COSC460

RESEARCH PROJECT

DEPARTMENT OF COMPUTER SCIENCE

UNIVERSITY OF CANTERBURY

A NOTATION FOR EXPRESSING RESTRICTING CONDITIONS ON A CONTEXT-FREE GRAMMAR

S. A. LINES

INDEX

I INTRODUCTION

Context-Free grammars have several features which make them suitable for defining programming languages. For instance, such grammars are: "recursive, have relatively simple and well understood recognizers, and a readable notation"[9]. However, no Context-Free grammar can completely define a language such as ALGOL 60. In the REVISED REPORT ON THE ALGORITHMIC LANGUAGE ALGOL 60[6] the language is defined by a Context-Free grammar and a set of restricting conditions. The main characteristic of all these restrictions is shown in Appendix A to be that two substrings in a sentence in the language must be either identical or different.

By defining an Extended Context-Free Grammar based on the Context-Free grammar of [6], and having additionally a facility for syntactically expressing this class of restricting conditions, we endeavour to completely define a programming language by the Extended Context-Free Grammar. We take the view presented by Strachey[11] that "it is the business of the syntax ... To determine if any particular piece of text is a program in the language ... And that of the semantics to ... Determine what the outcome of any well-formed program should be.". We believe that our Extended Context-Free Grammar fully defines the syntax of ALGOL 60.

The basis of the Extended Context-Free grammar remains the original Context-Free grammar. Thus many algorithms developed for parsing Context-Free languages can be used as the basis of any parser for an Extended Context-Free language.

II BACKGROUND

The Backus Naur Form metalanguage has become widely used for defining and describing the syntax of programming languages. However, BNF is restricted to expressing the class of Chomsky Context-Free languages and it seems unlikely that any non-trival programming language would by definable completely by a Context-Free grammar. For example, Floyd[4] showed many years ago that it was not possible to state all the "formation rules" of ALGOL 60 as a Phrase Structure (ie context-free) grammar, "so that there must necessarily be syntactic rules stated in other ways". These "rules" are normally given by "informally stated restrictions". Shortly after, Carracciolo di Torino[2] noted that in the ALGOL 60 Report "...many syntactical rules which remain unformalised ...have been given in the explanations and comments accompanying the metasyntactical formulas" (ie BNF rules). He pointed out the "strong context dependent character imposed (on ALGOL 60) by the unformalised restricting conditions".

Floyd[5] later suggested "any rule requiring that two or more constituent phrases of a construction be identical (or different) is almost certainly beyond the scope of Phrase Structure definition". Post's Correspondence Problem[1] shows that the question of whether a Turing Machine can say that two random sequences in a string are identical is recursively unsolvable. ALGOL 60 has exactly the requirement that an identifer must be declared before it is used, and it seems likely that no finite Phrase-Structure grammar could be written for ALGOL 60.

Attempts have been made previously to define completely the syntax of programming languages, in particular, ALGOL 60. The first example is, of course, ALGOL 60 whose authors defined a Context-Free grammar and added further restrictions to it to completely define the

language. In this system valid ALGOL programs form a subset of the language defined by the BNF rules.

In 1963, Carracciolo di Forino[2] suggested a method for dynamically defining ALGOL. He considered "an ALGOL program as a block (or more accurately a nest of blocks) which are a mixture of two types of strings, declarative strings (including label attachments) which represent local linguistic conventions giving rise to a local set of productions and sentential strings belonging to locally defined Context-Free languages". Thus entry to a block adds a new set of productions for defining identifiers and the statements of the block form a Context-Free language whose grammar is the set of productions locally valid for that block.

Other researchers have developed systems completely distinct from BNF. For example, a Canonic System[3] "has the capability to cross-reference between elements of the sentence structure that it generates". Canonic Systems define Context-Sensitive languages, but they are so powerful that they introduce undecidability problems.

Recognising the limitations of Context-Free grammars, the authors of ALGOL 68 have introduced the concept of two-level grammars and define the new language ALGOL 68 by this method.

Other examples of syntactic description languages can be found in [2].

III PRELIMINARY INVESTIGATION

The aim of this research was to develop an extension to BNF that would enable it to express syntactic relationships currently beyond its power. The language ALGOL 60 and syntax-directed and table-driven compilers suggested one characteristic reason for the inadequacies of Context-Free grammars. This is the inability of such a grammar to express the necessity that two phrases in a sentence should be either identical or different. We examined the restricting conditions imposed upon the Context-Free grammar used to define ALGOL 60 in [6] to test the validity of this proposition. The restrictions are all stated as semantics, comments or restrictions and are detailed in Appendix A, part I.

All the restricting conditions applied to the Context-Free grammar are based on comparing two or more phrases of the sentence (program) and the context in which they appear. For example, quoting from [6]: "The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations", can be expressed as: Each identifier declared may not be identical to any other identifier declared in the same block head. All restrictions can be similarly expressed, and are detailed in Appendix A part II.

We can augment an automatic parser with a procedure to compare two phrases of a sentence (program). With the syntax of the language indicating which restrictions must be applied, and when, to the sentences, we can develop automatic parsing procedures that do not require many tests of syntactic features to be included in the semantic routines.

Two concepts prove useful in the development of the Extended Context-Free grammar:

1) The occurance of a production in a parse indicates when a particular type of event has occured in the program (eg a declaration) and

2) A program or a subset of a program is uniquely defined by the sequence of production numbers that result from a parse of the program. In particular, a non-terminal is uniquely defined by both the sequence of terminal symbols and the sequence of production numbers.

For example, <identifier>'s can be defined by the followins productions:

1 <identifier>::=<identifier><letter>

2 <identifier>::=<letter>

3,4,5,.. <letter>::= A ! B ! C ...

Thus an <identifier> such as ABC is parsed - 3,2,4,1,5,1 - and no other identifier has this parse sequence. Further a non-terminal may be uniquely defined by only a subset of the production numbers from the parse, eg 3,4,5.

IV THE EXTENDED CONTEXT-FREE GRAMMAR

Let a Context-Free grammar be defined as follows:

DEFINITION 1

A Context-Free grammar, G, is a 4-tuple

$$G=(N,V,P,S)$$

where

N is a finite set of nonterminal symbols

V is a finite set of terminal symbols, disjoint from N

P is a finite set of productions of the form

$$A::=\beta,\ A\in N,\ \beta\in(N\cup V)^*$$

S is the start symbol.

We also define a set of restrictions over the productions of a Context-Free grammar.

DEFINITION 2

Let L_p be a finite set of labels attached to the productions of a Context-Free grammar, so that a production has the form

$$\delta:A::=\beta,\ \delta\in L_p,\ A\in N,\ \beta\in(N\cup V)^*$$

Let L_r be a finite set of labels which shall be attached to the restrictions

Let T be a finite set of 'types' of productions. We shall class a production as one of three types, ie declarations, statements, and block ends.

We define R, a finite set of restrictions of the form:

$$\delta:A\left\{\begin{matrix}=\\ \neq\end{matrix}\right\}\ \langle\langle C\rangle\rangle\ t,\ \delta\in L_r,\ A\in N,\ C\in(L_p\times(N\cup V))^*,\ t\in T,$$

where

δ is the 'restriction label'

A is an element from the right hand side of a production, the

'Applicant Production'

C is a finite set of 'Comparision Elements', connected by a logical operator (∧,∨). Each element of C consists of a 'Lookfor Label' and a 'Comparing Element'. The 'Comparing Element' is an element from the right hand side of a production (the 'Comparing Production') with the label='Lookfor Label'.

$\{=, \neq\}$ Is the 'relationship' between the 'Applicant Production' and the 'Comparing Production's,

with = the 'equivalent' relationship,

and ≠ the 'not equivalent' relationship.

t is the type of the applicant production.

We can now define our Extended Context-Free grammar

DEFINITION 3

Given a Context-Free grammar,G, and a set of restricting conditions on G which can be expressed as a set of RESTRICTIONS (as defined in Definition 2 above), we define our Extended Context-Free grammar, G , as a 4-tuple

$$G = (N, V, F, S)$$

where

N is a finite set of nonterminal symbols

V is a finite set of terminal symbols, disJoint from N

F is a finite set of formation rules

$$F = (P \cup R)$$

Where P is a finite set of Context-Free productions of the form

$$\gamma : A ::= \beta,\ A \in N,\ \gamma \in (L_p \times (L_r \cup e^+)),\ \beta \in (N \cup V)^*$$

And R is a finite set of restrictions over P of the form

$$\delta : A \{=, \neq\} \langle\langle C \rangle\rangle\ t,\ \delta \in L_r,\ A \in N,\ C \in (L_p \times (N \cup V))^*,\ t \in T$$

S is the start symbol,

$^+$ e is the empty symbol

The Extended Context-Free grammar distinguishes a set of Context-Free productions and a set of restrictions in a form as similar as possible to that in which a Context-Free grammar is expressed. The set of Context-Free productions remains essentially identical to the productions of the Context-Free grammar. We add labels which are specifically defined in the grammar as an important part of the productions.

The greatest difference between a Context-Free grammar and an Extended Context-Free grammar is the set of restrictions. The need for a restriction to apply is recognised as a result of a particular production being applied during a parse of a sentence. It may be necessary to add nonterminals and some single productions to an existing Context-Free grammar so that the need for a restriction can be uniquely and accurately defined.

For instance, the grammar of Appendix B has the productions

11 <call statement>::= call <identifier>

28 <procedure heading>::= procedure <identifier>

There is also a restricting condition applying to procedure calls that says: "An identifier used in a call statement must be declared a 'procedure'." Productions 11 & 28 are related by this restriction.

It is the task of the restriction to express the relationship between the two productions. We let the restriction have a label "3" (in keeping with Appendix B) and indicate in production 11 that restriction 3 must apply by including this label in the production in the following manner.

11/3: <call statement>::= call <identifier>

Restriction 3 states that the identifier which has been found, on parsing, to occur in production 11 must be identical to an identifier which had been found to occur in production 28.

We express this in the form:

: <identifier> = << 28:<identifier> >>

estriction 3 requires that the two identifiers be identical. If a

estriction requires that the element of the applicant production be ifferent from any other (as occurs on a declaration) then the

elationship would be "not equivalent to"(≠). A restriction may apply o more than one production provided that the element from each pplicant production is the same. A restriction may compare an pplicant element to several comparison elements. We place all the omparison elements inside the symbols "<< >>". An element may need o be compared against one or all of the elements thus enclosed in the racketing symbols . We use the boolean operators and(∧) & or(∨) to how this. Examples of these events are given in Appendix B.

It has been observed, however, that if the relationship is equivalent to" (=) the 'or' (∨) operator invariably applies and if he relationship is "not equivalent to" (≠) the 'and' (∧) operator nvariably applies. Since no evidence can be given to show that this ill be true for all restrictions on all grammars the alternatives emain expressed in the general form of the restrictions.

The 'type' of a restriction is a useful indication for avoiding nnecessary complexity in the parser. All productions that appear as omparision elements of a restriction are of type 'declaration', roductions which make certain 'declarations' unavailable for further se are of type 'block end' and all others are 'statements'.

heoretical Aspects

We believe that an Extended Context-Free grammar will completely efine the language ALGOL 60. This language cannot be completely efined by a Context-Free grammar and no proof has yet been given as o whether ALGOL 60 can or cannot be defined by a Context-Sensitive rammar or an Unrestricted grammar. We therefore have difficulty in lacing the Extended Context-Free grammar in relation to the Chomsky ierarchy.

An Extended Context-Free grammar without any restrictions is quivalent to a Context-Free grammar. Thus the Context-Free grammars re a subset of the Extended Context-Free grammars. We suggest that xtended Context-Sensitive grammars, Extended Unrestricted grammars nd Extended Regular grammars could also exist. The set of extended rammars being related to the Chomsky grammars by being based on the he respective Chomsky grammar and including a set of restrictions. he Chomsky grammar would be a subset of the respective Extended rammar and the Extended grammars would be a hierarcy within hemselves.

PARSING STRINGS OF LANGUAGES DEFINED BY AN EXTENDED CONTEXT-FREE GRAMMAR

The definition of the Extended Context-Free grammar has several implications to a parsing algorithm for languages defined by an extended Context-Free grammar. Obviously some mechanism is required for comparing two strings in the grammar and some method for identifying the strings that need to be compared. It is at this stage that the second concept mentioned in Section III becomes useful. If an identifier can be uniquely defined by the production numbers output as a result of parsing it, then the output of the parser can be used for comparisons as successfully as could the terminal symbols composing the identifier. Further, the output of the parser is already stored by the compiler for the generation of machine or assembly code. However, for any non-trival programming language the number of rules generated by a parser is exceedingly large. To overcome this problem we introduce the concept of the Augmented Restrictions. Augmented Restrictions are those which include an extra, "dummy" restriction. This "dummy" restriction serves to define those productions whose occurrence would not otherwise call a restriction but which are required by other restrictions during some comparison stage.

For example, in Appendix B productions number 3,14,15,16,17,18,19 call the "dummy" restriction. It should be noted that these productions all define <identifier>'s and that it is only the <identifier>'s that are compared in the restrictions of this grammar. The use of the "dummy" restrictions avoids unnecessary complexity in the parser for the grammar.

Conceptually, the parser functions on two levels. The first level can be any of the automatic bottom-up parsers for a Context-Free language. The second level is a restrictions parser. The algorithms

or table generation and parsing suggested below are for an Extended R(k) parser. They give only an outline of a possible method for implementing a parser and do not encompass the full potential of an extended Context-Free grammar.

able-Generation

We have not implemented, on a computer, the table generation rocedures given below. The hand generation of the tables in Appendix , for the grammar of Appendix B, suggested Algorithm 1.

We assume an LR(1) table generator already exists and is based on lgorithms 5.8, 5.9, 5.11 of Aho & Ullman[1]. We modify the existing able generator to create two extra tables. The 'r' table, the elating table, contains in the i^{th} element the restriction label ttached to the production which has the production label i. For xample see table 1 Appendix C. We also create a table, the 'a' table f applied restrictions for the restrictions parser by Algorithm 1 elow. For example see Table 2, Appendix C.

We have altered the definition of LR(k) tables associated with he items of a grammar G to be: (ref. Aho & Ullman pp 392)

EFINITION 4

Let G be an Extended Context-Free grammar and let S be a ollection of sets of LR(k) items for G. T(c), the LR(k) table ssociated with the set of items c in S, is a 4-tuple of functions f,g,r,a>. 'f' is called the "parsing action function", 'g' the "go o function", 'r' the "relating function", and 'a' the "applied estriction function".

1 defines the 'f' function as in (1) in Aho & Ullman pp392

2 defines the 'g' function as in (2) in Aho & Ullman pp 392

3 'r' the relating function specifies the restrictions that ust apply to the productions. r(i)= the restriction that must apply o production I in the grammar if a restriction exists otherwise

(i)=empty.

4 'a' , the applied restriction function is a table of data or the restrictions parser. 'a' specifies the elements that have to e compared and the action to be taken on finding a successful omparsion.

We extend the table generation procedure of Aho & Ullman to nclude table generation for the restrictions of the grammar. lgorithm 1 given below would be applied to the grammar upon ompletion of the algorithm 5.11.

lgorithm 1

estrictions table generator

nput: Restriction

utput: Table of 'a' functions

ethod:

) For every restriction in the grammar do 2 to 6 below.

2) scan for "restriction label" (rlabel)

3) scan for element of the applicant production, if found enter it in table a/n at a/n[rlabel,0]

4) scan for relationship

(a) is it "="? Yes: success[rlabel]=correct
fail[rlabel]=error
endofstack[rlabel]=error

(b) is it "≠"? Yes: success[rlabel]=error
fail[rlabel]=rlabel
endofstack[rlabel]=correct

5) while comparing productions still exist enter label in a/lookfor[0,rlabel,*] and element in a/lookfor[1,rlabel,*]

6) scan for production "type"

(a) is it "declaration"? Yes: store[rlabel]=save

(b) is it "statement"? Yes: store[rlabel]=remove

(c) is it "block end"? Yes: store[rlabel]=unstack

(d) if non of these then store[rlabel]=stack

7) for every distinct nonterminal in a/n and a/lookfor, parse the nonterminal "top-down" entering each parse sequence in a table parse sequence and indexing to the parse sequence in a/n and a/lookfor.

Parsing

As stated in the introduction, a parser for an Extended Context-ree language requires a procedure for comparing two subsets of its utput strings. We distinguish the parser as two parts. A Context-ree LR(1) parser which is a slightly modified version of algorithm .7 of Aho & Ullman, and a restrictions parser.

odification of Algorithm 5.7

R(k) Parsing Algorithm

nput: = input of algorithm 5.7

utput: = output of algorithm 5.7

ethod: perform steps 1 & 2 until acceptance occurs or an error is ncountered. If acceptance occurs, the string in the output buffer is he right parse of z.

(1) = (1) of algorithm 5.7)6 (2) the parsing action function 'f' of the table on top of the pushdown list is applied to the lookahead string u

(a) = 2(a) of algorithm 5.7

(b) if f(u)= reduce i then

if r(i) is not equal to no apply the restriction (algorithm 2 below can be used here)

if RESULT is not error then

if production i is A->β then 2|β| symbols are removed from

the top of the pushdown list and production number i is placed in the output buffer. A new table T' is then exposed as the top table of the pushdown list, and the goto function of T' is applied to A to determine the next table to be placed on top of the pushdown list. We place A and this new table on top of the pushdown list and return to step (1).

(c) = 2(c) of algorithm 5.7

(d) = 2(d) of algorithm 5.7

The restrictions parser works on the output of the context-free parser. By using augmented restrictions we can select the particular production that are involved in restrictions and therefore simplify and accelerate the parser. We present here a parser which operates on stack structure. Algorithm 3 is a procedure for comparing two sets of stack elements and Algorithm 2 is the restrictions parser, that is entered from 2(b) above.

Algorithm 2

Restrictions parser

Input: Table of 'a' functions, production label of the applicant production.

Output: Correct or Error

Method:

1) If r(i) > 0 do — Initialize Result

2) initialise a stack pointer c to the top of stack ~~position~~ pointer

3) While RESULT is not (correct or error) and stack pointer c Is not the last stack element do

(i) 4) set the next stack element

(ii) 5) is this element in a/lookfor[/,r(i),*]

yes: compare stack pointer c and stack pointer a (using algorithm 3)

no: decrement stack pointer c.

) if store[r(i)]=save then increment pointer a and stack i, reset top of stack pointer to pointer a.

if store[r(i)]=remove then decrement pointer a to top of stack pointer.

if store[r(i)]=unstack then decrement pointer a and top of stack pointer to pointer c.

if store[r(i)]=stack then increment pointer a and stack i.

) return RESULT to main parser

lsorithm 3

omparision Algorithm

nput: Restrictions stack and two pointers

utput: RESULT

ethod:

.) While stack pointer a is an element of the parse sequence of the applicant element and stack pointer c is an element of the parse sequence of the comparing element and if the applicant element = the comparing element then pointer a = pointer c then decrement both pointers.

:) if neither stack pointer c is in the parse sequence of the comparing element nor stack pointer is in the parse sequence of th applicant element then RESULT= success otherwise RESULT = r(i) = fail.

I CONCLUSIONS

We presented the Extended Context-Free grammars as a particular olution to the problems that arise from defining programming anguages by Context-Free grammars. The extended Context-Free grammar s quite simply a Context-Free grammar that has the facility of xpressing restricting conditions in a risorous notation. The grammar nd notation presented here can express the restricting conditions on LGOL 60. It will not, in its present form handle a condition such as ppears in BASIC (that statement numbers must be sequentially ncreasing throughout the program), however the method is extendible.

An Extended Context-Free grammar could completely define the ntax of ALGOL 60. The grammar thus defines languages that cannot be efined by Context-Free grammars. The Context-Free grammars form a ubset of the Extended Context-Free grammars. Because an Extended ontext-Free grammar with no restrictions is a Context-Free grammar it eems likely that Extended grammars could be defined over the entire homsky hierarchy.

No automatic process is given for specifying the syntax of a rogramming language in an Extended Context-Free grammar. However ince there is no universally applicable method[1] for specifying the ntax of a programming language in any grammar, this should not etract from the usefulness of the Extended Context-Free grammars.

Adding a restrictions table generator to the already existing R(k) table generator, will cause the table generation process for an xtended Context-Free grammar to be slower than the table generation rocess for a Context-Free grammar. However, the number of estrictions should usually be sufficiently small, and the speed of he restrictions table generator sufficiently fast, to make the verall effect of the restrictions insignificant in comparison with he notoriously slow table generation process for an LR(k) grammar.

We have implemented a parser based on the parsing algorithms of section V. Tests have shown that the addition of a restrictions parser to an LR(1) parser cause a reduction in the speed of an extended LR(1) parser as compared with an LR(1) parser. This is not an insignificant factor , but it is small compared to the total time required to parse a string. Considering a parser as part of a compiler, we expect that an Extended parser will reduce compilation time, because the syntax analysis is completed by the parser and does not require ad hoc checking in the semantics routines. The particular case of on-line interactive compilation of input programs should be greatly enhanced by the use of an Extended parser. Syntactic errors can be detected almost immediately they occur and not after the entire program has been input.

We have given algorithms for including the restrictions in an LR(k) parser, but any bottom-up parser for Context-Free grammars could be altered by analogous methods.

ACKNOWLEDGEMENTS.

I would like to thank Professor J.P.Penny and Mr. G.D.Freeth for their help and advice with all aspects of this research project. I would also like to thank Dr. Z.Ley for helpful suggestions on meta-notation.

BIBLIOGRAPHY

AHO & ULLMAN

THEORY OF PARSING, TRANSLATION AND COMPILING. VOLS 1 & 2

1972 PRENTICE-HALL

CARRACCIOLO DI FORINO

"Some Remarks on the Syntax of Symbolic Programming Languages"

In: CACM 6 [June 1963], pp456-460.

DONAVAN J.J.

SYSTENS PROGRAMMING (Canonic Systems pp240-250)

1972 MCGRAW-HILL

FLOYD R.W.

"On the Nonexistence of a Phrase Structure Grammar for ALGOL 60"

In: CACM 5 [Sept 1962], pp483-484.

FLOYD R.W.

"Syntax of Programming Languages"

In: IEEE Transactions on Electronic Computers 13 [Aug 1964], pp346-353.

NAUR ET AL

" Revised Report on the Algorithmic Language ALGOL 60"

In: CACM 6 [Jan 1963], pp1-17.

STEEL T.B. (ED) IFIP WORKING CONFERENCE ON FORMAL LANGUAGE
ESCRIPTION LANGUAGES FOR COMPUTERS, 1964 VIENNA
966 AMSTERDAM, NORTH-HOLLAND PUBLISHING CO.
ne followins papers:

CARRACCIOLO DI FORINO "On the Concept of Formal Linguistic Systems"

WALK K. "Entropy and Testability of Context-Free Languages "

NAUR P. "Documentation Problems in ALGOL 60"

1 STRACHEY C. "Towards a Formal Semantics"

2 THOMPSON I.J.

" A Comparative Study of Formalisms for Programming Language Definition

Report No 19, June 1975 Massey University Computer Unit

3 WALTERS D.A.

"Deterministic Context-Sensitive Languages"

In: INFORMATION AND CONTROL 17,1970,pp14-61

PPENDIX A

RESTRICTING CONDITIONS ON ALGOL 60

ART 1 : Examples of Syntactic Restrictions on ALGOL 60 Not Expressed n the Syntax[6]

"The same identifier cannot be used to denote two different quantities except when these quantities have disJoint scopes as defined by the declarations." (2.4.3 pp 5)

"Certain identifiers should be reserved for the standard functions of analysis." (3.2.4 pp 6)

"The type of the value of a particular variable is defined in the declaration for the variable itself of for the corresponding array identifier." (3.1.3 pp 6)

"Variables and function designators entered as boolean primaries must be declared BOOLEAN ." (3.4.4 pp 8)

"The type associated with all variables and procedure identifiers of a left part list must be the same." (4.2.4 pp 10)

"If the type (of left parts) is BOOLEAN, the expression must likewise be BOOLEAN." (4.2.4 pp 10)

"If the type (of left parts) is REAL or INTEGER, the expression must be arithmetic." (4.2.4 pp 10)

"Apart from the boolean expression of if clauses the constituents of simple arithmetic expressions must be of types REAL or INTEGER." (3.3.4 pp 7)

"Assignment to a procedure identifier may only occur within the body of a procedure defining the value of the function designator." (4.2.3 pp 10)

) "The actual parameter list of a procedure statement must have the same number of entries as the formal list of the procedure

declaration headins." (4.7.4 pp 12)

1 "The kind and type of each actual parameter must be compatible with the kind and type of the correspondins formal parameter." (4.7.5 pp 12)

2 "A so to statement defines its successor explicitely by the value of a desisnational expression." (4.3.3 pp 10)

3 "No so to statement can lead from outside into a block." (4.3.4 pp 10)

4 "Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable." (3.1.4.1 pp 6)

5 "Apart from labels and formal parameters of procedure declarations....all identifiers of a prosram must be declared." (5 pp 13)

6 "A declaration of an identifier is valid for one block." (5 pp 13)

ART 2 : Expressins Restrictins Conditions on ALGOL 60 as Comparisions etween Two Strinss

Each identifier declared may not be identical to any other identifier declared in the same block head.

Each identifier declared may not be identical to any identifier reserved for the standard functions of analysis.

Each identifier used as a boolean primary must be identical to an identifier that has been declared BOOLEAN.

Each identifier as a left part list must be declared to be the same type as every other identifier in the left part list.

If an identifier in a left part list has been declared boolean the expression must be boolean.

If an identifier in a left part list has been declared real or integer the expression must be arithmetic.

Apart from identifiers in if clauses any identifier in an arithmetic expression must be identical to an identifier that has been declared real or integer.

If the identifier of a left part has been declared a procedure the statement must occur in an uncompleted procedure declaration.

The number of parameters of a procedure statement must be identical to the number of parameters of the procedure declaration.

The identifer in an actual parameter list must have been declared to have a type compatible with the declaration for the corresponding identifier in the formal parameter list.

The identifier in a go to statement must be identical ~~dentical~~ to the label of a statement or a switch identifier.

The label of a statement or the desisnational expressions in a switch list of a switch identifier referenced in a go to statement must be identical to a label that is local to the go to statement.

The number of subscripts of the subscript list of a subscripted variable must be identical to the number of subscripts in the declaration of the array identifier.

Any identifier used in a statement must be identical to an identifier in a declaration.

Any identifier used in a statement must be identical to an identifier in a declaration that is in a block to which the current block is local.

'PENDIX B

AN EXAMPLE GRAMMAR

RT 1 : Context-Free Productions

<s>::=<program>

<program>::=<block> .

<block>::=<declarations> <statement>

<declarations>::=<variable> <procedure> ! <procedure>

<procedure>::=<procedure><procedure heading><procedure body> ! ?

<identifier list>::=<identifier> ! <identifier list>,<identifier>

<statement>::=<assignment statement> ! <call statement>

<call statement> ::= call <identifier>

<assignment statement>::=<left part>::=<expression>

<expression>::=<identifier>

<identifier>::=<identifier> <letter> ! <letter>

<letter>::= a ! s ! d ! s

<expression>::= <integer>

<integer>::=<integer> <digit> ! <digit>

<digit>::= 9 ! 0 ! 1 ! 2

<variable>::= variable <identifier list> ;

<procedure heading>::= procedure <identifier> ;

<prodecure body)::= <block> ;

<left part>::=<identifier>

RT 2 : Restricting Conditions

An identifier may be declared only once.

All identifiers used in assignment statements must be declared 'variable'.

An identifier used in a call statement must be declared a 'procedure'.

Identifiers are local to the blocks in which they are declared.

ART 3 : An Equivalent Extended Context-Free Grammar

roductions

<s>::=<program>

<program>::=<block> .

<block>::=<declarations> <statement>

<declarations>::=<variable> <procedure ! <procedure>

<procedure>::=<procedure><procedure heading><procedure body> ! ?

/1 <identifier list>::=<identifier>

/1 <identifier list>::=<identifier list>,<identifier>

<statement>::=<assignment statement> ! <call statement>

1/3 <call statement>::= call <identifier>

2 <assignment statement>::=<left part>:=<expression>

3/2 <expression>::=<identifier>

4/0 <identifier>::=<identifier><letter>

5/0 <identifier>::=<letter>

6/0 <letter>::= a

7/0 <letter>::= s

8/0 <letter>::= d

9/0 <letter>::= g

0 <expression>::= <integer>

1 <integer>::=<integer> <digit> ! <digit>

3 <digit>::= ! 0 ! 1 ! 2

7 <variable>::= variable <identifier list> ;

8/1 <procedure heading>::= procedure <identifier> ;

9/4 <prodecure body)::= <block> ;

0/2 <left part>::=<identifier>

estrictions

:

: <identifier> ≠ <<7:<identifier>.

```
                    8!<identifier>.
                    28!<identifier> >> declaration
!<identifier> = << 7!<identifier>+
                     8!<identifier> >> statement
!<identifier> = << 28!<identifier> >> statement
!             = << 28 >> block end
```

PENDIX C

TABLES GENERATED FOR THE EXAMPLE GRAMMAR

ble 1: the r function

oduction/restriction	r	production/restriction	r	production/restriction	r
0	-	11	3	21	-
1	-	12	-	22	-
2	-	13	2	23	-
3	-	14	0	24	-
4	-	15	0	25	-
5	-	16	0	26	-
6	-	17	0	27	-
7	1	18	0	28	1
8	1	19	0	29	4
9	-	20	-	30	2
10	-				

ble 2 : the a function

RULE	NONTERM	LOOKFOR	STORE	SUCCESS	FAIL	ENDOFSTACK
0			stack			
1	<identifier>		save	error	1	correct
	<identifier>	7				
	<identifier>	8				
	<identifier>	28				
2	<identifier>		remove	correct	2	error
	<identifier>	7				
	<identifier>	8				
3	<identifier>		remove	correct	3	error
	<identifier>	28				
4		29	unstack	correct	error	error

arse Seauence(<identifier>)=14,15,16,17,18,19

missing rules in table for Parse sequence