DYNAMIC TEMPLATE TRANSLATORS:

A Useful Model for the Definition of Programming Languages

by

Keith Philip Mason, B.Sc.(Hons.)

A thesis submitted for the degree of

Doctor of Philosophy

in the Department of Computer Science,

University of Adelaide.

February 29th, 1984.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF DIAGRAMS AND EXAMPLES

SUMMARY

A new device, called a Dynamic Template Translator (DTT), for the specification of programming languages is defined, and some of its properties are investigated.

DTTs are extensions of context free translation schemes in two directions: (1) their dynamic structure enables production rules to be created and destroyed according to the context seen as the parse progresses, and (2) their template structure enables information contained within symbols to be manipulated.

The purposes of a formal definition are discussed in relation to the notation that a formal definition should have useful human and mechanical characteristics. That is, that the definition should be easily understandable, and that it should be amenable to mechanical processing.

Other work in the area of formal definitions is reviewed, and it is shown that two main streams of research have evolved: (1) proposals encapsulating the syntactic structure of programming languages as a basis for semantic models, and (2) semantic models based on context free languages. The review is followed by an informal introduction to DTTs which shows that DTTs belong to the former category.

A formal model of a DTT is presented, and the

relationship between the input and output languages of a DTT is defined in terms of the concept of a derivation. This is then used to show that DTTs describe the recursively enumerable languages.

Although this is a property shared by many other proposals, DTTs have in addition the characteristic that table driven devices may be constructed from a large class of DTT specifications. To generate the driving tables, the construction algorithm extracts a context free grammar from the DTT, and applies to it an algorithm that takes account of information in the DTT lost to the context free grammar. The other components of these devices are straightforward generalizations from context free translation schema.

A demonstration of the application of DTTs to the languages Asple and Pascal is provided. Asple was chosen as it has already been used to compare the merits of several definition methods, and can therefore be used in the evaluation of DTTs. Pascal was chosen to illustrate that DTTs are capable of describing the structures found in realistic languages.

The structure of the DTT model is investigated and compared with that of other models. In particular, the mechanism of handling syntactic restrictions involving identifier names is found to have a unified approach in DTTs

not found in most other proposals. The descriptive properties of DTTs are compared with that of other models, and DTTs are seen to have useful human and mechanical properties not generally shared by other models.

The thesis concludes by summarizing the characteristics of DTTs, discussing the extent to which they are a useful model for the definition of programming languages, and by suggesting some directions for further research.

DECLARATION

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

If this thesis is accepted for the award of the degree, permission is granted for the thesis to be photo-copied.

Keith P. Mason

February 29th, 1984.

ACKNOWLEDGEMENTS

I wish to thank my supervisor, Dr. W. P. Beaumont, for his encouragement and criticism throughout the course of this research.

I also wish to thank Dr. B. P. Kidman for supervising my work while Dr. Beaumont was on leave.

Financial support for this research was obtained from the University of Adelaide Research Grant, and is gratefully acknowledged.

Chapter One

INTRODUCTION

1.1 Translators

The term translator has a long history. With the advent of computer languages, the term was quickly applied to devices developed to convert the evolving notations into machine code. However, whereas the the traditional role of a translator was in operating between two well defined languages, the new applications concerned source languages that were poorly defined. For example, problems with the definition of Algol 60 are documented by Knuth [24]. The task of constructing a translator from an inadequate definition has always been formidable, especially when even small differences in the translation may have significant implications. The problems this causes are well known. Poole [46] has exhibited differences between Fortran compilers, and Sale [51] has compared several Pascal implementations for inconsistencies. The difficulty of specifying an adequate definition is such that no model of definition has yet been generally accepted as being satisfactory.

1.2 Definitions of Programming Languages

A model for the formal definition of programming languages must, given an alphabet of symbols V, provide a method for selecting the set L of legal programs (a subset of V^*) and the model must also specify the meaning of each program p in L.

Considerably differing interpretations have been applied to the terms 'legal' and 'meaning' in the above. These differences give rise to the following questions:

(1) What is a valid program? Is it one that satisfies the syntactic restrictions of the language, or one that does not infringe the run-time semantic requirements, or one that always terminates with 'correct' output for all inputs?

(2) In what terms is the 'meaning' of a program specified? Is it in terms of a compiler, or of the transitions of an abstract machine, or of a mathematical mapping from the input to the output of a program, or of a logical system that defines all true statements about the program?

(3) How should a formal model reject erroneous programs? Should this be implicit, by giving rules for generating correct programs only, or should the model give a method for distinguishing correct programs from incorrect?

(4) Should a formal model acknowledge realistic restrictions such as bounded storage and finite numeric ranges? If so, how should it specify where restrictions or choices can be made by an implementor, and where the language must be modelled exactly?

The answers to these questions depend very much on one's standpoint. The view taken in this thesis is that the

formal definition of a programming language must be useful. It must be useful in relation to both human and mechanical factors, as explained below.

It must be useful in terms of human factors in the sense of being able to supply information about the language that a user may request. A human user must be able to obtain answers to questions, both general and specific, concerning the language from the definition. The quality of the answers obtained, and the ease of obtaining them, are measures of the usefulness and success of the human aspects of the definition.

The definition must be useful in terms of mechanical factors by providing a basis that is amenable to automatic processing. The areas of applications of a mechanically useful system include compiler generators and program verifiers. The quality of the systems generated, and the ease of producing them, are measures of the usefulness and success of the mechanical aspects of the definition.

There are three important applications which justify considerable effort to find a successful model for the formal definition of programming languages. They are:

(1) the theoretical study of the foundations of programming languages,

(2) the automatic implementation of compilers, and

(3) the automatic verification of programs.

Any formal model that is useful in the above sense, will also be relevant to these applications, since a model that presents accessible information about the language it defines may be expected to provide insights into these areas.

1.3 Scope of the Work

This dissertation reports the result of an experiment in the design of a model for the formal specification of programming languages. The aim of the experiment is to produce a formal model that is useful in the above sense.

A new device, called a Dynamic Template Translator (DTT), for specifying programming languages is introduced in this thesis as the result of the above experiment.

DTTs are informally described in chapter two after a review of the development of the field. DTTs are rigorously defined in chapter three, and important results that can be derived from the formal model presented. Chapter four shows how a table driven DTT may be constructed and evaluates such an implementation. Chapter five demonstrates the practicality of DTTs with two substantial examples, Asple [7] and Pascal [21]. Asple was chosen because it has already been used to compare definitional methods, and Pascal to show DTTs are designed for realistic languages. An evaluation of DTTs and comparison with the most prominent of existing definitional methods is given in chapter six. The concluding chapter, chapter seven, provides a summary of the

characteristics of DTTs, discusses the extent to which DTTs have achieved the goals mentioned above, and gives an outline of directions for further research.

Chapter Two

REVIEW

2.1 Introduction

The formal definition of programming languages had its genesis in the Algol 60 Report [41]. Although linguists had long divided natural language into syntax, semantics and pragmatics [34,39], it was not until Algol 60 that the distinction between syntax and semantics was established in programming languages. The formal notation BNF (which is equivalent to Chomsky's context free grammars [6]) was introduced to specify the syntactic structure of Algol 60. Its semantics and the syntactic component not expressible in BNF were described by prose text. Despite the considerable effort that was put into the formulation of the prose text, it was incomplete, inconsistent and ambiguous [25], characteristics inherent of natural language descriptions. These deficiencies were, however, offset by the clarity and simplicity of BNF. The success of BNF and its variants was so great, that it is still being used to define contemporary languages even though the above problems are well known. For example, Simula [8], Pascal [21], Modula-2 [63] and BPL [61] have this dichotomous form of definition.

2.2 Formal Models

A formal model may be the basis of a theory. BNF provided the foundations for the development of the theory of parsing that was to continue for over a decade. The

automated LL(1) and LALR(1) parsing strategies and syntax directed translation schemes were results of the most fruitful areas of this research (see [4] for some historical notes) which has transformed the problem of parsing context free grammars into a well understood process.

In contrast however, prose descriptions are not amenable to formal treatment. The resulting theoretical void has provided no sound basis for automated systems of any form.

The theoretical work inspired by Algol 60 on the formal specifications of programming languages therefore progressed in two directions. One group of workers proposed formal models with the syntactic structures of programming languages as a basis for a (to be developed) semantic model. Other workers investigated semantic models using context free grammars as a basis. In addition, engineers found in context free grammars a sufficient basis to implement some context dependent language semantics by grafting on symbol table management schemes; names encountered in a program and attributes describing them and the context of their occurrence were entered in tables so that context sensitive syntax requirements could be enforced.

2.3 Restricted Rewriting

The models which tried to encapsulate the syntactic structure of programming languages were generally of a generative nature and obtained their power by placing some

restrictions on the method of rewriting nonterminal symbols. Control Grammars [5] illustrate the type of mechanism found in the simpler proposals of this nature.

Each production rule of a Control Grammar has a context free structure and is assigned a label. The order in which productions may be used in a derivation is specified by a regular set over the production labels. For example the Control Grammar of table 2.1 generates sentences of the language $L=\{a^n b^n c^n \mid n>=1\}$

Table 2.1 - Control Grammar for $L=\{a^n b^n c^n \mid n>=1\}$

rule number	production	control
0	S → ABC	
1	A → aA	
2	A → a	
3	B → bB	$0(135)^*246$
4	B → b	
5	C → cC	
6	C → c	

The order of production applications in the derivation is controlled by the regular set $0(135)^*246$. As productions 1, 3 and 5 must be applied the same number of times, it is clear that this grammar generates precisely L.

Control Grammars generate the recursively enumerable languages, which is their downfall, as it is in general not

possible to construct a recognizing device equivalent to the generative model. This failure disqualifies Control Grammars from being useful in the sense given in chapter one, and is a common characteristic of restrictive rewriting techniques.

To show the variety of such devices that have been investigated, the following list (in approximately chronological order) briefly identifies the restrictive mechanism of each model.

(1) Matrix Grammars [1,20] - Productions are grouped into sequences called matrices. When a matrix is chosen during a derivation all of its rules must be applied in sequence.

(2) Order Grammars [13] - A partial ordering is defined over the production labels p_j, and a production p_i is never applied if there is another production p_j that can also be applied such that $p_j < p_i$.

(3) Indexed Grammars [3] - A string of indices may be associated with non-terminal symbols. The descendants of a nonterminal inherit its indices, thus allowing distant parts of the derivation to share common information (index strings). Although index languages are properly contained within the context sensitive languages, a practical recognizing device for them has not been put forward.

(4) Programmed Grammars [49] - Two sets of production labels, S and F are associated with each

production. If an attempt to apply a production in the derivation is successful, then the label of the next production to be tried must be in S, otherwise in F.

(5) W-Grammars - Two sets of rules, the meta-productions and the hyper-rules are used to generate a potentially infinite set of context free productions. The original use of W-grammars in the Algol 68 Report [58] was to define the context sensitive syntax of the language. The importance of Algol 68 ensured that there was substantial research into the implementation of W-grammars. This took the form of translating a W-grammar into an Affix Grammar [27], for which parsing strategies had been constructed. More recent work [43] however, has shown that W-grammars are sufficiently powerful to completely specify the semantics of programming languages, even to the extent that programs with infinite loops can not be generated. Unfortunately recognizers for these grammars can not be constructed, and therefore they are impractical and not useful.

(6) Macro Grammars [11] - Parameter lists may be appended to nonterminal symbols which are treated as macros and expanded according to a chosen strategy.

(7) Scattered Context Grammars [16] - An extension of Chomsky's context sensitive grammars where the adjacency requirement is removed. A rule may be applied if its context symbols are somewhere (in sequence) to the left or right of the rewritten symbol.

(8) Property Grammars [56] - An extension of context free grammars to allow an infinite set of terminal symbols. Symbols contain two parts, a finite part and an element of an infinite set, called a table.

(9) Time-Varying Grammars [52] - The active productions of a grammar are a function of the length of the derivation.

(10) State Grammars [24] - A state is associated with each side of a production. A production may only be used in a derivation if the generating device is in the state of the left hand side. The new state is that of the right hand side.

(11) Dynamic Syntax [17] - Productions may be added to the grammar as the parse progresses, allowing the production set to reflect the context seen. The dynamic mechanism that provides production rules mapping syntactic categories (such as variables) into sets of identifiers is formulated in terms of the lambda calculus.

(12) Canonic Systems [32] - Based on the formal systems of Post [47] and Smullyan [55], these devices

consist of a set of canons (logical rules stating that certain premises imply certain conclusions) in which variables of predicates may be n-tuples. This allows information to be passed throughout a derivation tree. Language restrictions are manifested by tests for containment of a particular symbol or string in a list.

(13) Direction Controlled Grammars [50] - An extension of programmed grammars in which not only is the next production to be used in a derivation specified, but also whether it is to be applied to the left or right of the current production.

(14) Random Context Grammars [59] - Similar to Scattered Context Grammars except that the context symbols may appear anywhere in the sentential form.

(15) Production Systems [30,31] - A powerful generative device, based on Canonic Systems, but with similarities to BNF. A global list, called an environment, rather than the structures passed through the derivation tree in Canonic Systems, is used to enforce context sensitive language restrictions. This power enables the definition of syntactically legal programs and their translation into a target language. A syntactically complex subset of PL/1 has been described using Production Systems.

(16) Coupled Grammars [57] - A generative device that obtains its power by interrelating parallel sequences of context free derivations.

(17) Dynamic Grammar Forms [15] - This is a two level device in which the grammar form is a template of context free rules. The program is scanned to determine an interpretation of the grammar form which includes a function for mapping entities such as identifiers into syntactic classes. The dynamic nature of the device allows a different mapping for different parsing environments. A subset of PL/1 has been described in this notation.

(18) Notation for Static Semantics [61] - A formal notation derived from BNF that defines static semantics using actions associated with productions to explicitly manipulate data structures such as stacks, variables and strings.

(19) Interdependent Translations [14] - A scheme for specifying translations dependent on the structure of both the input and output languages. Although these are assumed context free, the contextual dependence yields greater power.

(20) Dynamic Production Grammars [44] - An extension of context free grammars where guarded commands are used within productions. The guarded commands are allowed to reference sets of global variables, and may modify the grammar as the parse progresses.

All of the above models were initially defined as syntactic devices; Production Systems and W-grammars only have been substantially extended. All of the authors of the above proposals were hoping to obtain a formalism that reflected the syntactic structure of real programming languages which would then provide a useful basis for a semantic theory. However the formalisms were generally not useful enough to solve the problem of distinguishing legal from illegal sentences, even though some could answer more profound questions about the language defined. Hence interest in these models has diminished.

2.4 Semantics for Context Free Languages

The alternative approach of building a semantic framework upon a context free basis and then extending it to encompass real programming languages has also attracted much attention and has, in contrast, produced many significant results. These methods include:

(1) Attribute Grammars [26] - Values (called attributes) are associated with symbols of the grammar, and rules to evaluate attributes are associated with productions. Information is passed through the parse tree by evaluation rules which may reference the attributes of the parent symbol (inherited attributes) or of descendant symbols (synthesized attributes). The meaning of a program is associated with the attribute of the goal symbol of the grammar.

(2) Operational Semantics - The central aim of this approach is to define an abstract machine for interpreting (abstracted) programs of the language. The model consists of the definition of an abstract machine state that contains all essential information about the progress of the computation, and of the specification of allowable transitions between states (a transition function).

(3) Denotational Semantics - The denotational approach considers a program as specifying a function mapping input to output. A denotational description consists of abstract syntax, domain definitions to describe data structures, and function definitions to specify the meaning of nonterminal symbols.

(4) Axiomatic Semantics - An axiomatic definition enables the the proof of any true statement (and no false ones) about the execution of any program or program segment by using a set of axioms which provide a minimal set of constraints about the language and rules of inference for every language construct. An axiomatic definition is most useful for the construction of proofs that a program possesses certain formal properties, but gives no detail of how they can be achieved.

Operational definitions have been used in many forms. Two of the most important are (1) Landin's SECD machine [29] which was used to provide semantics for Algol 60, and (2)

the technique which became known as the Vienna Definition Language [60] developed at the IBM Vienna laboratory to define the formal semantics of PL/1. Denotational definitions can be traced back to McCarthy [35,36]. However the recent work is based on the theory of Scott and Strachey [53]. Denotational semantics have been used to describe a wide variety of languages including Algol 60, CLU, Snobol 4 and Pascal. Axiomatic semantics developed from Floyd's inductive assertion method [12] for program verification. Hoare's axiomatic approach [18] uses the program text to specify relations between assertions.

The formal nature of these systems has provided a basis for compiler generation schemes. Many of the early systems were little more than parser generators. YACC [23] for example, requires the user to place code to check semantic restrictions immediately following each rule. A user procedure is called each time a reduction occurs in the BOBS system [10]. The CDL system [28] accepts input in the form of an affix grammar and generates a recursive descent parser that uses each nonterminal as an action or a predicate defined by a macro with parameters and local variables. NEATS [22] is an attribute grammar based compiler writing system that has fixed domains for language constructs such as types and environments. NEATS translates the source program into an output stream by calling a user provided procedure every time an output symbol is generated. HLP [48] constructs a parse tree and evaluates attributes, which are Algol procedures, in alternating passes. HLP has generated

compilers for many languages including Simula and Euclid, however it is too large and slow for practical use. SIS [40] was the first compiler generator that did not require the user to encode semantic routines. It uses formal descriptions of syntax and denotational semantics of the language to construct the parse tree of a program and then applies the semantic functions to it and interprets the result. SIS uses an untyped lambda calculus extended with lists and tuples to represent programs and compilers. It has processed several small languages such as Loop and M-Lisp, but takes several minutes to process a six line program. Semantic Grammars [45] combine denotational semantics and attribute grammars and have been applied to Pascal and Fortran subsets to generate compilers that can handle programs of several pages in length.

In just over a decade, semantic models relying on a context free basis have developed significantly. However the models proposed do not fully satisfy the requirement of being useful as presented in the introduction. For example, definitions using denotational semantics are frequently impossible to read and even harder to understand. Operational definitions require an inquisitive user to be familiar with the detail of the interpretation states and transitions, which is very low level information and tedious to use. Axiomatic semantics give no detail of how to implement a language. Lack of clarity is one of the greatest impediments to understanding and using the above models.

2.5 Overview of DTTs

2.5.1 Introduction

DTTs were developed from context free translation schemes such that the clarity that built BNF's success was compromised as little as possible. One of the benefits of retaining BNF's simplicity is that a practical implementation of a large class of DTTs can easily be constructed. The accessibility of a DTT to its reader (whether human or mechanical) is one of its most useful characteristics, and one not generally shared by the above models. As insufficient detail of DTTs has been presented so far, detailed criticism must be deferred until chapter six when an evaluation and comparison of DTTs with other definitional methods is presented (although passing comments will be made in the following). Below is an intuitive description of DTTs; a rigorous definition is given in chapter three.

2.5.2 Structure

DTTs are an extension of a context free translation schemes in two directions; DTTs have dynamic structure and DTTs have template structure.

2.5.2.1 Dynamic Structure

The dynamic structure is specified by action sequences which may be associated with production rules. There are four basic actions that may be composed into sequences. They

are

(1) add production, which creates a production

(2) delete production, which removes a production from the production set

(3) add action, which inserts an action into the action sequence of an existing production, and

(4) continue, which does not alter the production system, but is used to mark a recovery point in case an action fails.

The action sequence of a production is interpreted when the production is recognized. Each action of the sequence is interpreted in turn, except that if an action fails (such as an attempt to delete a production that does not exist, or to add an action to a production that does not exist) then further actions are skipped until a continue action is encountered.

As an example of this first extension, table 2.2 contains a DTT that recognizes the language $L=\{a^n b^n c^n \mid n>=1\}$ using the dynamic nature. To illustrate the mechanism by which the DTT recognizes an element of the language L, a trace of the parse of the string 'aabbcc' is given in table 2.3. A trace of the parse of the string 'aabc' is given in table 2.4 to show how the DTT may reject a string which is not in the language.

Table 2.2 - A DTT for L = $\{a^n b^n c^n \mid n>=1\}$

```
(goal → s)
(s → a b c          [delete production,(b → b bsym)]
                    [delete production,(goal → s)]
                    [continue]
                    [delete production, (c → c csym)]
                    [delete production, (goal → s)]
                  )
(a → asym           [add production, (b → bsym)]
                    [add production, (c → csym)]
                  )
(a → a asym         [add production,
                      (b → b bsym
                        [delete production,(b → b bsym)])]
                    [add production,
                      (c → c csym
                        [delete production,(c → c csym)])]
                  )
```

Seven meta-symbols are used in the notation for DTTs in this thesis. Parentheses are used to delimit productions. The arrow → is used to separate the left and right hand sides of a production. Square brackets are used to enclose

---

Table 2.3 - Parse of 'aabbcc'

stack	input remaining	action
	aabbcc	shift
asym	abbcc	reduce by (a → asym) and add productions (b → bsym) and (c → csym)
a	abbcc	shift
a asym	bbcc	reduce by (a → a asym) and add productions (b → b bsym [delete production (b → b bsym)]) and (c → c csym [delete production (c → c csym)])
a	bbcc	shift
a bsym	bcc	reduce (b → bsym)
a b	bcc	shift
a b bsym	cc	reduce by (b → b bsym) and delete production (b → b bsym)
a b	cc	shift

a b csym	c	reduce by (c → csym)
a b c	c	shift
a b c csym		reduce by (c → c csym) and delete production (c → c csym)
a b c		reduce by (s → a b c) and attempt to delete (b → b bsym), however this fails, so skip the next action and recover at continue. Attempt to delete (c → c csym), however this also fails and so skip the last action.
s		reduce (goal → s)
goal		accept input as a sentence of L.

---

an action of an action sequence. Braces are used to enclose the translation string (there are no translation strings in the example of table 2.2).

The mechanism of a DTT is similar to that used by a shift-reduce parser recognizing a sentence. Symbols are shifted onto a stack, and when the right hand side of a production matches the stack, it is reduced (symbols corresponding to the right hand side of the production are popped off of the stack, and the left hand side symbol

pushed onto the stack). The process of symbol matching is described below in the section on the template structure of DTTs. A reduction must also interpret the action sequence of the production and write its translation string.

In the above DTT, asym is to correspond to the token 'a' on input, bsym to 'b' and csym to 'c'.

Dynamic character is included in three existing models, Dynamic Syntax, Dynamic Grammar Forms and Dynamic Production Grammars. However the dynamic mechanism of DTTs differs from these in that the dynamic control is part of a DTT production, and may itself change as the parse progresses, and is not, as in the former models, part of the external driving control of the device.

Table 2.4 - Parse of 'aabc'

stack	input remaining	action
	aabc	shift
asym	abc	reduce by (a → asym) and add productions (b → bsym) and (c → csym)
a	abc	shift
a asym	bc	reduce by (a → a asym) and add productions (b → b bsym

		[delete production (b → b bsym)]) and (c → c csym [delete production (c → c csym)])
a	bc	shift
a bsym	c	reduce by (b → bsym)
a b	c	shift
a b csym		reduce by (c → csym)
a b c		reduce by (s → abc) and attempt to delete (b → b bsym) which succeeds and then attempt to delete (goal → s) which succeeds. Continue by attempting to delete (c → c csym) which succeeds, then attempt to delete (goal → s) which fails.
s		As the parse can not progress and is not in the accept state, the input is rejected as a member of L.

2.5.2.2 Template Structure

The second extension is in the structure of DTT symbols. Whereas a symbol in a context free grammar is considered as an indivisible token (even if its usual representation is a string) within a DTT a symbol is considered as a string. That is, a single token in the input stream is given a string representation within the workings of a DTT. Further, subject to some constraints, symbols in DTT productions may also contain templates to correspond to arbitrary substrings. The constraints and correspondence are described below.

As an example of this second extension, table 2.5 contains a DTT that recognizes the language $L=\{a^n b^n c^n \mid n>=1\}$ using the template structure. Table 2.6 illustrates the mechanism of the DTT with a parse of the string 'aabbcc'.

The parse of any string that is not in L would proceed until the reduction by production five fails (e.g. the first application for 'aabc'), when, as no other production will succeed, the device halts rejecting the validity of the input.

Table 2.5 - A DTT for $L=\{a^n b^n c^n \mid n>=1\}$

1. (s.a. → asym)
2. (s.Qa. → s.Q. asym)
3. (s.Qb. → s.Q. bsym)
4. (s.Qc. → s.Q. csym)
5. (s.aXbYcZ. → s.aaXbbYccZ.)
6. (goal → s.aabbcc.)
7. (goal → s.abc.)

The notion used throughout this thesis is that upper case characters denote templates, whereas lower case characters and punctuation marks are used in the working string. When recognizing a sentence the templates on the right hand side will match strings on the stack. Templates on the left hand side (and those in translation strings, and those to be substituted in productions contained within action sequences) will have the string matched by their namesake on the right hand side of the production substituted in the construction of the symbol to be pushed on the stack. Thus every template used on the left must appear at least once on the right. Multiple occurrences on the right require identical strings to be matched. Templates can not start symbols, end symbols, be adjacent to another template in the symbol, or match the null string.

Table 2.6 - Parse of 'aaabbbccc'

stack	input remaining	action
	aaabbbccc	shift
asym	aabbbccc	reduce 1
s.a.	aabbbccc	shift
s.a. asym	abbbccc	reduce 2 (Q = 'a')
s.aa.	abbbccc	shift
s.aa. asym	bbbccc	reduce 2 (Q = 'aa')
s.aaa.	bbbccc	shift
s.aaa. bsym	bbccc	reduce 3 (Q = 'aaa')
s.aaab.	bbccc	shift
s.aaab. bsym	bccc	reduce 3 (Q = 'aaab')
s.aaabb.	bccc	shift
s.aaabb. bsym	ccc	reduce 3 (Q = 'aaabb')
s.aaabbb.	ccc	shift
s.aaabbb. csym	cc	reduce 4 (Q = 'aaabbb')
s.aaabbbc.	cc	shift
s.aaabbbc. csym	c	reduce 4 (Q = 'aaabbbc')
s.aaabbbcc.	c	shift
s.aaabbbcc. csym		reduce 4 (Q = 'aaabbbcc')
s.aaabbbccc.		reduce 5 (X='a',Y='b',Z='c')
s.aabbcc.		reduce 6
goal		accept input as in L.

The mechanism of a template match is similar to the BREAK pattern of Snobol4, where the 'break character' is the character immediately to the right of the template. The string matched is all characters from the starting position up to, but not including the first occurrence of the 'break character'.

The template nature of DTTs has parallels in two existing models, W-grammars and Property Grammars. The notation used for DTTs is very similar to that used for W-grammars, and one could wrongly gain the impression that a DTT template is like a W-grammar meta-notion with a much weaker structure. However a DTT is a single level device and quite different from a W-grammar. A DTT symbol is notionally much closer to the symbol of a property grammar. Every DTT symbol has a prefix which can never be matched by a template of a production (since a symbol may not commence with a template), and the set of these prefixes is finite. The remainder of the symbol corresponds to the table of a property grammar, as it is a member of a possibly infinite set. This view of a DTT symbol becomes very important when considering table driven DTTs as it is the part of the symbol that belongs to the finite set that is used as the handle to turn the driving mechanism of the recognizing device.

2.5.3 Conclusion

Although it has just been shown that either the dynamic nature or the template nature of a DTT is sufficient to

describe context dependent features, both are needed when describing programming languages. The dynamic nature is essential in catering for the changes to the parsing environment, and the template nature is needed to enforce complex context dependent restrictions.

As an introduction, this discussion of DTTs has been informal. A rigorous definition of DTTs will be given in chapter three. Chapter four will describe in detail the table driven parser on which the above parses are modelled, and give criteria to determine which parsing action should be performed in any particular instance.

2.6 Summary

The review of other relevant work has shown that two main approaches to research in the area of formal definitions have developed: (1) proposals that encapsulate the syntactic structure of programming languages as a basis for a semantic model, and (2) semantic models based on context free languages. It was also seen that no existing model provides a satisfactory method of defining programming languages.

The overview of DTTs has demonstrated that DTTs are a strong syntactic device with the ability to define language semantics via a translation into a known language. DTTs therefore belong to the first of the two approaches mentioned above.

Chapter Three

FORMAL MODEL

3.1 Introduction

In this chapter the definitions of DTTs and related concepts are presented, and are used to derive some formal results about the nature of the languages processed by DTTs.

3.2 Notation

The notation used in this thesis for cartesian products and discriminated unions is similar to that of Hoare [9], and follows very closely the record and record variant data structures of Pascal by naming components of tuples.

Denote the cartesian product C of sets X_1, X_2, ..., X_n by

$$C = (s_1:X_1,\ s_2:X_2,\ \ldots,\ s_n:X_n)$$

If for each i = 1, 2, ..., n x_i is an arbitrary element of X_i, then

$$X = (x_1,\ x_2,\ \ldots,\ x_n)$$

is an arbitrary element of C, and write $X.s_1 = x_1$, $X.s_2 = x_2$, ..., $X.s_n = x_n$.

Denote the union U of sets X_1, X_2, ..., X_n discriminated by set $Y = \{y_1, y_2, \ldots, y_m\}$ by

$$U = ([y:Y]y_1:C_1;\ y_2:C_2;\ \ldots;\ y_m:C_m)$$

where $C_i = (s_{i_1}:X_{i_1};\ s_{i_2}:X_{i_2};\ \ldots;\ s_{i_p}:X_{i_p})$.

If c_i is an arbitrary element of C_i then $S = (y_i,c_i)$ is an element of U, and we write $S.s_{i_1} = x_{i_1}$, $S.s_{i_2} = x_{i_2}$, ..., $S.s_{i_p} = x_{i_p}$ (note that $S.s_{jk}$ is undefined for j different

from i).

Frequently it is necessary to denote several strings of characters that are adjacent to each other. To avoid ambiguities, the symbol $ is inserted as a separator where required. {$} is also denoted by $ if no confusion can arise.

The function first which is used in the following, is defined here.

$$\text{first}(z_1\ z_2\ \ldots\ z_n) = \begin{cases} z_1 \text{ if } n >= 1 \\ \text{null string if } n = 0 \end{cases}$$

3.3 Definition of a Dynamic Template Translator

3.3.1 Specification

A DTT is a 9-tuple $\langle I,S,T,O,v,p_o,L_I,L_O,s\rangle$ where

(1) I, S, T and O are alphabets, called the input, symbol, template and output alphabets respectively. S and T are disjoint and finite.

(2) v is a set of finite elements of S^*, called the terminal prefixes.

(3) p_o is the initial set of productions, which are defined below.

(4) L_I and L_O are one to one functions. $L_I: I \rightarrow S^+$ is the input lexical function and $L_O: (S^+\$)^+ \rightarrow O$ is the output lexical function.

(5) s is an element of S^+ and is called the distinguished (or goal) symbol.

The motivation for these concepts and names is derived from the intended use of DTTs to translate programs into object code. Each input token (an element of I) is mapped into entities called symbols, which are strings over the alphabet S (the DTT working alphabet), by the input lexical function L_I. The recognition process is defined by the initial set of productions p_0, which is expressed in terms of v, S, T and s, and is described in detail below. During this process the translation is produced; initially as strings over S, but these are later mapped into the output language O by the output lexical function L_0. The process halts when the mechanism successfully constructs the distinguished symbol s, or it can proceed with no legal operation.

3.3.2 Production Rules

A DTT production is an element of the set $P=(lhs:M;rhs:M^*;q:A^*;w:M^*)$ where

(1) $M = S^+(T^+S^+)^*$

Elements of M are called template symbols.

(2) q is called the action sequence and A is the set of actions which are defined below.

(3) w is called the output string.

(4) lhs may not be prefixed by any element of v.

(5) elements of T^+, called templates, in lhs and w must appear at least once in rhs.

The context free grammar basis of DTTs is evident in the (lhs, rhs) structure of DTT productions. The context free equivalent of requirement (4) is not allowing terminal symbols to define productions. Restriction (5) comes from the intended mechanism of DTTs matching the right hand side of a production against a sentential form and then substituting for templates in the left hand side and output string w.

3.3.3 Actions and Action Sequences

The dynamic nature of a DTT is specified entirely in the action sequences associated with productions.

Let action = {add production,
delete production,
add action,
continue}.

Using the notation presented above for discriminated unions, define A = ([a:action]
add production:(pa:P);
delete production:(pd:P);
add action:(q:A;p:P);
continue:()).

That is, an action may either add the production pa, delete the production pd, add the action q to production p, or continue (without altering any productions in the system).

The effect of these actions on a production set is formalized below.

If Q is an element of A then we may define a function $K_Q: P \rightarrow P$ called the action function, describing the effect of an action on a production set p by

$$K_Q = \begin{cases} p+\{Q.pa\} & \text{if } Q.a=\underline{\text{add production}} \\ p-\{Q.p\}+\{(lhs,rhs,q_p{}^{\circ}Q.q,w)\} & \text{if } Q.a=\underline{\text{add action}} \text{ and } Q.p = (lhs,rhs,q_p,w) \\ p-\{Q.pd\} & \text{if } Q.a=\underline{\text{delete production}} \\ p & \text{if } Q.a=\underline{\text{continue}} \end{cases}$$

where + denotes set union, - set difference, and $^{\circ}$ is the sequential composition of actions, which is defined below. (Note that the second alternative deletes the existing production Q.p and adds the production which is formed by appending the action Q.q to production Q.p)

The evaluation of K_Q is said to be successful unless the result of either set subtraction is the original set (i.e. the production to be removed does not exist) when it is said to be unsuccessful.

The sequential composition (denoted by $^{\circ}$) of the action Q_1 to be followed by Q_2, applied to production set p is defined by

$$(Q_1 \circ Q_2)(p) = \begin{cases} Q_2(Q_1(p)) & \text{if } Q_1 \text{ is successful, and is successful if } Q_2 \text{ is, or if } Q_1.a \text{ is continue and is successful if } Q_2 \text{ is.} \\ Q_1(p) & \text{otherwise (that is, } Q_1(p) \text{ is unsuccessful and } Q_2.a \text{ is not continue) and is unsuccessful} \end{cases}$$

3.3.4 Symbol Matching and Template Substitution

The power of a DTT production is in its template structure. Templates are designed to match arbitrary substrings and allow the information so found to be manipulated. Informally, for a template symbol to match a string, each template must correspond to a non null component of the string that does not contain the character that follows the template (c.f. BREAK in Snobol 4), and the strings between templates must appear between the strings matched by the templates (all matches must be in sequence). The mechanism of matching templates is now formalized.

A template symbol $Y = t_o g_1 t_1 \ldots g_n t_n$ (t_i in S^+, g_i in T^+) is said to match a symbol $W = r_o s_1 r_1 \ldots s_n r_n$ (r_i, s_i in S^+) if $r_i = t_i$ (i=0..n) and if s_j does not contain first(r_j) (j=1..n). Such a match is denoted $Y \sim W$, and s_j is said to be associated with g_j.

When a production is recognized, the templates matched and referenced in the left hand side template symbol need to be substituted to form a symbol. This mechanism is now defined.

Let X and Y_i (i=1..n) be template symbols and W_i (i=1..n) be symbols (e.g. X could be the lhs of a production, Y the rhs and W the symbols matched) where

$X = r_{0,0}g_{0,1}r_{0,1} \cdots g_{0,N(0)}r_{0,N(0)}$

$Y_i = r_{i,0}g_{i,1}r_{i,1} \cdots g_{i,N(i)}r_{i,N(i)}$

$W_i = r_{i,0}s_{i,1}r_{i,1} \cdots s_{i,N(i)}r_{i,N(i)}$

such that $Y_i \sim W_i$ (i=1..n) [therefore $s_{u,v}$ is associated with $g_{u,v}$]. Note that double subscripting is necessary as there are many rhs symbols and that N(i) is the number of templates in Y_i (for i > 0), or in X (if i = 0).

Define the symbol, denoted by $\hat{X}$ formed by substituting X's templates with their associated strings by

$\hat{X} = r_{0,0}f_i r_{0,1} \cdots f_{N(i)}r_{0,N(0)}$

where $f_i = s_{u,v}$ if $g_{0,i} = g_{u,v}$ for some u > 0.

That is, $\hat{X}$ is formed by substituting all templates with associated strings in X by their associated strings. Note that if a template name has several occurrences ($g_{0,i} = g_{u,v}$ for at least two u,v with u > 0), then the same string must be associated with each occurrence.

There is a more general case of template substitution that needs to be considered, and that is in a production

that is a parameter of an action. In this case there are three possible interpretations of any template. The template:

(A) may be associated with a string in the right hand side of the recognized production and is to be substituted. These templates are sometimes referred to as being bound to their occurrence.

(B) may denote a template in the production (e.g. the production to be added may contain templates).

(C) may be used to match a substring of a production that is to be found in the production set (e.g. a production to be deleted may be referenced in the action by a parameter with templates to match substrings). Note that these templates can not be referenced elsewhere in the action sequence or the production.

For example, consider the production

```
(lhs_A. → rhs_A_B.
   [delete_production,(l_A_B_CCCC_X. → r_X_Y_ZZZZ.)])
```

Templates A and B in the parameter production of the delete action correspond to case (A), X and Y to case (B), CCCC and ZZZZ to case (C).

Case (A) can easily be distinguished from cases (B) and (C) by the scope of the template name; if the name of the template appears anywhere in the right hand side of the

recognized production, it is to be substituted. There is however no such convenient rule for choosing between (B) and (C). The following convention is used: If a search is to be made of the production set to find a production, and the production parameter contains a template whose name contains four or more letters, then that template may match a substring (that may not contain templates) of existing productions (i.e. case C), otherwise the template must correspond to a similar template (i.e. A template that has the same effect. Names only distinguish templates within a production - a systematic renaming of templates does not alter a production) in the object production (i.e. case B).

The substituted symbol is defined similarly to above except that f_i could be the template $g_{0,i}$ if no associated string exists (case B), and that (case C) W_i would have to be generalized to template symbols. As the generalization is clear, superfluous notation is not introduced.

A related problem occurs in the interpretation of production parameters of actions. For example, in the action

[add action,(p1),[delete production,(p2)]]

the production to be deleted could be either

(a) p2 literally, or

(b) a production in the system that matches p2.

This second alternative is denoted by a quote. viz.

[add action,(p1),[delete production,'(p2)]]

3.3.5 Derivation

Having discussed all the parts of a DTT, it is now shown how they fit together and relate to the input and output languages with the relation called derivation. This is made clearer by introducing the notion of a proto-derivation, which is a generalization of the context free derivation to incorporate the template nature of the DTT (i.e. not considering the dynamic nature of the DTT). The proto-derivation is defined in terms of a set of productions P' that contain all productions that could ever possibly be added to the system (and possibly some that the dynamic nature would preclude).

Define P' recursively by: P' is a set of productions containing

(1) all productions in p_o, and

(2) all productions that could be added to the DTT. More specifically, the production pa from every action [add production,(pa)] in (1) P' and in (2) the set C, where C contains all actions in action sequences of productions in P', and C also contains the action q of every action [add action,p,q)] in C.

However, if pa contains any templates that are bound to their occurrence, then a rule is added to P' for each possible string that each of the templates could match.

(3) no other productions.

A proto-derivation is, roughly speaking, a sequence of transitions, each from a form a_i in $(S^+)^+$ to a_{i+1} by the application of a production in P'.

Formally, the relation proto-derivation $\rightarrow$ is defined by

$$a\ r\ c \rightarrow a\ t\ c$$

(where a, c and t are in $(S^+\$)^*$, and r is in $S^+\$$) if there exits a production (lhs,rhs,q,w) in P' such that rhs ~ t and r = $\hat{lhs}$.

The dynamic nature of a DTT demands that a production exists when used in a reduction. This characteristic is not checked in the proto-derivation as P' contains all possible productions and does not model the changes that occur with time. This characteristic is however incorporated in the notion of derivation.

A derivation, denoted $\Rightarrow$, is a sequence of transitions as in a proto-derivation, but with the added restraint that in the i^{th} step of the derivation, if production (lhs_i, rhs_i, q_i, w_i) is used, then it must be in the production set p_i and $p_{i+1} = K_{q_i}(p_i)$, where K is the action function defined earlier.

Further, if $a_j = s_1\$ \ldots s_n\$$ where each s_k is prefixed by an element of v then $i = L_I(s_1)L_I(s_2) \ldots L_I(s_n)$ is in the input language I, and $o = L_O(w_j \ldots w_2\ w_1)$ is in the output language O.

The concept of derivation has defined the input and output languages of a DTT in terms of the mechanism of the DTT.

3.4 Scope of DTT Languages

The importance of the derivation is in formulating the mechanism of a DTT, and particularly in enabling a precise specification of the input and output languages of a DTT. Such a formal characterization of DTT languages allows formal results concerning the scope of DTT languages to be obtained.

In this section it is shown that:

(1) For any Turing Machine that accepts a language L, there exists a DTT D with input language $I_D = L$.

(2) For any DTT D with input language I_D, there exists a Turing Machine that accepts a language $L = I_D$.

(3) For any DTT D with input language I_D and output language O_D, there exists a DTT D' with input language $I_{D'}$ and output language $O_{D'}$ such that $I_D = O_{D'}$ and $O_D = I_{D'}$.

The importance of these three results is in establishing that the input and output languages of a DTT belong to the class of recursively enumerable languages.

It was shown in chapter two that most of the restrictive rewriting systems proposed as models of programming languages generated the recursively enumerable

languages, and that this was the cause of their failing as it is not in general possible to construct recognizing devices equivalent to the generative model. DTTs do not share this fate as it will be shown in chapter four that there is a subclass of DTT whose initial production set p_o satisfies some constraints which enable table driven translators to be automatically constructed. It is this subclass of DTTs that provide a useful model for the definition of programming languages. Proofs of the above theorems showing the necessity of constructing a subclass follow.

Theorem 3.1:

For any Turing Machine that accepts a language L, there exists a DTT with input language I_D = L.

Proof:

Without loss of generality assume that the Turing Machine has only one tape, and that the transition function is specified by quadruples (q_i,x,y,q_f) [Notation : (q_i,x,y,q_f) denotes an allowable transition from state q_i with x under the read/write head, to state q_f. If y is a symbol of the Turing Machines alphabet, it is written at the current head position. Alternatively y may be either of L or R, which are special symbols not in the Turing Machine's alphabet. If y is L, the head moves one square left, or if y is R, the head moves one square right.]

The proof consists of constructing a DTT that uses a specially constructed symbol to represent the state, tape

contents and head position of the Turing Machine, and has productions for each transition quadruple of the Machine. The structure of the symbol is shown by the template symbol tape_Q_L.LEFT^H.R.RIGHT_ which will match a tape that has at least two symbols to the left of the head (the one adjacent to the head is associated with the template L) and at least two to the right (the one adjacent to the head is associated with the template R, and the character under the head with H). Q matches a string representation of the machine states. It is assumed that the blank symbol of the Turing Machine is b, and that the punctuation marks _, ^ and . are not in the alphabet of the Turing Machine.

Construct a DTT as follows (to avoid an excess of notation, only the production set p_o will be given):

For each quadruple (q_1,x,L,q_2) of the transition function for the Turing Machine, construct in p_o the productions

(tape_q_2_LEFT^Y.xRIGHT_ → tape_q_1_Y.LEFT^xRIGHT_)

(tape_q_2_.^Y.xRIGHT_ → tape_q_1_Y.^xRIGHT_)

(tape_q_2_.^b.xRIGHT_ → tape_q_1_.^xRIGHT_)

For each quadruple (q_1,x,R,q_2) construct

(tape_q_2_x.LEFT^RIGHT_ → tape_q_1_LEFT^x.RIGHT_)

(tape_q_2_x.LEFT^b._ → tape_q_1_LEFT^x._)

For each quadruple (q_1,x,y,q_2) where y is not L or R, construct

(tape_q_2_LEFT^yRIGHT_ → tape_q_1_LEFT^xRIGHT_)

Also include in p_o the productions

(s → tape_q_f_X_) for each final state q_f of the Turing Machine,

(tape_q_s.^A._ → g_A_) for each start state q_s of the Turing Machine,

(g_A.x_ → g_A_ x) and

(g_x_ → x) for every symbol x in the Turing Machine's alphabet.

p_o contains no other productions.

The DTT will derive a string i if and only if the Turing Machine it simulates can halt accepting i. Thus the input language of the DTT is exactly that accepted by the Turing Machine.

Theorem 3.2:

For any DTT D with input language I_D there exists a Turing Machine that accepts a language L = I_D.

Proof:

Let n be the maximum number of distinct templates that occur in the right hand side of any production of D.

The proof consists of constructing an n+3 tape nondeterministic Turing Machine which simulates the DTT. To avoid an excess of notation, instead of listing a large number of transition quadruples, the mechanism of the Turing Machine will be discussed.

Tape 1 contains the input (a sequence of tokens from the input alphabet of the DTT)

Tape 2 is used for a standard representation of the active productions; templates are systematically replaced by special symbols referring to tapes 4 to n+3.

Tape 3 is used as a parse stack.

Tapes 4 to n+3 are used to remember strings matched by templates; for each production a particular template name is associated with a particular tape.

The states, tapes, alphabet and transition function of the DTT are constructed to do the following:

(1) Initially have p_o on tape 2 in the appropriate format (this could be done by finite control).

(2) To halt when tape 3 contains the goal symbol and all the input has been read.

(3) To repeatedly non-deterministically choose between reading the next input symbol or attempting to recognize an arbitrary production. The machine may halt as in (2) above, however if it chooses to read when all the input has been consumed, or the match as described below fails at any point, then the Turing Machine must enter an infinite loop.

If it chooses to read the next input symbol, the Turing Machine must advance tape 1 and write the string form of the symbol onto tape 3.

If it chooses to recognize a production, the Turing

Machine must initialize tapes 4 to n+3, non-deterministically decide which production to recognize, move the head of tape 2 to the last symbol of the right hand side of the production, and match each symbol of the production against the symbols on the top of the stack (tape 3). Strings corresponding to templates must be copied (with end markers) onto the appropriate tape (tapes 4 to n+3). When the matching is complete tapes 4 to n+3 must be checked to ensure that multiple occurrences of any template matched the same string, and also that the null string was not matched. The stack (tape 3) must be popped and the associated symbol of the left hand side of the production pushed onto tape 3. The action sequence must be interpreted; any productions or actions to be added must be written onto tape 2 appropriately (shifting other productions if necessary), and deleted productions removed from tape 2.

The Turing Machine will recognize an input string and halt if and only if the DTT it simulates can derive the string. Thus the language accepted by the Turing Machine contains only and all sentences in the DTTs input language.

Theorem 3.3:

For any DTT D with input language I_D and output language O_D, there exists a DTT D' with input language $I_{D'}$ and output language $O_{D'}$ such that $I_D = O_{D'}$ and $O_D = I_{D'}$.

Proof:

Let D = <I,S,T,O,v,p_O,L_I,L_O,s> be a DTT.

The proof constructs a DTT D' whose derivations have a similar structure to D except that

(1) all leaves of the parse tree of D (string representations of tokens of I) are rewritten to null in D', and they are also written to the translation in D'.

(2) All translation strings of D are appended to the parse tree as terminals in D'.

This effectively swaps the input and output languages.

Construct D' = <O,S U {_},T,I,v',p_O',L_O^{-1},L_I^{-1},s> where

(1) _ is a character not is S or T,

(2) p_O' contains all productions of p_O modified as follows: (lhs,rhs,q,w) in p_O becomes (lhs, rhs w, q,) in p_O',

(3) p_O' also contains the following rules for each x member of v:

(x,,,x)

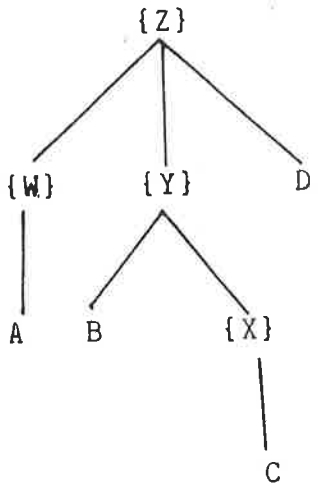
(x#,,,x#) for each character # in S

(xX#,,,xX#) for each character # in S

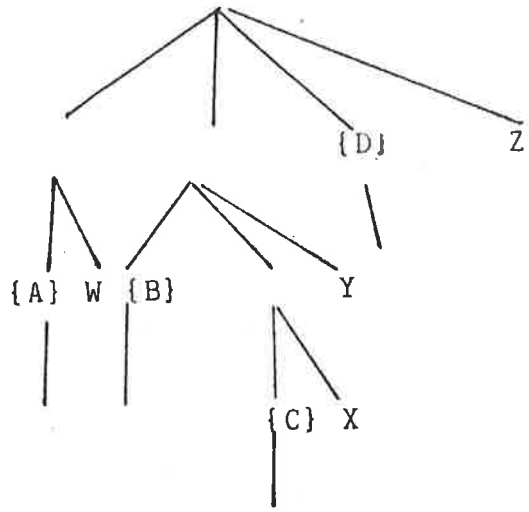
(4) v' is all elements of w appended to rhs in (2).

It follows from the construction that D' satisfies the requirements. An illustration of the construction is given in Diagram 3.1.

Diagram 3.1 - Schematic of parse trees



Parse Tree in D
Input String: ABCD
Output String: WXYZ



Parse Tree in D'
Input String: WXYZ
Output String: ABCD

Corollary 3.4:

The input and output languages of a DTT belong to the class of recursively enumerable languages.

Proof:

From theorems 3.1 and 3.2 it can be seen that the DTT input languages are equivalent to the languages accepted by Turing Machines, namely the recursively enumerable languages. Theorem 3.3 shows that the DTT output languages are also recursively enumerable.

Chapter Four

TABLE DRIVEN DTTS.

4.1 Introduction

The weakness of the restrictive rewriting systems described in chapter two is their inability to be be used to construct a recognizing device equivalent to the generative model for a language. This chapter describes a large subclass of DTTs which do not share this property, as table driven translators may be constructed for them. After describing the construction of such a translator, an appraisal is given of a working implementation of an automated DTT generator system.

4.2 Constructing a Translator

4.2.1 Extending a Context Free Basis

As context free grammars were extended to DTTs, it is reasonable to expect that the recognizing device for a context free grammar, the push-down automaton, may be extended to form a recognizing device for DTTs. Before discussing these extensions, some of the terms used in the recognition of context free grammars by table driven shift/reduce parsers and translators will be introduced, as these will be used in describing the translating device for DTTs.

4.2.2 Context Free Translation

4.2.2.1 Shift/Reduce Translators

A push-down automaton consists of a push-down stack, input tape and finite control. A shift/reduce parser is a particular form of a push-down automaton in which the control is defined by a universal driving routine and a set of data called parse tables that are dependent on the grammar defining the language. There are two sets of tables, the 'f' tables, which are arrays indexed by terminal symbols to obtain a parsing action, and the 'g' tables, which are arrays indexed by terminal and non-terminal symbols to obtain a pair of 'f' and 'g' tables. The driving routine continually consults the tables to determine which parse action should be performed. A parsing action may either

(1) indicate a successful parse: the success action,

(2) indicate an unsuccessful parse: the error action,

(3) indicate that a symbol is to be read and pushed onto the stack: the shift action, or

(4) indicate that a production has been recognized, and that the right hand side of the production (which is currently on the top of the stack) should be replaced by its left hand side: the reduce action.

The driving routine is described by the code segment in table 4.1 where insymbol returns the next input token in sym. Other entities have meanings implied by their names.

Table 4.1 - Driving Routine of a Shift/Reduce Parser

```
insymbol(sym);
current_tables := start_tables;
initialize_stack(current_tables);
cycle
  begin
    action := current_tables.f[sym];
    case action of
      accept : exit(success);
      error : exit(failure);
      shift :
        begin
          push_stack(sym,current_tables);
          insymbol(sym);
          current_tables := current_tables.g[sym]
        end;
      reduce(production) :
        begin
          for length(production.rhs) steps do pop_stack;
          current_tables :=
            top_of_stack_tables.g[production.lhs];
          push_stack(production.lhs,current_tables)
        end
    end
  end
```

Translation schemes based on shift-reduce parsers produce a portion of output whenever a production is recognized, i.e. when a reduction is performed.

4.2.2.2 Table Generation

Several strategies for constructing the 'f' and 'g' tables have been proposed. These methods all have a common basis; they iterate all possible parse configurations and group them into equivalence classes called item sets. As the driving routine for DTTs works on tables that are constructed by a modified parse table generator, it is necessary to introduce the method of parse table construction in order to discuss the modifications. This discussion of LR(1) table generation is based on Aho [4], where greater detail may be found.

Each item of an item set represents a partially recognized production. For example, the LR(1) item [X →A.B,a] denotes a state where A has been recognized, and if symbols derivable from B are recognized next on input with right context of 'a', then X will have been recognized. 'a' is called the look-ahead. Two functions, 'Closure' and 'Successor', are important in constructing the collection of item sets.

Closure adds items to the current set that could be recognized in the current context. For example, Closure of the above item would add items of the form [B →.Q,a] to recognize B. Closure of an item set I is defined by the

procedure in table 4.2.

Table 4.2

```
procedure Closure(I);
  begin
    repeat
      for each item [A → X . B Z, f] in I and each
                production B → Y in the grammar and
                each b in first(Zf) such that
                [B → . Y, b] is not already in I
      do add [B → . Y, b] to I
    until no more items can be added to I;
    return I
  end
```

Successor is the function that defines which item set will represent the state of the parse in the next step. The successor of item set I on recognition of symbol X is defined by the procedure in table 4.3.

The procedure to construct, by iteration of all possible parse configurations, the collection of item sets of a grammar G with goal production GOAL → S is given in table 4.4.

Table 4.3

```
procedure Successor(I,X);
  begin
    let Q be the set of all items [A → J X . K, f]
    such that [A → J . X K, f] is in I;
    return closure(Q)
  end
```

The 'f' and 'g' tables that drive a shift-reduce parser are constructed from the collection of item sets; each set corresponding to a pair of 'f' and 'g' tables. The 'f' table represents the information contained in the items of the set and the 'g' table represents the item sets formed by the successor operation on the set. The following rules are used to construct the 'f' and 'g' tables for item set I.

1. If [A → J . a K, b] is in I where 'a' is a terminal symbol of the grammar, then f[a] is set to shift.

2. If [A → X . , a] is in I, then set f[a] to reduce by A→X.

3. If [GOAL→S . ,null] is in I, then set f[null] to accept.

Table 4.4

```
procedure Items(G);
  begin
    C := {Closure({[GOAL → . S, null]})};
    repeat
      for each set of items I in C and each grammar
                symbol X such that Successor(I,X) is
                not empty and not already in C
      do add Successor(I,X) to C
    until no more sets of items can be added to C
  end
```

---

4. All entries in the 'f' table not defined by the above are set to error.

5. If Successor(I,X) returns item set T, then g[X] is set to the tables corresponding to set T.

Unfortunately some context free grammars can give rise to item sets that contain items which result in conflicting entries in the 'f' table; either a reduce action conflicting with a shift action (a shift/reduce conflict) or a reduce action conflicting with a different reduce action (a reduce/reduce conflict). In this case the tables produced are inadequate for parsing as there is at least one sentence

of the language for which the choice in the case statement of the shift-reduce parser in table 4.1 is not uniquely defined. Such instances are called parse table conflicts and their absence (or presence) defines that the grammar is (or is not) LR(1).

The LR(1) table generation algorithm produces an impractically large number of tables when applied to programming languages. An economical generator that is only slightly less general, is the LALR(1) table generator. LALR(1) tables may be generated from item sets formed by merging LR(1) item sets with common cores (items without look-aheads), however more efficient algorithms for LALR(1) table construction exist (see [4] for detail on LALR(1) devices).

4.2.3 Strategy for Constructing a Table Driven DTT

4.2.3.1 Introduction

The dynamic and template characteristics of DTTs are extensions of a context free basis. It is this underlying structure that provides the handle to a table driven mechanism for DTTs. The following procedure is used to exploit this structure to obtain a table driven DTT.

(1) A context free grammar is extracted from the DTT.

(2) Using algorithms derived from the standard table generation algorithms given in tables 4.2, 4.3 and 4.4 above, 'f' and 'g' parse tables are constructed

for the context free grammar. These algorithms are augmented to use information in the DTT that is lost in the context free grammar to resolve table conflicts.

(3) The shift-reduce parser described in table 4.1 is extended to cater for the structure of DTT symbols, for the dynamic nature of a DTT, and also for two additional parsing actions that the augmented generator may generate to resolve conflicts.

Provided that there are no unresolvable conflicts in the parse tables, the translator so produced is a faithful implementation of the DTT. Each of the above extensions is described below. As (3) is the easiest to describe, it is presented first.

4.2.3.2 Extensions to the Parser

Some conflicts in the parse tables may only be resolvable at parse time. To cater for this, the repertoire of parser actions has been augmented by two. In addition to the actions described in section 4.2.2.1 an action may indicate that the parser should:

(1) choose between conflicting reduce actions depending on which one exists in the production set (it will indicate an error in a similar manner to reduce if no matching production exists, but is undefined in the case that more than one production exists as this should never happen if the tables are

correctly generated) : the resolve r/r action, and

(2) choose between conflicting reduce actions and a shift action depending on whether or not a production for the reduction is present (the table generator must ensure that the default shift is always appropriate if a reduction production can not be found) : the resolve s/r action.

The extensions to the shift-reduce translator to cope with the string nature of DTT symbols require that the shift action should convert a token on input into its string form when pushed onto the stack, which has also been modified to accommodate strings. The reduce action must match the right hand side templates against the stack symbols and appropriately substitute the strings matched.

Extending the parser to cope with the dynamic nature is slightly more demanding. As the driving routine calls for reduction by a production class only, the parser must be extended to maintain copies of the active productions so that the matching mechanism can determine that a production exists which matches the top of stack. The reduce cycle of the parser must also interpret the action sequence of a recognized production, as well as writing the translation string.

The modified driving routine is shown in table 4.5. In that segment of code, the routine insymbol incorporates the input lexical function of the DTT and returns the string form of the input token in sym. The routine execute

corresponds to the action function K of chapter 3. Class(sym) is the corresponding symbol in the context free grammar obtained from the DTT, the match function returns a production with the required syntactic structure that matches the top of stack, but is undefined if no production matches, and the procedure execute interprets the action sequence of the recognized production. The s/r and r/r actions can only distinguish between two conflicting actions, however the generalization is apparent.

The question of efficiency of a DTT may be raised, as the overheads in both time and space required by the extensions could be large. These costs can be reduced as the context free part of the system need not be stored. However the term 'useful' from the introduction was not restricted to meaning 'most efficient'. Indeed, that a formal definition can provide a model for recognizing a language is a very useful characteristic. It is never doubted that more efficient mechanisms can be found to recognize particular languages; a formal model provides a general rule that others can use as a standard.

4.2.3.3 Extracting a Context Free Grammar from a DTT

The symbols of the context free grammar to be constructed from a DTT are simple lexical prefixes, called syntactic classes, of DTT symbols. If t is a DTT symbol then denote its syntactic class by class(t). For example, if

---

Table 4.5 - Driving Routine for a DTT

```
insymbol(sym);
current_tables := start_tables;
push_stack(current_tables);
cycle
  begin
    action := current_tables.f[class(sym)];
    case action of
      accept : exit(success);
      error : exit(failure);
      shift :
        begin
          push_stack(sym,current_tables);
          insymbol(sym);
          current_tables := current_tables.g[class(sym)]
        end;
      reduce(prod_class) :
        begin
          prod_match := match(stack,prod_class);
          if undefined(prod_match) then exit(failure);
          for length(prod_class.rhs) steps do pop_stack;
          current_tables :=
            top_of_stack_tables.g[prod_class.lhs];
          push_stack(prod_match.lhs,current_tables);
```

```
    execute(prod_match.action_seq)
  end;
resolve s/r(prod_class) :
  begin
    prod_match := match(stack,prod_class);
    if undefined(prod_match) then
      begin
        push_stack(sym,current_tables);
        insymbol(sym);
        current_tables :=
          current_tables.g[class(sym)]
      end
    else
      begin
        for length(prod_class.rhs) steps do
          pop_stack;
        current_tables :=
          top_of_stack_tables.g[prod_class.lhs];
        push_stack(prod_match.lhs,current_tables);
        execute(prod_match.action_seq)
      end
  end;
resolve r(prod_class_1)/r(prod_class_2) :
  begin
    prod_match_1 := match(stack,prod_class_1);
    prod_match_2 := match(stack,prod_class_2);
    if undefined(prod_match_1)
      and undefined(prod_match_2) then
```

```
            exit(failure)
          else
            if undefined(prod_match_2) then
              begin
                for length(prod_class_1.rhs) steps do
                  pop_stack;
                current_tables :=
                  top_of_stack_tables.g[prod_class_1.lhs];
                push_stack(prod_match_1.lhs,current_tables);
                execute(prod_match_1.action_seq)
              end
            else
              begin
                for length(prod_class_2.rhs) steps do
                  pop_stack;
                current_tables :=
                  top_of_stack_tables.g[prod_class_2.lhs];
                push_stack(prod_match_2.lhs,current_tables);
                execute(prod_match_2.action_seq)
              end
        end
    end
  end
```

---

class(term_real.) is term, class(multop) is multop and class(fact_int.) is fact then the DTT production

(term_real. → term_real. multop fact_int.)

will result in the following context free rule.

term → term multop fact

To avoid ambiguities, no syntactic class can be a prefix of any other.

The choice of the function class can affect the effectiveness of the context free grammar in reflecting the structure of the DTT. If not enough characters are included then many DTT symbols of differing context may be allocated to the same syntactic class. Such redundancy hides structure that could be used in parsing. Conversely, taking the longest string possible may result in unnecessary distinctions being drawn between symbols that should have the same syntactic class. Just as a 'good' BNF definition will provide meaningful names for its symbols, a 'good' DTT will provide meaningful prefixes (syntactic classes) which will aid a human reader's understanding of the language, and be useful props for a mechanical interpretation.

Formally, let G = <V,N,S,p> be the context free grammar obtained from a DTT, D = <I,SS,T,O,v,p_O,L_I,L_O,s>, where

V = {class(t)|t is a symbol of D}

N = {class(lhs)|(lhs,rhs,q,w) is in P'}

S = class(s)

p = {(class(lhs),class(rhs))|(lhs,rhs,q,w) is in P'}

where s is the distinguished symbol of D, and P' is the set of all possible DTT productions,as defined in chapter three.

4.2.3.4 Conflict Resolution

4.2.3.4.1 Introduction

Applying a standard parse table generation algorithm to G may produce tables without conflicts, in which case they are adequate for driving the translator. However conflicts may be present in the parse tables, and these could indicate:

(1) an inappropriate choice of the syntactic classes obscuring the structure of the language,

(2) information contained in the symbol structure or dynamic nature of the DTT which could be used to resolve conflicts being unavailable to the table generator, or

(3) the structure of the DTT being such that tables adequate for parsing can not be constructed.

Unfortunately, the interpretation of a conflict is not always immediately apparent. If it does belong to (2) and we are sufficiently astute, we may be able to see a mechanism for resolving the conflict. However conflicts belonging to (1) and (3), and those of (2) for which a suitable resolution procedure can not be found require a restructuring of the context free grammar or of the DTT, if adequate driving tables are to be found. The absence (or existence) of conflicts for which resolution methods can not be found shows that a particular DTT does (or does not) belong to the subclass for which table driven translators

can be constructed. The class of languages for which table driven DTTs can be constructed therefore contains those languages for which at least one DTT that has adequate driving tables can be constructed. It is clear that as recognizers for this class of languages exist, they are contained in the set of recursive languages, however the exact characterization of the class is an open question. Hence it is possible that the structure of the language that the DTT describes is such that no adequate parse tables can constructed for it (i.e. belonging to the class of recursively enumerable languages, but not the recursive languages).

4.2.3.4.2 Categories of Resolution

4.2.3.4.2.1 Introduction

Part of the work of this thesis has been to study mechanisms for using information in the DTT to resolve conflicts in the parse tables. The techniques naturally fall into two categories:

(1) Those that resolve conflicts by considering the dynamic nature of the DTT.

(2) Those that resolve conflicts by considering the symbol (including template) structure of the DTT.

Of course, some conflicts may need consideration of both the dynamic and symbol structure of the DTT to be resolved.

In some cases the correct path to be taken may depend on the input seen and only be resolvable at translation time. However, for such resolution to be acceptable, the generator must be able to ensure that all occurrences of the conflict will be resolvable. In such cases the actions in conflict will be replaced by a new action, either resolve s/r or resolve r/r. The mechanism of these actions has already been presented.

4.2.3.4.2.2 Resolution using the Dynamic Structure

Conflicts in the parse tables produced by the syntactic structure of a pair of productions which can never coexist as active productions during a parse are examples of candidates of this category of resolution. Determining at generation time that two productions can not coexist is however not trivial. One approach is to extend an item set to include the configuration of active productions. The standard LR(1) table generation algorithm (or one of its variants) could be extended to keep track of added and deleted productions (and every production's action sequence), and whenever different parse paths could lead to different production sets, produce item sets for each parse configuration formed.

Any item like

$$[X \rightarrow . B, f] \qquad (1)$$

which has been added in the closure of an item such as

[A → J . X K, g] (2)

will mean that X → B will be recognized before A → J X K. Any changes the action field of (1) makes to the active productions will have to be passed on to the successor set on X of the item set containing (2). Other items like

[X → . Q, h] (3)

with different action sequences may also be added in closure of (2). As the recognition of the productions in (1) or (3) will result in different parse states, two versions of the successor set on X of the set containing (2) are needed to correspond to the two different configurations. The parse tables formed from this larger collection of item sets will not contain any conflicts of type (1) above.

The practicality of this approach is however in serious doubt. The LR(1) family of generators is known to be exceedingly greedy in its space and time requirements. This method further differentiates between parse states and forms even more item sets, which only exacerbates the problem.

4.2.3.4.2.3 Resolution using Symbol Structure

Conflicts in the parse tables produced by the syntactic structure of a pair of productions that have symbols whose structure is such that both productions can not simultaneously match any stack configuration is an example

of candidates for this category of resolution. To illustrate this table 4.6 contains DTT rules that yield a reduce 3 / reduce 4 conflict. Yet it is clear that productions 3 and 4 could never both be recognized, and so there is in fact no conflict.

Table 4.6

DTT Rules

(s → a)	(1)
(s → b)	(2)
(a → x_.)	(3)
(b → x.)	(4)

Syntactic Classes

s, a, b, x.

Table 4.7

DTT Rules

(s → a)	(1)
(s → b)	(2)
(a → x_A. y)	(3)
(b → x_.)	(4)

Syntactic Classes

s, a, b, x, y.

A second illustration is given in table 4.7 which will produce a shift / reduce 4 conflict. In this case it is the inability of a template to match the null string that ensure that there is no conflict. The inclusion of templates in a symbol makes the task of determining resolvability much more complex.

It has already been noted that an inappropriate choice of syntactic classes can result in unresolvable conflicts. As this is related to symbol structure, the example in table 4.8 to illustrate that extension of the syntactic class may resolve conflicts, has been included in this section.

Table 4.8

DTT Rules

(s → a)

(s → b)

(a → ident_a.) (1)

(b → ident_b.) (2)

Syntactic Classes

class(s) = s

class(a) = a

class(b) = b

class(ident_a.) = ident

class(ident_b.) = ident

Resulting Context Free Grammar

s → a

s → b

a → ident (a)

b → ident (b)

The parse tables associated with this grammar contain conflicts as the parser can not decide which of the two productions (a) or (b) has been recognized after the syntactic class ident is recognized. Extension of the syntactic class for ident to the entire symbol will resolve the conflicts of this example, however it is not considered a practical general method of resolution.

4.2.3.5 Error Recovery

The translator constructed above will be (if the parse tables do not contain unresolvable conflicts) a faithful implementation of the DTT. However, to be a practical tool, even for language designers,the system must have some form of error recovery built into it.

Being driven by a parser based on a context free strategy, error recovery for DTTs naturally falls into two categories.

(1) Context free errors which are detected by the driving routine when an error action is

encountered.

(2) Context sensitive errors that are detected by a mismatch of the production set with either the stack when a reduction is called, or an action production parameter when the action is executed during a reduction.

There is also a third category of errors, execution semantic errors such as values out of range and non-terminating programs, that are not handled explicitly by the DTT, but are only detected in relation to the semantics of the translation language when the translation is interpreted.

Context free error recovery in the shift-reduce parser that table driven DTTs are based on has been extensively studied (e.g. see reference list of [54]). The techniques that have been developed are all applicable to the recovery from context free errors in table driven DTTs, and can be easily accommodated.

Context sensitive information is stored in the trailing strings of DTT symbols. This may be tested explicitly during a reduction, or implicitly by using the recovery mechanism of failed actions. An explicit error is manifested by a mismatch of the stack with all of the active productions of the required syntactic structure during a reduction (note that a context free error occurs if the syntactic classes don't match). The skipping after an action executed as part of an action sequence fails, may be used to test the state of the system. If an error is detected, a vital production

rule may be deleted, with the effect of blocking a successful parse.

The implicit mechanism is impractical as a basis if an error recovery scheme because the point of recognition of an error (when the vital production is called in a reduction) is distant form the point of occurrence (and detection). A better approach would be to allow an action to report an error explicitly so that recovery could start at the point of detection.

A simple approach to the recovery mechanism for context sensitive errors manifested by a mismatch of the stack, is to revert to context free parsing until the effect of the error is eliminated. In this method, whenever a reduction can not be performed because a match can not be found, a general reduction according to the syntactic structure of the required rule is performed. If the left hand side symbol requires template substitution that can not be satisfied, it is flagged so that in any future matches its syntactic class only is used (but other stack symbols are treated fully) and symbols that reference the templates of the faulty symbol are also flagged. When all flagged symbols are popped from the stack, the effect of the error will have been completely eliminated from the parse (with the exception of any added productions).

4.3 An Implementation

4.3.1 organization

A table driven DTT generator system has been successfully implemented at the University of Adelaide on a VAX-11 system, and the implementation of this generator is now discussed.

The major components of the generator system (all written in Pascal) are

(1) the analyser,

(2) the table generator, and

(3) the translator.

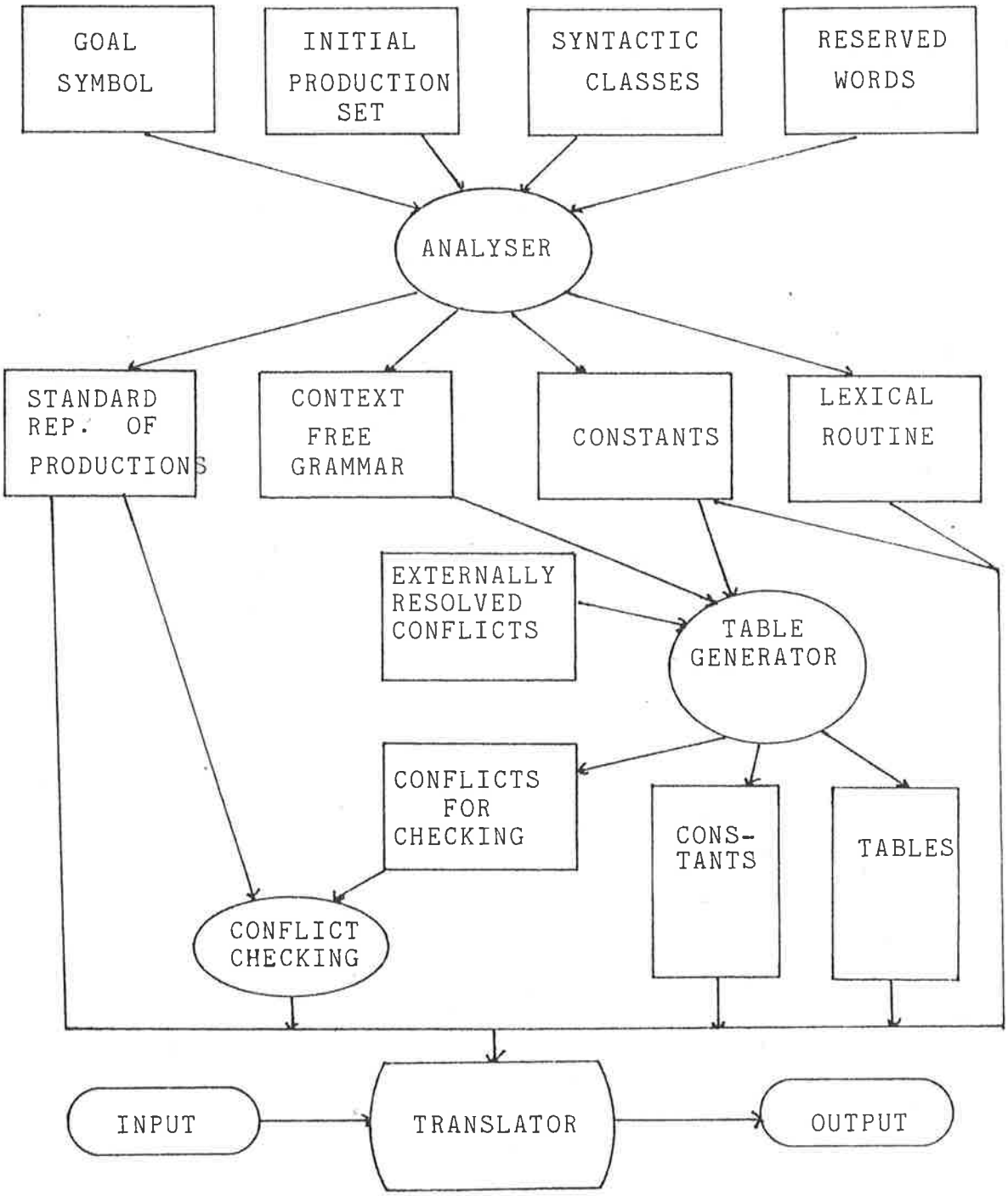
The relation between the various components is shown in diagram 4.9.

4.3.2 The Analyser

The analyser has as its input, the initial production set p_o, the goal symbol s and also files to indicate the syntactic classes (i.e. the function class) and instead of L_I, a list of reserved words for a lexical routine (which assumes the domain of programming languages and handles standard special symbols as well as the given reserved words).

The analyser has to produce four files:

Diagram 4.9 - Schematic of a Table Driven DTT Generator



(1) a context free grammar for the table generator,

(2) a lexical routine for the translator,

(3) a list of constants for the table generator and translator, and

(4) the initial production set for the translator.

4.3.3 The Table Generator

The tables generated are based on the LALR(1) tables, however they have been extended to include the resolve s/r and resolve r/r actions.

The method of resolving conflicts by considering the dynamic structure of a DTT is of dubious value and has not been implemented. Conflicts that are resolvable by considering the symbol structure of a DTT have been only implemented in a restricted form that is described below.

Consider the following DTT rules:

```
(x → start list end)
(list → a_X_.)                    (a)
(a_REM. → a_X_REM.)               (b)
```

Syntactic classes : x, start, list, end, a.

(This could be an extract from list processing rules of a DTT. In rule (b) X corresponds to the head item in the list, and REM to the remainder of the list, each item separated by underscores.)

Examining the rules we see that there is no stack configuration that could simultaneously satisfy both of rules (a) and (b) - since templates can't match null strings. However the collection of item sets would contain the following item sets (where g is in first(END f)).

[X→ . START LIST END , f] (*)

successor of (*) on START

[X→ START . LIST END , f]

[LIST→ . A , g] (**)

[A→ . A , g]

successor of (**) on A

[LIST→ A . , g] (1)

[A→ A . , g] (2)

Items (1) and (2) form a reduce/reduce conflict. Note that if DTT rules had initially been

(x→ start list)

(list→ a_X_. end)

(a_REM.→ a_X_REM.)

then a shift/reduce conflict would have occurred.

Clearly these conflicts can be resolved by the generator. Sufficient conditions are:

(1) one of the productions in the conflict is recursive, and

(2) that the first DTT symbol of all possible occurrences of both productions can not simultaneously match any stack string.

Checking the recursive nature of a rule, as required in criterion (1) is straightforward. However, criterion (2) can not be determined by the generator described above as it has access to the underlying context free grammar, but not the DTT productions. Although a fully integrated system would immediately consult the DTT productions, the simpler approach by this system is to note the conflict for subsequent checking, and generate the appropriate resolve action if criterion (1) is satisfied. When table generation is completed, the DTT productions are consulted to determine if the noted conflicts are satisfy criterion (2). No action is taken if they do, however if criterion (2) is not satisfied, the generation process is aborted and the tables that have been constructed erased.

Only this restricted case of conflicts using the symbol structure that has been implemented, as the effort of implementing further strategies does not seem worthy of the diminishing return.

The generator does have the facility to allow the user to specify that particular conflict pairs are resolvable at parse time. Although this is a compromise of the automatic nature of the system, there are resolvable conflicts, such as those due to the dynamic structure that is used to ensure uniqueness of declared identifiers, whose resolvability is

guaranteed by the structure (i.e. uniqueness) but is extremely difficult to prove formally. This facility is necessary to allow the user to point out that such conflicts are resolvable as a sufficiently powerful deductive system has not been produced.

4.3.4 The Translator

The translator is a straightforward implementation of the algorithm given in table 4.5 with syntactic classes represented as scalar types.

4.4 Summary

This chapter has shown the construction of table driven devices from DTT definitions. Further, the construction has been almost completely automatic. The points where user intervention is required are:

(1) Supplying the syntactic categories for the generator, and

(2) Specifying that particular conflicts (particularly those related to dynamic structure) are resolvable.

The first point is so fundamental that it could be considered as part of a tuple definition of a table driven DTT.

The second point maybe seen as a weakness of DTTs, as it can be expected that in programming language applications a user will need to take some action, as the complexity of a deductive system seems intractable. However the author

contends that for a well written DTT with clear syntactic classes, it is not difficult for one to decide the resolvability of conflicts.

Establishing the construction of table driven DTTs is important because it provides a mechanism for determining the behaviour of a DTT (i.e. can be used to answer questions about the language). In particular it defines the set of strings that are not in the language. It is this characteristic that is absent in most alternative proposals.

Chapter Five

APPLICATIONS OF DTTS

5.1 Introduction

This chapter discusses the application of DTTs to describe the languages Asple and Pascal.

5.2 Asple

5.2.1 Overview

Asple was designed [7] as a vehicle to demonstrate clearly the power of W-grammars. To meet this end, it has reasonable complexity without being too lengthy. Asple is therefore, a language which can be used as a metric of the usefulness of a definitional method. Indeed it has already been used to evaluate the merits of various definitional methods [37]. For this reason, Asple was chosen as the first major example of a DTT, and a comparison and evaluation of the DTT for Asple with alternative definitions for the language is given in chapter six.

To give the reader an overview of the language, an informal description of Asple follows. The definitive specification of Asple can however, be found in [7].

Asple is a simple Algol-like language. An Asple program consists of two parts; a declaration section and a statement section. All identifiers used in the statement section must be declared exactly once in the declaration section.

APPLICATIONS

A declaration associates a mode with an identifier. There are two primitive modes in Asple; integer and Boolean. Unsigned integers are denotations of mode integer. Boolean denotations are true and false. From the primitive modes an infinite number of modes may be constructed. These take the form integer, reference-to-integer, reference-to-reference-to-integer, reference-to-reference-to-reference-to-integer, etc., and similarly for Boolean. Values of the first three of these are commonly called constants, variables and pointers.

Integer expressions may be formed from integer values with the operators addition and multiplication. Boolean expressions may be formed from Boolean values with the operators and and or, or by comparing integer values with the operators eq and ne. Addition and or are both denoted by +, multiplication and and by *.

An assignment statement has a value on the right hand side which becomes the value referred to by the identifier on the left hand side (thus the right hand side mode must be that referenced by the mode of the left hand side).

Asple allows one kind of mode coercion called dereferencing. When, on the right hand side of an assignment statement, an identifier of mode reference-to-M is in a context requiring a value of mode M, it may be dereferenced to obtain the value the identifier references. Dereferencing may be applied several times if necessary to obtain a value of the required mode.

Asple has two transput statements, input and output, for communication between Asple program and external files. In [7], an Asple file is defined as a sequence of integral and Boolean constants separated by commas, however for our purposes the commas will be replaced by end of line markers.

The object for the translation is a well known intermediate code, p-code [42]. P-code was chosen as it is well documented, well defined by a machine implementation, and easy to use. A lower level object for the translation, such as machine code or assembly instruction was not chosen as they can be extremely difficult to read, and are not always convenient to use. A higher level object for the translation was not chosen because the question of its definition could be open. The possibility of choosing a non-machine based object is discussed in chapter six, however a code based translation was chosen to demonstrate the application of DTTs to compiler generation.

Full details of p-code may be found in [42], however an overview of the structure of a p-code program is follows. A p-code program consists of two assembly records; the first contains the program code, the second is a contains a jump to the start of the main program code. Each assembly record consists of a sequence of assembly instructions terminated by the identification line 'q'. These generally have three fields, an instruction mnemonic and two operand fields which may be symbolic. Instructions may define values for constants such as labels. Code segments also have every

tenth line identified by its number preceded by the letter 'i'.

5.2.2 Formal Description

The DTT for Asple is D_{asple} = <I,S,T,O,p_o,v,L_I,L_O,s> where

I = {begin, end, int, bool, ref, input, output, if, then, else, fi, while, do, od, true, false, :=, +, *, =, <>} + {all identifiers, such as count, x, sum, etc} + {0, 1, 2, ...}

S = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', '_', '^', '.'}

T = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z'}

O = S

p_o is shown in Appendix A. All lines (other than comments) are labelled for reference (labels are not part of the productions).

v = {beginsym, endsym, intsym, boolsym, inputsym, outputsym, ifsym, thensym, elsesym, fisym, whilesym, dosym, odsym, becomessym, plussym, starsym, eqsym, nesym, constant, ident}

s = goal

L_I = is tabulated in table 5.1

L_O maps any character other than '_' or '$' into itself. '_' is mapped into the blank character. A string of '$'s is mapped into the decimal representation of one less than the length of the string (so that zero may be represented). A line number identification is inserted every ten lines.

5.2.3 Discussion

5.2.3.1 Introduction

As the productions are the heart of a DTT, most of the discussion will be centred on them. The other components of the DTT for Asple are straightforward and will not be commented on in detail.

The first point to note about the production set of Appendix A is its small size (138 lines containing 62 production classes), which means that reading the DTT is not an awesome task. None the less, the action sequences of some productions are verbose, and a more compact notation is desirable. Although the use of a macro mechanism was considered, it was not found to be sufficiently general to justify the extension to the basic model. The second is that there is a degree of modularization of the productions, each module giving insight onto one aspect of the language. Although the modules are interrelated and do not stand alone, the DTT could not be criticized for being monolithic. The benefit of these two feature are in the ease of

comprehension of the DTT by a human reader.

It is the form of individual productions that determine the readability and hence the usefulness of a definition. The following discussion is broken into sections commenting on the productions in each of the main modules.

5.2.3.2 Modes and Declarations

The infinite number of modes in Asple is not a problem within itself. However, as will be discussed in the section on the assignment statement, in order to obtain conflict free parse tables, it is necessary to syntactically distinguish between identifiers that reference a primitive mode, and other identifiers. This causes some redundancy in the DTT, as rules describing a single aspect of the language must be repeated for both syntactic cases. The syntactic classes 'mode' and 'rmode' both correspond to modes, and 'legal' and 'rlegal' correspond to identifiers. Despite this redundancy, the treatment of modes (lines A61 to A64) is clear and cleanly handled by the DTT.

Similarly, the addition of a production to the system for each identifier declared is straightforward, however the volume of similar rules (lines A13 to A60) makes this section laborious. The action sequence of these productions can be thought of as procedural algorithms for checking the state of the system, and reporting an error by modifying the productions if the uniqueness criteria are violated. This process, while not at all complex, becomes quite cumbersome

because of the redundancy in the syntactic categories for modes causing duplication of production rules. This results in the description of what is a small part of the language taking up almost one third of the physical size of the definition.

Table 5.1 L_I for D_{asple}.

Token	L_I(Token)
begin	beginsym
end	endsym
int	intsym
bool	boolsym
ref	refsym
input	insym
output	outsym
if	ifsym
then	thensym
else	elsesym
fi	fisym
while	whilesym
do	dosym
od	odsym
true	constant_bool_true.
false	constant_bool_false.

:=	becomessym
+	plussym
*	starsym
=	eqsym
<>	nesym
identifier X	ident_X.
(e.g. count	ident_count.)
integer constant V	constant_int_V.
(e.g. 2	constant_int_2.)

The allocation of addresses to identifiers must be carried out during the processing of declarations. The incremental allocation of these addresses is handled as a 'side effect' of using the allocation symbol 'loc' (A130-A133). When, in situations like this, information stored in productions is manipulated by executing an action sequence, it should be thought of as being similar to invoking a procedure, rather than a surreptitious side effect of the parsing process.

5.2.3.3 Statements and Expressions

The best aspects of DTTs may be seen in the handling of statements and expressions as the productions succinctly embody the structure they describe. The productions (lines A65 to A109) are almost context free; the only extension is

the inclusion of the mode in expressions to enforce context sensitive restrictions. These productions (which take up about a quarter of the physical size of the DTT) very successfully convey to a reader the structure of Asple statements and expressions (which comprise the bulk of the language structure).

There are however two points that detract from the success of this part of the definition. The first is the need to break productions (such as those for iterative and conditional statements) whenever a label needs to be generated for the p-code. This clouds the structure of the constructs involved. This is a common characteristic of LR based systems as it is only permissible to perform an action when a structure (i.e. a production) has been recognized, but not during its recognition (as in LL systems). The second is that the strings 'i' and 'b' used to denote integer and Boolean modes could have more meaningful names, thereby aiding clarity. This approach has been used to show how expediency may compromise objectives. In this case the p-code instructions for loading values end with the first letter of the mode of the data type being moved. Thus a single rule with a template to match a single letter mode of the object and substitute it in the instruction generated, will adequately replace the iteration of cases that is otherwise needed (as for example in the case of multiplication whose mnemonic is 'mult' and and whose mnemonic is 'and').

5.2.3.4 Assignment

As a toy language used as the vehicle to demonstrate the power of W-grammars, Asple was designed to show how well a small fragment, centred on the assignment statement, of a language could be defined. The relationship between the modes of the source expression and destination variable can be stated simply: The mode of the source expression must be that referenced by the destination variable, where the source expression may be dereferenced if necessary.

For a DTT to follow this literally, requires the parsing of the source expression to be familiar with the mode of the destination so that appropriate dereferencing can be performed. This requires the formation of a conglomerate symbol containing the source and destination modes that has to be manipulated by several obscure productions with complex template matching and substitution. Not only does this make the section on assignment extremely turgid, but it also has the undesirable effect of distorting the form of the productions for expressions.

The source of the problem is in knowing when to dereference a value. The approach taken here is to divide assignment into those cases where the source value is of a primitive mode and may involve arithmetic expressions, and those where the mode of the source has a reference mode. To obtain conflict free parse tables, this necessitates a distinction on the syntactic level between references to primitive modes and other modes. The implementation of this

has already been discussed in section 5.2.3.2.

The productions for assignment therefore reflect the different syntactic cases. The rule to correspond to the conventional "variable := expression" assignment (lines A110,A111) is about as simple as this could be. The syntactic distinction of 'legal' or 'rlegal' for identifiers results in the need for rules (on lines A113 to A118) to tie the classes together, and given the redundancy, they do it quite well. However the remainder of the productions describing assignment between two identifiers of syntactic class 'rlegal' are not particularly easy to follow, as they involve processing of conglomerate symbols (although not as obscure as the above mentioned case). The rules involving the syntactic class 'refass' check that the source mode is compatible with the destination. The rules involving the syntactic class 'assderef' dereference the source if appropriate.

Assignment is a relatively simple concept and should therefore be defined by simple productions. In Asple, the infinite number of modes coupled with automatic dereferencing complicate the issue and result in the DTT specification of Asple lacking clarity at this point. This could be flaw in DTTs, or it could be the design of Asple that is at fault, as it can be argued that a good model for a language definition will highlight weaknesses in a language. This is a significant point and will be raised again in chapter six.

5.2.3.5 Allocation of space and labels

The rules for allocating space and labels are quite straightforward (the side effect characteristic has already been discussed in section 5.2.3.2). The only comment needed is that arithmetic is difficult within the DTT (although arithmetic could be defined in a string form, it is not helpful and certainly not useful to do so; a further comment on this will be made in relation to W-grammars in chapter six). Numeric quantities such as addresses and lengths of objects are best represented in a DTT by the length of a string of special characters that is converted into its decimal representation by the lexical output function L_0. For example, the output string 1_$$$$$$$$_ maps into 1 7.

5.2.4 Examples

To illustrate the DTT for Asple, examples 5.1 and 5.2 (after [37]) are presented.

Example 5.1 - The Factorial Function.

```
begin
  int X, Y, Z;
  input X;
  Y := 1;
  Z := 1;
  if (X <> 0) then
    while (Z <> X) do
      Z := Z + 1;
```

```
      Y := Y * Z
    od
  fi;
  output Y
end
```

Parsing of the second line of this Asple program causes the productions

```
(legal_$$$$$$$$$_Z_ref_i. → ident_Z.)
(legal_$$$$$$$$$$_Y_ref_i. → ident_Y.)
(legal_$$$$$$$$$$$_X_ref_i. → ident_X.)
```

to be added to the system so that subsequent occurrences of these identifiers may be recognized. For example, the reference to Z in line nine results in the top stack symbol, ident_Z., being reduced to legal_$$$$$$$$$_Z_ref_int.. The next reduction (using the production on line A92 of the DTT) will be to fact_int. which also generates p-code to load the appropriate location at runtime.

Example 5.2 - Reference Modes and Dereferencing

```
begin
  int INTA, INTB;
  ref int REFINTA, REFINTB;
  ref ref int REFREFINTA, REFREFINTB;
  INTA := 100;
  INTB := 200;
  REFINTA := INTA;
  REFINTB := INTB;
```

```
  REFREFINTA := REFINTA;
  REFINTA := INTB;
  INTB := REFINTA;
  input REFREFINTA;
  output REFINTB
end
```

This program illustrates reference modes and dereferencing in the assignment statement of Asple. The parsing of lines two to four adds the following productions to the system.

```
(legal_$$$$$$$$$$_INTA_ref_i. → ident_INTA.)
(legal_$$$$$$$$$_INTB_ref_i. → ident_INTB.)
(rlegal_$$$$$$$$$$$_REFINTA_ref_ref_i. → ident_REFINTA.)
(rlegal_$$$$$$$$$$_REFINTB_ref_ref_i. → ident_REFINTB.)
(rlegal_$$$$$$$$$$$$$_REFREFINTA_ref_ref_ref_i.
  → ident_REFREFINTA.)
(rlegal_$$$$$$$$$$$$_REFREFINTB_ref_ref_ref_i.
  → ident_REFREFINTB.)
```

The assignments on lines five, six and eleven of the program are recognized by the production on lines A110 of the DTT. In the case of the line eleven assignment, the source expression will be dereferenced by the productions on line A96 and line A98. The assignments on lines seven, eight and ten are handled by the production on line A113. In processing the assignment on line nine, the production on lines A123 is applied to produce the symbol

```
refass_$$$$$$$$$$$$$_ref_ref_ref_i_^$$$$$$$$$$$_ref_ref_i_.
```

The production on line A125 would be applied twice to obtain the symbol

refass_$$$$$$$$$$$$$_ref_i_^$$$$$$$$$$$$_i_.

The production on line A121 is applied (note that it could not have been applied previously, and that the production on line A125 can not now be applied) to yield

assderref_$$$$$$$$$$$$$^i_.

As the production on line A119 is not applicable, no dereferencing is to be performed. The rule on line A112 is however applicable, and can be used to recognize the assignment.

5.2.5 Table Driven Asple Translation

D_{asple} has been processed by the generator described in chapter 4 using syntactic classes given in table 5.2. The generator produced 99 tables (the item set collection consisted of 99 sets containing 2789 items) that contained no unresolvable conflicts.

118 instances of conflict were able to be resolved. 41 of these were due to the pair of context free productions

LEGAL → IDENT

RLEGAL → IDENT

As the dynamic nature of the DTT ensures the uniqueness of declared identifiers, this category of conflict will always

be resolvable at parse time, and hence these conflicts were

Table 5.2 Syntactic classes for the table driven D_{asple}.

goal beginsym dectrain semicolonsym strain endsym whst ifst ifthst sym lab addr loc declist decc mode idlist whsym ident commasym legal dref rdecc intsym rmode boolsym rlegal refsym stt assignment inn conditional iteration transput fact beginpart lparensym exp rparensym count space relop endpart eqsym nesym refass constant assref assderef term starsym plussym ifsym thensym elsesym fisym whilesym dosym odsym outsym insym becomessym

---

resolved externally to the generator. The remaining conflicts were due to the following pairs of context free productions

FACT → DREF

DREF → DREF

TRANSPUT → INN

INN → INN

ASSIGNMENT → ASSDERREF

ASSDERREF → ASSDERREF

ASSDERREF → REFASS

REFASS → REFASS

DECLIST → DECC ADDR

DECC → DECC ADDR

DECLIST → RDECC ADDR

RDECC → RDECC ADDR

Of the two criteria of section 4.3.3, each of the above satisfy criterion (1) for resolution, and by consultation of the corresponding DTT productions, criterion (2) is also shown to be satisfied. Hence these conflicts are all resolvable at parse time.

5.2.6 Summary

The discussion of D_{asple} given above has aimed to demonstrate how the DTT works, and to thereby highlight aspects of the definition. These will be discussed further in chapter six.

5.3 Pascal

5.3.1 Introduction

The most complex syntactic structures found in programming languages are those that incorporate a name scoping mechanism whereby inner declarations temporarily override outer ones. The difficulty this poses can be seen in the parsing of field identifiers of record structures in Pascal for example, where substantial changes to the parsing environment (altering the set of legal identifiers and

attributes to be associated with them) occur in the transition of the 'spot' symbol.

Structures containing scoping mechanisms are most naturally described by a stack mechanism. However as DTTs do not provide stack facilities, the description of this sort of structure is a major test of their ability to define structures that can not be given a natural internal representation, and is illustrated below.

Another example of a language feature without a corresponding structure in a DTT is in language restrictions involving subrange checking. If p_code (which does not contain any representation of arithmetic concepts) is the target language, this requires the DTT to contain production rules to associate arithmetical properties with the digit characters. However, such productions have features that are difficult to reconcile with the goals of chapter one. The definition of arithmetical concepts within DTT productions is pointless, as well as being cumbersome (resulting in the difficulty mentioned above), and is therefore an inappropriate use of a DTT.

This problem is further analysed later in the thesis, however rather than present an inappropriate DTT for complete Pascal, DTT productions for a subset of Pascal including record types and block structure may be found in Appendix B. Other formal components of the definition of the DTT are straightforward, and they have not been included. The form of these productions is discussed in section 5.3.2.

For compactness, features of Pascal that could be defined in such a DTT but do not illustrate any additional features of a DTT have been left out, however they have been described in section 5.3.3.

5.3.2 The Productions

5.3.2.1 Introduction

The discussion of the form of the productions in Appendix B is broken into sections corresponding to the major language components.

5.3.2.2 Statements

Statements (P458 to P538) are very well described in the DTT. Note that the dangling 'else' problem has been overcome by structuring the productions (e.g. P481,P482) so that the 'if-then' without an 'else' structure is not permissible as the 'then' component of an 'if-then-else' construct. In the processing of case statements, rules of syntactic class triple (e.g.P524) are added and tested (e.g. P521) to ensure uniqueness of case labels. Assignment is also easily handled. Note the distinction between assigning a scalar value (P485,P486) and assigning a variable of a named structured type (P487,P538).

5.3.2.3 Expressions

The structure of the productions (P539-P579) for expressions is very clear. Mixed mode arithmetic is treated

particularly cleanly.

5.3.2.4 Labels

The syntactic nature of labels is well stated in the DTT (P009-P032). However restrictions involving jumps into structured statements are difficult to specify. This is a well known difficulty with Pascal.

5.3.2.5 Record Types

As the production rules (P580-P709) to describe record types are lengthy, a commentary of their structure is provided.

The substantial changes to the parsing environment that occur during the recognition of a record field identifier are processed by a series of unit reductions. The sequence of symbols that would appear on the top of the parse stack during the recognition of a integer field named freq in a record structure named stats are shown below (except that spotsym and ident_freq. have been shifted onto the stack in line (4), the reduction was from recname to recid).

```
name_stats.                          (1)
legal_rec.                           (2)
recname                              (3)
recid spotsym ident_freq.            (4)
recfld_int.                          (5)
legal_int.                           (6)
```

The chain of unit reductions is used as a vehicle to make the necessary changes to the productions to model the changing parsing environment. For each field identifier, a rule of the form (4) to (5) is added by the reduction (3) to (4), and each of these added productions is deleted by the reduction (5) to (6). The rules to perform (3) to (4) and (5) to (6) are added by the reduction (2) to (3). The production defining the reduction (1) to (2) is added during the variable declaration, and contains an action to add the rule for (2) to (3) that contains appropriate action sequences. All transient productions are deleted when their function has passed, so not to interfere with the parse of future record structures.

The processing of the record type declaration initially adds rules to allow the (3) to (4) (e.g. P590) and (5) to (6) (e.g. P595) transitions above, and then for each field identifier adds actions (e.g. P638-P640) to the (3) to (4) production to add, and (e.g.P641-P643) to the (5) to (6) production to delete, appropriate productions recognizing field identifiers (of type (4) to (5) above (e.g. P640-P643)). On completion of the record declaration (P603-P611), a rule is added that allows (2) to (3) above (P604,P605) and includes actions adding copies of the constructed (3) to (4) and (5) to (6) (P606,P610) rules which are deleted (P607,P611). For each variable declared to be of the record type a rule of the (1) to (2) form (e.g. P263) is added that contains an action to add the (2) to (3) rule just constructed, and that is deleted after the

variable list is processed.

Uniqueness of field identifiers is ensured by adding a rule of the syntactic class single (P635) for each field identifier. Subsequent declarations check (P632) for existing rules of this form (indicating a uniqueness violation) and inhibit (P633) a successful parse if found.

To cope with nested records, a stack structure must be incorporated into the DTT rules. On entry to a nested structure (P581-P601), all existing rules are saved (to be restored) in the action sequence of the exit production of the inner declaration, and appropriate productions initialized for the inner declaration. The depth of nesting is maintained in the production (depth_D. ->).

The complexity of the syntactic structure of records results in the productions for this component of the language containing long and highly structured action sequences, and having many nontrivial interactions with other productions. The result is that these productions do not have the transparent clarity that was a feature of the simpler structure such as expressions and statements. None the less, record types are adequately described.

5.3.2.6 Block Structure

As with record types, it is the effect of local declarations temporarily overriding global ones requiring a stack mechanism that complicates the form of the productions. This mechanism is similar to that already

discussed for nested records, and will not be discussed in detail. On block entry all conflicting entries are saved in the action sequence of (block -> labpart constpart typepart varpart pfpart stpart) for latter restoration. Block structure is adequately described by these productions.

5.3.3 Other Features of Pascal

5.3.3.1 Introduction

The problems of the semantic aspects of a Pascal to p-code DTT do not detract from the description of the syntactic features of Pascal by a DTT, which contains clarity similar to that demonstrated for Asple. The features of such a DTT that were not included in the example of Appendix B because they do not illustrate any additional aspects of a DTT, are described below.

5.3.3.2 Code Generation

With the few exceptions referred to above, code generation for Pascal is very easily implemented; statements and expressions in particular are very cleanly handled.

5.3.3.3 Enumerated Types

Except for anonymous enumerated types (those where the type declaration occurs at the point of the variable declaration), there is no problem in defining enumerated types in DTTs. The dynamic nature specifically caters for information like this that is encountered during a parse.

Anonymous enumerated type are difficult as it is not easy to integrate the names of the constants of the type into the DTT, however this can be overcome with some effort. The utility of anonymous enumerated types has also been criticized in [45].

5.3.3.4 File Types

There is no inherent difficulty in implementing file types within a DTT. The need to prohibit file of file structures, and the requirement that all file program parameters are used in a 'file of' declaration results in some syntactically similar rules having slightly differing action sequences. The effect of this is to increase the physical length of the DTT.

5.3.3.5 Array Types

Arrays can easily be implemented. The equivalence of the form of a multidimensional array defined with dimensions separated by commas and the array of array form can be trivially accommodated.

5.3.3.6 Forward Declared Procedures and Functions

As a table driven parser can not be directed to perform a reduction that does not progress towards the accept state, checking for parameter lists processed in a forward declaration must be done by an action sequence, which is cumbersome. Changing the language design to incorporate parameters on the actual declaration only creates a

consistency checking problem. A different keyword on the actual definition would overcome the parsing ambiguity.

5.3.3.7 Constant Declarations

In a manner similar to type declarations, constant declarations can easily be accommodated.

5.3.3.8 Parameters

Processing parameters is not generally difficult, however subtlety is needed in the distinction of value and variable parameters to avoid a parse table conflict in the processing of a variable actual parameter in a parameter list (i.e. should a reduction to factor (for a value formal parameter) be performed, or should the the variable be passed as is (for a variable formal parameter). This difficulty may reflect on the poor distinction between variables and values in Pascal.

Procedure and function valued parameters are very difficult to implement as there is insufficient information about their parameters available.

5.3.3.9 Pointer Types

In most other cases, the symbol representing a type contains a complete elaboration of the type, even when the type has been defined by a pseudonym. This is not possible in the case of forward declared pointer types, and requires a production rule to be added to replace a forward declaration by its type (which may be a pseudonym) when

used. For consistency it is desirable for all pointers to be treated in the same manner, requiring the generation of a tag name for anonymous pointer types.

5.3.4 Table Driven Pascal Parsing

D_{pascal} has been processed by the generator described in chapter 4. The generator produced 268 tables (268 sets containing 14549 items) that contained no unresolvable conflicts.

There were 280 conflicts that were resolved. The vast majority of these were due to the following pairs of context free productions:

LEGAL → NAME
FUNCC → NAME

LEGAL → NAME
DEST → NAME

The scoping of identifier names is handled by the dynamic nature of the DTT (by hiding overridden declarations), and provides that these conflicts will always be resolvable at parse time. Thus the resolvability of these conflicts is determined externally to the generator.

Conflicts were also due to the rules:

FACT → LEGAL
NEXP → LEGAL

The string structure of the symbols of this DTT has been constructed so that it is always possible to distinguish between identifiers declared to be of a named type and those of an anonymous type, thus allowing these conflicts are resolvable at parse time. The resolvability of these conflicts was determined externally to the generator.

The remaining conflicts were due to the following pairs of rules:

VARDEC → VDEC
VDEC → VDEC

FIELD → RLIST
RLIST → RLIST

The criteria of section 4.3.3 are satisfied in these cases, and hence the generator determines that the conflicts are resolvable at parse time.

5.3.5 Conclusion

The specification of a translation from Pascal to p-code involves structures that have no natural representation within a DTT. Structures involving syntactic specifications, such as stack mechanisms, can be conveniently be synthesized within the DTT model. However, as those that involve semantic structures, such as restrictions requiring arithmetic concepts by the recognizer, can not be given an appropriate representation within the DTT model, a satisfactory DTT translating Pascal into p-code can not be constructed.

There is a point here that needs amplifying: a DTT is a syntactic device that defines the semantics of a language by a translation into a target language. Hence it is not appropriate to define semantic concepts such as arithmetic within the production rules. This raises the question of role of the target language, which although treated in chapter six, is mentioned here in relation to coded translations. Some problems (e.g. addressing fields of nested records) may be avoided by using a more powerful (yet realistic) coded target language. However others, such as those involving range checking would still be inappropriately expressed in the production structure of the DTT. Possible extensions to the structure of the DTT model, may be envisaged to invest the necessary information for these decisions in additional actions, thereby extending processing power of action sequences. Although considered, this approach was rejected as an 'ad hoc' extension that would need to be adapted to particular applications.

5.4 Summary

The DTT for Asple has shown that DTTs can successfully be used as a basis to define a translation. The DTT for the subset of Pascal has shown that DTTs are capable of describing the most complex syntactic structures found in programming languages.

P-code was chosen as the target language for Asple as a straightforward means of demonstrating the applicability of DTTs to compiler generation. Defects with the coded translation were noted in the fragmentation of the productions for Asple statements, and also in the appropriateness of a Pascal to p-code DTT. The role of a coded target language will be examined further in chapter six.

Chapter Six

EVALUATION

6.1 Introduction

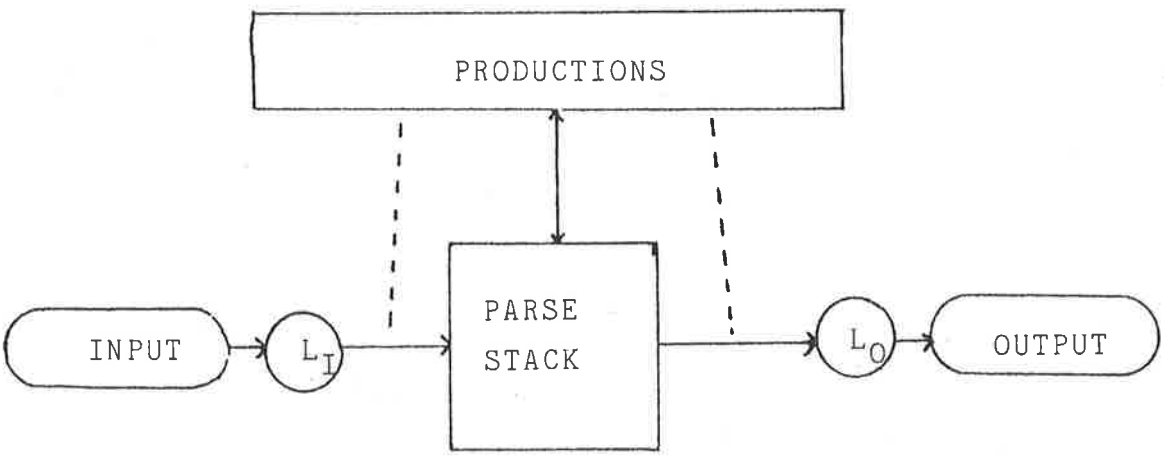
This chapter investigates several aspects of DTTs as a model for the definition of programming languages, and compares similar treatments in other models. The discussion is presented in sections considering: control structures, symbol management, information representations, descriptive properties, language design, and the choice of target languages.

6.2 Control Structure

Diagram 6.1 shows that control of all active components in a DTT is vested in the productions alone. The

---

Diagram 6.1 - Control Structure of a DTT

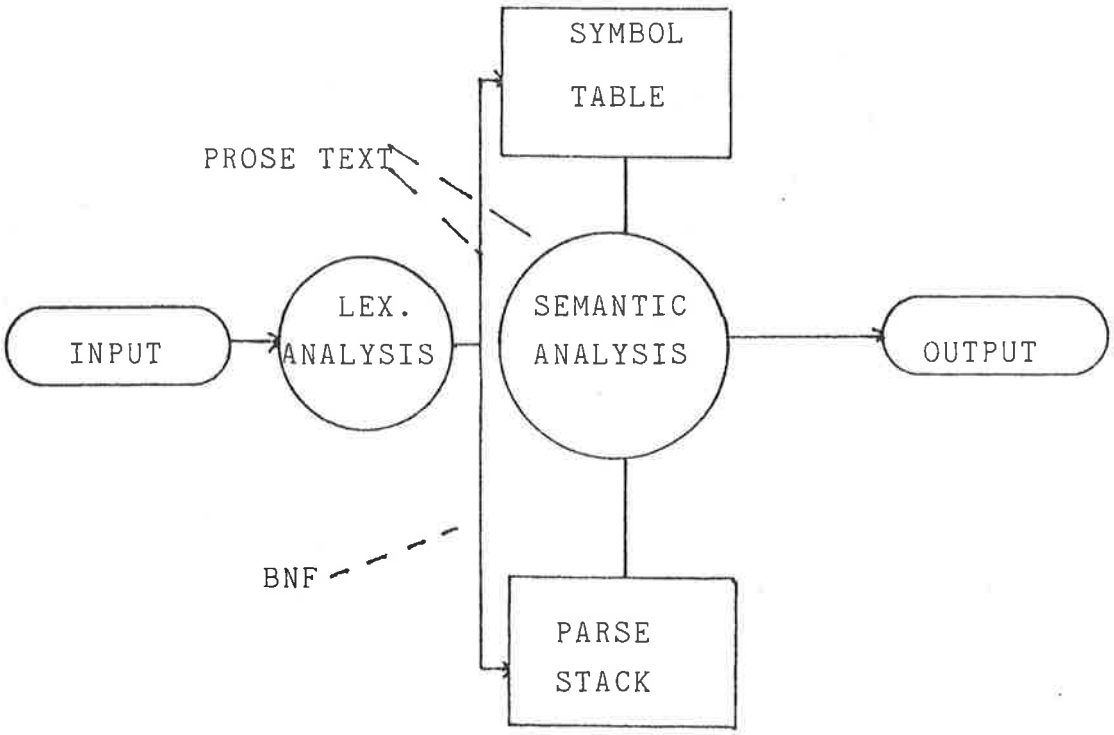


---

productions specify the changes that may occur during a reduction to the parse stack, which also defines the changes to the production set. Productions also implicitly control the flow of tokens through the (passive) lexical functions L_I and L_O from input to output. The salient features of this centralized control are its symmetry and its simplicity, and are reflected in the perspicuity of DTT definitions.

It is interesting to contrast the above structure with that of a compiler constructed from a conventional BNF plus prose definition.

Diagram 6.2 - Control Structure of a Conventional Compiler



In such devices the BNF describes the lexical nature of tokens, defines the structure of the parse tree and controls the changes that occur on the parse stack. The prose section defines the context sensitive syntax and code generation, and thereby controls the symbol table management scheme implementing these requirements as side effects of the context free driving mechanism.

The complicated and compound nature of this approach is apparent from the diagram. Neither the control of the device nor the specification of the control is centralized. Indeed, this can be seen as a classic example of coroutines [38]. This dislocation of control is a consequence of the perturbing effect that forcing a symbol table management scheme onto a context free basis has on the entire system. The contrast that has been shown between the unity of the control of a DTT and the heteroarchical control of the BNF plus prose models results in greatly differing mechanisms for the manipulation and storage of information within these systems. The variety of mechanisms used in these, and several other prominent proposals that were mentioned in chapter two will be investigated in sections 6.3 and 6.4.

6.3 Symbol Management

6.3.1 Introduction

As Ledgard has pointed out [30], the most difficult syntactic aspect of programming languages is the restrictions on the use of identifier names. Any model that

addresses this issue must, in one form or another, have a symbol table, and the question of implementing syntactic restrictions then resides in the management of the symbol table. In this section, the structures that correspond to a symbol table and to its management are identified for the most prominent models that have been proposed. In section 6.4 these structures are examined to find any common underlying characteristics.

6.3.2 DTTs

As DTTs have only been introduced in this thesis, the role of productions and action sequences will be discussed before identifying structures corresponding to a symbol table and its management.

A DTT production encapsulates a piece of information about the language. Recognition of a production corresponds to recognition of the fact (or structure) of the production by the system.

The role of action sequences in DTT productions is to provide a procedural algorithm to process the piece of information that has been recognized. The flow of control structure for actions was designed to be weak; only strong enough to allow minimal processing power. Although stronger control could be provided, the nature of DTTs would be substantially be altered if the role of knowledge recognition was transferred from the mechanism of recognition of productions to the execution of high power

action sequences. An extreme example of this would be a system that reads a program into a single conglomerate stack symbol which is then processed entirely by an action sequence.

The string nature of DTT symbols may be used for storage of structured information (whereas in a context free system a symbol can only represent a scalar). Thus a dynamically created production may be used to store information for future reference. If the production is used in a reduction then the information it contains maybe explicitly manipulated, however, information that is contained in productions may also be tested during execution of action sequences, although not explicitly manipulated.

Dynamically created productions can therefore take the role of a symbol table in a DTT, and the action sequences of productions can define the management of a such a symbol table. This has already been illustrated in chapter five where the actions of the DTTs for Asple and Pascal explicitly showed the mechanism used to process names encountered in a program by adding a production for each identifier declared, and testing existing productions to enforce constraints such as uniqueness.

6.3.3 BNF plus prose

The structure of devices developed from this category of models has been outlined in section 6.2. However the concepts of a symbol table and its manipulation are not

explicitly discussed in these definitions. In these definitions the language restrictions are described in general terms (e.g. 'all labels declared in a block must be distinct'), leaving the detail of such constraints up to the implementor.

6.3.4 W-grammars

W-grammars, like DTTs, are string based. However the power of the W-grammar meta-notion enables highly structured information to be stored within a symbol. Using this approach, the W-grammar compacts the complete symbol table into every symbol that needs to consult it for ensuring context sensitive restrictions. These restrictions are enforced by a sequence of derivations (specified by rules of the W-grammar) that only yields a finite subtree (a W-grammar criteria for a legitimate derivation) if the restrictions are satisfied.

W-grammars therefore represent the symbol table in string form in grammar symbols and define its management by rules of the W-grammar.

6.3.5 Production Systems

The concept of an environment (a global list) in Production Systems contains all information that would be found in a symbol table structure. Functions whose domains are environments are defined to specify the context sensitive restrictions and it is the use of these functions in productions that is equivalent to the management of a

symbol table.

6.3.6 Dynamic Grammar Forms

In this approach productions mapping syntactic classes into identifiers are dynamically managed by the interpretation of the grammar form according to the parsing environment. The context free rules added correspond to the symbol table and the specification of the dynamic mechanism of the interpretation function corresponds to the symbol table management.

6.3.7 Dynamic Production Grammars

In this approach productions mapping syntactic classes into identifiers are dynamically controlled by guarded commands (which may reference global data structures such as lists) within productions. The context free rules added correspond to the symbol table, and the guarded commands to the symbol table management.

6.3.8 Attribute Grammars

Although the original proposal [26] was directed towards semantics for context free grammars, various extensions have proposed systems that enforce constraints on the values to attributes (which may be lists). In such systems it is an attribute (list) that corresponds to the symbol table, and the specification of the constraints and attribute evaluation rules that correspond to the symbol table management.

6.3.9 Notation for Static Semantics

The concept of a symbol table is clearly embodied in the structures (such as stacks, strings and variables) that may be manipulated by the actions associated with productions, which are equivalent to the management of the symbol table. As the structure of this model is very similar to that of the devices of section 6.3.3, this technique can be seen as a formalization of the aspects of the definitions of section 6.3.3 that were left to the implementor.

6.3.10 Conclusion

The structure corresponding to a symbol table, and those that define its management in the most prominent models have been identified. Only DTTs and W-grammars were able to bring these aspects within a single conceptual framework; all the other proposals were compound in nature, usually defining structures auxiliary to the production basis.

It is worth noting that the models for language semantics were not considered in this section because they usually assume a syntactically valid program, and therefore do not address questions relating to restrictions involving identifier names.

6.4 Information Representation

6.4.1 Introduction

It is interesting to further examine the basic techniques used by the models discussed in section 6.3, to represent non context free information that is processed during a parse. Insight may be obtained into the structure of not only definitional methods, but also of programming languages, if any underlying similarities or common features can be found.

An investigation of the structures used for the symbol tables of the models discussed in section 6.3 shows that there are two basic approaches:

(1) to tag such information onto the existing context free structure (discussed in section 6.4.2), and

(2) to form an additional structure specifically to contain such information (discussed in section 6.4.3).

The mechanisms used for the management of these structures may similarly be studied, and the above dichotomy is also apparent (and is discussed in section 6.4.4).

6.4.2 Tagging

Tagging mechanisms generalize the scalar value of a context free symbol so that structured information may be represented. W-grammars and DTTs replace the scalar symbol

by a string. The approach by Attribute Grammars is to replace the scalar symbol by a record structure containing specifically typed data items.

The string mechanism is the more concise as it does not require the additional evaluation rules of Attribute Grammars. However a string representation is not always the most appropriate as it can be difficult to manipulate if unnatural structures are used. Arithmetic in W-grammars, for example, illustrates how cumbersome and complex a definition may become if unnatural structures are used. A general point may be drawn from this: that the utility of a system suffers if it uses unnatural structures for the objects it describes. This is particularly pertinent to W-grammars as they use strings for all structures, including those such as arithmetic which are inappropriate in string form. This problem does not arise in Attribute Grammars as the attributes may be defined to have natural structures for the data they contain. In DTTs, as shown in the examples of chapter five, the string nature of DTT symbols was generally used only to represent identifier names. As strings are the appropriate structure for identifier names, no awkwardness results in this aspect of the DTT. In the case of Pascal code generation, the need to use such structures was seen as inappropriate.

6.4.3 Adding

Structures created to hold information encountered during a parse are best divided into two categories:

(1) information added in the form of extra production rules, and

(2) information contained in auxiliary structures.

DTTs, Dynamic Production Grammars and Dynamic Grammar Forms all add production rules to the system to contain information encountered during a parse.

The Notation for Static Semantics uses auxiliary data structures (such as stacks) to contain such information. The environment concept of Production Systems is another example of an auxiliary structure.

6.4.4 Management Control

The representation of information needed to manage the symbol tables described above may also be categorized in terms of tagging existing structures and of creating additional structures. However, whereas tagging in section 6.4.2 was to extend the symbol structure, in this case it is associated with the control structure. In DTTs, Dynamic Production Grammars, Notation for Static Semantics and Attribute Grammars, actions or evaluation rules are tagged onto production rules.

Additional structures were created in some models, and also relate to the control of the device. In Dynamic Grammar Forms the specification of the interpretation function includes structures that dynamically manage the productions. In Production Systems, the additional structures are in the form of functions whose domains are environments.

6.4.5 Complexity

The complexity of a definition can, in some cases, be traced to the structures a model provides for the expression of detail, rather than to intrinsic properties of the language. Such complexity can be broken into two categories, that which is due to the complexity of the control structure, and that due to the structures from which the definition is built.

In models such as W-grammars, the control complexity is exacerbated by using a universal tagged structure, strings, to express detail such as arithmetic inappropriately.

Complexity in a definition adds to the difficulty a user encounters in initially understanding how the device works, and in the difficulty of using it, and therefore detracts from its utility.

6.4.6 Conclusion

Two methods of representing non context free information processed in a parse have been identified; tagging of existing structures and the adding of new ones.

Of the models examined in section 6.3, DTTs were the only model to exhibit both mechanisms in the representation of the symbol table. This can be seen to give DTTs a greater flexibility in the choice of structures available for the representation of information.

These two methods were also apparent in the methods used to represent information that managed the symbol table.

6.5 Descriptive Properties

6.5.1 Introduction

The most common metric that has been used to characterize the achievements of formal models in past work has been measures of the scope of applicability of the model. Aho has pointed out [2], 'the class of languages suitable for for completely representing programming languages undoubtedly lies within the class of context sensitive languages'. However, the scope of applicability is not an appropriate measure of the benefit of a definitional method as even the exact characterization of this class by a formal model will not guarantee that the model has any any other useful properties. The descriptive properties of a model, i.e. its ability to describe the language in a manner that enables the user to obtain an understanding of the concepts involved, are closely related to the human aspects mentioned in chapter one.

Objective evaluations of the descriptive properties of a definitional model are difficult to formulate as there are no numerical measures of the benefits in such a model. Any evaluation must therefore be subjective. Given this, it is desirable to base such an evaluation on neutral criteria. In a survey paper [37] a comparison of the several different formal definitions of Asple are presented and evaluated. The criteria used cover a wide range of characteristics and were designed to be fair; not highlighting the good points of one model to the exclusion of those of another. These criteria,

detached from DTTs, provide an unbiased means of assessment of DTTs.

6.5.2 Evaluating Definitional Models

6.5.2.1 Introduction

The criteria to be used for assessing a model are: completeness, simplicity, clarity of the defined syntax, clarity of the defined semantics, ability to show detail, ability to show errors, and ease of modification. The survey cited above assessed W-grammars, Production Systems, Vienna Definition Language and Attribute Grammars, and presented its results in tabular form. Reproduced in table 6.3 is the summary table of the survey with an additional column for DTTs. Brief comments about each of the categories of the evaluation follow.

6.5.2.2 Completeness

This is the ability of the formal system to define the entire programming language. A DTT defines the semantics of a programming language by a translation into a language whose semantics are well defined. Thus a DTT definition is relative rather than absolute. In this sense a DTT is not trying to completely specify a language. The realm of DTTs is the syntactic specification of a language coupled a with means of a translation into another form which specifies the semantics.

6.5.2.3 Simplicity of the model

This criterion assesses the initial difficulty that a user encounters in understanding the model. DTTs structure has been shown to be simpler than that of other models. This is undoubtedly due to its closeness to its context free basis.

6.5.2.4 Clarity of defined syntax

This category includes context sensitive syntax. DTTs strongest points are in the clear manner in which syntax is completely defined.

6.5.2.5 Clarity of defined semantics

This describes the ability of the model to specify the meaning of a program. DTTs do not directly address this question as they presuppose a target language whose semantics are well understood. While this could be seen as a weakness of the model, it could also be seen as a strength, as it does not introduce esoteric concepts and notations which hinder the users' comprehension. Instead it gives a mechanism, the translation language, for the specification of the semantics in the most appropriate form for the user.

Table 6.3 Evaluation of several Definitional Methods

	W-GRAMMARS	PRODUCTION SYSTEMS	AXIOMATIC APPROACH	VIENNA DEFINITION LANGUAGE	ATTRIBUTE GRAMMARS	DTT
COMPLETENESS	+	NA	NA	+	–	NA
SIMPLICITY OF MODEL	+	+	0	–	0	+
CLARITY OF DEFINED SYNTAX	–	+	NA	0	+	+
CLARITY OF DEFINED SEMANTICS	0	NA	0	0	0	NA
ABILITY TO SHOW ERRORS IN PROGRAMS	–	+	0	+	0	+
ABILITY TO SHOW DETAILS	0	+	0	+	0	+
EASE OF MODIFICATION	0	0	0	0	0	0

RATINGS:
+ Positive
0 Neutral
– Negative
NA Not Applicable

6.5.2.6 Ability to show errors

The ability to construct a recognizing device from a DTT description is a very strong point in their favour, as it is a characteristic not shared by other proposals. As questions about program legality would be the most common a user would ask, the ability to mechanically show errors must be seen as an advantage of DTTs.

6.5.2.7 Ability to show detail

This criterion measures the ease with which a user may obtain answers to questions about the language. To highlight the importance of this facet of a definition, a set of questions about Asple were presented in the survey together with detailed answers for one question. The question has been answered for DTTs in table 6.4. This answer is shorter and clearer than that provided by other methods.

6.5.2.8 Ease of modification

DTTs simple structure and modularity enhances their ease of modification, however objective measures of this quantity are very difficult to obtain. As a simple illustration of the ease with which DTTs can be modified, a major change to the way the DTT of Appendix A recognizes declarations so that identifiers are processed as encountered rather than after the entire declaration has been processed requires only superficial changes to a small number of rules.

None the less, this is not sufficient to justify a positive rating for DTTs, especially since all the other models were rated neutral, and hence DTTs have also been given a neutral rating.

Table 6.4

Question : Is the assignment of the constant 2 to the variable x valid in the following Asple program?

```
begin
  ref int x;
  x := 2;
end
```

This question is answered (1) as a table driven DTT would, and (2) as a human using the DTT definition would.

(1) The parse would proceed normally until after the 'becomes' symbol is read. At this stage the top stack symbol is becomessym. The only forward move that is defined by the parse tables for this particular state is with a lookahead syntactic class of ident. As the lookahead syntactic class is 'constant', the device reports an error (the earliest point possible) and either attempts error recovery or halts.

(2) A human user would notice that in parsing the declaration, a rule
(rlegal_x_ref_ref_int. → ident_x.)
is added to the production set. In the parsing of the assignment statement, this production is used to recognize the left hand side of an assignment,

reducing it to the syntactic class rlegal. As the only legitimate assignments with rlegal on the left hand side have either rlegal or legal on the right hand side, a human user would deduce that the input is not a legal Asple program.

6.5.3 Conclusion

The most significant observation from the above comparison is in the diversity of objectives of the various models. Production Systems, like DTTs, are a powerful syntactic device that provides a relative semantic definition via a translation rather than an absolute one. Axiomatic Semantics does not attempt to define syntax, only semantics. W-grammars provide a complete specification within a formal system. VDL provides a complete specification based on a model of an abstract machine. Attribute Grammars provide only a relative semantic definition.

The different objectives and priorities of these models makes an evaluation of their descriptive properties difficult. However it is clear that DTTs are not inferior to the above in terms of either objectives or achievements, and can even be seen to be superior in having useful human and mechanical characteristics not generally shared by the other models.

6.6 Language Design

The criteria in section 6.5 were developed to compare the performances of formal models at specifying a language. It is interesting to turn this around and use a formal model to determine the appropriateness of the design of language constructs, as it can be expected that complexities in a definition will reflect overly complex language constructs. A further point is that the highlighting of problem areas of programming language design by this method would pinpoint areas where effort is needed for improvement.

This aspect of DTTs has already been mentioned in relation to anonymous types and forward declared procedures, and also in the more important case of modes and dereferencing in Asple. Allowing an infinite number of modes causes an overly complex definition. In reality, only the modes corresponding to constants, variables and pointers have any value, as the rest are quite useless.

A cynic could suggest that faced with the alternatives of a pedestrian definition (that iterates all modes) of a desired language, or a sophisticated definition (modes defined by a single recursive rule and base rules) of a modified language, the designers of Algol 68 (on which Asple was based) were goaded into choosing the latter.

Whether based on any substance or not, this suggestion does raise the question of the values perceived to be desirable in the design of a language and its definition. Of more interest is the apparent conflict between the above

suggestion (that choosing a succinct definition led to language complexities) and the objective of the first paragraph of this section (that a good model would highlight complexities of a language).

The fault here lies with the use of very powerful constructs that have the appearance of simplifying the specification when in fact they only hide detail. In instances (such as in W-grammars) where the powerful constructs have no realizable representations, the simplicity of the description is a sham. It is only a model which describes its objects by constructs that are realizable that are useful, and that will be able to highlight overly complex components of a definition. The theoretical identification of definitional structures that correspond to realizable structures is not considered further in this thesis, however some obvious empirical results can be cited. For example, it has been shown that DTTs do reflect the design flaws of Asple, and that W-grammars do not.

6.7 Target Languages

A translator operates between two languages. Although an intermediate code (p-code) was chosen for the Asple example as a means of easily demonstrating the application of DTTs to compiler generation, a machine independent notation could have been used. The deficiencies of using a coded translation are in its machine dependent nature, its low level structures, and in (possibly) the nature of the

definition of the target language. These deficiencies were apparent in the demonstration with p-code, as it was seen that the form of the productions for Asple statements was fragmented, and that there were difficulties with a translator for Pascal to p-code.

Alternatively, it is possible to have as the target language any formalism that will adequately define the semantics of the programming language. The most obvious other choices are target languages which are based on either axiomatic semantics or denotational semantics. An axiomatic approach would produce a logical system that defined all true statements (and no false ones) about the program. A denotational approach would yield a function specifying a mapping from the input of the program to its output. As an example of the advantages over coded translations, it can be seen that denotational semantics, for example, contain sufficiently powerful mathematical structures that the definition of arithmetical concepts within the DTT productions is not necessary. Although some 'compile time' range checking would be passed to the translation, it seems quite feasible to propose a DTT with a denotational target language as a basis for a compiler generator, or a DTT with an axiomatic target as the basis for a program verifier.

Combining a syntactically strong model such as DTTs with a semantically strong model seems sensible, and its modularity contrasts well with the monolithic approach of models like W-grammars. Such a combination is however, not

a new idea. Ledgard et al [37] have used axiomatic semantics as the target of a specification using Production Systems. Paulson [45] has used denotational semantics as the target of a system based on attribute grammars.

The question of which is the best form for the translation language of a DTT is not simply answered. It is easy to find particular forms of translation languages that are suitable for particular applications. However, a model that encapsulates semantics in a general form which may be generally amenable to mechanical processing (rather than for specific applications) is not available.

Suggesting that the choice of the translation language depends on the intended application is not really satisfactory. This can be related to the concept of consistent and complementary definitions introduced in [19]. However, not only is it a major task to prepare several different definitions for different audiences, but the job of determining that they are consistent (i.e. that they all define exactly the same language) is overwhelming.

Therefore it is necessary to choose just one form for the translation language. The objectives of chapter one were to produce a definition with useful human and mechanical properties. The problem of choosing an adequate model is exacerbated by the variety of uses a formal definition must serve, and with human users, by their personal preferences. Theoreticians would find in denotational semantics the answers to most of their questions, whereas applications

programmers (the vast majority of users) would find none. As none of the currently available semantic models or coded translations seem to meet the chapter one criteria of being useful (in either the human or mechanical aspects, let alone both), the formulation of an adequate semantic model must be left for further research.

6.8 Conclusion

This chapter has evaluated DTTs and considered them in relation to other models. Control structures and mechanisms for representing symbol tables and their management were investigated in the most prominent models, and DTTs found to have a simpler and more unified approach. The variety of mechanisms used in the models were found to be built on two basic strategies, the tagging of existing structures and the adding of new ones. DTTs were the only device to use both strategies in the representation of the symbol table, which allowed the greatest possibilities for using natural representations of the objects involved. The descriptive capabilities of various models in relation to Asple were compared and DTTs found to syntactically strong. It was also demonstrated that DTTs have the ability to highlight anomalies in the design of programming languages. The possibility of choosing a semantic model for the translation language was also discussed, and the human properties of current semantic models were not found to be attractive as the ultra-mathematical formalism can result in meaning being hidden beneath abstruse notation. As coded translations

also have flaws, an optimal choice of the translation language does not seem to be possible at present, and further work in the area of semantic models is required.

Chapter Seven

CONCLUSION

7.1 Introduction

In section 1.3, DTTs were presented as the result of an experiment into the design of a formal model for the specification of programming languages. The aims of the experiment were to find a model that possessed useful human and mechanical characteristics. This chapter discusses the achievements of DTTs in respect of these criteria. Some areas for further research are also discussed. The thesis concludes by summarizing the qualities of DTTs.

7.2 Human Factors

The measures of success of a formal definition in its human characteristics were taken in section 1.2 to be the quality of, and the ease of obtaining answers to questions about the language defined.

It is not easy to quantify the quality of an answer to such questions, or the ease of obtaining them. Characteristics such as accuracy, detail, and succinctness all relate to this measurement. Although the discussion of section 6.5 gave much space to this issue, the sample of table 6.4 showing DTTs to be superior to existing methods is not sufficient evidence to justify a conclusive result. None the less, the author contends that DTTs have useful human characteristics.

7.3 Mechanical Factors

The measures of success of a formal definition in its mechanical characteristics were taken in section 1.2 to be the quality of, and the ease of obtaining automated systems concerning the language defined.

The automatic generation of a compiler for Asple has been demonstrated in chapter five. This compiler was produced very easily, and although not of commercial quality in terms of speed and size, it none the less performed its task satisfactorily.

Although the generation of other systems has not been investigated, the formal basis provided by DTTs could provide a basis for the implementation of systems such as program verifiers, especially if the translation language were expressed in terms of a semantic model. Therefore the author contends that DTTs have useful mechanical characteristics.

7.4 Further Work

There are however, still some points relating to DTTs that need to be investigated and have not been pursued in this dissertation. These include:

(1) An exact characterization of the class of languages processed by tables driven DTTs,

(2) An investigation of alternative actions to see if the current repertoire provides the most succinct description, and the best strategy for error

recovery,

(3) An investigation of efficiency mechanisms (although the system described in chapter four is table driven, it does perform a substantial amount of parse time pattern matching that is just mechanical substitution, and could undoubtedly be streamlined),

(4) An investigation of more general methods for resolving conflicts by considering the dynamic and template characteristics of a DTT, and

(5) The formulation of an ideal model to be used as the translation language for DTTs. Such a model should include sufficient structures for the appropriate representation of all semantic information. In addition it should satisfy the requirements of a wide variety of users: theoreticians, implementors, programmers, and language designers. This is the most pressing need of DTTs, as they are an adequate syntactic model needing a suitable semantic target.

7.5 Summary

The review of related work in chapter two showed that no satisfactory model for the definition of programming languages has been put forward, despite the large volume of research that has been invested in this problem.

In this thesis, DTTs have been compared with other proposals and found to be superior in many respects relating

to human and mechanical factors. In particular, some potentially powerful alternative systems suffer from having no equivalent recognizing device, however the design of DTTs has overcome this inadequacy. DTTs are able, within an unified framework, to completely specify the most difficult of syntactic components, the management of identifiers encountered in a program. Also DTTs have the ability to highlight anomalies in the design of programming languages.

Although not the ultimate solution to the problem of programming language definition, DTTs are certainly more useful than existing methods, and therefore provide a positive contribution to the theory of programming languages.

APPENDIX A

The initial production set, p0, of D_{asple}, mapping legal programs into p-code.

Goal production

```
A00 (goal →
A01     beginpart dectrain semicolonsym strain endpart)
```

Initialization and finalization

```
A02 (beginpart → beginsym
A03     {l_3} {_ent_1___l_4} {_ent_2___l_5})
A04 (endpart → endsym loc_S. count_T.
A05     {_retp} {l_4=_____S_} {l_5=_____T_} {q}
A06     {i__0} {_mst_________0} {_cup_0___l_3} {_stp} {q})
```

Productions to recognize declarations and modes

- A production is added to the system for each identifier declared to remember its name, mode and allocated address (e.g. A24, A36, A58, A60).
- Multiple declarations of an identifier result in the goal production being deleted, thereby inhibiting a successful parse (e.g. A16, A21, A28, A33, A40, A45, A52, A57).
- Symbols with the syntactic class 'legal' correspond to an identifier that references

a primitive mode. These symbols also contain the name, allocated address and mode of the identifier (e.g. A24).

- Symbols with the syntactic class 'rlegal' correspond to other identifiers. These symbols also contain the name, allocated address and mode of the identifier (e.g. A48).
- The mode integer is denoted in a symbol by _i. (A61), the mode Boolean by _b. (A62), a reference mode by _ref (A63, A64).
- Unique addresses are allocated by A130 - A133 in the symbol with syntactic class 'addr'.

```
A07 (dectrain → declist)
A08 (dectrain → dectrain semicolonsym declist)
A09 (decc_M^L_. → mode_M. idlist_L.)
A10 (rdecc_M^L_. → rmode_M. idlist_L.)
A11 (idlist_N. → ident_N.)
A12 (idlist_N_L. → idlist_L. commasym ident_N.)
A13 (decc_M^L. → decc_M^N_L. addr_A.
A14    [delete_production,
A15       (legal_AAAAA_N_ref_MMMMM. → ident_N.)]
A16    [delete_production,(goal → beginpart dectrain
A17       semicolonsym strain endpart)]
A18    [continue]
A19    [delete_production,
```

```
A20        (rlegal_AAAAA_N_ref_MMMMM.→ ident_N.)]
A21     [delete_production,(goal→ beginpart dectrain
A22        semicolonsym strain endpart)]
A23     [continue]
A24     [add_production,(legal_A_N_ref_M.→ ident_N.)])
A25 (declist→ decc_M^N_. addr_A.
A26     [delete_production,
A27        (legal_AAAAA_N_ref_MMMMM.→ ident_N.)]
A28     [delete_production,(goal→ beginpart dectrain
A29        semicolonsym strain endpart)]
A30     [continue]
A31     [delete_production,
A32        (rlegal_AAAAA_N_ref_MMMMM.→ ident_N.)]
A33     [delete_production,(goal→ beginpart dectrain
A34        semicolonsym strain endpart)]
A35     [continue]
A36     [add_production,(legal_A_N_ref_M.→ ident_N.)])
A37 (rdecc_M^L.→ rdecc_M^N_L. addr_A.
A38     [delete_production,
A39        (legal_AAAAA_N_ref_MMMMM.→ ident_N.)]
A40     [delete_production,(goal→ beginpart dectrain
A41        semicolonsym strain endpart)]
A42     [continue]
A43     [delete_production,
A44        (rlegal_AAAAA_N_ref_MMMMM.→ ident_N.)]
A45     [delete_production,(goal→ beginpart dectrain
A46        semicolonsym strain endpart)]
A47     [continue]
```

```
A48    [add_production,(rlegal_A_N_ref_M. → ident_N.)])
A49 (declist → rdecc_M^N_. addr_A.
A50   [delete_production,
A51      (legal_AAAAA_N_ref_MMMMM. → ident_N.)]
A52   [delete_production,(goal → beginpart dectrain
A53      semicolonsym strain endpart)]
A54   [continue]
A55   [delete_production,
A56      (rlegal_AAAAA_N_ref_MMMMM. → ident_N.)]
A57   [delete_production,(goal → beginpart dectrain
A58      semicolonsym strain endpart)]
A59   [continue]
A60   [add_production,(rlegal_A_N_ref_M. → ident_N.)])
A61 (mode_i. → intsym)
A62 (mode_b. → boolsym)
A63 (rmode_ref_M. → refsym mode_M.)
A64 (rmode_ref_M. → refsym rmode_M.)
```

Productions to translate Statements

- Unique labels are generated by A126-29 in symbols of syntactic class 'lab'.
- Templates are used to remember destinations of jumps.

```
A65 (strain → strain semicolonsym stt)
A66 (strain → stt)
A67 (stt → assignment)
A68 (stt → conditional)
A69 (stt → iteration)
```

```
A70 (stt → transput)
A71 (iteration → whst_S_T. dosym strain odsym
A72    {_ujp_______lS_} {lT_})
A73 (whsym_T. → whilesym lab_T. {lT_})
A74 (whst_S_T. → whsym_S. exp_b. lab_T. {_fjp_______lT_})
A75 (conditional → ifst_T. thensym strain fisym {lT_})
A76 (ifst_T. → ifsym exp_b. lab_T. {_fjp_____l_T_})
A77 (conditional → ifthst_T. strain fisym {lT_})
A78 (ifthst_L. → ifst_T. thensym strain elsesym lab_L.
A79    {_ujp_______lL_} {lT_})
A80 (transput → outsym exp_M. space_T. space_S.
A81    {_ldci_______10_} {_lda_0_____6_}
A82    {_csp_______wrM_}
A83    {_lda_0_____6_} {_csp_______wln_})
A84 (transput → insym legal_A_N_ref_M.
A85    {_lda_0____A_} {_lda_0_____5_} {_csp_______rdM_}
A86    {_lda_0_____5_} {_csp_______rln_})
A87 (inn_M_. → insym rlegal_A_N_ref_M. {_lao_______A_})
A88 (inn_M. → inn_ref_M. {_inda_______0_})
A89 (transput → inn_M_.
A90    {_lda_0_____5_} {_csp_______rdM_} {_lda_0_____5_}
A91    {_csp_______rln_})
```

Productions to recognize expressions

- productions defining 'legal' and 'rlegal' are added to the system during the processing of declarations (A7 - A64)
- Whenever a value is loaded onto the stack

a location is reserved via the symbol 'space'.

```
A92 (fact_M. -> legal_A_N_ref_M. space_T.
A93     {_ldoM_______A_})
A94 (fact_M. -> constant_M-V. space_T. {_ldcM_______V_})
A95 (fact_M. -> dref_M_. {_indM_______0_})
A96 (dref_M_. -> rlegal_A_N_ref_ref_M.  space_T.
A97     {_ldoa_______A_})
A98 (dref_M. -> dref_ref_M. {_inda_______0_})
A99 (fact_M. -> lparensym exp_M. rparensym)
A100 (fact_b. -> lparensym exp_i. eqsym exp_i. rparensym
A101     {_equi_})
A102 (fact_b. -> lparensym exp_i. nesym exp_i. rparensym
A103     {_neqi_})
A104 (term_M. -> fact_M.)
A105 (term_i. -> term_i. starsym fact_i. {_mpi_})
A106 (term_b. -> term_b. starsym fact_b. {_and_})
A107 (exp_M. -> term_M. )
A108 (exp_i. -> exp_i. plussym term_i. {_adi_})
A109 (exp_b. -> exp_b. plussym term_b. {_ior_})
```

Productions to process the assignment statement

- These are grouped into the three categories of assignment depending on the syntactic classes of the source and destination.
  (1) the destination mode references a primitive mode (syntactic class 'legal'),

(A110).

(2) the destination mode references a reference to a primitive mode (syntactic class 'rlegal'), and the source mode is a reference to the same primitive mode (syntactic class 'legal') (there is such a rule for both primitive modes), (A113 A116).

(3) other cases (both source and destination have syntactic class 'rlegal') (A112, A119-A125).

- Category 3 productions ensure that the mode of the source is compatible with that of the destination (A123-A125), and dereference the source if necessary (A119-A122).

```
A110 (assignment → legal_A_N_ref_M. becomessym exp_M.
A111    {_sroM_____A_})
A112 (assignment → assderef_AD^MS_. {_sroa_____AD_})
A113 (assignment → rlegal_AD_ND_ref_ref_i. becomessym
A114    legal_AS_NS_ref_i.
A115    {_lao_____AS_} {_sroa_____AD_})
A116 (assignment → rlegal_AD_ND_ref_ref_b. becomessym
A117    legal_AS_NS_ref_b.
A118    {_lao_____AS_} {_sroa_____AD_})
A119 (assderef_AD^MS. → assderef_AD^ref_MS.
A120    {_inda_____0_})
A121 (assderef_AD^MS. → refass_AD_MD_^AS_MS.
```

```
A122    {_lao_____AS_})
A123 (refass_AD_MD_^AS_MS_.→ rlegal_AD_ND_ref_MD.
A124    becomessym rlegal_AS_NS_MS.)
A125 (refass_AD_MD^AS_MS.→ refass_AD_ref_MD^AS_ref_MS.)
```

Productions allocating labels for p-code

```
A126 (sym_$$$$$$.→ )
A127 (lab_T.→ sym_T.
A128    [delete_production,(sym_T.→ )]
A129    [add_production,(sym_$T.→ )])
```

Productions that allocate identifier addresses

```
A130 (loc_$$$$$$$$$.→ )
A131 (addr_T.→ loc_T.
A132    [delete_production,(loc_T.→ )]
A133    [add_production,(loc_$T.→ )])
```

Productions to reserve space for temporary values

```
A134 (count_$$$$$.→ )
A135 (space_T.→ count_T.
A136    [delete_production,(count_T.→ )]
A137    [add_production,(count_$T.→ )])
```

APPENDIX B

The initial production set for a subset of Pascal based on record and block structure

Goal Production

```
P000 (goal -> programme)
```

Productions to Recognize a Program

```
P001 (programme -> progheader blokk spotsym)
P002 (progheader -> progsym ident_N. semicolonsym)
```

Productions to Process Block Structure

-the trailing string in LEVEL identifies the depth of nesting

```
P003 (blokk -> block level_$L.
P004     [delete_production,(level_$L. -> )]
P005     [add_production,(level_L. -> )])
P006 (level_$$. -> )
P007 (block ->
P008     labpart constpart typepart varpart pfpart stpart)
```

Productions to Process Label Declarations

```
P009 (labpart -> )
```

```
P010 (labpart -> labelsym declabs semicolonsym)
P011 (declabs -> labdec)
P012 (declabs -> declabs commasym labdec)
P013 (labdec -> constant_int-V. level_L.
P014    [delete_production,
P015       (lablegal_L_V. -> constant_int-V.)]
P016    [delete_production,(goal -> programme)]
P017    [continue]
P018    [add_action,
P019       (block -> labpart constpart typepart
P020          varpart pfpart stpart),
P021       [add_production,
P022          '(lablegal_LLLL_V. -> constant_int-V.)]]
P023    [delete_production,
P024       (lablegal_LLLL_V. -> constant_int-V.)]
P025    [continue]
P026    [add_action,
P027       (block -> labpart constpart typepart
P028          varpart pfpart stpart),
P029       [delete_production,
P030          (lablegal_L_V. -> constant_int-V.)]]
P031    [add_production,
P032       (lablegal_L_V. -> constant_int-V.)])
```

Productions to Process Type Declarations

- atype_N:T. matches a type T with name N
- atype_:T. matches an anonymous type T
- setype_T. matches a scalar type T

```
P033 (typepart -> )
P034 (typepart -> typesym dectypes semicolonsym)
P035 (dectypes -> typedec)
P036 (dectypes -> dectypes semicolonsym typedec)
P037 (typedec -> nname_N_L. eqsym atypeNM:-|T.
P038    [delete_production,(atype_XXXX. -> name_N_L$P.)]
P039    [delete_production,(goal -> programme)]
P040    [continue]
P041    [add_action,
P042       (block -> labpart constpart typepart
P043          varpart pfpart stpart),
P044       [add_production,
P045          '(atype_XXXX. -> name_N_LLLL$K.)]]
P046    [delete_production,
P047       (atype_XXXX. -> name_N_LLLL$K.)]
P048    [continue]
P049    [add_action,
P050       (block -> labpart constpart typepart
P051          varpart pfpart stpart),
P052       [add_production,
P053          '(legal_XXXX. -> name_N_LLLL$K.)]]
P054    [delete_production,
P055       (legal_XXXX. -> name_N_LLLL$K.)]
P056    [continue]
P057    [add_action,
P058       (block -> labpart constpart typepart
P059          varpart pfpart stpart),
```

```
P060        [add_production,
P061            '(funcc_XXXX. -> name_N_LLLL$K.)]]
P062     [delete_production,(funcc_XXXX. -> name_N_LLLL$K.)]
P063     [continue]
P064     [add_action,
P065        (block -> labpart constpart typepart
P066            varpart pfpart stpart),
P067        [add_production,
P068            '(procc_XXXX. -> name_N_LLLL$K.)]]
P069     [delete_production,(procc_XXXX. -> name_N_LLLL$K.)]
P070     [continue]
P071     [add_action,
P072        (block -> labpart constpart typepart
P073            varpart pfpart stpart),
P074        [delete_production,
P075            (atype_N:-¦T. -> name_N_L$P.)]]
P076     [add_production,(atype_N:-¦T. -> name_N_L$P.)])
P077
P078 (typedec -> nname_N_L. eqsym atypeNM:-r¦T.
P079     [delete_production,(atype_XXXX. -> name_N_L$P.)]
P080     [delete_production,(goal -> programme)]
P081     [continue]
P082     [add_action,
P083        (block -> labpart constpart typepart
P084            varpart pfpart stpart),
P085        [add_production,
P086            '(atype_XXXX. -> name_N_LLLL$K.)]]
P087     [delete_production,(atype_XXXX. -> name_N_LLLL$K.)]
```

```
P088    [continue]
P089    [add_action,
P090       (block -> labpart constpart typepart
P091          varpart pfpart stpart),
P092       [add_production,
P093          '(legal_XXXX. -> name_N_LLLL$K.)]]
P094    [delete_production,(legal_XXXX. -> name_N_LLLL$K.)]
P095    [continue]
P096    [add_action,
P097       (block -> labpart constpart typepart
P098          varpart pfpart stpart),
P099       [add_production,
P100          '(funcc_XXXX. -> name_N_LLLL$K.)]]
P101    [delete_production,(funcc_XXXX. -> name_N_LLLL$K.)]
P102    [continue]
P103    [add_action,
P104       (block -> labpart constpart typepart
P105          varpart pfpart stpart),
P106       [add_production,
P107          '(procc_XXXX. -> name_N_LLLL$K.)]]
P108    [delete_production,(procc_XXXX. -> name_N_LLLL$K.)]
P109    [continue]
P110    [add_action,
P111       (block -> labpart constpart typepart
P112          varpart pfpart stpart),
P113       [delete_production,
P114          (atype_N:-r|T. -> name_N_L$P.)]]
P115    [add_production,
```

```
P116        (atype_N:-r|T. -> name_N_L$P.
P117           [add_production,'(recname -> legal_:rec.)])])
P118
P119 (atype_int:-|int. -> integersym)
P120 (atype_real:-|real. -> realsym)
P121 (atype_char:-|char. -> charsym)
P122 (atype_bool:-|bool. -> booleansym)
P123
P124 (sclegal_int. -> legalX:int.)
P125 (sclegal_char. -> legalX:char.)
P126 (sclegal_bool. -> legalX:bool.)
P127 (subrange_N. -> constt_N-VV. rangesym constt_N-VW.)
P128 (atype_char:-|char. -> subrange_char.)
P129 (atype_bool:-|bool. -> subrange_bool.)
P130 (atype_int:-|int. -> subrange_int.)
P131 (sctype_int. -> atypeX:-|int.)
P132 (sctype_char. -> atypeX:-|char.)
P133 (sctype_bool. -> atypeX:-|bool.)
P134 (atype_:-|set_T. -> setsym ofsym sctype_T.)
```

Productions to Process Variable Declartions

(but not of structures that contain components of type record)

-the template symbol legal_N:T. matches a declared identifier of type T that is named N

-the template symbol legal_:T. matches a declared identifier of the anonymous type T

```
P135 (varpart -> )
P136 (varpart -> varsym decvars semicolonsym)
P137 (decvars -> vardec)
P138 (decvars -> decvars semicolonsym vardec)
P139 (vdecNT:-¦X_;T. -> idlist_X. colonsym atypeNT:-¦T.)
P140 (vardec -> vdecNT:-¦L_N_;T.
P141    [delete_production,(atype_XXXX. -> name_N_L$P.)]
P142    [delete_production,(goal -> programme)]
P143    [continue]
P144    [delete_production,(legal_XXXX. -> name_N_L$P.)]
P145    [delete_production,(goal -> programme)]
P146    [continue]
P147    [add_action,
P148       (block -> labpart constpart typepart
P149          varpart pfpart stpart),
P150       [add_production,
P151          '(atype_XXXX. -> name_N_LLLL$K.)]]
P152    [delete_production,(atype_XXXX. -> name_N_LLLL$K.)]
P153    [continue]
P154    [add_action,
P155       (block -> labpart constpart typepart
P156          varpart pfpart stpart),
P157       [add_production,
P158          '(legal_XXXX. -> name_N_LLLL$K.)]]
P159    [delete_production,(legal_XXXX. -> name_N_LLLL$K.)]
P160    [continue]
P161    [add_action,
P162       (block -> labpart constpart typepart
```

```
P163          varpart pfpart stpart),
P164       [add_production,
P165          '(funcc_XXXX. -> name_N_LLLL$K.)]]
P166    [delete_production,(funcc_XXXX. -> name_N_LLLL$K.)]
P167    [continue]
P168    [add_action,
P169       (block -> labpart constpart typepart
P170          varpart pfpart stpart),
P171       [add_production,
P172          '(procc_XXXX. -> name_N_LLLL$K.)]]
P173    [delete_production,(procc_XXXX. -> name_N_LLLL$K.)]
P174    [continue]
P175    [add_action,
P176       (block -> labpart constpart typepart
P177          varpart pfpart stpart),
P178       [delete_production,(legalNT:T. -> name_N_L$P.)]]
P179    [add_production,(legalNT:T. -> name_N_L$P.)])

P180 (vdecNT:-¦L_REM;T. -> vdecNT:-¦L_N_REM;T.
P181    [delete_production,(atype_XXXX. -> name_N_L$P.)]
P182    [delete_production,(goal -> programme)]
P183    [continue]
P184    [delete_production,(legal_XXXX. -> name_N_L$P.)]
P185    [delete_production,(goal -> programme)]
P186    [continue]
P187    [add_action,
P188       (block -> labpart constpart typepart
P189          varpart pfpart stpart),
```

```
P190        [add_production,
P191            '(atype_XXXX. -> name_N_LLLL$K.)]]
P192    [delete_production,(atype_XXXX. -> name_N_LLLL$K.)]
P193    [continue]
P194    [add_action,
P195        (block -> labpart constpart typepart
P196            varpart pfpart stpart),
P197        [add_production,
P198            '(legal_XXXX. -> name_N_LLLL$K.)]]
P199    [delete_production,(legal_XXXX. -> name_N_LLLL$K.)]
P201    [continue]
P202    [add_action,
P203        (block -> labpart constpart typepart
P204            varpart pfpart stpart),
P205        [add_production,
P206            '(funcc_XXXX. -> name_N_LLLL$K.)]]
P207    [delete_production,(funcc_XXXX. -> name_N_LLLL$K.)]
P208    [continue]
P209    [add_action,
P210        (block -> labpart constpart typepart
P211            varpart pfpart stpart),
P213        [add_production,
P214            '(procc_XXXX. -> name_N_LLLL$K.)]]
P215    [delete_production,(procc_XXXX. -> name_N_LLLL$K.)]
P216    [continue]
P217    [add_action,
P218        (block -> labpart constpart typepart
P219            varpart pfpart stpart),
```

```
P220        [delete_production,(legalNT:T. -> name_N_L$P.)]]
P221     [add_production,(legalNT:T. -> name_N_L$P.)])
```

Productions to Process Declarations of Variables with Components of type record

```
P222 (vdecNT:-r¦X_;T. -> idlist_X. colonsym atypeNT:-r¦T.)
P223 (vardec -> vdecNT:-r¦L_N_;T.
P224     [delete_production,(atype_XXXX. -> name_N_L$P.)]
P225     [delete_production,(goal -> programme)]
P226     [continue]
P227     [delete_production,(legal_XXXX. -> name_N_L$P.)]
P228     [delete_production,(goal -> programme)]
P229     [continue]
P230     [add_action,
P231        (block -> labpart constpart typepart
P232           varpart pfpart stpart),
P233        [add_production,
P234           '(atype_XXXX. -> name_N_LLLL$K.)]]
P235     [delete_production,(atype_XXXX. -> name_N_LLLL$K.)]
P236     [continue]
P237     [add_action,
P238        (block -> labpart constpart typepart
P239           varpart pfpart stpart),
P240        [add_production,
P241           '(legal_XXXX. -> name_N_LLLL$K.)]]
P242     [delete_production,(legal_XXXX. -> name_N_LLLL$K.)]
P243     [continue]
```

```
P244    [add_action,
P245       (block -> labpart constpart typepart
P246          varpart pfpart stpart),
P247       [add_production,
P248          '(funcc_XXXX. -> name_N_LLLL$K.)]]
P249    [delete_production,(funcc_XXXX. -> name_N_LLLL$K.)]
P250    [continue]
P251    [add_action,
P252       (block -> labpart constpart typepart
P253          varpart pfpart stpart),
P254       [add_production,
P255          '(procc_XXXX. -> name_N_LLLL$K.)]]
P256    [delete_production,(procc_XXXX. -> name_N_LLLL$K.)]
P257    [continue]
P258    [add_action,
P259       (block -> labpart constpart typepart
P260          varpart pfpart stpart),
P261       [delete_production,(legalNT:T. -> name_N_L$P.)]]
P262    [add_production,
P263       (legalNT:T. -> name_N_L$P.
P264          [add_production,'(recname -> legal_:rec.)])]
P265    [delete_production,(recname -> legal_:rec.)])

P266 (vdecNT:-r|L_REM;T. -> vdecNT:-r|L_N_REM;T.
P267    [delete_production,(atype_XXXX. -> name_N_L$P.)]
P268    [delete_production,(goal -> programme)]
P269    [continue]
P270    [delete_production,(legal_XXXX. -> name_N_L$P.)]
```

```
P271    [delete_production,(goal -> programme)]
P272    [continue]
P273    [add_action,
P274       (block -> labpart constpart typepart
P275          varpart pfpart stpart),
P276       [add_production,
P277          '(atype_XXXX. -> name_N_LLLL$K.)]]
P278    [delete_production,(atype_XXXX. -> name_N_LLLL$K.)]
P279    [continue]
P280    [add_action,
P281       (block -> labpart constpart typepart
P282          varpart pfpart stpart),
P283       [add_production,
P284          '(legal_XXXX. -> name_N_LLLL$K.)]]
P285    [delete_production,(legal_XXXX. -> name_N_LLLL$K.)]
P286    [continue]
P287    [add_action,
P288       (block -> labpart constpart typepart
P289          varpart pfpart stpart),
P290       [add_production,
P291          '(funcc_XXXX. -> name_N_LLLL$K.)]]
P292    [delete_production,(funcc_XXXX. -> name_N_LLLL$K.)]
P293    [continue]
P294    [add_action,
P295       (block -> labpart constpart typepart
P296          varpart pfpart stpart),
P297       [add_production,
P298          '(procc_XXXX. -> name_N_LLLL$K.)]]
```

```
P299    [delete_production,(proce_XXXX. -> name_N_LLLL$K.)]
P300    [continue]
P301    [add_action,
P302       (block -> labpart constpart typepart
P303          varpart pfpart stpart),
P304       [delete_production,(legalNT:T. -> name_N_L$P.)]]
P305    [add_production,
P306       (legalNT:T. -> name_N_L$P.
P307          [add_production,'(recname -> legal_:rec.)])])
```

Productions to Process Procedure and Function Declarations

```
P308 (pfpart -> )
P309 (pfpart -> decpfs semicolonsym)
P310 (decpfs -> pfdec)
P311 (decpfs -> decpfs semicolonsym pfdec)
P312 (pfdec -> pheader blokk)
P313 (pfdec -> fheader blokk)
P314 (pheader ->
P315    procsym nname_N_L. parlist_PP. semicolonsym
P316    [delete_production,(atype_XXXX. -> name_N_L$P.)]
P317    [delete_production,(goal -> programme)]
P318    [continue]
P319    [delete_production,(legal_XXXX. -> name_N_L$P.)]
P320    [delete_production,(goal -> programme)]
P321    [continue]
P322    [delete_production,(proce_XXXX. -> name_N_L$P.)]
```

```
P323    [delete_production,(goal -> programme)]
P324    [continue]
P325    [delete_production,(funcc_XXXX. -> name_N_L$P.)]
P326    [delete_production,(goal -> programme)]
P327    [continue]
P328    [add_action,
P329       (block -> labpart constpart typepart
P330          varpart pfpart stpart),
P331       [add_production,
P332          '(atype_XXXX. -> name_N_LLLL$K.)]]
P333    [delete_production,(atype_XXXX. -> name_N_LLLL$K.)]
P334    [continue]
P335    [add_action,
P336       (block -> labpart constpart typepart
P337          varpart pfpart stpart),
P338       [add_production,
P339          '(legal_XXXX. -> name_N_LLLL$K.)]]
P340    [delete_production,(legal_XXXX. -> name_N_LLLL$K.)]
P341    [continue]
P342    [add_action,
P343       (block -> labpart constpart typepart
P344          varpart pfpart stpart),
P345       [add_production,
P346          '(funcc_XXXX. -> name_N_LLLL$K.)]]
P347    [delete_production,(funcc_XXXX. -> name_N_LLLL$K.)]
P348    [continue]
P349    [add_action,
P350       (block -> labpart constpart typepart
```

```
P351          varpart pfpart stpart),
P352       [add_production,
P353          '(proce_XXXX. -> name_N_LLLL$K.)]],
P354    [delete_production,(proce_XXXX. -> name_N_LLLL$K.)]
P355    [continue]
P356    [add_production,(proce_P. -> name_N_L$P.)]
P357    [add_action,
P358       (block -> labpart constpart typepart
P359          varpart pfpart stpart),
P360       [delete_production,(proce_P. -> name_N_L$P.)]]
P361    [add_action,(blokk -> block level_$LL.),
P362       [add_production,
P363          '(block -> labpart constpart typepart
P364             varpart pfpart stpart)]]
P365    [delete_production,
P366       (block -> labpart constpart typepart
P367          varpart pfpart stpart)]
P368    [add_production,
P369       (block -> labpart constpart typepart
P370          varpart pfpart stpart)]
P371    [add_action,
P372       (block -> labpart constpart typepart
P373          varpart pfpart stpart),
P374       [delete_production,(blokk -> block level_$LL.)]]
P375    [add_action,
P376       (block -> labpart constpart typepart
P377          varpart pfpart stpart),
P378       [add_production,'(blokk -> block level_$LL.)]]
```

```
P379     [delete_production,(blokk -> block level_$LL.)]
P380     [add_production,(blokk -> block level_$LL.)]
P381     [delete_production,(level_L. ->)]
P382     [add_production,(level_$L. -> )])

P383 (fheader -> funcsym nname_N_L. parlist_PP.
P384     colonsym setype_T. semicolonsym
P385     [delete_production,(atype_XXXX. -> name_N_L$P.)]
P386     [delete_production,(goal -> programme)]
P387     [continue]
P388     [delete_production,(legal_XXXX. -> name_N_L$P.)]
P389     [delete_production,(goal -> programme)]
P390     [continue]
P391     [delete_production,(proce_XXXX. -> name_N_L$P.)]
P392     [delete_production,(goal -> programme)]
P393     [continue]
P394     [delete_production,(funce_XXXX. -> name_N_L$P.)]
P395     [delete_production,(goal -> programme)]
P396     [continue]
P397     [add_action,
P398        (block -> labpart constpart typepart
P399           varpart pfpart stpart),
P400        [add_production,
P401           '(atype_XXXX. -> name_N_LLLL$K.)]]
P402     [delete_production,(atype_XXXX. -> name_N_LLLL$K.)]
P403     [continue]
P404     [add_action,
P405        (block -> labpart constpart typepart
```

```
P406          varpart pfpart stpart),
P407       [add_production,
P408          '(legal_XXXX. -> name_N_LLLL$K.)]]
P409    [delete_production,(legal_XXXX. -> name_N_LLLL$K.)]
P410    [continue]
P411    [add_action,
P412       (block -> labpart constpart typepart
P413          varpart pfpart stpart),
P414       [add_production,
P415          '(funcc_XXXX. -> name_N_LLLL$K.)]]
P416    [delete_production,(funcc_XXXX. -> name_N_LLLL$K.)]
P417    [continue]
P418    [add_action,
P419       (block -> labpart constpart typepart
P420          varpart pfpart stpart),
P421       [add_production,
P422          '(procc_XXXX. -> name_N_LLLL$K.)]]
P423    [delete_production,(procc_XXXX. -> name_N_LLLL$K.)]
P424    [continue]
P425    [add_production,(funcc_P:T. -> name_N_L$P.)]
P426    [add_action,
P427       (block -> labpart constpart typepart
P428          varpart pfpart stpart),
P429       [delete_production,(funcc_P:T. -> name_N_L$P.)]]
P430    [add_action,(blokk -> block level_$LL.),
P431       [add_production,
P432          '(block -> labpart constpart typepart
P433             varpart pfpart stpart)]]
```

```
P434    [delete_production,
P435       (block -> labpart constpart typepart
P436          varpart pfpart stpart)]
P437    [add_production,
P438       (block -> labpart constpart typepart
P439          varpart pfpart stpart)]
P440    [add_action,
P441       (block -> labpart constpart typepart
P442          varpart pfpart stpart),
P443       [delete_production,(blokk -> block level_$LL.)]]
P444    [add_action,
P445       (block -> labpart constpart typepart
P446          varpart pfpart stpart),
P447       [add_production,'(blokk -> block level_$LL.)]]
P448    [delete_production,(blokk -> block level_$LL.)]
P449    [add_production,(blokk -> block level_$LL.)]
P450    [delete_production,(level_L. ->)]
P451    [add_production,(level_$L. -> )]
P452    [add_production,(dest_:T. -> name_N_L$P.)]
P453    [add_action,
P454       (block -> labpart constpart typepart
P455          varpart pfpart stpart),
P456       [delete_production,(dest_:T. -> name_N_L$P.)]])
P457 (parlist_$. -> )
```

Productions to Process Statements

```
P458 (stpart -> beginsym stlist endsym)
```

```
P459 (stlist -> statement)
P460 (stlist -> stlist semicolonsym statement)
P461 (statement -> lstt)
P462 (lbalst -> balst)
P463 (lbalst -> lablegal_L_V. colonsym balst)
P464 (elemst -> beginsym stlist endsym)
P465 (balst ->
P466    ifsym exp_bool. thensym lbalst elsesym lbalst)
P467 (balst -> whilesym exp_bool. dosym lbalst)
P468 (elemst -> repeatsym stlist untilsym exp_bool.)
P469 (elemst -> forsym sclegal_T. becomessym exp_T.
P470    tosym exp_T. dosym lbalst)
P471 (elemst -> forsym sclegal_T. becomessym exp_T.
P472    downtosym exp_T. dosym lbalst)
P473 (balst -> elemst)
P474 (elemst -> assignment)
P475 (elemst -> proc_P.)
P476 (elemst -> casest)
P477 (elemst -> )
P478 (elemst -> gotosym lablegal_L_V.)
P479 (lstt -> stt)
P480 (lstt -> lablegal_L_V. colonsym stt)
P481 (stt -> ifsym exp_bool. thensym lstt)
P482 (stt -> ifsym exp_bool. thensym lbalst elsesym lstt)
P483 (stt -> whilesym exp_bool. dosym lstt)
P484 (stt -> elemst)
P485 (assignment -> destX:T. becomessym exp_T.)
P486 (assignment -> destX:real. becomessym exp_int.)
```

```
P487 (assignment -> dest_N:T. becomessym nexp_N:T.)
P488 (dest_X. -> legal_X.)
P489 (nest_$$. -> )
P490 (casest -> choice nest_$D.
P491    [delete_production,(nest_$D. -> )]
P492    [add_production,(nest_D. -> )])
P493 (choice -> casenest selegal_T. ofsym select_T. endsym)
P494 (casenest -> casesym nest_D.
P495    [delete_production,(nest_D. -> )]
P496    [add_production,(nest_$D. ->)]
P497    [add_action,(casest -> choice nest_$DD.),
P498       [add_production,
P499          '(choice -> casenest selegal_T. ofsym
P500              select_T. endsym)]]
P501    [delete_production,
P502       (choice ->
P503          casenest selegal_T. ofsym select_T. endsym)]
P504    [add_production,
P505       (choice ->
P506          casenest selegal_T. ofsym select_T. endsym)]
P507    [add_action,
P508       (choice ->
P509          casenest selegal_T. ofsym select_T. endsym),
P510       [delete_production,(casest -> choice
 nest_$DD.)]]
P511    [add_action,
P512       (choice ->
P513          casenest selegal_T. ofsym select_T. endsym),
```

```
P514        [add_production,'(casest -> choice nest_$DD.)]]
P515     [delete_production,(casest -> choice nest_$DD.)]
P516     [add_production,(casest -> choice nest_$DD.)])
P517 (select_T. -> constlist_T. colonsym statement)
P518 (select_T. -> select_T. semicolonsym constlist_T.
P519     colonsym statement)
P520 (constlist_T. -> constt_T-V. nest_D.
P521     [add_action,(triple_D_V. -> ),[continue]]
P522     [delete_production,(goal -> programme)]
P523     [continue]
P524     [add_production,(triple_D_V. -> )]
P525     [add_action,
P526        (choice ->
P527           casenest solegal_T. ofsym select_T. endsym),
P528        [delete_production,(triple_D_V. -> )]])
P529 (constlist_T. ->
P530     constlist_T. commasym constt_T-V. nest_D.
P531     [add_action,(triple_D_V. -> ),[continue]]
P532     [delete_production,(goal -> programme)]
P533     [continue]
P534     [add_production,(triple_D_V. -> )]
P535     [add_action,(choice ->
P536        casenest solegal_T. ofsym select_T. endsym),
P537        [delete_production,(triple_D_V. -> )]])
P538 (nexp_N:rec. -> legal_N:rec.)
```

Prodctions for Arithmetic, Boolean

and Set Expressions

P539 (fact_T. -> legal:T.)

P540 (fact_T. -> constt_T-V.)

P541 (fact_int. -> legalX:int.)

P542 (fact_real. -> legalX:real.)

P543 (fact_bool. -> legalX:bool.)

P544 (fact_char. -> legalX:char.)

P545 (fact_T. -> funcc_X:T.)

P546 (fact_T. -> lparensym exp_T. rparensym)

P547 (term_T. -> fact_T.)

P548 (sexp_T. -> term_T.)

P549 (exp_T. -> sexp_T.)

P550 (fact_bool. -> notsym fact_bool.)

P551 (term_int. -> term_int. divsym fact_int.)

P552 (term_int. -> term_int. modsym fact_int.)

P553 (term_int. -> term_int. starsym fact_int.)

P554 (term_real. -> term_real. starsym fact_real.)

P555 (term_real. -> term_real. slashsym fact_real.)

P556 (term_real. -> term_real. starsym fact_int.)

P557 (term_real. -> term_real. slashsym fact_int.)

P558 (term_real. -> term_int. starsym fact_real.)

P559 (term_real. -> term_int. slashsym fact_real.)

P560 (term_real. -> term_int. slashsym fact_int.)

P561 (term_bool. -> term_bool. andsym fact_bool.)

P562 (sexp_int. -> sexp_int. addop_R. term_int.)

P563 (sexp_real. -> sexp_real. addop_R. term_real.)

P564 (sexp_real. -> sexp_int. addop_R. term_real.)

P565 (sexp_real. -> sexp_real. addop_R. term_int.)

```
P566 (sexp_bool. -> sexp_bool. orsym term_bool.)
P567 (exp_bool. -> sexp_T. relop_R. sexp_T.)
P568 (exp_bool. -> sexp_int. relop_R. sexp_real.)
P569 (exp_bool. -> sexp_real. relop_R. sexp_int.)
P570 (exp_bool. -> sexp_T. insym sexp_set_T.)
P571 (exp_bool. -> sexp_T. insym lsqsym rsqsym)
P572 (relop_equ. -> eqsym)
P573 (relop_neq. -> nesym)
P574 (relop_les. -> ltsym)
P575 (relop_leq. -> lesym)
P576 (relop_grt. -> gtsym)
P577 (relop_geq. -> gesym)
P578 (addop_ad. -> plussym)
P579 (addop_sb. -> minussym)
```

Productions to Recognise Record Structures

```
P580 (depth_$. ->)
P581 (recdec -> recordsym depth_D.
P582    [add_action,(atype_:-r|rec. -> endrec),
P583       [add_production,'(endrec -> recdec fld endsym)]]
P584    [delete_production,(endrec -> recdec fld endsym)]
P585    [add_production,(endrec -> recdec fld endsym)]
P586    [add_action,(endrec -> recdec fld endsym),
P587       [add_production,'(recid -> recname)]]
P588    [delete_production,(recid -> recname)]
P589    [continue]
P590    [add_production,(recid -> recname)]
```

```
P591    [add_action,(endrec -> recdec fld endsym),
P592       [add_production,'(legal_X. -> recfld_X.)]]
P593    [delete_production,(legal_X. -> recfld_X.)]
P594    [continue]
P595    [add_production,(legal_X. -> recfld_X.)]
P596    [delete_production,(depth_D. -> )]
P597    [add_production,(depth_$D. -> )]
P598    [add_action,(endrec -> recdec fld endsym),
P599       [delete_production,(depth_$D. -> )]]
P600    [add_action,(endrec -> recdec fld endsym),
P601       [add_production,(depth_D. -> )]])

P602 (endrec -> recdec fld endsym)
P603 (atype_:-r|rec. -> endrec
P604    [add_production,
P605       (recname -> legal_:rec.
P606          [add_production,'(legal_X. -> recfld_X.)])]
P607    [delete_production,(legal_X. -> recfld_X.)]
P608    [add_action,
P609       (recname -> legal_:rec.),
P610          [add_production,'(recid -> recname)]]
P611    [delete_production,(recid -> recname)])
P612 (fld -> rfields variant)
P613 (rfields -> field)
P614 (rfields -> rfields semicolonsym field)
P615 (variant -> )
P616 (rlist_X|LST_|T.D. ->
P617    idlist_LEV_LST. colonsym atype_X|T. depth_D.)
```

```
P618 (rlistNT:-¦LST¦T.D. -> rlistNT:-¦N_LST¦T.D.
P619    [add_action,(single_N_D. -> ),[continue]]
P620    [delete_production,(goal -> programme)]
P621    [continue]
P622    [add_production,(single_N_D. -> )]
P623    [add_action,(endrec -> recdec fld endsym),
P624       [delete_production,(single_N_D. -> )]]
P625    [add_action,(recid -> recname),
P626       [add_production,
P627          (recfldNT:-¦T. -> recid spotsym ident_N.)]]
P628    [add_action,(legal_X. -> recfld_X.),
P629       [delete_production,
P630          (recfldNT:-¦T. -> recid spotsym ident_N.)]])
P631 (field -> rlistNT:-¦N_¦T.D.
P632    [add_action,(single_N_D. -> ),[continue]]
P633    [delete_production,(goal -> programme)]
P634    [continue]
P635    [add_production,(single_N_D. -> )]
P636    [add_action,(endrec -> recdec fld endsym),
P637       [delete_production,(single_N_D. -> )]]
P638    [add_action,(recid -> recname),
P639       [add_production,
P640          (recfldNT:-¦T. -> recid spotsym ident_N.)]]
P641    [add_action,(legal_X. -> recfld_X.),
P642       [delete_production,
P643          (recfldNT:-¦T. -> recid spotsym ident_N.)]])
P644 (rlistNT:-r¦LST¦T.D. -> rlistNT:-r¦N_LST¦T.D.
P645    [add_action,(single_N_D. -> ),[continue]]
```

```
P646    [delete_production,(goal -> programme)]
P647    [continue]
P648    [add_production,(single_N_D. -> )]
P649    [add_action,(endrec -> recdec fld endsym),
P650       [delete_production,(single_N_D. -> )]]
P651    [add_action,(recid -> recname),
P652       [add_production,
P653           (recfldNT:-|T. -> recid spotsym ident_N.
P654              [add_production,
P655                 '(recname -> legal_:rec.)])]]
P656    [add_action,(legal_X. -> recfld_X.),
P657       [delete_production,
P658          (recfldNT:-|T. -> recid spotsym ident_N.)]])
P659 (field -> rlistNT:-r|N_|T.D.
P660    [add_action,(single_N_D. -> ),[continue]]
P661    [delete_production,(goal -> programme)]
P662    [continue]
P663    [add_production,(single_N_D. -> )]
P664    [add_action,(endrec -> recdec fld endsym),
P665       [delete_production,(single_N_D. -> )]]
P666    [add_action,(recid -> recname),
P667       [add_production,
P668           (recfldNT:-|T. -> recid spotsym ident_N.
P669              [add_production,
P670                 '(recname -> legal_:rec.)])]]
P671    [delete_production,(recname -> legal_:rec.)]
P672    [add_action,(legal_X. -> recfld_X.),
P673       [delete_production,
```

```
P674          (recfldNT:-¦T. -> recid spotsym ident_N.)]])
P675 (variant -> semicolonsym casesym nname_N_L. colonsym
P676    setype_T. ofsym vfld_T. depth_D.
P677    [add_action,(single_N_D. -> ),[continue]]
P678    [delete_production,(goal -> programme)]
P679    [continue]
P680    [add_production,(single_N_D. -> )]
P681    [add_action,(endrec -> recdec fld endsym),
P682       [delete_production,(single_N_D. -> )]]
P683    [add_action,(recid -> recname),
P684       [add_production,
P685          (recfld_T:-¦T. -> recid spotsym ident_N.
P686             [add_production,
P687                '(recname -> legal_:rec.)])]]
P688    [add_action,(legal_X. -> recfld_X.),
P689       [delete_production,
P690          (recfld_T:-¦T. -> recid spotsym ident_N.)]])
P691 (vfld_T. -> vfld_T. semicolonsym vfield_T.)
P692 (vfld_T. -> vfield_T.)
P693 (vfield_T. -> dconstlist_T. colonsym lparensym
P694    fld rparensym)
P695 (dconstlist_T. -> constt_T-V. depth_D.
P696    [add_action,(double_T_D. -> ),[continue]]
P697    [delete_production,(goal -> programme)]
P698    [continue]
P699    [add_production,(double_T_D. -> )]
P700    [add_action,(endrec -> recdec fld endsym),
P701       [delete_production,(double_T_D. -> )]])
```

```
P702 (dconstlist_T. ->
P703    dconstlist_T. commasym constt_T-V. depth_D.
P704    [add_action,(double_T_D. -> ),[continue]]
P705    [delete_production,(goal -> programme)]
P706    [continue]
P707    [add_production,(double_T_D. -> )]
P708    [add_action,(endrec -> recdec fld endsym),
P709       [delete_production,(double_T_D. -> )]])
```

Miscellaneous Productions

```
P710 (nname_N_L. -> ident_N. level_L.)
P772 (idlist_L_NME. -> nname_NME_L.)
P773 (idlist_L_LST_NME. ->
P774    idlist_L_LST. commasym ident_NME.)
P775 (name_N_$$L. -> ident_N. level_L.)
P776 (constpart -> )
P777 (constt_X. -> constant_X.)
```

BIBLIOGRAPHY

[1] Abraham,S.; 'Some Questions of Phrase Structure Grammars'. Computational Linguistics 4(1965)61-70.

[2] Aho,A.V.; 'Indexed Grammars - An Extension of Context Free Grammars'. Ph.D. Thesis, Princeton University 1967.

[3] Aho,A.V.; 'Indexed Grammars - An Extension of Context Free Grammars'. Journal A.C.M. 15(1968)4:647-671.

[4] Aho,A.V. and Ullman,J.D.; 'Principles of Compiler Design'. Addison Wesley 1978.

[5] Altman,E. and Banerji,R.; 'Some Problems of Finite Representability'. Information and Control 8(1965)251-263.

[6] Chomsky,N.; 'Three Models for the Description of Language'. IRE Transactions on Information Theory IT-2(1956)3:113-124.

[7] Cleaveland,J. and Uzgalis,R; 'Grammars for Programming Language'. Elsevier Computer Science Library, Programming Language Series vol 4. Elsevier North-Holland 1977.

[8] Dahl,O.J.,et al; 'Simula - An Algol Based Simulation Language'. Comm. A.C.M. 9(1966)671-678.

[9] Dahl,O.J., Dijkstra,E.W. and Hoare,C.A.R.; 'Structured Programming' Academic Press 1972.

[10] Eriksen,J.H.; 'The BOBS-System'. Tech. Report DAIMI PB-71, Computer Science Dept., Aarhus Denmark. Feb. 1976.

[11] Fischer,M.C.; 'Grammars with Macro-like Productions'. Proceedings of the 9th Annual IEEE Symposium on Switching and Automata Theory (1968)131-142.

[12] Floyd,R.W.; 'Assigning Meaning to Programs'. Proceedings of Symposium in Applied Mathematics, vol. 19 'Mathematical Aspects of Computer Science'. 1967.

[13] Fris,J.; 'Grammars with Partial Orderings of the Rules'. Information and Control 12(1968)415-425.

[14] Georgeff,M.P.; 'Interdependent Translations'. J. Computer and System Science 22(1981)198-219.

[15] Ginsburg,S. and Rounds,E.M.; 'Dynamic Syntax Specifications Using Dynamic Grammar Forms'. I.E.E.E. Trans. Software Engineering SE-4(1978)1,44-55.

[16] Greibach,S. and Hopcroft,J.; 'Scattered Context Grammars'. J. Computer and System Science 3(1969)233-247.

[17] Hanford,K.V. and Jones,C.B.; 'Dynamic Syntax: A Concept for the definition of the Syntax of Programming Languages'. IBM Technical Report TR 12.090, 1971.

[18] Hoare,C.A.R.; 'An Axiomatic Approach to Computer Programming'. Comm. A.C.M. 12(1969)10,576-580.

[19] Hoare,C.A.R. and Lauer,P.E.; 'Consistent and Complementary Definitions of the Semantics of Programming Languages'. Acta Informatica 3(1974)135-153.

[20] Ibarra,O.; 'Simple Matrix Languages'. Information and Control 17(1970)359-394.

[21] Jensen,K. and Wirth,N.; 'Pacsal User Manual and Report'. Springer Verlag 1974.

[22] Jesperson,P., Madsen,M. and Rils,H.; 'NEATS - New Extended Attribute Translation System'. Tech. Report Computer Science Dept. Aarhus University, Denmark.

[23] Johnson,S.C.; 'YACC: Yet Another Compiler Compiler'. Tech. Report CSTR 32, Bell Laboratories Murray Hill NJ July 1975.

[24] Kasi,T.; 'An Hierarchy Between Context Free and Context Sensitive Languages'. J. Computer and System Science 4(1970)492-508.

[25] Knuth,D.E.; 'The Remaining Trouble Spots in Algol 60'. Comm. A.C.M. 10(1967)10:611-618.

[26] Knuth,D.E.; 'Semantics of Context Free Languages'. Mathematical Systems Theory 2(1968)127-145.

[27] Koster,C.H.A.; 'Affix Grammars' in 'Algol 68 Implementation', North-Holland 1971.

[28] Koster,C.H.A.; 'Using the CDL Compiler-Compiler' in 'Compiler Construction: An Advanced Course' Bauer,F.L. and Eickel,J. (eds). Springer Verlag 1976.

[29] Landin,P.J.; 'The Mechanical Evaluation of Expressions'. Computer Journal 6(1964)308-320.

[30] Ledgard.H.F.; 'Production Systems: Or Can We Do Better Than BNF?' Comm. A.C.M. 17(1974)2,94-102.

[31] Ledgard,H.F.; 'Production Systems: A Notation for Defining Syntax and Translation'. I.E.E.E. Trans. Software Engineering SE-3(1977)2,105-124.

[32] Ledgard,H.F. and Donovan,J.J.; 'A formal System for the Specification of Syntax and Translation of Programming Languages' in Proceedings of AFIPS JFCC 31(1971)553-569.

[33] Lewis,P.M. and Stearns,R.E.; 'Syntax Directed Transduction'. Journal A.C.M. 15(1968)3:465-488.

[34] Lucas,P.; 'On the Formalization of Programming Languages' in 'The Veinna Development Method: The Meta-Language' Bjorner,D. and Jones,C.B. (eds). Lecture Notes in Computer Science vol. 61, Springer Verlag 1978.

[35] McCarthy,J.; 'Recursive Functions of Symbolic Expressions and their Computation by Machine'. Comm. A.C.M. 3(1960)4,184-195.

[36] McCarthy,J.; 'A Basis for the Mathematical Theory of Computation' in 'Computer Programs and Formal Systems' Braffort,P. and Hirschberg,D. (eds) 35-69. North-Holland 1963.

[37] Marcotty,M., Ledgard,H.F. and Bochman,C.V.; 'A Sampler of Formal Definitions'. Computer Surveys 8(1976)191-275.

[38] Marlin,C.D.; 'Coroutines. A Programming Methodology, a Language Design and an Implementation'. Lecture Notes in Computer Science vol. 95, Springer Verlag 1980.

[39] Morris,C.; 'Signs, Language and Behaviour'. Braziller 1955.

[40] Mosses,P.D.; 'SIS - Semantics Implementation System: Definition, Manual and User Guide'. Tech. Report DAIMI MD-30 Computer Science Dept., Aarhus University Denmark. 1979.

[41] Naur,P. et al; 'Revised Report on the Algorithmic Language Algol 60'. Comm. A.C.M. 6(1963)1:1-17.

[42] Nori,K.V. et al; 'The Pascal P Compiler - Implementation Notes (Revised Edition)'. Breicte Nr. 10, Institute fuer Informatik, Eidgenoessiche Technishe Hochschule, Zurich 1976.

[43] Pagan,F.G.; 'Semantic Specification Using Two Level Grammars: Blocks, Procedures and Parameters'. Computer Languages 4(1979)171-185.

[44] Partridge,D.; 'Dynamic Production Grammars'. Unpublished Manuscript 1981.

[45] Paulson,L.C.; 'A Compiler Generator for Semantic Grammars'. Ph.D. Thesis Stanford University 1982.

[46] Poole,P.C.; 'Portable and Adaptable Compilers' in 'Compiler Construction: An Advanced Course' Bauer,F.L. and Eickel,J. (eds). Springer Verlag 1976.

[47] Post,E.L.; 'Formal Reductions of the General Combinatorical Decision Problems'. American J. Mathematics 65(1943)197-215.

[48] Raihi,K.J. et al; 'The Compiler Writing System HLP'. Tehc. Report A-1978-2 Dept. of Computer Science, University of Helsinki, Finland 1978.

[49] Rosenkrantz,D.J.; 'Programmed Languages and Classes of Formal Languages'. Journal A.C.M. 16(1969)1:107-131.

[50] Rozenberg,G.; 'Direction Controlled Programmed Grammars'. Acta Informatica 1(1972)242-252.

[51] Sale,A.H.J.; 'Pascal Compatibility Report'. Tech. Report R77-5 Dept. of Information Science, University of Tasmania, Australia, 1977.

[52] Salomaa,A.; 'Periodically Time Variant Grammars'. Information and Control 17(1970)294-311.

[53] Scott,D. and Strachey,C.; 'Towards a Mathematical Semantics for Computer Languages' in 'Computers and

Automata' Fox,J. (ed) 19-40. John Wiley 1972.

[54] Sippu, S. and Soilsalon-Soininen,E.; 'Practical Error Recovery in LR Parsing' in Conference Record of the 9th ACM Symposium on the Principles of Programming Languages (1982)177-184.

[55] Smullyan,R.M.; 'Theory of Formal Systems'. Annals of Mathematical Studies No.47'. Princeton University Press 1961.

[56] Stearns,R.E. and Lewis,P.M.; 'Property Grammars and Table Machines'. Information and Control 14(1969)524-549.

[57] Vaishnavi,V.K. and Basi,S.K.; 'On Coupled Grammars and Languages and Translations'. Int. J. Computing Mathematics Series A 6(1977)33-54.

[58] van Wijngaarden,A. et al; 'Revised Report on the Algorithmic Language Algol 68'. Springer Verlag 1976.

[59] van der Walt,A.P.J.; 'Random Context Grammars' in 'Information Processing 71' 66-68. North-Holland 1972.

[60] Wegner,P.; 'The Vienna Definition Language'. Computer Surveys 4(1972)5-63.

[61] Williams,M.H.; 'A Formal Notation for Static Semantics'. Computer Languages 5(1980)37-55.

[62] Williams,M.H.; 'The Programming Language BPL'. Computer Journal 25(1982)3:289-306.

[63] Wirth,N.; 'Modula-2'. Breicte Nr. 36, Institute fuer Informatik, Eidgenoessiche Technishe Hochschule, Zurich 1980.