TEMP
load  r,C
mult  r,D
add   r,TEMP
```

With two registers, the save, the use of a temporary storage

---

[12] This approach, of combining these usually distinct conceptions of program structure into one for purposes of optimizing code generation, was recently applied also in the design of PQCC [22].

location and the final memory to register add can all be reduced to a single register to register add:

```
load  r1,A
mult  r1,B
load  r2,C
mult  r2,D
add   r1,r2
```

With an infinite number of registers (or a stack machine) optimal register use could be guaranteed for any expression, no matter how complex. However, registers are generally limited to a small number (eight on the PDP-11 and, for purposes of multiplication and division, eight also on the IBM 360/370), and although few programs use large expressions, it is precisely those few that need good code generation techniques to render their translation competitive with hand written assembly code.

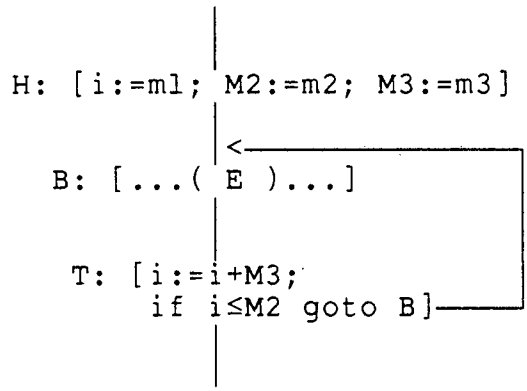
The same kind of analysis that generated good code for program segment P7 can also determine good register use for segment P8, when combined with basic block analysis. Code generation will not be so straightforwardly one-pass now, however, as in the case of P7. Aho and Ullman ([3], pp 526-527) describe a process for finding out which intermediate and final values computed in one basic block remain useful for its successors -- and should, therefore, where possible, be retained in registers. This process requires two passes over the code: one forward, and one backward, scanning each successor basic block for use of variables and intermediate values. DAG representations are also used in Aho and Ullman's loop optimizations (pp 454-499), heuristic reordering of evaluation

sequence (pp 538-540), and global register allocation (pp 533-537).

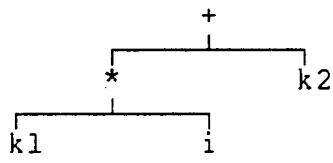
Consider again the example on page 93:

```
for i:=m1 until m2 by m3 do
  ...  (k1*i + k2) ...
  end
```

This is represented by a flow graph of the form



The loop is represented by three blocks, H, B, and T (for convenience, there are three blocks, although B and T together form a basic block). The expression E is a graph:



The induction variable i is easily extracted from the original program in this case, since it is in a for loop. In general, induction variables may be discovered by analysis of the data flow (Aho and Ullman [3], pp 466-471). The use of the induction variable in expression E results in a highly regular sequence of values:

```
i=m1          k1*m1 + k2
i=m1+m3       k1*m1 + k2 + k1*m3
i=m1+2(m3)    k1*m1 + k2 + 2(k1*m3)
...
i=m1+n(m3)    k1*m1 + k2 + n(k1*m3)
```

Just as we make the for loop more primitive by moving the computation of m2 and m3 to the header block, and re-expressing the control flow as an if/goto combination, so we make expression E more primitive by re-expressing the general equation (E=k1*i + k2) as an iterative series derived from the above analysis:

```
E[0] = k1*m1 + k2
E[n] = E[n-1] + k2*m3
```

or, in terms of programming constructs,

```
i1 := k1*m1 + k2;
do i1 := i1 + k1*m3 od
```

Since "k1*m3" is a constant value, it can be moved outside the loop as well:

```
i1 := k1*m1 + k2;
i2 := k1*m3;
do i1 := i1 + i2 od
```

We can now merge this (infinite) loop with the controlling loop:

```
H: i := m1;
   M2 := m2;
   M3 := m3;
   do B: ... (E) ...;
      T: i := i + M3
   while i ≤ M2 od
```

We do this by replacing "(E)" by "(i1)" and by placing the loop-invariant computation in the header block H, and the iteration in the computation-independent tail block T (to insure that updating i1 will not affect any computation performed in the body B). The result is:

```
H: i := m1; M2 := m1; M3 := m3;
   i1 := k1*m1 + k2;
   i2 := k1*m3;
   do B: ... (i1) ...;
      T: i := i + M3;
         i1 := i1 + i2
   while i ≤ M2 od
```

A similar form of analysis, merging independent portions of a loop, was used by Dijkstra in a derivation of GCD ([34], section 4).

The term peephole optimization was first applied by McKeeman [83] to describe last minute code improvements, especially for code generated by syntax directed translations, where the context was severely limited by the grammar. Many of these are subsumed by other methods of optimization that are more generally applicable to fully integrated translation efforts. McKeeman's specific examples of redundant load elimination from sequences like the following:

```
{ X := C;          { load  r1,C          { load  r1,C
  Y := X }    =>     store r1,X    =>     store r1,X
                     load  r1,X           store r1,Y }
                     store r1,Y }
```

and pre-evaluation of constant expressions, are both better performed at higher levels, where more general methods will almost certainly produce better code. However, when techniques like multi-stage translation are used, especially in software porting strategies, such local pattern matching and improving transformations are extremely valuable, and relatively simple to implement, considering that each stage of the translation is certain to introduce inefficiencies that are best removed before

they multiply in the next stage. Richards [102] comments on the importance of such techniques with specific reference to the portable BCPL compiler; see also Cattell [21] for a critique of multi-stage translators, particularly because of their tendency to lose valuable optimizing information (paired with a critique of one-stage translators for being unable to cope well with contextual information).

3.3 Error handling in a syntactically structured environment

"In an ideal world [says Horning], where neither humans nor machines ever make mistakes, the compiler-writer could limit his attention to correctly translating correct programs (and some naive compiler writers do so). In reality, however, most of the programs processed by any compiler will be to some extent incorrect" ([61], p 532).

There is a considerable literature on context free syntactic error detection, recovery, and, to a lesser extent, correction. Modern parsing techniques (LL and LR especially) detect syntactic errors at the earliest possible moment, and relatively good recovery techniques exist so that the parser can continue, after an error, and detect further errors as well (Aho and Ullman [3], pp 397-402). However, many of these techniques interfere with the semantic portion of syntax directed compilers (Gries [53], p 321), and this can be cited as yet another argument for separating the parse from the translation. Gries

also notes: "The nice part about top-down error recovery is that the partially constructed tree conveys much usable information...not as readily available in the bottom-up method" (p 325); the hidden point here is, of course, that a syntactic structure gives much more support for complex decisions involving programs (and the lack of depth in most discussions on error recovery indicates that it is nothing if not complex) than does an unstructured string. Elsewhere, Gries notes that perhaps the usual approach to error correction, which involves finding the minimum sequence of string transformations that will make a "correct" program out of an incorrect one, would be better defined in terms of transformations on a tree ([54], p 629).

A major annoyance in developing a program using a one pass compiler is the redundancy of error messages. Consider the program ([53], p 319):

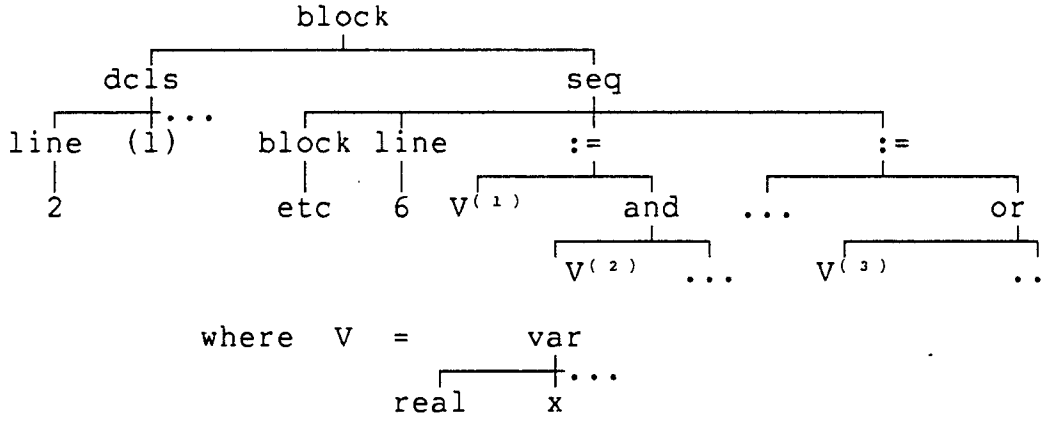
```
begin                                    (1)
  real x; ...                            (2)
  begin                                  (3)
    boolean x; ...                       (4)
    begim ... end;                       (5)
    x := x and y; z := x or y            (6)
  end end                                (7)
```

It contains one error, namely the misspelled begin of line 5. However, this error causes a mismatching of begin-end pairs, so that the statements on line 6 will be parsed in an environment where x is declared to be real, not boolean. We get, in the average compiler, three messages telling us that x is used incorrectly in boolean expressions on line 6; if it were not

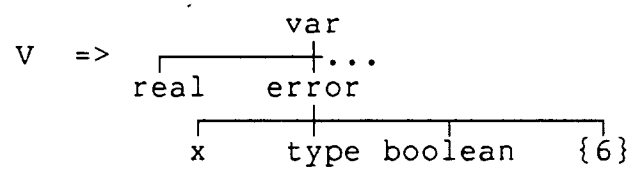
declared on line 2, we would get another three messages telling us that as well.

Some redundancy is unavoidable in these situations. In the absence of spelling correction (which is rarely implemented, and frequently so unreliable that users turn it off in annoyance after its first major blunder) we can expect a note about the odd symbol "begim", a note about the redundant second end on line 7, and one note about the misuse of x on line 6; if x were "misused" on other lines, we would still like just one note, giving us a list of offending lines.

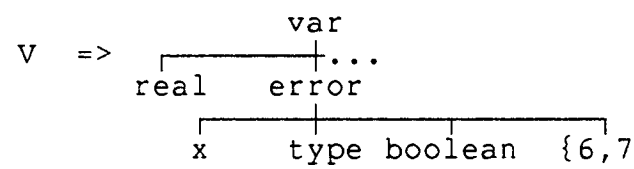
There are, to be sure, known techniques for overcoming this deficiency, generally involving the use of symbol tables, since the difficult cases for context free parsers are always the context sensitive ones, and in programming languages that is always those cases involving the use of symbols (Gries [53], pp 318-320). Let us consider the classical techniques in terms of a graph transformation approach, however. The above program would be eventually parsed as



(Note line number information retained by parse.) In a type checking pass, the second occurrence of x, marked "V(2)", is encountered in a boolean operation ("x and y"); this use is incorrect, and is marked as such by altering the "symbol table entry" represented by subgraph V:



(x is involved in a type error at line(s) {6}, where it is used in a context requiring a boolean.) The next problem is with the assignment of the result of a boolean expression to a real variable represented by "V(1)". Subgraph V is already marked with a type error involving incorrect use of x in a boolean context on line 6, so nothing is done. Similarly, the use of x represented by "V(3)" adds no new error information. However, if this statement were on line 7, the line number information would be augmented:



The above effectively re-expresses the use of symbol tables in error reporting in a graph transformation form. Errors should also be recorded in a list -- which may be sorted, at the end of the compilation, into some canonical order based on error types, line number of first occurrence, or some other reporting criterion. If the information recorded is the "error" subgraph

shown above, recording any further errors about this symbol will automatically cause them to be recorded in the error list as well.

Consider also the program

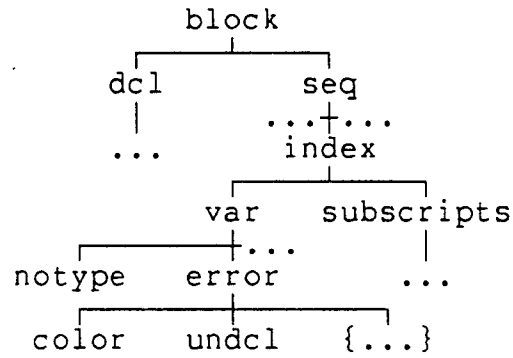
```
begin array[1:7, 2:5, -1:1] of integer colour;
  ...color[2, 3, 0]...
  end
```

This involves a spelling error,[13] but will normally generate two error messages: one to complain about "color" being undeclared; the other to complain that the dimensions of the subscript do not match those "declared" for the variable, or possibly that there is a type incompatibility, or even (if square brackets are not available, and parentheses are used instead) that the procedure call is incorrect (Aho and Ullman [3], p 403; Gries [53], p 318). The second message is extraneous, and may confuse the programmer.[14] It should be suppressed by the same method used earlier: the program graph generated by the parser will be of the form

---

[13] Possibly introduced by having the program pass through the hands of two programmers of different nationality.

[14] Experienced programmers begin to "see through" redundancies of this sort after a while, simply because it is such a common experience. In my experience teaching introductory programming courses, however, redundant messages are especially confusing to beginning programmers, who often cannot see that the second message is a function of the first, and yet are most in need of help from the compiler. Minimizing redundant and extraneous messages is, therefore, of even greater importance in student language processors than in processors aimed at professionals.



When the subscript error is discovered, the fact that "color" has already been marked as an undeclared variable is enough to suppress the error message about type incompatibility. Naturally, this will be the wrong action in the rare situation where the program had two errors in it:

```
begin array[1:7, 2:5] of integer colour;
  ...color[2, 3, 0]...
  end
```

Fixing up the name only results in a new message involving mismatched actual and declared subscript lists. However, properly speaking the second error does not yet exist in this program, since "color" may have been intended to be a separate structured variable having three dimensions. The sensible rule is that it is better to be annoying occasionally than to be annoying continually.

3.4 Automatic program development and improvement

The earlier portions of this chapter were concerned primarily with a presentation of traditional translation concerns in terms of a uniform model. That the concept of syntactic transformation can be successfully applied to these

concerns with a great deal of elegance argues in its favor as a model for translation.

However, there are several new concerns in programming language studies and closely related areas that are beginning to take well-defined form. These are not yet included in texts on the subject, because they have yet to emerge into a uniform relationship to the traditional subjects normally covered in such texts. In this section we will review what appear to be the major points in these new concerns: it should become reasonably clear from this review that they, too, can be considered in terms of syntactic transformations.

Programming disciplines and program correctness concerns have gone hand in hand since their emergence from the first challenges to traditional methods of program development that spawned them more than a decade ago. At the same time, the idea of automatic synthesis of a running program from specifications of its abstract characteristics has developed into a closely related area.

Program development by stepwise refinement[15] is a discipline for creating programs out of abstract specifications by making the specification increasingly concrete. Wegbreit [138] lists transformation rules whereby abstract specifications

---

[15] A phrase coined by Wirth [141].

may be reduced in stages to machine instructions: the process may be partially automated by making available a system which automatically does the book-keeping chores of applying such transformations under direction of a human agent (the programmer). Similar systems of transformation rules, and their application in a semi-automatic manner, are described by Burstall and Darlington [19], Bauer et al. [12], and Arsac [6]. These systems are, to some extent, a development of graphics editors and "top-down" programming tools whose transformation rules are simply the recursive syntax definition rules of BNF. For instance, Hansen [55] describes a system (Emily) for graphic editing of hierarchic text (not necessarily program text, although that was its first application) using as basis a generative grammar. In developing such hierarchic text, one begins with a single symbol, displayed on the screen (the distinguished symbol of the generative grammar):

<statement>

Using a menu, the programmer transforms this into one of its possible productions, which is displayed appropriately indented:

for <var> := <expr> until <expr> do
  <statement sequence>
  end

At each step, the programmer may expand any nonterminal symbol (indicated here in the classical BNF style by "<...>"). The final result, one consisting purely of terminal symbols, is guaranteed to be a program correct in its context free syntactic structure. In program development systems, the programmer begins with an abstract expression of, say, the eight queens

problem:[16]

```
initialize;
repeat try current step;
   if successful then advance else regress
until underflow or overflow
```

This is a general statement of a solution method by backtracking; in the eight queens problem, we mean by this: at any point j, place the j-th queen on the board ("try current step"), test the partially complete board for the mutual nonagression condition ("successful"), and then either advance to queen j+1 or regress to queen j-1. "We mean by this" is a statement that, as far as the programmer is concerned, indicates that he can replace the appropriate abstraction by the more concrete instruction, so that he ends up with the program:

```
variable j;
initialize;
repeat place the j-th queen on the board;
   if present placement meets nonagression condition then
      proceed to queen j+1
   else reconsider queen j-1
 until underflow or overflow
```

It is possible to start off ("initialize") with the eight queens lined up at the top of the board, one to a column, and to "place the j-th queen" by advancing her one row. "Underflow", corresponding to a regression past the first column, means that we have a problem with no solution; while "overflow", corresponding to an advancement past the last column, means that we have achieved a solution. So we now have a still more

---

[16] Place eight chess queens on a board so that none of them is in a position to capture any of the others. Recall that a queen can move any number of squares in a row, column, or diagonal.

concrete program:

```
variable j;
j := 1;
line up the queens one to a column, on row 0;
repeat advance the j-th queen one row;
   if present placement meets nonagression condition then
      proceed to queen j+1
   else reconsider queen j-1
 until j < 1 or j > maxcolumn
```

This is still a very abstract expression of the algorithm: no data structures, except the variable j, have been declared; no specific meaning has been attached to the real workhorse, the predicate "present placement meets nonagression condition". In fact, we might be tempted to expand "proceed to queen j+1" as simply "j := j+1", and "reconsider queen j-1" as "j := j-1", but while the expansion for "proceed" is essentially correct, the expansion for "reconsider" must take into account that we do not wish to return to a previous condition, nor do we wish to ignore any intermediate conditions: this algorithm works only if it works its way monotonically through all reasonable permutations. So "reconsider queen j-1" must be expanded as:

```
procedure reconsider queen i is
   return queen i+1 to the top of her column;
   j := i end
```

This procedure may be expanded inline by replacing its "call" by its body, and its "parameter" i by j-1, optimizing the resulting expression "j-1+1" as "j": note that one has to be careful not to introduce the Algol 60 call by name problem [90].

Before we go much farther, it will become necessary to invent a representation (a data structure) for the board, since

further refinements will certainly have to make explicit reference to this data structure; but we have gone far enough for the flavor of this method to become clear.[17] The semi-automatic program development systems mentioned above generally proceed along these lines, by making the process easier, the same way Emily, and similar systems, make syntactic expansion into programs easier. In fact, an Emily-like system would be a good start for a program development system.

Emily-like systems are especially appropriate in teaching environments, where the problem with confusing and somewhat redundant syntactic error messages, described in section 3.3, can be overcome entirely by making available a program editor that will produce only syntactically correct programs. Cornell, where a syntax-correcting compiler for a PL/I subset was developed earlier, has been experimenting with a program editor of this kind [123]. Syntax graphs can be used effectively in such a setting because they can represent the context sensitive syntax of the program and can thus assure the correct declaration of names, because they represent the hierarchical structure and can be displayed at any level of detail, and because the graph manipulation techniques outlined here are appropriate to structure editing.

---

[17] Wirth [141] gives a slightly different, but essentially similar exposition of this process, from which the above is derived.

Proofs of program correctness are simplified by beginning from an abstract version because, for instance, "f(S1,S2,...,Sn)" is a refinement of "P" if and only if (using Hoare's notation for predicate transformers):

{p}f(S1,S2,...,Sn){q} implies {p}P{q}
for all predicates p,q

But the proof of "f(S1,S2,...,Sn)" satisfying the requirements that its execution uniformly transforms p into q will be much more detailed than that for "P". So if we prove "P", and then derive the expression "f(...)" by applying well-understood, even proven, equivalence transformations, then a simple proof suffices, and the correctness of the refinement is guaranteed. (Incidentally, Dijkstra [34,35][18] proceeds in the opposite direction, deriving a program from its predicate transforming properties: the resulting program will necessarily be correct -- with respect to the predicate transforming properties ascribed to it.)

Another semi-automatic basis for creating correct programs was given a formal treatment by Manna and Waldinger: "In order to construct a program satisfying certain specifications, a theorem induced by these specifications is proved, and the desired program is extracted from the proof" ([86] from the abstract). It was further developed, along somewhat different lines, by Floyd [43], by Darlington and Burstall [30], and by Wang [137]. The concept of abstraction forms a basis for

---

[18] See also Knuth [72], pp 287-289.

several programming languages designed for verification, like Mesa [48], CLU [80] and Alphard [145]; recently, Morris described an implementation strategy for abstractions using replacement by open procedures (a form of macro expansion, and a method already mentioned in relation to abstraction in section 3.1.1): however, this leads, as Morris points out, to the Algol 60 problem of implementing parameters through a call-by-name mechanism [90].

A denotational semantics may also be used as a basis for proving correctness and this, too, leads to the possibility of automatic program development. Mosses [91] describes a translator writing system which synthesizes translations from a denotational semantics model of the language. Huet and Lang [62] give a methodology for applying Burstall and Darlington style program transformations and a method for proving their correctness with respect to their intended effect using a denotational semantics. Thus the concern for correctness in programs is transferred to a concern for correctness in translators: considering the immense importance of correctness to the users of a translator, this is no small matter, and any work aimed at making the translation process more regular, and thus more amenable to a demonstration of correctness will be of value; we will return to this question in section 4.2.4.

In all these instances, the central principle upon which they operate is some notion of transformation. In the next

chapter, a design for a translator writing system is outlined based entirely on the idea of representing programs as directed graphs and defining various forms of program manipulation as transformations on these graphs. Whether this design will carry over successfully into applications of the kind described in this section, is not presently obvious: program development systems like that of Darlington and Burstall tend to be interactive, and dependent on a visual display, while compilers tend to be batch oriented "black box" processors. However, compilers once tended to be card oriented with ad hoc parsers: it may be that some combination of the concepts of abstraction and of program synthesis will have the same ultimate effect on the nature of translators as the concept of recursive syntactic definition once had; namely, to alter that nature permanently. If so, we can look forward to interactive program development tools based not only on a rigorous notion of syntax, but also on a rigorous notion of translation, and that notion will, if the preceding overview has looked in the right places, almost certainly be based on some concept of syntactic transformation.

Chapter 4: Toward a translator writing system

The general progress of this dissertation has been from the abstract to the concrete. Chapter 2 presented a general consideration of translation while establishing the importance of syntactic structure and the pervasiveness of the concept of transformation in any discussion of the subject. Chapter 3 presented a series of specific problems in translation within the general framework of syntactic graph transformation, but left many details of implementation in the air. This chapter is intended to make concrete many of the issues that were deliberately left abstract in the preceding chapters: this does not mean that it will take the form of a user manual for an actual programming language, but it does mean that it will be made clear how such a language might be designed. It will concern itself especially with the considerations that would necessarily be brought to such a design, and with the conclusions those considerations would probably lead to. Where appropriate, algorithms guaranteeing its efficient implementation will be presented.

It is both too early and too distracting to present a programming language or translator writing system here. This dissertation is intended primarily to define and bring into its traditional context a technique for describing translations that has been, as shown above, inherent in many earlier discussions, but was never given a comprehensive treatment that would

demonstrate its considerable descriptive power. That such a translator writing system is possible, and that it would live up to the claims made for it here is supported to some extent by empirical evidence which will be discussed in section 4.4; but the main support for this technique lies in its ability, as demonstrated in chapter 3, to give uniform expression to a wide range of translation-related topics. Additional empirical evidence, presumably in the form of a more firmly defined translation language and a larger, more "realistic" range of actual applications, would not necessarily have lent extra support to the primary thesis. On the contrary, such evidence would have required a more detailed presentation and, hence, the definition of an actual programming language, which might well have distracted attention from the core ideas by introducing questions of style and even taste in language. To avoid this excess of detail, therefore, and to keep the discussion to a necessary minimum for the purpose of convincing the reader that the ideas presented in chapter 3 are feasible, the subsequent presentation is limited to giving concrete form only to such aspects of translation as might be given form in a translator writing language. The discussion will focus on the advantages these forms give, and the ways in which they might be implemented in a practical manner.

The remainder of this chapter is divided into four parts: section 4.1 outlines the form of a translator written in terms of graph transformations; section 4.2 treats specific questions

of representation (of graphs and graph patterns), of implementation (of pattern matching and transformation), and of program structuring in a graph transducer language; section 4.3 presents an overall strategy for structuring translators, both for purposes of simplifying translations and, it follows, for directing and considering the implementation of a translator; and section 4.4 presents actual experience with a primitive implementation of a graph transforming translator, and the effect this had on the presented design.

4.1 Overall structure of the system

One tends, in reading computer program documentation, to ignore the word "system" as a useful bit of noise referring, in a vague and general way, to the program being described. Its use in the term "translator writing system" is, however, generally correct, since many TWSs are not single programs, but systems of separate programs to perform the separate functions of translation. Feldman and Gries make this quite clear: "Since compiler writing is a large programming task with many aspects, it is not surprising that many different techniques have been proposed as aids to compiler writers. In a very real sense, any system feature (e.g. trace, edit) which helps one produce large programs is a compiler-writing tool" ([39], p 78). It is therefore difficult to define precisely where the TWS ends and the operating system begins; it may be part of a larger program development system that is integrated but is not itself a

complete operating system.[1]

Part of the reason for translator writing tools to tend to separate into individual programs lies in the differences in the languages traditionally used to express its different functions: syntax is most often, and to great profit, described in a variant of BNF; semantics (everything else) is most often described in a general purpose programming language (for example, XPL [84] and TRUST [133]), and where the two are combined, as in many syntax directed translation techniques, the usual programming problems involving an integration of logically separate functions will tend to crop up. Sandewall separates the functions of parsing, compiling or interpreting, and program printing, especially because, in a program development system, these may need to be applied in some order other than the traditional sequence in a compiler ([110], pp 37-38); McKeeman's discussion of "vertical fragmentation" in compiler construction exaggerates, but also thereby emphasizes the separability of compiler functions ([85], pp 20-27).

Simply, a translator consists of the following sequence of programs:

---

[1] Sandewall: "Programming system, is used [here] to mean an integrated piece of software which is used to support program development, including but not restricted to a compiler" ([110], p 35).

parser
transducer (one or more in sequence)
flattener

A parser is any program which takes a textual representation and, if it is correctly formed, turns it into an internal representation; transducers change one internal representation into another, and represent the actual process of translation; finally, a flattener is any program that takes a standard internal representation, usually a non-linear structure, hence the name "flattener", and produces some form of interpretive code, machine code, or even external representation. Of these, only the transducers are of direct interest to this dissertation: by and large, the emphasis in the past, in any discussion of translation, has been heavily on the side of parsing techniques, and it is out of place here, in what is intended to be a discussion of a general purpose technique for translation, to spend more than the necessary minimum time on parsing.

However, in spite of the fact that what is being developed in this dissertation is not a syntax directed technique for the expression of translation, it is nevertheless a technique highly dependent on a syntactic analysis of the program under translation. Before we can develop a language for the expression of graph transformation, therefore, and apply this to the translation of programming languages, we must first consider syntactic recognition and acceptance to a certain level of detail. In particular, the argument presented in chapter 3

depends entirely on the assumption that the program under translation has received the benefit of a complete parse, in which all the context sensitive aspects of program acceptance have been dealt with as well as the context free structure, and that both these syntactic domains are completely and adequately representable in a graph structure, without recourse to external data structures like symbol tables.

It has been remarked before in this dissertation that the context sensitive aspects of programming languages are traditionally dealt with in the form of a symbol table. Furthermore, in the absence of a sufficiently expressive language for syntactic definition, the symbol table will be the only recourse, and it must be stressed that any implementation of the ideas developed in this chapter will probably depend on some form of symbol table to implement context sensitive recognition. However, in the design of a programming language -- in the case of the present discussion, a language for the programming of translators -- it is particularly important that its component concepts be raised to their highest level of abstraction in order to reduce the intellectual effort required both to program in it and to read the resulting programs as algorithms. For example, most programming languages since Algol 60 have used the concept of a stack, a concept that requires considerable mathematical sophistication to express rigorously [59, 79], in order to implement the concept of name scoping; nevertheless, the concept of name scoping, or locality of

variables, has been incorporated in these programming languages in a way that does not require the introduction first of the concept of a stack. Similarly, just as BNF is a long-proven abstraction for syntactic definition that is conceptually cleaner than the same definition presented in terms of a parser written in a general purpose programming language, so an extension of BNF is used here to define the relationship between context sensitive syntax and syntactic graph representations in a manner that is cleaner than writing a tree touring algorithm to perform the same function.[2]

The basis for our discussion will be augmented grammars, whose rules are of the form ([87], pp 250-265):

```
<dcl train>↓symbol-table1 ↑symbol-table2
   ::= <declaration>↓symbol-table1 ↑symbol-table2
     | <declaration>↓symbol-table1 ↑symbol-table3
       ";"
       <dcl train>↓symbol-table3 ↑symbol-table2 .
```

This is a pure augmented rule which, as indicated on page 35, may be thought of as a recursive recognition procedure with parameters symbol-table1 and symbol-table2, and local variable symbol-table3. Symbols following a downward arrow, "↓", represent inherited attributes; symbols following an upward arrow, "↑", represent synthesized attributes. In terms of a recursive recognition procedure, the left hand side's attributes

---

[2] This insight was derived from experience with the translator discussed in section 4.4.1, which was implemented without benefit of such a parser and had, therefore, to perform the translation of context free tree into context sensitive graph itself: this transformation required an entire pass, which was as large as the translation pass that followed.

are like formal parameters where, in the language of Algol W, the inherited attributes are value parameters and the synthesized attributes are result parameters; on the right hand side these attributes are like variables whose values are, if inherited, passed to, or, if synthesized, returned by the expansion (recursive call) of the nonterminal symbol to which they are attached. In practice, these rules are further augmented with predicates, which must evaluate to true for the rule to succeed, and with procedure calls which perform "semantic" actions. Bochmann has shown that the number of passes necessary to evaluate an attribute grammar can be derived from the grammar (it may be unbounded), and that regular context free parsing techniques are applicable to such grammars [15].

Obviously, if semantic actions can be written in a general purpose language, the result is a syntax directed translator. However, that is overstepping the bounds of the augmented grammar technique's usefulness in its present context (and, if my hypothesis is correct, in any context), and I shall propose here a restricted set of operations and predicates that will make it possible to define, in a relatively succinct fashion, the relationship of the grammar of a language to its abstract representation as a syntax graph.

Consider again the language L, first presented on page 29:

```
(1 ) L ::= BLOCK
(2 ) BLOCK ::= begin DCLS; STMS end
(3a) DCLS ::= TYPE VAR
(3b)        | TYPE VAR ; DCLS
(4 ) TYPE ::= real | int | ...
(5 ) VAR ::= a | b | ... | z
(6a) STMS ::= STM
(6b)        | STM ; STMS
(7a) STM ::= VAR
(7b)       | BLOCK
```

A simple augmentation of this grammar is capable of defining both the form of the abstract syntax tree that represents it and the relationships between the definitions and uses of symbols:

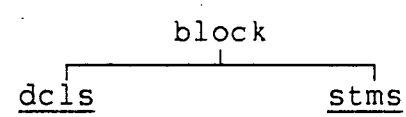
```
<L>↑tree ::= <Block>↓{} ↑tree .

<Block>↓symbols ↑tree
  ::= "begin" [locsymbols := copy(symbols)]
      <Dcls>↓locsymbols ↑dcls
      ";"
      <Stms>↓locsymbols ↑stms
      "end"
      [tree := "block"<dcls,stms>] .

<Dcls>↓symbols ↑tree
  ::= <Dcl>↓symbols ↑tree
    | <Dcl>↓symbols ↑dcl
      ";"
      <Dcls>↓symbols ↑dcls
      [tree := "dcls"<dcl,dcls>] .

<Dcl>↓symbols ↑tree
  ::= <Type>↑type
      <Var>↑var
      [tree := "var"<type,var> ;
       symbols@var := tree] .

<Type>↑type
  ::= "real" [type := "real"]
    | "int"  [type := "int"]
    | ...

<Var>↑var
  ::= "a" [var := "a"]
    | "b" [var := "b"]
    | ...
    | "z" [var := "z"] .
```

```
<Stms>↓symbols ↑tree
  ::= <Stm>↓symbols ↑tree
    | <Stm>↓symbols ↑stm
      ";"
      <Stms>↓symbols ↑stms
      [tree := "stms"<stm,stms>] .

<Stm>↓symbols ↑tree
  ::= <Var>↑var
      [tree := symbols@var]
    | <Block>↓symbols ↑tree .
```

The primary rules are augmented with the inherited attribute "symbols", which represents a symbol table, and the synthesised attribute "tree", which represents the abstract parse tree; it will, in fact, be a DAG. The symbol table is accessed by the subscripting notation

symboltable@symbol

which refers to the current definition of the "symbol" in the table. The only other notation is that of assignment, and a simple list notation in which a tree of the form



is represented as

"block"< dcls , stms >

The function copy creates a copy of the structure represented by its argument. It is used here to insure that blocks inherit all variables declared global to the block, but use only a local copy of the symbol table to declare local variables, so that these do not return to the higher level blocks. A stacking mechanism would be a more efficient implementation of this (see page 31), but would have increased the size of this definition slightly; for purposes of a definition, the two are equivalent.

4.2 Toward a graph transducer language

In chapter 3 we examined many different cases of translation and translation related concerns which would be expressed in terms of graph transformations. We saw that even such translation methodologies as dividing the translator up into a sequence of independent stages could be discussed in terms of transformation, where each stage is implemented as a graph transducer. Many details of the actual expression of translation algorithms were left to the reader's imagination as being incidental to an abstract discussion of translation such as presented there. However, it is now time to consider at least some of these details, both as to the expression of graph transformations (and syntactic transductions in general) and the reasonable implementation of these concepts, so that there can be no question that the transformational approach to translation is practical as well as elegant.

The details are addressed in three stages, each building on the preceding one: section 4.2.1 considers the definition of graphs and graph patterns in a program; section 4.2.2 defines a graph pattern matching operator and gives an algorithm for its efficient implementation; and section 4.2.3 defines a graph transformation operator.[3] Section 4.3 considers a means of

---

[3] In chapter 3, the two functions were combined in the single operator "=>"; here they will be first separated, for purposes of discussion, and then recombined by means of a unifying construct.

combining individual transformations into a graph transducer by means of what practical experience suggests to be a natural, and current program correctness concerns suggest to be a well formed, programming construct. Where appropriate, implementation algorithms are described, and historical background is surveyed.

4.2.1 Describing graphs in programs

Throughout chapter 3, we represented graphs pictorially. In one sense, this is the most natural representation for a graph -- a picture, as the anonymous wise man said, is worth a thousand words -- and it served us well enough in communicating the basic concepts of graph transformation. There is a severe problem with this representation, however, if the representation is to be embedded in a programming language: programs are, with some exceptions, strings of text. Some early programming languages were line oriented, and there are occasional suggestions for a return to this form, especially in giving two dimensional representations to certain constructs, like sub- and superscripts:

$$A_{i,j} := x_i^2 + y_j^2$$

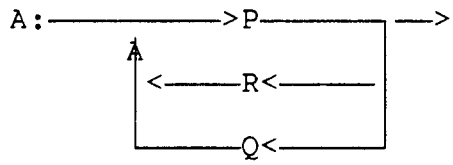
(which, admittedly, is more like natural mathematical notation than "A(i,j) := x(i)**2 + y(j)**2"). However, most of our compiler technology, especially the immense body of lexical scanner and parser technology built up in the wake of Algol 60,

depends on a stream orientation for languages, and we depart from this orientation only at the risk of losing that technology.

An interesting example of this need for "linearization" may be found in parser description techniques. Although, as we determined in section 2.1.1, BNF is a simple and powerful language for the expression of syntax, it has certain limitations when it comes to producing efficient parsers for arbitrary BNF specifications, that have proved very difficult to characterize except through an algorithm for generating parsers from BNF descriptions.[4] However, recent years have seen the comeback of transition diagram representations of grammars, a method that dates back to the early sixties [27]. A grammar of the form

```
A ::= P                          A' ::= RA
      PA'                               QA
```

may be expressed diagramatically as



This notation was put to use systematically in the definition of Pascal [64], and several "languages" have been invented since to express these diagrams in linear form for parser generators, all

---

[4] Compare Aho and Ullman [3], chapter 6. This problem is analogous to the problem of giving precise descriptions of programming language semantics without resorting to specifying a compiler.

resembling the relationship between flowcharts and (goto replete) programming languages like Fortran. A recent (1979) use was in extending such diagrams to "semantic" processing of programming languages [28], an extension to syntax diagrams of the syntax directed translation technique.

Since programming language syntax is highly hierarchical (a result of its being almost context free), a primarily list structured representation with a few labels and "gotos", would suffice to represent syntax graphs. Such a representation, which was introduced as a tree description language in the augmented grammar of section 4.1, was used as the basis for a preliminary graph representation language. This proved to be relatively pleasant to work with, primarily because the number of labels and "gotos" tended to be small: for example, the parse graphs of L programs were described in section 4.1 without once resorting to a label. A representation of the if-then pattern on page 70 would look like this, for instance:

```
L: "if-then" < c ,
               "seq" < s ,
                       "goto" < !L > > >
```

(Here the "L:" construct labels a node, and the "!L" construct represents an arc to the node so labelled.)

The other major approach to linearizing complex patterns is to build them up out of simpler subparts, in a manner analogous to the way in which BNF specifications define an entire language by recursively defining its basic parts. Pfaltz and Rosenfeld

[98] present a grammar of picture components as a basis for formally describing the manipulation of complexly related objects (games, surfaces, biological systems). Shaw's Picture Description Language [116] is a calculus for forming pictures out of simple basic components like oriented lines, arcs, etc. Feder [38] uses a similar description language for pictures, based on grammars, and investigates its formal properties in relation to string grammars. Sugito et al. [121] present a graph manipulation language based on Pfaltz and Rosenfeld's web grammars, while Shapiro and Baron [115] do the same in a calculus resembling Shaw's. Knowledge representation languages depending on complex relationships between components are often linearized in this fashion [14], and have in turn influenced the form of TCOL's linear representation language for programming language constructs [78].

Although these representations are generally adequate and many can be shown to be complete in the sense that they are able to represent any programming language construct, or indeed any construct generable by a context sensitive grammar, they are not as natural to a discussion of programs as the linear text of the program they are ultimately intended to represent. For example, the expression

L: if c then s; goto L fi

is more natural than the graph pattern above, or even the graph that pattern linearizes. At the same time, while the program is natural, when it is used in a transformation, as in its original

setting on page 70,

```
while c do s od     =>
     L: if c then s; goto L fi
```

it introduces the questions: What is the structural relationship of this fragment to any program it is a part of, before and after the transformation? What relationship, if any, does the label L have to other labels in the rest of the program? and What syntactic entities do the variables c and s represent? (The last question relates to the syntactic macro problem discussed in section 3.1.1).

The answers to these questions lie in the complete syntactic definition of the language. Just as the human programmer "knows" (sometimes incorrectly) the syntactic structure of such a program fragment when he is writing it down or reading it, whereas the compiler must painstakingly eke out this structure (always, we expect, correctly), so too the human author of a translator knows what he means by such a transformation, whereas the translator writing system must have the wherewithal to acquire this knowledge from a "declaration" of the syntax involved.

We would like a mechanism for this declaration, therefore: we would like to be able to say that "while c do s od" is a program fragment representing a syntactic entity; to declare the precise structure of that entity; to declare the same information about the construct "L: if c then s; goto L fi"; and

to present these declarations in a form that makes it possible to extract the answers to our questions about the translation defined above.

To be more specific, let us consider the following program as a specification of the above translation:

```
begin
  language Struct is
    <stm>↑tree
      ::= "while" <exp>↑exp "do"
            <stm>↑stm "od"
          [tree := "while"<exp,stm>]
        | "stm" [tree := "stm"] .
    <exp>↑tree
      ::= "exp" [tree := "exp"]
    end;

  language Unstruct is
    <stm>↓ids ↑tree
      ::= <id>↑id ":" <stm>↓ids ↑tree
          [ids@id := tree]
        | "if" <exp>↑exp "then" <stms>↓ids ↑stms "fi"
          [tree := "if"<exp,stms>]
        | "goto" <id>↑id
          [tree := "goto"<ids@id>]
        | "stm" [tree := "stm"] .
    <stms>↓ids ↑tree
      ::= <stm>↓ids ↑tree
        | <stm>↓ids ↑stm ";" <stms>↓ids ↑stms
          [tree := "stms"<stm,stms>] .
    <exp>↑tree ::= "exp" [tree := "exp"] .
    <id>↑id ::= [defined lexically]
    end;

  Struct: { while exp do stm od }
       => Unstruct: { L: if exp then stm; goto L fi }
  end
```

Two of the questions asked in connection with the example on page 135 are answered here: the structural relationship of the two fragments with any containing structure is defined by the syntax of the two languages in conjunction with the rules for graph replacement defined in chapter 3; and the relationship

of the label "L" in the target to any other labels "L" in the containing program are made explicit by the complete syntactic definition of Unstruct, also in conjunction with the rules for graph replacement (there is no such relationship).

What has occurred here is a process of abstraction: the programmer defines once for all what he means when he says a piece of text (enclosed in braces) is a program in language Struct, say; he can then use such pieces of text as abstract entities, knowing fully that, underneath, the compiler has translated his text into a graph representation, and that this is what he is really manipulating. This notation will be extended, in the next section, to deal with graph patterns, by turning it into a kind of typed lambda notation, and thus answer the third of our questions about the syntactic transformation example, namely, What do "c" and "s" represent?

At this point we have come through the full circle described in the introduction. From an analysis of earlier translation efforts we concluded, in chapter 2, that there was something innately attractive about the view of translation that treats it as a textual transformation, but that this attractive vantage is often obscured behind the very real limitations of string, and even context free (tree) patterns in dealing with the full range of translation concerns. In chapter 3 we examined a graph transformation formalism which, it was demonstrated by examples, promised to give just such unambiguous

expression to this full range of translation concerns. However, the graphs, it was readily perceived, presented many of the same problems in representing translation as do trees in representing program structure, or flowcharts in representing control flow. Only now, by the promised process of abstracting the underlying graph representation in such a way that the program fragments can once again be represented as strings of text, have we been able to have it both ways: on the one hand we have a rigorously defined program structure, and on the other we have a translation superficially defined in terms of a textual transformation.

The only question remaining is a practical one: We know that there are parsing techniques for complete programs in a language, but how can we be sure that program fragments are generally parsable? The answer is twofold, once bottom up and once top down: first, assuming the fragment reduces to a single nonterminal (a highly reasonable assumption for translation, where the syntax directed translation model has thrived under exactly this assumption), any bottom up method will produce such a nonterminal for such a fragment, provided it is capable of parsing the language, whether or not it is embedded in a larger program (i.e., whether the nonterminal is part of a larger reduction, or an end product); second, a general top down parsing strategy first defined by Earley[5] will generalize to

---

[5] Earley's paper [36] is a bit vague; the best description to my knowledge is by Graham and Harrison [49]. Earley's algorithm

take any number of nonterminals, up to the entire nonterminal vocabulary, as its starting point, and is therefore guaranteed to find a parse for even a program fragment.

4.2.2 Graph pattern matching

The preceding section was concerned with presenting an abstraction of the concept of syntactic structure. This abstraction was such that the relationship of the structure to the written form of a language could be defined once, in all its finicky detail, and subsequently constructs in the language could be written down in a natural "high level" formulation without losing the precision inherent in the abstracted structure. We examined one example of a syntactic transformation amounting to a translation, in the expression

---

is notorious for being costly: in the worst case, that of an ambiguous grammar, it operates in time proportional to a function of the cube of the length of the input $O(n^3)$; only for deterministic grammars, which are not always the most convenient description of the language, does it operate in time proportional to the length of the input $O(n)$; and for most grammars, the unambiguous but nondeterministic grammars in which most programming languages are defined, its complexity is $O(n^2)$. Furthermore, none of these measures takes into account the enormous overhead involved in performing the computation described by this algorithm. However, we are not talking here about input strings of upward of a thousand symbols, such as normal programs, but of tiny program fragments of from ten to fifty symbols, and for these Earley's algorithm is quite acceptable. (An extremely recent contribution to general context free parsing by Graham *et al.* [50] generalizes the work of Earley and others, and promises lower overhead; the worst-case complexity remains the same, however.)

```
Struct: { while exp do stm od }
     => Unstruct: { L: if exp then stm; goto L fi }
```

Here, the material in the brackets represents two program fragments in a not very useful pair of programming languages. In general, however, we will want to express not merely program fragments, but patterns in the language, so that translations of the (possibly infinite) set of all programs in the language can be expressed in a finite (indeed small) set of transformations. Referring again to the above example, we want to translate not only the program fragment "while exp do stm end" in this manner, but all program fragments of its syntactic form, where "exp" and "stm" stand for arbitrarily complex sub-fragments of the syntactic forms <exp> and <stm>, respectively; i.e., "exp" and "stm" are to become variables in a transformation pattern.

The use of variables in this context is analogous to the use of variables in procedure definition. The definition.

```
f = lambda (x) (x+x)
```

defines a kind of pattern for any program in which f occurs, and defines as well what effect f has on its environment, by using the variable x, representing f's immediate successor, or argument, in its definition. For example, the expression

```
a + f b - c
```

is, by the above definition, equivalent to the expression

```
a + (b + b) - c
```

Similarly, the expression

```
Struct (e , s) : { while e do s od}
```

represents a syntactic pattern in the language Struct, where "e" and "s" act as variables, such that the pattern matches any of the following expressions:

```
while exp do stm od
while exp do while exp do stm od od
while exp do while exp do while exp do stm od od od
etc.
```

However, we are up against the serious question: How do e and s relate to the syntax of the language? Specifically, how do we restrict the pattern to matching

```
while exp do while exp do stm od od
      ‾‾‾     ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾
       e                 s
```

and not

```
while exp do while exp do stm od od
      ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾     ‾‾‾‾‾
             e                   s
```

(the syntactic macro problem discussed in section 3.1.1)? This question has important ramifications if the abstraction is to work correctly, since it depends on there being known parsing techniques that are able to handle its details. It is resolved by resorting once again to the analogy with procedure definition: if we have to define the procedure f so that it will operate not only on integer and real values, but on Boolean values as well, and if we assume that the operator "+" is not defined for Boolean arguments, we must define the procedure with respect to the type of its argument:[6]

---

[6] Algol 68 and Smalltalk are examples of languages providing this kind of abstraction facility. The language Ada calls this type of definition overloading, and provides for it in its definition.

```
f = lambda (int x) (x + x)
f = lambda (Bool x) (x or x)
```

In syntactic pattern matching, the syntactic definition not only provides an abstraction for the underlying syntactic structure of a program segment, but also a set of syntactic types defined by the nonterminal symbols in the defining grammar. Our pattern now becomes

```
Struct ( <exp> e, <stm> s) : { while e do s od }
```

which is unambiguous. In effect, the abstraction mechanism is asked to discover the underlying syntactic structure of this program fragment by applying a parser for language Struct to the sentential form

```
while <exp> do <stm> od
```

which, it can determine with no difficulty, is a production of the symbol <stm>.

Having defined a pattern description terminology, we can now turn our attention to pattern matching. Let us define a pattern matching operator "::", to be a Boolean operator such that

```
G :: P
```

is true if graph G matches pattern P. That is to say, if there is a subgraph of G, rooted at the same node as G, which has the same "shape" as P and, insofar as the nodes of P are labelled, has the same labels on corresponding nodes. Thus,

```
Struct : { while exp do stm od }   ::
    Struct (<exp> e) : { while e do stm od }
```

is true, while

```
Struct : {while exp do while exp do stm od od}   ::
    Struct (<exp> e) : { while e do stm od }
```

is false. On the other hand,

```
Struct : {while exp do while exp do stm od od}   ::
    Struct (<exp> e, <stm> stm) : { while e do stm od }
```

is true.

The only practical consideration remaining in this discussion of graph pattern matching is the efficient implementation of a pattern matching algorithm. In general, the problem of pattern matching over the set of directed graphs is NP-complete,[7] but the following algorithm will demonstrate that, given the syntax graphs used here, and reasonable restrictions on the match, rooting it at the defining nodes of the subject and pattern graphs, the complexity of the problem is proportional to the number of nodes in the pattern graph.

Preliminaries

The following is an algorithm for the evaluation of "S::P", where S is a node defining a graph (K,G,S) and P is a node defining a graph (L,H,P), as defined in chapter 3. P, the pattern, may define a pattern graph (one containing terminal nodes with special symbols); S, the subject, never defines a

---

[7] Garey and Johnson [45], p 202: Not only is determining whether a graph G contains a subgraph isomorphic to another graph P NP-complete, but with the restriction on G that it be a directed acyclic graph, and on P that it be a tree, the problem is NP-complete as well. These restrictions are far more stringent than those we will be willing to impose on syntax graphs and their pattern matches.

pattern graph. The algorithm is recursive, and is performed in the following environment:

- V is a set of variable symbols used, if at all, in labelling nodes of P only.

- A is an initially empty set which will accumulate variable assignments during the pattern match. Thus, where the pattern contains a node labelled by a variable symbol $v \in V$, and this node is matched against a node in the subject, the graph defined by the subject node s is "assigned" to v by being associated with it, as $(v,s) \in A$, in step 3 of the algorithm.

  T is an initially empty set which prevents cycles in the match by recording, in step 6, that a node p of the pattern graph has been matched with a node s of the subject graph, as $(p,s) \in T$; steps 1 and 2 of the algorithm prevent any attempt to either "rematch" p with another node $s' \neq s$ once it has been matched with s, and any attempt to go through the recursive match on p's children a second time.

The algorithm consists of seven steps at any one of which the match may succeed or fail, and the algorithm thus terminate, depending on a condition tested at that step. Thus, at step n, we can always assume that the conditions determining steps 1 through n-1 have tested false.

Algorithm (S :: P)

(1) If $(P,S) \in T$, the match succeeeds.

(Recall that T is initially empty. In this case, the pattern node P has been visited before (see step 6) and was then also matched against the subject node S. We can assume, therefore, that a recursive match of the children of S and P will tell us nothing new -- and would, in fact, cause the match S::P to recur infinitely.)

(2) If $\exists n \neq S$ such that $(P,n) \in T$,
or $\exists m \neq P$ such that $(m,S) \in T$, the match fails.

(In this case, the pattern node P has been visited before, but was matched against some other node in the subject graph, or the subject node S has been visited before, but was matched against some other node in the pattern graph. Since there cannot be two such matches, this one must fail.)

(3) If $L(P)=v \in V$ then

$A \leftarrow A \cup \{ (v,S) \}$, and the match succeeeds.

(In this case, the pattern graph was a variable, represented by v. The set A is updated to indicate that v has been assigned the graph defined by S -- pending successful completion of the entire pattern match.)

(4) If $L(P) \neq L(S)$, the match fails.

(In this case, P and S have different labels; that is, the graphs defined by nodes P and S have different roots, and therefore do not match.)

(5) If $\exists n,m \in N$ such that

$G(S,n) \neq 0$ and $G(S,n+1) = 0$ and
$H(P,m) \neq 0$ and $H(P,m+1) = 0$ and $n \neq m$, then

the match fails.

(In this case, P and S have different numbers of children.)

(6) (At this point the nodes P and S have been found to be the same: P is not labelled with a variable, and has not been visited before; P and S have the same labels and the same number of children. It remains only to compare their children, in order of age.)

$T \leftarrow T \cup \{ (P,S) \}$, and

$i \leftarrow 1$, and

while $G(S,i) \neq 0$ do:
if $G(S,i)$ :: $H(P,i)$ fails then S::P fails.
else, $i \leftarrow i + 1$.

(Set T records that P has been visited and was matched with node S: this prevents an advance beyond step 2 in future visits to P, and thus prevents infinite loops. The match then proceeds recursively, matching children of S and P, in order, stopping as soon as two children fail to match.)

(7) The match succeeds.

Cleaning up

Once the match is over, the set A contains a pairing of assignments of nodes to variables. If the match was successful, these assignments need to be actually performed, and sets A and T can then be made empty sets again for the next pattern match. Notice that no attempt is made to prevent a variable v from being assigned two values in A. This will happen only if v occcurs as a label on two distinct nodes of the pattern graph: such a pattern would be ill-formed, and in an actual implementation it would have to be prevented.

The primary reason for presenting this algorithm is to demonstrate that it is possible to restrict graph pattern matching in such a way that the complexity of the matching process is proportional to the size of the graph. The algorithm permits us to make the following assertion:

> In the evaluation of the pattern matching algorithm, no arc of the pattern graph is traversed more than once.

Proof:

> If the root nodes fail to match, of course, no arcs are traversed, but let us suppose the root nodes match, and a recursive match of the children takes place. In that case

-- see step 6 of the algorithm -- the set T is updated to indicate that the pattern node has been visited (and matched against the subject node), and on any subsequent visit to the pattern node, the algorithm cannot get past step 2, and thus cannot traverse the arcs leading out of the pattern root node a second time. By induction on the number of nodes in a pattern, it follows that no arc can be traversed more than once in a match.

From this it follows (1) that the complexity of a pattern match is proportional to the number of arcs in the pattern graphs (because they are traversed at most once), and (2) that all pattern matches terminate (since pattern graphs are finite, and no arc can be traversed more than once).

4.2.3 Graph transformation

We have, so far in section 4.2, progressed to the definition of a syntactic pattern matching facility based on an abstraction of a graph representation of the language's complete syntactic structure. The next stage is a definition of graph transformation and, as in the definition of graph patterns, we will do well to look for existing models on which to base such a definition.

Among the earliest transformational systems are "production" systems and Markov systems, which precede and are

closely resembled by both BNF and the Snobol string transformation statement form:

<subject> <pattern> = <new string>

(the part of the <subject> string matched by the <pattern> -- if at all -- is replaced by the <new string>), and numerous other string and symbol manipulation systems. Floyd defined a symbol manipulation language based on production systems [40], later modified by Evans, which made it possible to program a parser with semantic actions that transform the parse stack. Ledgard [76,77] has extended production systems to a more syntactically structured model, by extending it from string manipulation to tree manipulation, also for purposes of syntax directed translation.

Another major use of a transformational model is in the Vienna Definition Language, where the abstract syntax of a program is represented as a tree with labelled nodes and labelled branches. The operational semantics of a language are defined in terms of interpretive transformations over the abstract syntax of the language. These transformational manipulations are performed by the mu operator, which can (a) add branches to a node, (b) delete branches, and (c) change the subtree connected to a branch. The label on a branch acts as a selector for the subtree at the other end. VDL is a very primitive structure manipulating language modelled on Lisp; for the uses it has been put to, there was, according to Wegner, no need for either more complex syntactic structuring, nor for a

more abstract transformational operator ([139], p 12). The reason for this is clear: VDL was intended, not as a programming language, but as a definitional model; as such it had to be small enough to be easily defined in itself, and the least amount of abstraction would have increased the size of such a definition; for the same reasons, a context free model for syntax was the preferable one, because of its simplicity in definition, even though a symbol table technique had then to be implemented in the language in order to deal with the context sensitive aspects of of programming languages. Much of the value of VDL and similar operational models (for instance, Semanol [4]), has been diminished by the subsequent appearance of the denotational model which, less dependent on an underlying computational model, is also less cumbersome.

We have been using, up to now, the operator "=>" to represent transformation. In chapter 3 where this operator and its effect on graphs was defined formally, it combined, for simplicity of notation, the functions of pattern matching and transformation in expressions like

Pascal: {complex} => Pascal : { record r,i: real end }

(page 70). Here, the pattern consisting of the identifier "complex", matched against a nebulous notion of the subject program, caused a transformation, in the subject program, of the subgraph representing the matched identifier into a subgraph representing the record declaration (an example of the implementation of language extensions). We now have the pattern

match operator "::" and its extensions defined below to make specific reference to the subject graph being matched, and can use the operator "=>" purely as a transformation operator. Thus,

```
if program :: Pascal: {complex} then
  program => Pascal: { record r,i: real end } fi
```

This transformation will happen often enough that a convenient shorthand notation may be used, similar to that of Snobol:

```
program matching Pascal: {complex}
  => Pascal: { record r,i: real end }
```

4.3 Using graph transducers in translations

We now have at hand the basic operations of graph pattern matching and graph transformation, and a means of combining them into a single operation, analogous to the Snobol string replacement operation. These two operators, along with the representation of syntactic graph patterns and the syntactic abstraction mechanism defined earlier, present us with enough manipulative power such that, if they were appropriately embedded in a standard programming language, it would be possible to write translators, in that language, based on the syntactic transformation model.

However, the concept of a "standard" programming language varies rather wildly depending on one's perception of

programming; and, since there have been several developments in programming methodology in the last decade that, I believe, bear directly on the transformational view we have been taking of translation, it is worth bringing up these topics here, so that they may shed some light on any considerations we may have for a transformational translator writing system.

We have already remarked, in section 3.1.3, and will do so again in section 4.4, that transformations often can be structured as a sequence of subtranslations. Particularly, the transformational view of translation, in which one construct is transformed into another construct, can be extended to a higher level view in which a program in one language is transformed into an equivalent program in another language. Experience with a transformational TWS has shown not only that this view is transportable to the design of a translator (the Pascal S translator reported in section 4.4.1 contains two major procedures representing, respectively, the Pascal S to Pcode, and the Pcode to Assembler translations), but that it imposes a highly regular structure on the implementation of each translator step (which we have chosen to call a transducer), such that it is primarily a set of rules for transforming individual syntactic constructs in the subject program. For example (from Kernighan [68]):

```
program matching Ratfor(<expr> e, <stms> s1,s2):
                    { if (e) { s1 } else { s2 } }
  => Fortran(<expr> e, <stms> s1,s2):
       { if (e) goto 10 ;
           s1 ;
           goto 20 ;
       10 s2 ;
       20 continue ; }

program matching Ratfor(<stm> s, <stms> ss): { s ; ss }
  => Fortran(<stm> s, <stms> ss): { s ;
                                    ss ; }

program matching Ratfor(<expr> e, <stms> s):
                    { while (e) { s } }
  => Fortran(<expr> e, <stms> s):
       { 10 if (e) goto 20 ;
              goto 30 ;
         20   s ;
              goto 10 ;
         30 continue ; }

etc.
```

Fortran expresses itself poorly (note use of semicolons as newline markers), and there is some question whether anything but an ad hoc parser will handle even this much of the language, but the point is clear: the Ratfor to Fortran transducer consists of a set of independent transformations, and the only thing not specified above, other than the actual grammars for Ratfor and Fortran, is how the transducer is to tour the Ratfor graph to turn it into a Fortran graph.

Fortran statement numbers act like labels in this context. The above translation does not imply that a Ratfor program containing, say, two occurrences of an *if* statement will be translated into a Fortran program incorrectly containing two occurrences of the statement number 10. The statement numbers merely express a syntactic relationship between the statements

represented in these fragments, a relationship that is purely local to the statement fragments. As with labels in earlier examples, they will disappear in the process of turning these fragments into pattern graphs, the translation will transform similarly structured graphs in the same manner, and it will then be the function of the Fortran "flattener", which performs the inverse function of a parser, to assign unique statement numbers to the statements referenced by goto statements.

This transducer-like structure of translators is not surprising: it is a direct result of the structure of the problem, which is in turn determined by the structure of programming languages. The syntax directed translation model was not so successful for so long without good reason. Indeed, Rosen has studied tree and DAG transformation systems analogous to string transformation systems like grammars [104,105], and has come up with conditions guaranteeing that a set of transformations will have a unique result, independent of the order in which they are applied -- a result analogous to ambiguity results for grammars. Rosen suggests applications in a limited area of translation -- proofs of optimization correctness -- but the basic principle clearly has potential for wider application, so that various subtasks of translation can be expressed and proved correct in this fashion. Dijkstra has used a similar approach to the expression of algorithms in such a way that they can be easily expanded into remarkably elegant, and provably correct, programs, by use of the guarded command

construct [34]. A closely related, and nearly contemporary paper by Chirica and Martin [23] uses Floyd-Hoare inductive assertions to prove the correctness of translations performed by compilers, using as its basis the notion that a compiler transforms a syntactically complete program fragment in the language into an equivalent program fragment in another language, where two programs may be defined to be equivalent if they perform the same predicate transforming function. For example, using relationships derived by Floyd, Hoare [42,59] and others, it is possible to show that

{p} do S until C {q}
and {p} L: S; if ¬C goto L {q}

for all predicates p and q; this implies that the two program fragments are equivalent and that each therefore represents a correct translation, into another idiom, of the other.[8]

There are also indications, both theoretical [119,104] and practical [17,21] that translators can best be structured as a collection of independent "passes" or transformations, most usually a sequence but, in Sklansky et al. [119], for instance, a network of transducers. DeRemer and Kron [32] have argued that constructing a system differs from the construction of a local process not only in magnitude, but also in the kind of considerations required of the programmer: primarily in dealing

---

[8] I am using here my own formulation [129], developed independently of, but somewhat later than Chirica and Martin's, as a proposal (unimplemented) for a translation synthesizer along the lines of Arsac and others [6].

with the local processes as well defined "black box" functions.

In connection with this, we may choose to think of programming language transducers as black boxes that can be described by the well known "T diagram" which is one of the legacies of the Uncol project and is useful for describing a bootstrap: a translator from language A to language B, written in language L, can be thought of as a function T(A,B,L), and two such functions can be combined in a prescribed manner to deliver a third such function; the process of building up functions toward a desired result is generally useful in developing a bootstrap sequence. For instance, if we have a translator from language A to language B written in (high level) language L, and we want to implement it on machine M, which has no compiler from L to M, we use Uncol U in a three stage bootstrap as follows:

(1) We define the only meaningful combination of two translators to be T(K,L,M)°T(I,J,K) producing T(I,J,L) (i.e., a K-to-L translator applied to an I-to-J translator written in K produces an I-to-J translator in L).

(2) We write two additional translators, both of which should be, if U is a proper Uncol, much easier to write than the T(A,B,M) translator we desire:

T(U,M,M) and T(L,U,U)

(3) We can now complete the process by three translator combinations:

T(U,M,M)°T(L,U,U) producing T(L,U,M)

T(L,U,M)°T(A,B,L) producing T(A,B,U)

T(U,M,M)°T(A,B,U) producing T(A,B,M)

Variations of this process have been at the back of much of the use of intermediate languages in portability experiments [93], and in designing complex translators as a sequence of steps [17], or even as a network of translations [9,44] resembling Sklansky et al.'s network of transductions [119].

The importance of this kind of program construction consists in its ability to keep the complexity of programs down: an important asset in the construction of a translator. Thus, once a transducer is written, and we are assured of its correctness by whatever means we choose to depend on,[9] we may safely use it as a "black box" through which a program in language A may pass and emerge transformed as an equivalent program in language B.

For the above reasons, it is worth briefly considering the construction and combination of transducers. We introduce, for purposes of discussion, a notation resembling Dijkstra's guarded

---

[9] I use the term "assured" advisedly. I am well aware of the immense and thankless task involved in formally proving the correctness of even a small program using existing proof methodologies (compare, in this context, the arguments of DeMillo et al. [31]). However, most conscientious programmers go through some semi-formal process whereby they assure themselves of the correctness of their programs rather than submitting them blindly to the machine to be run; the more a language is designed to simplify this process, the more effective that language will be as a programming tool, therefore.

command notation:

```
transform Program matching
  pattern1 => program1 ¤
  pattern2 => program2 ¤
    ...                ¤
  pattern  => program  end
```

The portions indicated by "Program", "pattern" and "program" are expressions evaluating to, respectively, a graph, a graph pattern, and a graph. Free variables introduced on the left hand side of a transformation operator carry on to the right hand side, so that we can now write the Ratfor to Fortran transducer as follows:

```
RF := transform program matching

        Ratfor(<expr> e, <stms> s1,s2):
          { if (e) {s1} else {s2} }
            => RF(e); RF(s1); RF(s2);
               Fortran (...):  {...}  ¤

        Ratfor(<stm> s, <stms> ss): { s ; ss }
            => RF(s); RF(ss);
               Fortran (...):  {...}  ¤

        Ratfor(<expr> e, <stms> s):
          { while (e) { s } }
            => RF(e); RF(s);
               Fortran (...):  {...}  ¤

        etc. end
```

Notice that the transducer is now given a name by being assigned to a variable, and is defined recursively to transform the subparts of the statements. All of this is, of course, shorthand for:

```
RF := procedure (program);

        if program :: Ratfor(...): {...} then
          RF(e);...;RF(s2);
          program => Fortran(...): {...} fi;

        if program :: ... etc.
```

The reasons for introducing the new notation are that: (1) It is cleaner looking, especially because the subject graph appears only once in the transducer, instead of two times the number of transformation rules, and because the patterns now appear in a prominent position at the beginning of each rule so that transformations are once again, as in chapter 3, identified by their pattern, and the subject has become a well defined but, for the purposes of discussion of individual transformations, abstract entity; and (2) Assertions about a set of transformations collected into a transducer, whether stated formally or informally, are now more readily formulated and proved [34].

What has been presented here is enough of a graph transformation language to indicate the form a translation might take when expressed in such a language. Such a translation would take full advantage of the concept of syntactic transformation, but would not lose any of the considerable advantages provided by syntax directed translation. A transducer such as the one above represents in essence a one pass syntax directed translation; we have even given it the production system style of construction found in, for instance, BNF specifications. The only difference is, that the

transformations are not tied strictly to any particular parsing technique, whether top down, bottom up, left or right reducing, operator precedence, k-symbol lookahead, or even ad hoc. The translation is, in fact, as independent of the concrete syntax of the language as it is necessary to be.

The next section will report on two trial implementations of translators, using this approach. One of these was substantial, and relatively realistic, and, although based on a more primitive set of constructs than those defined above, the results showed rather conclusively that a very large part of the process of translation is expressed succinctly by these or similar constructs. It was this experience, in fact, which led to a redesign resulting in the above considerations.

4.4 Experience with a graph transducer language

In the preceding sections, we examined some of the considerations that would lead to the design for a translator writing system based on the concept of translation as syntactic transformation. The partial design presented there represents a second version of such a language, based on practical experience with a first attempt. This section is intended to present that experience, and to indicate what effect it had on the design.

A graph transducer "language" was implemented as a set of Lisp function definitions. These were directly related to a

minimal design for an Algol 60-style language; the language itself was deliberately restricted to the following set of constructs:

```
subject :: pattern
subject => object
ordinary arithmetic and Boolean operators
variable := expression
procedure (A1,...,A) E1;...;E end
procname (E1,...,E)
if E0 then E1 elsif E2 then ... else E fi
```

Graph patterns were constructed by

```
label: node < pattern1,...,pattern >
!label
?  or  ?variable
```

Declarations could occur anywhere in the program, their range being limited to the "nest"[10] in which they were declared, and were of the forms

```
var x
var x := memory
```

The second form defines a symbol table, which is accessed by

```
x [ expression sequence ]
```

Symbol tables were used only in programming the context sensitive (first) pass over the parse tree, and were the first to go in the revised design, above, which relegates all syntactic matters to the parse. In this more primitive version, the input was a context free parse tree, and the contextual information had to be gathered in a pass over the tree: here, symbol tables were an absolute necessity, but we have seen (section 4.1) how symbol tables may be abstracted by a more sophisticated syntax definition (which may, in fact, be

---

[10] A term taken from Algol 68 [131].

implemented as a tour of a context free tree [15]). As a result, the definition of the translation is completely freed from such implementation dependent considerations.

Graph patterns were "evaluated" by being turned into actual graphs. These primitive forms were, as I said in section 4.2.1, not unpleasant to work with, but they tended to become extremely cumbersome. Compare, for instance, the two equivalent expressions:

```
program matching "if"<?var expr,?var true,?var false>
  => "stms"< expr,
     "stms"< "FJP"<!f>,
     "stms"< true,
     "stms"< "UJP"<!e>,
  f: "stms"< false,
  e:         "NUL" >>>>>
```

and

```
program matching PascalS(<expr> expr, <stms> true,false):
 { if expr then true else false }
  =>              Pcode(<expr> expr, <stms> true,false):
       { expr ;
         FJP f ;
         true ;
         UJP e ;
       f: false ;
       e: NUL }
```

The original design included a simple looping construct and value returning general exit:

```
label: repeat
         ...
         exit label with expression end;
         ...
         end
```

However, this was never actually used. I included a looping construct because my programming style is such that it uses iteration about as often as recursion; it is not my programming

style, therefore, that dictated the absence of iteration in my translations. Instead, it stands to reason, the absence of iteration, and overwhelming dependence instead on recursion, is an artifact of the structure of the problem which, as we have remarked before, is in turn an artifact of the recursive definition of programming language syntax. It was by considering this, what at first appeared to be surprising, result, that I came to consider the use of the simple "guarded command" style construct of section 4.3

A note about efficiency of implementation is appropriate here. The procedure calling mechanism imposed by the architecture of "modern" computers on modern programming languages has, with the notable exception of a series of Burroughs machines designed to support a dialect of Algol 60, been forced to be singularly inefficient, with the result that the term "recursive" has more often than not been taken as synonymous with "accursed". Recall, however, that the recursion here is a definitional abstraction of a tree touring algorithm, which implies the need for a stack, but none of the complicated register saving/restoring and display maintaining code involved in normal procedure calling.[11] A considerable portion of the work on high level optimization has been concerned with just such recursion elimination for the sake of backward machine design, and would be particularly applicable here.[12] The fact

---

[11] Aho and Ullman [3], pp 356-364; Gries [53], pp 193-203.
[12] Knuth [72], pp 280-282; Darlington and Burstall [30].

that this kind of recursion will be common in a translation language dependent on syntactic structure, and easily isolated, means that it will be readily optimized.

This was a typeless language, and was so defined because (1) it is easy enough to implement a typeless language in Lisp, and (2) there were no clear reasons for insisting on type declarations. However, the considerations of section 4.2.2, on graph pattern matching, now indicate that typing, at least of pattern matching variables, is useful in creating a more expressive language for patterns. Practical considerations, such as the fact that typed languages provide better error detection, and are more efficiently implementable than typeless languages, are another reason for including typing in the design of the second version. Note, however, that the language has a flexible typing facility, like the most modern languages, and that this was a direct result of the syntactic abstraction introduced in section 4.2.2

Marking the graph nodes as to their syntactic types may have a further benefit: other structure manipulating languages, like VDL and the PQCC, tag the arcs out of a node, so that individual arcs can be addressed, much as individual fields in a data structure are normally given names so that they can be addressed symbolically. In the original version of the graph transformation language, this capacity was not present, and the only pass in which it was badly missed was the context dependent

information gathering pass. It may be, therefore, that it will not now be necessary; but this conclusion is too much affected by other changes to the language (especially by the removal of context sensitive processing to the parser) to be reachable without further experience.[13]

In the next two subsections we will examine the two translations that have been expressed in the experimental graph transformation language. These sections will concentrate especially on the contribution made to these translations by the graph transformational view.

4.4.1 A Pascal-S compiler

The larger of the two efforts at using the experimental graph transformation language in expressing a translator was a Pascal S compiler. I felt that a reasonably complete example of a programming language translation was necessary (1) to provide an empirical proof that this approach to translation is at least

---

[13] Recall that, as in the above examples, individual fields are selected in a pattern match by individual identifiers, and may then be manipulated, as in the instruction

```
RF ( s )
```

However, if a total transformation is not necessary, but only a modification -- as often happens in context sensitive passes, where a single subgraph may be replaced, added, or deleted -- it is useful to be able to address the fields individually, as in

```
statement@<expr> := newvalue
```

reasonable, and (2) to try out one approach to the design of such a translation language on which to base possible, and indeed almost certain, revisions and extensions of the base concept. I also saw this as an opportunity to try out the effectiveness of the "graph transducer" basis for building up complex translators out of relatively manageable subtranslators.

To keep this effort simple enough to be feasible within the time allotted, it had to be restricted to an interesting but small language. I also felt it had to be an existing language that was relatively well known so that this report could be reasonably certain of communicating its primary information without having to include an entire description of the language, but especially so that there could be no opportunity of biases (actual or imagined) entering into the choice of common language constructs to include for translation -- particularly biases toward reducing the work by simplifying the problem. The language finally settled on was Pascal S [142] because (1) it is a proper subset of a well known language based on modern considerations of programming language design [64], (2) it is large enough to be considered a "teaching subset", and is therefore not a trivial or "toy" language, and (3) Wirth has published a program (in Pascal) which implements an interpreter for the language and I hoped to compare the two versions for size.

To demonstrate some of the translation scheme's

flexibility, I also decided: (1) to do the translation in two phases, from Pascal S to Pcode [94,13], and from Pcode to Assembler; and (2) to implement a language extension mechanism in the form of inline procedures. As a result of this decision, the size comparison proved to be out of the question, because the two implementations were too dissimilar: Wirth's version translates to Pcode and interprets this, mine continues to Assember; Wirth's is a syntax directed translator which makes it difficult to separate out even the Pascal S to Pcode translation for partial comparison to the corresponding pass in the transformational translator.

Aside from the two translation passes there was also a pass to collect contextual information (identifier types, identifier ranges, expression types). This pass was approximately as large as the Pascal S to Pcode translation pass, and it was this unacceptable size that resulted in the reconsideration of syntactic recognition reported in section 4.1

The ability to tour the syntactic structure at will proved particularly valuable in translating the Pascal case statement. Case statements usually create problems for one pass translators, although well known techniques exist for code generation in one pass which makes the case statement no worse than any other statement requiring backward jumps, except quantitatively [140]. The resulting translation is usually from

```
case selector of
  labels1 : statement1;
    ...
  labels : statement;
  end
```

to

```
     selector
     jump L1
S1:  statement1
     jump L2
      ...
Sn:  statement
     jump L2
L1:  test selector
     if out of bounds jump L2
     jump L3+selector
L3:  jump S?
      ...
     jump S?
L2:  NUL
```

(compare the Zuerich interpreter [94]). The above involves, besides being as well-structured as a plate of spaghetti, one more jump per evaluation than is absolutely necessary. A translation with fewer jumps, however, is possible only if code need not be emitted the moment a construct is reduced to its syntactic class, as in all syntax directed translations. A transformational translation would transform the subgraphs representing the statements into equivalent Pcode subgraphs, then generate the selector evaluation, jump table (containing pointers to the statements), and finally the statements.

In BCPL, which has a simpler case construct, it is not generally feasible to generate a jump table, although for a certain range and spread of selector values this is certainly preferable to the Lisp style conditional involving iterative

testing and jumping over alternatives or a table of value/address pairs. BCPL compilers implementing an intelligent code generation algorithm which decides between these two options inevitably must make a complete pass over the code of the case statement in order to decide which is preferable followed by another pass to generate the code. Here especially, a graph representation of the program is useful.

Pascal has something of a language extension facility, as mentioned on page 70, in its type definition statement. A definition of the form

type complex = record r,i : real end

is most easily implemented by replacing every occurrence of the identifier "complex" (within the range of the definition) by the subgraph representing the record definition. The Pascal S implementation included this simplification, which took place during the contextual information gathering pass. (A bit of a cheat: as discussed on page 86, language extension should be a separate pass for proper functional separation.)

It was decided also to include a parameterized language extension facility in the form of inline procedures. A definition of the form

macro M (A_1,...,A_n) S_1;...;S_m end

is implemented by replacing every occurrence of

$M(P_1,...,P_n)$

by (a subgraph representing):

$S';...;S'_m$

where S'_i is a copy of S_i in which every occurrence of argument A_j has been replaced by P_j ($1 \le i \le m$, and $1 \le j \le n$). This involved creating a copy of the subgraph representing

$S_1;...;S_m$

and substituting, for any occurrence of a symbol A_j, the subgraph representing P_j. No copy of P_j needs to be made; it is, of course, more efficient in space and time to substitute a reference to the same graph, especially if it represents an expression which might, subsequently, be subject to common subexpression optimization techniques. However, it is necessary to make a copy of "$S_1;...;S_m$" for every macro expansion. It was tempting to define a special language "feature" which performed this function, including the substitution of parameters; Lisp systems often have such a function for the parameterized copying of list structures, but it felt like a cheat, here, where its only obvious use was in macro replacement. In any case, the copy and substitution procedure proved to be reasonably small, not only because it did not have much to do, but also because it only needed to consider that portion of the Pascal S syntax (approximately half) concerned with computation, and could

ignore the portion concerned with declaration.

4.4.2 A SASL compiler

The other application of graph transformation to a translation problem was in a "compiler" for an applicative language SASL. Turner [126,127] describes a method for translating expressions in an applicative language (lambda calculus, pure Lisp, SASL, etc.) into a form in which there are no bound variables, and consisting only of applications of monadic functions. Only a finite number of new symbols are introduced in Turner's method, so that, for instance, the expression

```
lambda (x) (x+1)
```

is translated to

```
S ( S ( K plus ) ( K 1 ) ) I
```

(dyadic operators like "+" and "*" are turned into monadic functions "plus" and "times" returning as their value a monadic function -- a process known as Currying), where the new symbols S, K, and I are defined as

```
S f g x = f x (g x)
  K x y = x
    I x = x
```

The evaluation of the successor function above, applied to a value, 7, say:

```
lambda (x) (x+1) 7 => 7+1
                   => 8
```

now becomes

```
S ( S ( K plus ) ( K 1 ) ) I 7
  =>  S ( K plus ) ( K 1 ) 7 ( I 7 )
  =>² ( K plus ) 7 ( K 1 7 ) 7
  =>² plus 1 7
  =>  8
```

(six steps instead of two). However, Turner introduces optimizing transformations:

```
(1) SASL(<comb> e1,e2):       { S ( K e1 ) ( K e2 ) }
      => SASL(<comb> e1,e2):     { K e1 e2 }

(2) SASL(<comb> e1):          { S ( K e1) I }
      => SASL(<comb> e1):        { e1 }

(3) if (1) and (2) do not apply then
    SASL(<comb> e1,e2):       { S ( K e1 ) e2 }
      => SASL(<comb> e1,e2):     { B e1 e2 }

(4) if (1), (2) and (3) do not apply then
    SASL(<comb> e1,e2):       { S e1 ( K e2 ) }
      => SASL(<comb> e1,e2):     { C e1 e2 }
```

and new combinator symbols

```
B f g x = f (g x)
C f g x = f x g
```

For the successor function example, only rules (1) and (2) are necessary for the transformation:

```
S(S(K plus)(K 1))I => S(K(plus 1))I          [rule (1)]
                   => plus 1                 [rule (2)]
```

The factorial function uses all four transformations, and thus also requires the introduction of the symbols B and C. The optimization reduces its applicative form from 29 function and constant symbols to 15. These results are interesting in that they demonstrate an optimization by transformation in a computationally "clean" calculus, with the result that the optimization is, unlike ad hoc optimizations in conventional programming languages, provably correct.

A pure combinator language has the additional attraction for a consideration of optimization that the expression optimizations discussed in section 3.2.3 can be applied as well: that is, common subexpressions can be merged in a DAG representation of the program and need be evaluated only once. Turner also reports that the combinatory code has "some remarkable self optimizing properties including that constant calculations are automatically moved outside of loops..." ([127], p 32). This result alone should be enough to make those who struggle with optimizing compilers for standard programming languages sit up and take notice: it may be a chance phenomenon, but it may also be the beginning of a more regular, theoretically based approach to optimization.

Since the optimizations and the description of a compiler and combinator evaluator were all expressed by Turner as transformations on syntactically well formed applicative expressions, we decided it was worthwhile to implement at least some of these in the graph transformation language. The result was an extremely clean implementation of part of Turner's technique[14] expressed in terms very close to those used in the paper and, consequently, very short and readable.

For a comparison of the two styles of translation

---

[14] Only the optimizing and variable removing transformations were implemented; a complete implementation beckoned most invitingly but was abandoned in favor of more pressing matters.

expression, old and new, see the appendix, where the SASL optimizer is given in both styles. Compare these, especially the second, with Turner's papers, to get a feel for the near "naturalness" of the expression style. The text of the old style Pascal S translator is available for examination as well, but is too large to be included here for the minimal interest it is likely to receive; it will, however, be furnished on request.

Chapter 5: Evaluation and conclusions

This dissertation has presented arguments for the extension of programming language translation technology from a purely syntax directed model to a syntax transforming model. The result of such an extension, it was argued, would be to bring within reach of a single, all-embracing technique the various translation related programming activities that do not coincide so neatly with a purely syntax directed view but must, if implemented at all for translators produced by automatic means, be added on by ad hoc techniques.

A recapitulation is in order here of the specific aspects of translation that have been brought into a uniform relationship during the presentation of this thesis.

(1) Programming languages exhibit both context free and context sensitive relationships between their constituent parts that together form the complete syntactic (meaning-free) description of programs within the language. Traditional syntax- or, rather, parser-directed translation techniques have been limited, by theoretical limitations on automatic parsing techniques, to context free descriptions augmented by a symbol table or, where multiple passes were possible, to context free descriptions augmented by transformations of a symbol table. In a graph transformation model of translation, however, both context sensitive and context free descriptions are possible

simultaneously, in the same formalism, and without recourse to external concepts like symbol tables.

(2) Programs are not only described syntactically, but also with reference to the notion of control flow. Any sort of program improvement effort will be based on a description of this control flow, which can be represented and manipulated by the same mechanisms that make possible the representation and manipulation of context sensitive syntactic aspects. In examples presented above, we saw that the treatment of labels in such cases is essentially the same as that of variables -- except that the variables do not introduce cycles in the graph structure, while labels generally do. As a matter of fact, the traditional "tuple" representation of programs in discussions of translation and especially optimization can be seen to be essentially a directed graph representation, from which all syntactic structure has been purged, leaving only the control flow structure. Given this unification, we can see that such traditionally distinct areas as code optimization, program development by stepwise refinement, translation of "abstractions" such as macros, inline procedures, type declarations, and more sophisticated contemporary conceptions of abstraction, all represent variations on a base concept of program manipulation by syntactically structured substitution, and are in essence no different from straight production-level to machine-level language translation.

(3) Like many other descriptions of complex processes, descriptions of programming language translation suffer from any discipline which forces them to fit a monolithic, all-at-once model rather than allowing them to be described incrementally in terms of a number of stages or "passes." Even where the traditional syntax directed view of translation has permitted multiple passes, the syntactic structure of the program under translation remained static; and yet, efforts in compiler writing and program portability have increasingly turned to a multi-stage model in which descriptions of the translations involved are often based on an abstract intermediate language with a syntactic structure, albeit primitive, uniquely its own and distinct from either the source or target languages of the overall translation. The syntactic transformation model permits a treatment of the stages in such a translation in a manner uniform with the treatment of one-stage translations, and an external representation of these stages that permits the intermediate languages to be displayed in much the same way as they are displayed in papers on the subject of multistage translation techniques.

Most of these arguments were presented in chapters 1 and 2. In chapter 2 also, the groundwork was laid for an approach to translation intended to be successful in dealing uniformly with the traditional translation concerns while at the same time retaining as much as possible the highly natural use of syntactic structure in translation. The uses of syntactic

structure were outlined and a representation using directed graphs instead of trees was developed. The idea of graph transformation was presented there as an alternative to tree touring combined with code emission (the equivalent of syntax directed translation).

Chapter 3 was primarily concerned with examining the traditional concerns of translation and program manipulation as aspects of syntactic transformation. The concepts of graph and graph transformation were formally specified and the model they defined was used in the subsequent discussion. Language extension, porting, editing, code generation, optimization, and multi-pass translation were all examined in the light of this model; more recent directions in programming language manipulation were also shown to incorporate aspects of syntactic transformation.

Chapter 4 completes the survey of traditional concerns in terms of syntactic transformation by presenting the essential aspects of an essentially traditional translator writing system. Although this TWS design is somewhat unconventional in its emphasis on translation rather than syntactic acceptance, it is nevertheless traditional in its "batch" oriented design, performing the three steps of accepting, transducing, and code generating in a straightforward sequence. (Later on in this chapter, recent trends in translator writing will be examined: these differ significantly in their approach to the expression

of translation from the syntax directed model.)

Much of this dissertation, then, is concerned with a synthesis of traditional translation by means of a descriptive technique which differs from the syntax directed technique in that, where the latter depends entirely on a static syntactic representation of a program, and is driven by the actions induced on a parser by a grammar, the former dynamically alters the syntactic representation to suit the various subtechniques and to move the program by stages from the domain of one static syntactic description (that of the source language) to the domain of another (the target language), passing through as many intermediate stages, concrete or abstract, as is either necessary or convenient. Instead of a process which tours the syntactic structure determined by the syntax of the language under translation, and emits the translation in a left to right discipline, the proposal was to transform the syntactic structure on a local, incremental basis, until it had become a syntactically structured representation of the translated program. Chapter 4 also developed a language for the definition of transducers in which the programmer may treat the transduction as one, not on graphs, but on textual program fragments in their natural representation.

This presentation of traditional techniques alone, if it was at all successful in establishing that a uniform restructuring of translation techniques was possible, forms a

strong argument for the merits of syntactic transformation in translation. However, the argument is not complete, and that of chapter 3, because of the breadth of the subject, was no more than cursory in its examination of this traditional material. Except for a short example in the appendix, and the somewhat more elaborate project briefly described in section 4.4.1, the effectiveness of the model is thus far weak in empirical evidence.

Such evidence can only accumulate from acceptance of this technique, or some variation on this technique involving a similar approach to translation, by a significant portion of the community concerned with programming language translation. Much like the widespread acceptance of the syntax directed model. One possible route to this acceptance would be the implementation and dissemination of a translator writing system based on the design in chapter 4. This represents a large, but feasible effort, and it is one of my long-term goals to produce such a system. Since it <u>is</u> a "system" as presented in chapter 4, it can be constructed in parts, some of which may prove to be interesting computational <u>tools</u> in their own right. Let us briefly examine the primary parts making up such a system, and a strategy for their implementation, here.

<u>Graph support tools</u>

Under this heading falls the core of software tools

necessary to the other parts. These include the software to allocate space for, and construct, graphs, to perform pattern matching and graph transformation -- essentially to implement the material described formally (and with an eye to implementation) at the beginning of chapter 5 and in section 4.2.2. Such tools would be useful not only in a translator writing system based on syntactic (graph) transformation, but in many applications whose underlying model is a directed graph which it may be necessary to manipulate: knowledge representation networks, picture description networks (see section 4.2.1), and other complex hierarchies, such as databases. Indeed, the recursive structure of a relational database closely resembles the recursive structural relationships imposed on a program by the grammar of the language it is expressed in; moreover, the conversion of relational databases to meet changing demands on their ability to store and retrieve information is a continuing problem that requires at least as much complex (and therefore careful) consideration as the translation of a programming language [118].

Syntactic acceptors

By separating syntactic recognition and translation, we not only free the translator from the limitations imposed by context free (and even more restricted) syntactic recognition, we also free the parse, if such is being used for syntactic acceptance,

from the demands placed on it by the translation. Indeed, we free altogether the programs that perform the task of creating syntax graphs from the need to be parsers: there is no reason why translation cannot be one of the functions performed by a program editor, just as text formatting is frequently one of the tasks performed, albeit not always very successfully, by text editors; a program editor like Emily (see section 3.4) bypasses many of the functions of a parser --although it is syntax directed-- because it forces the creation of a syntactically correct program, whereas a parser exists in part to ensure the syntactic correctness of input by issuing error messages and even by attempting automatic correction of errors. Incremental compilers and program development systems are increasingly important aspects of translation, and programming environments encouraging the use of these will benefit from translator development tools which encourage the separation of input and translation phases -- so that one can be completely changed without affecting the other. Parsers of all kinds have a wide range of uses as standalone software tools, as is demonstrated by the many, often surprising applications the parser generator YACC has been put to [66]. YACC is the product of a design philosophy that directs systems of software to be built out of manageably sized "tools."

"Code" generators (graph flatteners)

In section 4.1 the last phase of a traditional translation

was described as an inverse parser. As with parsers, it is convenient to be able to replace one kind, an object module generator, for instance, with another which "prettyprints" the target program, or one which stores an intermediate representation of the translated program's graph structure in a file using a linear graph notation. Recall that, in the discussion on optimization in this model, optimization is not considered one of the functions of a graph flattener (hence the avoidance of the term "code generator"): optimization is one of the aspects of graph transformation. Again, as with syntactic acceptors, separating the function of graph flattening from that of translation (graph transformation) has its benefits: just as syntactic recognition may not be necessary in a translation (the program may have been generated by use of a syntactic editor; the previously parsed program may have been stored in a linear graph notation and need not be parsed again), so a code generation may not be necessary either. In particular, the syntactic graph need not be translated at all, but may be interpreted instead.

These three subsystems are each programming tools in their own right: the graph manipulation package has applications, as indicated, far beyond its use as a programming tool; parser generators like YACC are essentially a form of input processor with a wide range of uses outside of language processing; a syntactic editor (an alternative to parsing) would be applicable also to document editing [117] and database manipulation [118];

a syntax graph flattener could serve at least as a program prettyprinter, when the input graph is not transformed. They can exist, therefore, and be useful even without their being part of a TWS, and they can be constructed, one by one, each being immediately available long before a full TWS has come into existence. Furthermore, once they exist, they can themselves be used in the construction of a TWS.

A translator writing system is itself a translator, and can be implemented as such. Usually, in fact, it is at least three translators in the sense that each has its own description language: one to generate symbol scanners from lexical descriptions expressed in a language specific to scanner generators; one to generate parsers from syntactic descriptions expressed in a language specific to parser generators; and one to generate translators from semantic descriptions. However, the TWS outlined in chapter 4 combined these into a more uniform description. Once the basic translator writing tools exist, these can be used to construct a translator for programs written in the language outlined in chapter 4; the first program written in this language and compiled with this translator should be a proper description of the meta-translator itself.

Even before the TWS exists -- even when only its associated library of tools exists -- a number of applications become possible, and can be built up out of the tools, using some other programming language, most likely a general purpose algorithmic

one, to control the use of the tools. This approach of building systems from tools is deliberately modelled on the "software tools" approach to system construction developed mostly at Bell Labs and advocated by Kernighan and Plauger [69]. This approach depends on getting the basic tools out into the world and refining them iteratively as a wide base of experience is acquired in their use; it appears to be a successful method which has been applied specifically to language development tools in several different settings [1,67].

The primary division of the translation tools into three subparts reflects the traditional view of translation as a single continuous process with several distinct phases (which may be performed either strictly sequentially or by cooperating sequential processes). However, once these three portions are seen as collections of closely related tools, it becomes just as natural to restructure the translation process.

It was already suggested in this chapter that the function of syntactic acceptor normally performed by a parser could be performed by an interactive program editor instead. Such an editor would, if it acted like Emily or the Cornell Synthesizer [123], be used to build syntactically correct program graphs which could then be passed through a graph transformer. More than this, however, once the program graph exists, it can be edited, whenever semantic changes are necessary, by a process of graph transformation. This process would produce new graphs

which can either be printed by a prettyprinting graph flattener, or stored, or translated. The program can be stored as a graph, using a linear graph notation as in the PQCC, or as a kind of object module with relocatable links to be resolved, next time the program is edited, by a kind of loader, or, again, as a prettyprinted program. A structure editor would perform many of the functions of a program development system such as discussed in section 3.4. A library of syntactic transforms for systematic program development [6,30] could be maintained : such a library could be developed in the syntactic transformation language and compiled analogously to the way subroutine libraries are maintained for conventional general purpose programming systems.

At the other end, the transformed graphs representing translated or partially translated programs, need not be "flattened" at all, either by a prettyprinter or into object modules. One aspect of programming language "translation" not dealt with in chapter 3, but certainly traditional in the sense that it has been known as a technique, and applied for as long as compilers have, is program interpretation. Furthermore, the idea of basing an interpreter on a syntactic representation of the program already has a considerable history behond it. The Vienna Definition Language (VDL), an interpretive automaton, uses an abstract syntactic representation of the program it is interpreting -- in essence a semantically augmented tree with both branches and nodes labelled (branches are not implicitly

ordered with respect to a node) -- and performs its interpretation by successively transforming the portions of the tree used to represent the data and the program until some final state results -- in the simplest case a single node representing the value computed by the program [139].

VDL is not a language for the definition of usable interpreters; that is not its function, which is a purely definitional one. However, Turner [127], whose work on partial compilation of expressions in an applicative language was discussed in chapter 4, has based the design of an interpreter for the resulting variable-free expressions on the use of possibly cyclic list structures to represent the expressions, and a transformational interpreter to evaluate them. One of the benefits of this approach is that it permits a relatively efficient form of normal order reduction, which is used because it can guarantee termination provided the evaluation <u>can</u> terminate. The substitution of unevaluated arguments for function parameters in evaluating a function call, on which normal order reduction depends and which normally causes partially evaluated programs to grow out of proportion with their original size, is no more costly in this model than the more usual applicative order regime, by virtue of the DAG-forming subgraph replacement scheme it uses to implement normal order evaluation (analogous to the discussion of common subexpression optimization in section 3.2.1). Several relatively recent proposals have advocated a similar "lazy

evaluation" approach, particularly for performing computations that are essentially non-terminating, but achieve interesting partial results along the way (the sieve of Eratosthenes, which generates all primes, is an ancient example of such an algorithm): see Turner's paper for a discussion and further references.

Rosen [105] has also suggested graph transformation as a basis for the formal definition of interpreters.

The preceding design was presented in part as an indication of the directions of research deriving directly from the material developed in this dissertation. It is also presented as a workable outline for others wishing to pursue these ideas: in his summary on the practical value of software tools in the Unix setting, Johnson [67] remarks that whereas few successful tools were "designed" in the traditional sense -- by being turned out as finished products by a skilled team of toolwrights --, the most common process was instead one of iterative refinement of the primitive tools by a community of skilled tool users. If the concept of syntactic transformation has the kind of merit claimed for it here as a basis for translator definition, it will come to be adapted to many different forms of translation, under many different circumstances, and necessarily by many different people, often quite independently of each other. The result will be a similar iterative refinement of the basic concepts to suit more nearly the direct

concerns of translator writers. The formalized definitions at the beginning of chapter 3 and in section 4.2.2 are so constructed that they can be used directly in deriving a practical implementation, as a further incentive to the use of these tools.

Fortunately, this process already appears to be taking place. Very recently there have been reports on several parallel efforts in the area of code generation that have depended on abstract syntax graphs (primarily trees) to represent the programs, and pattern matching and graph transformations as the means to effect optimizations on the program [20,51,78]. These also have, in common with the material presented here, the assumption that code generation is to be considered separately from input acceptance. The three research efforts cited here run the gamut from being strongly deterministic [20] to highly heuristic [78] in their approach to selecting appropriate optimizations. The translator writing system presented in chapter 4 compares with these on the deterministic end of the spectrum, in that the programmer, not a set of heuristics, determines the transformations that will take place; the overall philosophy of translation presented in this dissertation is highly compatible, however, with the approach taken at the other end of the spectrum (the PQCC project), in its use of successive transductions of a graph representation to construct a translator. In fact, the primary aim of this dissertation was to establish the highly uniform application of

this (syntactic) transformation technique to all aspects of translation. In that sense, it represents the design of a toolkit for all such transformational systems, now and in the forseeable future.

Appendix: A SASL compiler

In section 4.4.2, we discussed a transformational "compiler" for the applicative language SASL, based on a recent algorithm for bracket-abstraction developed by Turner in which the number of new symbols introduced is small [126, 127]. This appendix presents an implementation of Turner's algorithm in terms of syntactic transformations: the language of Turner's papers in fact implies that syntactic transformation would be a natural way of thinking about the algorithm.

This appendix proceeds by first defining a subset of SASL over which translations will be defined, and then giving two versions of the bracket abstraction algorithm due to Turner: (1) The algorithm in the old formulation of graph transformation, the one for which an interpreter was actually implemented, and thus the only one of the two formulations which has been empirically verified; and (2) The algorithm in the new formulation of graph transformation developed in chapter 4, which is unimplemented and therefore unverified. In both cases the language of the formulation is defined informally before the algorithm is presented. The reader will probably find the second, more abstract formulation of the algorithm easier to read, and indeed the first, more primitive version is here only for completeness, in case there is something subtly wrong with the second, unverified version.

A: A SASL subset

SASL is a "syntactically sugared" lambda calculus, or applicative language. Where in lambda calculus one would write

```
let suc = lambda (x) (plus x 1)
```

in SASL the same expression is written

```
def suc x = plus x 1
```

For the sake of syntactic simplicity, we will use no infix notation, which is permitted in SASL, but will represent all operators, like "plus" in the above examples, in prefix form. Moreover, all functions will be expected to take only a single argument: any multi-argument function will be considered to have been reduced, by a process known as Currying, to a single-argument function. Both these operations can, it is commonly known, be applied without loss of generality.

A BNF definition of this basic subset is then as follows:

```
expression ::= definition | factor

definition ::= "def" symbol "=" factor
             | "def" symbol symbol "=" factor

factor ::= primary | factor primary

primary ::= symbol | "(" factor ")"
```

A program consists of a sequence of expressions; symbols are either names, representing functions or variables, or constants. SASL programs would normally be evaluated in an incremental, interpretive manner.

The sublanguage generated by the symbol "factor" is a pure applicative subset. Turner's algorithm, like other bracket abstraction algorithms, is aimed at reducing all expressions to this form, and eliminating in the process all variables: in the successor function "suc", for example, "x" is a variable. This is accomplished in two stages. First, functions with a bound variable, those generated by application of the BNF rule

definition ::= "def" symbol symbol "=" factor

are reduced to functions without a bound variable by a process Turner calls abstraction,[1] which is defined by the transformations

```
def f x = E   =>  def f = [x] E
[x] (E1 E2)   =>  S ([x] E1) ([x] E2)
      [x] x   =>  I
      [x] y   =>  K y
```

where S, K and I were defined in section 4.2.2. This process is represented in the two implementations by the procedure "Abstract". The second step is an optimizing one, in which excessively complex applications of S, K and I are reduced, in most cases, to simpler ones. The optimizing transformations are

```
S (K E1) (K E2)  =>  K (E1 E2)
     S (K E1) I  =>  E1
    S (K E1) E2  =>  B E1 E2 {if no earlier rule applies}
    S E1 (K E2)  =>  C E1 E2 {if no earlier rule applies}
```

where B and C were defined in section 4.2.2. This process is implemented below by the procedure "Optimize".

---

[1] Not to be confused with the use of this term in the work of Wulf and others, which is its use in the main body of this dissertation.

B: Old formulation

The graph transformation language used in the experimental implementation was discussed, but not defined in section 4.2. The definition which follows is somewhat informal, and only defines enough of the language to make reading of the algorithm possible. This informal description should be less cumbersome than a formal one and, for the purposes of this appendix, sufficiently precise. The following are the legal language forms.

var x

The variable x is declared. The variable has as its scope the smallest nest in which it is contained; where language forms define a nest, this will be noted in their definition, below. The value of this expression is the name of x; thus "var x" can be used wherever "x" can be used, and will have the same value.

x := E

The expression E is evaluated and its value assigned to the variable x. The value of this expression is the value of E.

procedure (A1,...,An) E end

The value of this expression is a procedure with n formal parameters, A1 through An, and body E. A procedure is thus a manipulable entity, like any constant, and may be assigned to

a variable, which then becomes its "name". A procedure defines a nest, in which the formal parameters are automatically declared.

F (A1,...,An)

The procedure assigned to the variable F is evaluated with actual parameters A1 through An. The value of this expression is the value resulting from the evaluation of the procedure with these actual parameters.

if E1 then E2 [elsif E]* [else E] fi

Conditional expression in the (unrevised) Algol 68 style. A conditional expression defines a nest.

P1 :: P2

Boolean expression performing the pattern match described in section 4.2.2.

? or ? x

In a graph pattern match only, the symbol ? matches any subgraph. If it is attached to a variable, as in "? x", a successful pattern match will cause an assignment of the subgraph matched by "?" to the variable.

"string" < P1,...,Pn >

The value of this expression is a pattern whose root node consists of the value "string" and whose children are formed

by P1 through Pn, which may be patterns, simple (non-pattern) values, or variables.

E1 => E2

The graph resulting from evaluating expression E1 is transformed, by the method defined in chapter 3, into the graph resulting from evaluating expression E2.

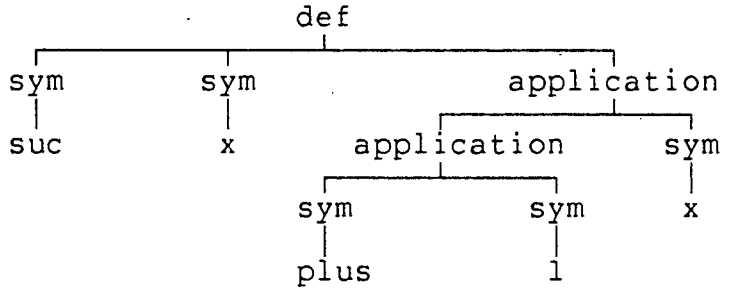
For this version of the graph translation language, we must also supply a parser/treebuilder and a "flattener", which constructs a readable output from the transformed graph. Since the input and output of the SASL "compiler" are graphs in the same language, the flattener performs the inverse function of the parser/treebuilder. We need only define the relationship between the grammar and the resulting graph, therefore. In the following, each grammatical form in the language is shown with its resulting graph as "form => graph". The graphs are given in the above-defined graph expression sublanguage.

```
expression ::= definition => definition
expression ::= factor => factor

definition ::= "def" symbol "=" factor
                 => "def" <symbol,"void",factor>
definition ::= "def" symbol1 symbol2 "=" factor
                 => "def" <symbol1,symbol2,factor>

factor ::= primary => primary
factor ::= factor primary
             => "application" <factor,primary>

primary ::= symbol => "sym" <symbol>
primary ::= "(" factor ")" => factor
```

The result is a parse tree for the "abstract" syntax, leaving out many of the disambiguating but otherwise redundant symbols and one-on-one productions. For example, the expression

```
def suc x = plus x 1
```

results in the parse tree



(Notice that "plus" is here Curried. What is applied to x is a function resulting from the application of "plus" to 1.)

We are now ready to examine the program. The top procedure merely orders the two stages:

```
var Improve := procedure (E)
   Abstract(E);
   Optimize(E)
   end ¢ Improve ¢
```

```
var Abstract := procedure (E)

    ¢ { def f v = body }  =>  { def f = [v] body } ¢

    if E :: "def"<?var f,"sym"<?var v>,?var body> then
      Abstract(body);
      E => "def"<f,"void","abstraction"<v,body>>

    ¢ { def f = body } ¢

    elsif E :: "def"<?var f, "void", ?var body> then
      Abstract(body)

    ¢ { [v] (L R) }  =>  { S ([v] L) ([v] R) } ¢

    elsif E :: "abstraction"<?var v,
               "application"<?var L,?var R>> then
      Abstract(L); Abstract(R);
      E => "application"<"application"<"sym"<"S">,
                                       "abstraction"<v,L>>,
                         "abstraction"<v,R>>

    ¢ { [v] v }  =>  { I } ¢

    elsif E :: "abstraction"<?var v,?var vl> and v :: vl then
      E => "sym"<"I">

    ¢ { [v] i }  =>  { K i } ¢

    elsif E :: "abstraction"<?var v, ?var i> then
      E => "application"<"sym"<"K">, i>

    ¢ { L R } ¢

    elsif E :: "application"<?var L, ?var R> then
      Abstract(L); Abstract(R)

      fi
    end ¢ Abstract ¢
```

```
var Optimize := procedure(E)

  ¢ { S (K E1) (K E2) } => { K (E1 E2) } ¢

  if E :: "application"<
            "application"<"sym"<"S">,
              "application"<"sym"<"K">,
                           ?var E1>>,
              "application"<"sym"<"K">,
                           ?var E2>> then
    Optimize(E1);
    Optimize(E2);
    E => "application"<"sym"<"K">,
                      "application"<E1,E2>>

  ¢ { S (K E1) I } => { E1 } ¢

  elsif E :: "application"<
              "application"<"sym"<"S">,
                "application"<"sym"<"K">,
                             ?var E1>>,
              "sym"<"I">> then
    Optimize(E1);
    E => E1

  ¢ { S (K E1) E2 } => { B E1 E2 } ¢

  elsif E :: "application"<
                     "application"<"sym"<"S">,
                       "application"<"sym"<"K">,
                                    ?var E1>>,
                       ?var E2> then
    Optimize(E1);
    Optimize(E2);
    E => "application"<"application"<"sym"<"B">,E1>,
                      E2>

  ¢ { S E1 (K E2) } => { C E1 E2 } ¢

  elsif E :: "application"<
              "application"<"sym"<"S">,?var E1>,
              "application"<"sym"<"K">,?var E2>> then
    Optimize(E1);
    Optimize(E2);
    E => "application"<"application"<"sym"<"C">,E1>,
                      E2>

  elsif E :: "application"<?var E1,?var E2> then
    Optimize(E1);
    Optimize(E2);
    Optimize(E)
    fi
  end ¢ Optimize ¢
```

The reader may wish to to be assured that this is correct by tracing the short sequence:.

```
E = { def suc x = plus 1 x }
E' = Abstract(E)
   = { def suc = S (S (K plus) (K 1)) I }
E" = Optimize(E')
   = { def suc = plus 1 }
```

and, if he has the courage, the much longer sequence:

```
E = { def fac n = cond (eq n 0)
                       1
                       (times n (fac (minus n 1))) }
E' = Abstract(E)
   = { def fac = S (S (S (K cond) (S (S (K eq)
                   I) (K 0))) (K 1)) (S (S (K
                   times) I) (S (K fac) (S (S
                   (K minus) I) (K 1)))) }[2]
E" = Optimize(E')
   = { def fac = S (C (B cond (C eq 0)) 1)
                   (S times (B fac (C minus 1))) }[3]
```

---

[2] This is slightly different from Turner's version [127], p 34), in which

```
(S (S (K eq) I) (K 0)))
```

is replaced by the entirely equivalent

```
(S (S (K eq) (K 0)) I))
```

a result that cannot be obtained by strict application of Turner's rules. The equivalence of these two expressions, which test for equality to zero, can be verified by applying them both to a variable, say x, and reducing both expressions so formed to "eq 0 x".

[3] There is a typographical error in Software -- Practice and Experience ([127], p 35), which leaves out the second "C".

C: New formulation

The "new" formulation of syntactic transformation was already defined informally in some detail during the discussions of chapter 4. However, for the sake of uniformity with the preceding section on the old formulation, and especially for the reader's convenience, the following is a short synopsis of enough of the language to make reading the SASL "compiler" possible.

Once more it is necessary to imagine that a parser/treebuilder and a "flattener" exist, or can be constructed within a translator writing system based on syntactic transformation. However, this time each translation program includes a syntactic definition of the source and target languages which defines the relationship between the source and target languages and the syntactic graphs being manipulated.[4] This definition in turn makes it much easier to read and comprehend the translation program -- as a comparison of the program in this section with that in section B should demonstrate.

We begin by defining the language forms which differ from the old formulation:

---

[4] In the case of our SASL example, the source and target languages are both SASL, so only one syntactic definition is needed.

language L is [syntactic definition] end

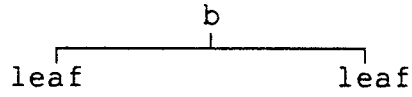
The syntactic definition consists of an augmented grammar in which nonterminal symbols are delimited by angle brackets and terminal symbols by quotes, attributes may be attached to any nonterminal symbol with the operators "↑" (synthesized attribute) and "↓" (inherited attribute),[5] syntactic rules are separated by periods ".", left and right hand sides of rules are separated by the BNF production symbol "::=", alternative right hand sides of rules are separated by the BNF alternative separator "|", and "semantic actions", consisting of assignments of subtree expressions (as used in section B) to attribute symbols are enclosed in square brackets "[" and "]". The rule

```
<bin>↑b ::= <bin>↑b1 <bin>↑b2
              [b := "b"<b1,b2>]
          | "leaf" [b := "leaf"]
```

for example, ambiguously defines the set of all binary trees with leaves labelled "leaf" and nodes labeled "b", such that the expression

{leaf leaf}

corresponds to the graph (a binary tree)



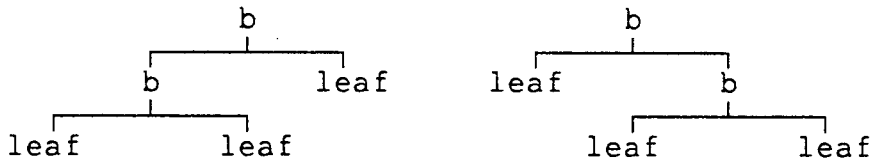
and the expression

{leaf leaf leaf}

may correspond to either of the graphs (binary trees)[6]

---

[5] The definition of SASL is context free and requires no inherited attributes.




```
transform variable matching
   pattern  E    ;...;  =>E    ;...;  E
            1  l1            li          ln
   ¤...¤

   pattern  E    ;...;  =>E    ;...;  E
            k  k1            kj          km
   end
```

This defines a transducer, a procedure which may be applied in parallel to any number of syntactic graph expressions, as in

Abstract (E1, E2)

in the following program: "Abstract" is a variable possessing a transducer, and is applied to E1 and E2 (possibly in parallel). Any graph "variable", the formal argument of the transducer, is matched against the k graph patterns one at a time. If any of them succeeds, the subsequent expressions are evaluated. Any expression consisting of a transformation symbol followed by an expression yielding a graph causes a transformation in the pattern graph.

---

[6] Ambiguity is to be frowned on in syntactic definitions. In the preceding, neither the reader of the phrase {leaf leaf leaf} nor the compiler can be entirely sure of the underlying structure. If the compiler determines on one in favor of the other, any pattern match in which this tree is used as a pattern is liable to fail. The syntax for SASL, given below, is not ambiguous.

Now we are ready to look at the SASL compiler in the revised form:

```
var SASL_compiler := procedure (Parsed_SASL_program)
  language SASL is
    <expression> ↑tree ::= <definition> ↑tree
                         | <factor> ↑tree .
    <definition> ↑tree ::= "def" <symbol> ↑sym ↑fact
                             [tree := "def"<sym,"void",fact>] .
                         | "def" <symbol> ↑sym1 <symbol> ↑sym2
                             "=" <factor> ↑fact
                             [tree := "def"<sym1,sym2,fact>] .
    <factor> ↑tree ::= <primary> ↑tree
                     | <factor> ↑fact <primary> ↑prim
                         [tree := "application"<fact,prim>
                     | <abstraction> ↑tree .
    <primary> ↑tree ::= <symbol> ↑tree
                      | "(" <factor> ↑tree ")" .
    <abstraction> ↑tree ::= "[" <symbol> ↑sym "]"
                              <factor> ↑fact
                            [tree := "abstraction"<sym,fact>] .
    <symbol> ↑tree ::= [defined lexically]
    end;


  var Improve := procedure (E)
    Abstract (E);
    Optimize (E)
    end ¢ Improve ¢;
```

```
var Abstract := transform E matching

  SASL(<symbol> f,v; <factor> body):
    { def f v = body }
      Abstract(body);
    => SASL(<symbol> f,v; <factor> body):
        { def f = [v] body }

¤ SASL(<symbol> f; <factor> body):
    { def f body }
      Abstract(body)

¤ SASL(<symbol> v; <factor> L; <primary> R):
    { [v] (L R) }
      Abstract (L, R);
    => SASL(<symbol> v; <factor> L; <primary> R):
       { S ([v] L) ([v] R) }

¤ SASL(<symbol> v): {[v] v}
    => SASL: { I }

¤ SASL(<symbol> v,i): {[v] i}
    => SASL(<symbol> i): {k i}

¤ SASL(<factor> L; <primary> R): {L R}
      Abstract (L, R)

end ¢ Abstract ¢;
```

```
  var Optimize := transform E matching

    SASL(<primary> E1,E2): {S (K E1) (K E2)}
        Optimize (E1, E2);
      => SASL(<primary> E1,E2): {K (E1 E2)}

  ¤ SASL(<primary> E1,E2): {S (K E1) E2}
        Optimize (E1, E2);
      => SASL(<primary> E1,E2): {B E1 E2}

  ¤ SASL(<primary> E1,E2): {S E1 (K E2)}
        Optimize (E1, E2);
      => SASL(<primary> E1,E2): {C E1 E2}

  ¤ SASL(<factor> E1; <primary> E2): {E1 E2}
      => Optimize (E1, E2)

  end ¢ Optimize ¢;


  Improve (Parsed_SASL_Program)

  end ¢ SASL_Compiler ¢
```

The compiler is called with a parsed SASL program or expression, which it transforms into a "compiled" or variable-free SASL expression. For example,

```
var E := Parser();
SASL_Compiler (E);
Flattener (E)
```

References

[1] H. Abramson, T. Rushworth, and T. Venema
TOSI: A tree oriented string interpreter for the design and implementation of semantics
Software -- Practice and Experience 7 (1977), pp 663-670

[2] A.V. Aho and J.D. Ullman
The theory of parsing, translation, and compiling; Volume I: Parsing
Prentice-Hall, 1972

[3] A.V. Aho and J.D. Ullman
Principles of compiler design
Addison-Wesley, 1977

[4] E.R. Anderson, F.C. Belz, and E.K. Blum
SEMANOL(73): A metalanguage for programming the semantics of programming languages
Acta Informatica 6 (1976), pp 109-131

[5] W.F. Appelbe
A semantic representation for translation of high-level algorithmic languages
PhD Thesis, Department of Computer Science, The University of British Columbia, 1978

[6] J.J. Arsac
Syntactic source to source transforms and program manipulation
Communications of the ACM 22 (1979), pp 43-54

[7] P.R. Bagley
Principles and problems of a universal computer-oriented language
Computer Journal 4 (1962), pp 305-312

[8] J.L. Baker
Notes for a course in automata theory
Department of Computer Science, The University of British Columbia, 1976

[9] V.R. Basili
The SIMPL family of programming languages and compilers
Technical report TR-305, University of Maryland Comp. Sci. Center, 1974

[10] F.L. Bauer and J. Eickel (eds.)
Compiler construction: An advanced course
Springer-Verlag, 1976

[11] F.L. Bauer
Historical remarks on compiler construction

in [10], pp 603-621

[12] F.L. Bauer, M. Broy, R. Gnatz, W. Hesse, B. Krieg-Brueckner, H. Partsch, P. Pepper, and H. Woessner
Towards a wide spectrum language to support program specification and program development
SIGPLAN Notices 13 12 (December 1978), pp 15-24

[13] R.E. Berry
Experience with the PASCAL P-compiler
Software -- Practice and Experience 8 (1978), pp 617-627

[14] D.G. Bobrow and T. Winograd
An overview of KRL, a knowledge representation language
Cognitive Science 1 (1977), pp 3-46

[15] G.V. Bochmann
Semantic evaluation from left to right
Communications of the ACM 19 (1976), pp 55-62

[16] G.V. Bochmann
Compiler writing system for attribute grammars
The Computer Journal 21 (1978), pp 144-148

[17] H. Boom
The organization of the object code generator in Algol 68 H
Technical report IW 33/75, Stichting Mathematisch Centrum, Amsterdam, 1975

[18] P.J. Brown
The ML/I macro processor
Communications of the ACM 10 (1967), pp 618-623

[19] R.M. Burstall and J. Darlington
A transformation system for developing recursive programs
Journal of the ACM 24 (1977), pp 44-67

[20] J.L. Carter
A case study of a new code generation technique for compilers
Communications of the ACM 20 (1977), pp 914-920

[21] R.G. Cattell
A survey and critique of some models of code generation
Technical report, Carnegie-Mellon University, 1977

[22] R.G. Cattell, J.M. Newcomer, and B.W. Leverett
Code generation in a machine-independent compiler
SIGPLAN Notices 14 8 (August 1979), pp 65-75

[23] L.M. Chirica and D.F. Martin
An approach to compiler correctness
SIGPLAN Notices 10 6 (June 1975), pp 96-103

[24] N. Chomsky
On certain formal properties of grammars
Information and Control 2 (1959), pp 137-167

[25] A.J. Cole
Macro processors
Cambridge University Press, 1976

[26] S.S. Coleman, P.C. Poole, and W.M. Waite
The Mobile Programming System, JANUS
Software -- Practice and Experience 4 (1974), pp 5-23

[27] M.E. Conway
Design of a separable transition diagram compiler
Communications of the ACM 6 (1963), pp 396-408

[28] J.R. Cordy, R.C. Holt, and D.B. Wortman
Semantic charts: A diagrammatic approach to semantic processing
SIGPLAN Notices 14 8 (August 1979), pp 39-49

[29] D. Crowe
Generating parsers for affix grammars
Communications of the ACM 15 (1972), pp 728-732

[30] J. Darlington and R.M. Burstall
A system which automatically improves programs
Acta Informatica 6 (1976), pp 41-60

[31] R.A. DeMillo, R.J. Lipton, and A.J. Perlis
Social processes and proofs of theorems and programs
Communications of the ACM 22 (1979), pp 271-280,614

[32] F.L. DeRemer and H. Kron
Programming-in-the-large versus programming-in-the-small
SIGPLAN Notices 10 6 (June 1975), pp 114-121

[33] F.L. DeRemer
Transformational grammars
in [10], pp 121-145

[34] E.W. Dijkstra
Guarded commands, nondeterminacy and formal derivations of programs
Communications of the ACM 18 (1975), pp 453-457

[35] E.W. Dijkstra
A discipline of programming
Prentice-Hall, 1976

[36] J. Earley
An efficient context-free parsing algorithm
Communications of the ACM 13 (1970), pp 94-102

[37] E.F. Elsworth
Compilation via an intermediate language
The Computer Journal 22 (1978), pp 226-233

[38] J. Feder
Plex languages
Information Sciences 3 (1971), pp 225-241

[39] J. Feldman and D. Gries
Translator writing systems: An exploration of concepts and principles
Communications of the ACM 11 (1968), pp 77-113

[40] R.W. Floyd
A descriptive language for symbol manipulation
Journal of the ACM 8 (1961), pp 579-584

[41] R.W. Floyd
On the nonexistence of a phrase structure grammar for Algol 60
Communications of the ACM 5 (1962), pp 483-484

[42] R.W. Floyd
Assigning meaning to programs
Proceedings of the Amer. Math. Soc., Symposia in Applied Math., 19 (1967), pp 19-31

[43] R.W. Floyd
Toward interactive design of correct programs
Technical report no. CS-235, or AI Memo AIM-150, Computer Science Department, Stanford University, 1971

[44] R.A. Fraley
Unlanguage grammars and their uses
Technical report 77-6, Department of Computer Science, The University of British Columbia, 1977

[45] M.R. Garey and D.S. Johnson
Computers and intractibility: A guide to the theory of NP-completeness
W.H. Freeman and Company, 1979

[46] J.V. Garwick
GPL: A truly general purpose language
Communications of the ACM 11 (1968), pp 634-638

[47] F. Geiselbrechtinger, W. Hesse, B. Krieg, and H. Scheidig
Language layers, portability and program structure
in [130], pp 79-99

[48] C.M. Geschke, J.H. Morris Jr, and E.H. Satterthwaite
Early experiences with Mesa
Communications of the ACM 20 (1977), pp 540-553

[49] S.L. Graham and M.A. Harrison
Parsing of general context-free languages
Advances in Computers 14 (1976), pp 77-185; Earley's algorithm pp 122-139

[50] S.L. Graham, M.A. Harrison, and W.L. Ruzzo
An improved context-free recognizer
ACM Transactions on Programming Languages and Systems 2 (1980), pp 415-462

[51] S.L. Graham
Table-driven code generation
IEEE Computer, 13 8 (August 1980), pp 25-34

[52] S.A. Greibach
Theory of program structures: Schemes, semantics, verification (Lecture notes in computer science, 36)
Springer-Verlag, 1975

[53] D. Gries
Compiler construction for digital computers
John Wiley & Sons, 1971

[54] D. Gries
Error recovery and correction: An introduction to the literature
in [10], pp 627-638

[55] W.J. Hansen
Creation of hierarchic text with a computer display
PhD Thesis, Stanford University, 1971

[56] D.R. Hanson
RATSNO -- An experiment in software adaptability
Software -- Practice and Experience 7 (1977), pp 625-630

[57] L.E. Heindel and J.T. Roberto
LANG-PAK -- An interactive language design system
American Elsevier, 1975

[58] P.G. Hibbard and S.A. Schuman (eds.)
Constructing quality software (Proceedings of the IFIP Working Conference on Constructing Quality Software)
North-Holland Publishing Co., 1978

[59] C.A.R. Hoare
An axiomatic basis for computer programming
Communications of the ACM 12 (1969), pp 576-580,583

[60] C.A.R. Hoare and N. Wirth
An axiomatic definition of the programming language Pascal
Acta Informatica 2 (1973), pp 335-355

[61] J.J. Horning
What the compiler should tell the user
in [10], pp 526-548

[62] G. Huet and B. Lang
Proving and applying program transformations expressed with second-order patterns
Acta Informatica 11 (1978), pp 31-55

[63] E.T. Irons
A syntax directed compiler for Algol 60
Communications of the ACM 4 (1961), pp 51-55

[64] K. Jensen and N. Wirth
PASCAL: User manual and report
Springer-Verlag, 1974

[65] S.C. Johnson
YACC: Yet another compiler-compiler
Computing Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1978

[66] S.C. Johnson and M.E. Lesk
Unix time sharing system: Language development tools
Bell System Technical Journal 57 6 (July-August 1978), pp 2155-2175

[67] S.C. Johnson
Language development tools on the Unix system
IEEE Computer, 13 8 (August 1980), pp 16-20

[68] B.W. Kernighan
RATFOR -- A preprocessor for a rational FORTRAN
Software -- Practice and Experience 5 (1975), pp 395-406

[69] B.W. Kernighan and P.J. Plauger
Software tools
Addison-Wesley, 1976

[70] D.E. Knuth
Backus Normal Form vs. Backus Naur Form
Communications of the ACM 7 (1964), pp 735-736

[71] D.E. Knuth
Semantics of context-free languages
Mathematical Systems Theory 2 (1968), pp 127-145; Math. Sys. Th. 5 (1971), p 95

[72] D.E. Knuth
Structured programming with goto statements
Computing Surveys 6 (1974), pp 261-301

[73] C.H.A. Koster
Affix grammars
in [95], pp 95-106

[74] C.H.A. Koster
Using the CDL compiler-compiler
in [10], pp 366-426

[75] B.M. Leavenworth
Syntax macros and extended translation
Communications of the ACM 9 (1966), pp 790-793

[76] H.F. Ledgard
Production systems: or, Can we do better than BNF?
Communications of the ACM 17 (1974), pp 94-102

[77] H.F. Ledgard
Production systems: A notation for defining syntax and semantics
IEEE Transactions on Software Engineering 3 (1977), pp 105-124

[78] B.W. Leverett, R.G.G. Cattell, S.O. Hobbs, J.M. Newcomer, A.H. Reiner, B.R. Schatz, and W.A. Wulf
An overview of the Production Quality Compiler-compiler project
Technical report CMU-CS-79-105, Department of Computer Science, Carnegie-Mellon University, 1979

[79] B. Liskov and S. Zilles
Specification techniques for data abstractions
SIGPLAN Notices 10 6 (June 1975), pp 72-87

[80] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert
Abstraction mechanisms in CLU
Communications of the ACM 20 (1977), pp 564-576

[81] E.S. Lowry and C.W. Medlock
Object code optimization
Communications of the ACM 12 (1969), pp 13-22

[82] M.D. McIlroy
Macro instruction extensions of compiler languages
Communications of the ACM 3 (1960), pp 214-220

[83] W.M. McKeeman
Peephole optimization
Communications of the ACM 8 (1965), pp 443-445

[84] W.M. McKeeman, J.J. Horning, and D.B. Wortman
A compiler generator
Prentice-Hall, 1970

[85] W.M. McKeeman
Compiler construction
in [10], pp 1-36

[86] Z. Manna and R.J. Waldinger
Toward automatic program synthesis
Communications of the ACM 14 (1971), pp 151-165

[87] M. Marcotty, H.F. Ledgard, and G.V. Bochman
A sampler of formal definitions
Computing Surveys 8 (1976), pp 191-276

[88] J.R. Metzner
A graded bibliography on macro systems and extensible languages
SIGPLAN Notices 14 1 (January 1979), pp 57-68

[89] C.N. Mooers
TRAC, A procedure-describing language for the reactive typewriter
Communications of the ACM 9 (1966), pp 215-219

[90] J.B. Morris
Data abstraction: A static implementation strategy
SIGPLAN Notices 14 8 (August 1979), pp 1-7

[91] P.D. Mosses
Mathematical semantics and compiler generation
PhD Thesis, The University of Oxford, 1975

[92] P. Naur
The European side of the last phase of the development of Algol 60
SIGPLAN Notices 13 8 (August 1978), pp 15-44

[93] M.C. Newey, P.C. Poole, and W.M. Waite
Abstract machine modelling to produce portable software -- A review and evaluation
Software -- Practice and Experience 2 (1972), pp 107-136

[94] K.V. Nori, U. Amann, K. Jensen, and H.H. Naegeli
The PASCAL "P" compiler: Inplementation notes
Technical report, ETH, Zuerich (undated)

[95] J.E.L. Peck (ed.)
Algol 68 implementation
North Holland Publishing Company, 1971

[96] A.J. Perlis and K. Samelson
Preliminary report -- International Algorithmic Language
Communications of the ACM 1 12 (December 1958), pp 8-22

[97] A.J. Perlis
The American side of the development of Algol
SIGPLAN Notices 13 8 (August 1978), pp 3-14

[98] J.L. Pfaltz and A. Rosenfeld
Web grammars
Proc. Int. Joint Conf. on Artificial Intelligence,
Bedford, MA, 1969, pp 609-619

[99] R.H. Pierce and J. Rowell
A transformation-directed compiling system
The Computer Journal, May 1977, pp 109-115

[100] P.C. Poole
Towards improved reliability and efficiency through
hybrids
in [58], pp 63-73

[101] J.C. Reynolds
COGENT programming manual
Research and development report ANL-7022, Argonne National
Laboratory, 1965

[102] M. Richards
The portability of the BCPL compiler
Software -- Practice and Experience 1 (1971), pp 135-146

[103] M. Richards
Bootstrapping the BCPL compiler using INTCODE
in [130], pp 265-270

[104] B.K. Rosen
Tree-manipulating systems and Church-Rosser theorems
Journal of the ACM 20 (1973), pp 160-187

[105] B.K. Rosen
Deriving graphs from graphs by applying a production
Acta Informatica 4 (1975), pp 337-357

[106] S. Rosen (ed.)
Programming systems and languages
McGraw-Hill, 1967

[107] S. Rosen
The Algol programming language
in [106], pp 48-78

[108] M.A. Sabin
Portability -- Some experiences with FORTRAN

Software -- Practice and Experience 6 (1976), pp 393-396

[109] J.E. Sammet
The early history of COBOL
SIGPLAN Notices 13 8 (August 1978), pp 121-161

[110] E. Sandewall
Programming in an interactive environment: The LISP experience
Computing Surveys 10 (1978), pp 35-71

[111] M. Sassa
A pattern matching macro processor
Software -- Practice and Experience 9 (1979), pp 439-456

[112] B.R. Schatz, B.W. Leverett, J.M. Newcomer, A.H. Reiner, and W.A. Wulf
TCOL(ADA): An intermediate representation for the DOD Standard Programming Language
Technical report CMU-CS-79-112, Department of Computer Science, Carnegie-Mellon University, 1979

[113] D.V. Schorre
META-II: A syntax-oriented compiler writing language
Proceedings of the ACM 19th National Conference, 1964, section D1.3

[114] B. Schwanke
Survey of scope issues in programming languages
Technical report CMU-CS-78-131, Department of Computer Science, Carnegie-Mellon University, 1978

[115] L.G. Shapiro and R.J. Baron
ESP^3: A language for pattern description and a system for pattern recognition
IEEE Transactions on Software Engineering 3 (1977), pp 169-183

[116] A.C. Shaw
Parsing of graph representable pictures
Journal of the ACM 17 (1970), pp 453-481

[117] A.C. Shaw
A model for document preparation systems (draft)
Department of Computer Science, University of Washington, 1978

[118] N.C. Shu, B.C. Housel, and V.Y. Lum
CONVERT: A high level translation definition language for data conversion
Communications of the ACM 18 (1975), pp 557-567

[119] J. Sklansky, M. Finkelstein, and E.C. Russell
A formalism for program translation
Journal of the ACM 15 (1968), pp 165-175

[120] C. Strachey
A general purpose macrogenerator
The Computer Journal 8 (1965), pp 225-241

[121] Y. Sugito, Y. Mano, and K. Torii
On a two-dimensional graph manipulation language GML
Systems•Computers•Controls 7 (1976), pp 1-9

[122] A.S. Tanenbaum
A general-purpose macro processor as a poor man's compiler-compiler
IEEE Transactions on Software Engineering 2 (1976), pp 121-125

[123] T. Teitelbaum and T. Reps
The Cornell program synthesizer: A syntax directed programming environment
Technical report, Cornell University, May 1980

[124] R.D. Tennent
The denotational semantics of programming languages
Communications of the ACM 19 (1976), pp 437-453

[125] V.F. Turchin
A supercompiler system based on the language REFAL
SIGPLAN Notices 14 2 (February 1979), pp 46-54

[126] D.A. Turner
Another algorithm for bracket abstraction
The Journal of Symbolic Logic 44 (1978),pp 67-70

[127] D.A. Turner
A new implementation technique for applicative languages
Software -- Practice and Experience 9 (1979), pp 31-49

[128] P. van den Bosch
The design and implementation of a document processor
M.Sc Thesis, Department of Computer Science, The University of British Columbia, 1974

[129] P. van den Bosch
On the joys of axiomatic semantics
Term paper for a course on topics in programming languages, Department of Computer Science, The University of British Columbia, 1977

[130] W.L. van der Poel and L.A. Maarssen (eds.)
Machine oriented higher level languages
North-Holland, 1974

[131] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, and C.H.A. Koster
Report on the algorithmic language Algol 68
Numerische Mathematik 14 (1969), pp 79-218; W-grammars defined in section 1.1, pp 88-93

[132] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.G.L.T. Meertens, and R.G. Fisker
Revised report on the algorithmic language Algol 68
Springer-Verlag, 1976

[133] T. Venema
TRUST user's guide
Technical manual, Department of Computer Science, The University of British Columbia, 1976

[134] W.M. Waite
A language-independent macro processor
Communications of the ACM 10 (1976), pp 433-440

[135] W.M. Waite
The Mobile Programming System: STAGE2
Communications of the ACM 13 (1970), pp 415-421

[136] W.M. Waite
Optimization
in [10], pp 549-602

[137] A. Wang
A case study in program transformation
BIT 16 (1976), pp 322-331

[138] B. Wegbreit
Goal-directed program transformation
IEEE Transactions on Software Engineering 2 (1976), pp 69-80

[139] P. Wegner
The Vienna Definition Language
Computing Surveys 4 (1972), pp 5-63

[140] G. Winiger
A note on one-pass CASE statement compilation
SIGPLAN Notices 11 1 (January 1976), pp 32-36

[141] N. Wirth
Program development by stepwise refinement
Communications of the ACM 14 (1971), pp 221-227

[142] N. Wirth
PASCAL-S: A subset and its implementation
Technical report #12, ETH Zuerich, 1975

[143] W.A. Woods
Context-sensitive parsing
Communications of the ACM 13 (1970), pp 437-445

[144] J.M. Wozencraft and A. Evans, Jr.
Notes on programming linguistics
Department of Electrical Engineering, Massachussetts Institute of Technology, 1971

[145] W.A. Wulf, R.L. London, and M. Shaw
An introduction to the construction and verification of ALPHARD programs
IEEE Transactions on Software Engineering 2 (1976), pp 253-265