GENERATING A RECOGNIZER

FOR A BNF GRAMMAR

BY

JAY EARLEY

Carnegie Institute of Technology
Pittsburgh, Pennsylvania
June 1, 1965

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

This paper describes an algorithm which takes the BNF definition of the grammar of a language and constructs a set of productions for recognizing sentences of the language. These productions, as introduced by Floyd [4] and modified by Evans [3], constitute a language in which may be written a one-pass, one-push-down-stack recognizer, which involves no closed subroutines. Thus it goes directly from the basic definition of the grammar of a programming language to an efficient parser for the language which could easily be incorporated into a compiler.

Production Language*. The production language consists of a set of productions of the form

```
L0      a  b  c |  →              G |          *G1
```

Let us assume we are parsing a sentence by productions. Each time we scan a character, it is placed at the top of a push down stack known as the syntax stack. Then we sequence through the productions, comparing the top few characters in the syntax stack to each production. The first bar in the production represents the top of the stack, and the characters stretching immediately to its left (known as the stack of the production) represent the top few characters of the syntax stack. A σ in a production stack represents any single character in the syntax stack. When a production stack first matches the corresponding topmost characters of the syntax stack, we apply the production to the syntax stack. This transforms these topmost characters into what is represented by the second bar and the characters to its immediate left. If the stack is to be left unchanged, there will be no → or second stack. Then we examine the link of the production to see what to do

* Linguists often use "production" for an alternative of a BNF definition and "reduction" for the parser. With this in mind our terminology should not cause confusion.

next. If there is an * we scan the next character. In any case, we then transfer to the label in the link of the production and start sequencing down the productions again. Any of the productions may have labels.

This production language is the basis for at least one working Algol compiler [3] and also comprises a significant part of FSL, a language for writing compilers [5].

Definitions and Conventions. A BNF grammar consists of a set of BNF definitions. Each definition consists of a non-terminal character followed by ::= and a set of alternatives separated by bars |. The non-terminal is known as the definition of each of its alternatives. The characters in the alternatives may be non-terminals or terminals--actual symbols from the alphabet of the language. Non-terminals will be represented by English capitals, terminals by English lower case, and strings by Greek letters; X, Y, and Z will stand for either terminals or non-terminals. A string β is a substring of α if there exist strings φ and Ψ such that $\alpha = \varphi\beta\Psi$. A substack of a stack is defined analogously. A head of a string is a substring which includes the first character. A tail is a substring which includes the last character. Λ is the empty string. We will also adopt some terminology from Floyd. [1]

We are interested in simple phrase structure grammars, and our BNF is equivalent to a simple phrase structure grammar if we require that there be exactly one non-terminal R which does not appear to the right of ::= in any definition. This character, which we call the root, defines the sentential forms of the grammar. We also require that every character which appears in the BNF must be used in the derivation of a sentence from R. We will only consider unambiguous grammars--that is, grammars in which no sentence has more than one derivation.

The Algorithm. The algorithm which constructs the productions will not work correctly for all allowable grammars. If a grammar contains two or more

alternatives which have a certain relationship, then it will not work. These are called culprit strings. After explaining the algorithm, it will be shown how these can be detected and deleted from the grammar.

Two sets of productions can be constructed corresponding to each character of the grammar. A set of productions is a section of the productions which acts as a unit because only its first production has a label which can be transferred to. The 0-productions for a character X (labeled X0) are constructed so as to appear at the point in the parse where we expect to find an instance of X in the sentence starting with the last symbol scanned. These will be constructed only for non-terminals. The 1-productions for a character X (labeled X1) are constructed to appear at the point in the parse when X is second in the syntax stack, and we need to decide what to do next according to its context.

0-productions. Let B be the non-terminal for which we are constructing 0-productions. Let I(B) be the set of all terminal characters with which B can begin. I(a) is defined for terminals by setting I(a) = a. We want to construct one production for each member of I(B). Therefore, we examine each alternative in the definition of B in the following way. If it begins with a terminal character, then we construct a production from that string. If it begins with a non-terminal C, we examine the definition of C to determine I(C) in the same way that we examined B. The production which is constructed depends on the context of the initial terminal character a in the alternative. There are three cases:

(1) If the alternative contains just a ,

G ::= a | ...

the production is

a | → G | *G1

(2) If a is followed by a terminal character,

G ::= ab... | ...

the production is

a | | *a1

(3) If a is followed by a non-terminal,

G ::= aH... | ...

the production is

a | | *H0

After constructing these productions, we examine them to see if two have the same stack element. If this is so, we replace these two by one production having the form of (2).

It is fairly easy to see why the productions are constructed in this way. In cases (1) and (3) we know that there is only one initial string of B which begins with a, so we just do what is indicated by the string. In (1) we replace the a by its definition, scan the next character, and go to G1 with G second in the stack. In (3) we go to H0, knowing that we must find an instance of H next in a legal sentence. In (2) or in the case that more than one production has the same stack, we want to postpone any parsing decision until another character has been scanned.

1-productions. Let X be the character for which we are constructing 1-productions. Let C(X) be the set of all places in the BNF in which X appears. We want to construct one 1-production for each member of C(X) according to the following five cases:

(1) If X is last in the alternative,

G ::= αX | ...

the production is

α X σ | $\rightarrow$G σ | G1

(2) If X is followed by a non-terminal,

G ::= αXH... | ...

one production is constructed for each member i of I(H). They are of the form,

α X i | | H0

(3) If X is followed by a terminal character which ends the alternative,

G ::= αXc | ...

the production is

α X c | → G | *G1

(4) If X is followed by two terminals,

G ::= αXcd... | ...

the production is

α X c | | *c1

(5) If X is followed by a terminal and then a non-terminal,

G ::= αXcH... | ...

the production is

α X c | | *H0

As with the 0-productions, there is a rule for combining productions which have similar stacks. However, its explanation must wait until after culprits have been explained. Unlike the 0-productions, the order of the 1-productions is important. This will also be explained later.

Culprits. In the 1-productions it is important that if a production should be applied to the syntax stack, another production which precedes it cannot also be applied to the same syntax stack. In this case we say that the preceding production precludes the other. If there exist two productions such that whichever one is placed first will preclude the other, then this will cause serious trouble

in the sets of 1-productions. Therefore we want to discover what alternatives will cause this; these are the culprit strings.

In order to explain culprits, we must define the following:

(1) Let δ be an alternative with definition G. Then a string β is a legal predecessor to δ if there exists a sentential form ...βG.... A terminal b is a legal successor to δ if there exists a sentential form ...Gb....

(2) Two alternatives δ and γ form a head culprit if there exists a common substring α_1 such that $\delta = \alpha_1\ldots$, $\gamma = \beta\alpha_1\ldots$, and β is a legal predecessor to δ. Note that any two alternatives which begin with the same character form a head culprit ($\beta = \Lambda$). If two alternatives have the above form, but β is not a legal predecessor to δ, then they form a head pseudo-culprit but need not be deleted.

(3) Two alternatives δ and γ form a tail culprit if there exists a common substring α_2 such that either

(a) $\delta = \ldots\alpha_2$ and $\gamma = \ldots\alpha_2$

or (b) $\delta = \ldots\alpha_2$ and $\gamma = \ldots\alpha_2$X...

where a member of I(X) is a legal successor to δ.

Two alternatives are culprits if they form a head culprit, a tail culprit, and the common substrings α_1 and α_2 coincide. Note that two identical alternatives form a culprit and that the empty alternative must form a culprit with some other alternative.

In order that no productions will be precluded, the 1-productions are ordered by the following rule, except in one special case. All productions with σ's at the top of their stacks go last after the others. Within these two categories, the productions can go in any order unless the stack of one is a substack of the stack of another. Then the longer one goes first.

We will first examine problems of preclusion between two productions α and β from the same category. In the σ category, for α to preclude β, (1) α must have a

stack of which β's stack is a substack, so that α is ordered first, and (2) the alternatives from which they were constructed must form a head culprit so that α can match the same syntax stack as β.

stack of α = γδσ|

stack of β = δσ|

They are culprits.

However, since they are σ-productions with identical tails, they form a tail culprit. These conditions make them culprit strings, so we don't need to worry about them because they will have been deleted. In the other category, for preclusion to occur, again one production stack must be a substack of the other, and they must form a head culprit. In this case, the alternatives are not necessarily culprits, since both production stacks have a terminal at the top. However, since they both have the same terminal at the top, we can postpone the parsing decision until we scan the next character. So we construct just one production in place of these two, using the smaller stack and having the form of type (4) 1-productions. This is analogous to combining 0-productions with identical stacks.

stack of α = γδb|

stack of β = δb|

Replace by: δb| | *b1

This just leaves us to consider possible problems between two productions from different categories. First of all, the alternative of the σ production α must have as one of its legal successors the top stack element of the other production β. This means the alternatives will be tail culprits. Now if α, which goes last, is a substack of β except for their top symbol, then as before they must be culprit strings to produce preclusion. But if β is the substack, then it will preclude α whether or not they form a head culprit.

stack of β = $\delta b|$

stack of α = $\gamma\delta\sigma|$

Put α before β.

This is the one special case to the ordering rule. When there are two productions such that their alternatives form a tail culprit, a head pseudo-culprit, and the shorter one would normally go first, reverse their order.

Again, with careful examination, it is now obvious why we have the five categories under 1-productions. In all cases except (4), we know that this is the only production with this stack, so we can again replace the string by its definition in (1) and (3), and go to the appropriate 0-productions in (2) and (5). The attentive reader may notice that some of the types of 0- and 1-productions can be omitted with some loss in efficiency.

It is assumed that all sentences will be preceded by ⊢ and followed by ⊣. So the D1 productions will consist of

D1 ⊢ D ⊣ | → | SUCCESS EXIT

Also, we add, after every set of productions, the production

σ | | ERROR EXIT

In parsing a sentence with the productions, the parser starts by scanning ⊢ and the first character and transferring to D0. So in constructing the productions we start by constructing the D0 productions, and then examining the links of all productions to see what other sets should be constructed. This is continued until a set is constructed for each distinct link. Since all sets are closed with respect to the rest of the productions, their order in the total set of productions is irrelevant.

Theorems 1 through 5 at the end of the article prove that the algorithm constructs a recognizer for any culprit-free grammar.

Deleting Culprits. We have explained how the algorithm works, and what culprits are; now we must show how they may be deleted without changing the grammar appreciably. Let us assume that we have found two alternatives to be culprits and have chosen to delete one of them, α.

$$A ::= \ldots \mid \alpha \mid \ldots$$

We are going to remove α from the definition of A, so we must find every place in the BNF in which A appears. At each of these places we add to the definition of the alternative β which contains A, an extra alternative having the same form as β, with A replaced by α.

$$B ::= \ldots \mid \overbrace{\gamma A \delta}^{\beta} \mid \ldots \mid \overbrace{\gamma\alpha\delta}^{\text{added}}$$

If A occurs n times in β, 2^n-1 extra alternatives must be added. After making these additions, we delete α from A. Note that one of these β's may be α itself, if A is recursively defined.

These additions are made so that the deletion not only preserves the sentences which the BNF generates, but only changes the structure of the parse of any sentence in a minor way. That is, in the above example, at first the derivation was: $B \rightarrow \gamma A \delta \rightarrow \gamma\alpha\delta$. After the deletion, it is $B \rightarrow \gamma\alpha\delta$. In fact, if desirable, the string $\gamma\alpha\delta$ which was added could be tagged in such a way as to indicate that α should be parsed first as an A. Then the production set in which the parsing occurs would contain:

c1	$\gamma A \delta \sigma$ |	$\rightarrow$	$B\ \sigma$ |	B1	[from β]
	$\gamma\alpha\delta\sigma$ |	$\rightarrow\ \gamma\ A\ \delta\ \sigma$ |		c1	[from added string]

where c is the last character of δ. This means that the structure of the parse for the changed BNF is exactly the same as the original. Of course, if A had been the root, then we would have changed the grammar by this deletion, since we would be deleting a class of sentences from the language, but it is not necessary to make deletions from the root to remove a culprit.

We must now ask, does this deletion remove the culprit? The answer is: not necessarily. What we have done is to increase the context in which α appears in BNF. This is a step in the right direction, but it could easily take more than one deletion to remove the culprit, if it can be removed at all. The problem is that these deletions may only increase the context of α in one direction. However, one may introduce other transformations on the BNF which will increase α's context in the other direction while still preserving the properties of the grammar. It seems probable that a second type of culprit deletion could be formalized, and it could be proven that, using the two, all culprits can be removed from any bounded context grammar. [1] However, the important point is that in most if not all practical cases (useful programming languages) the culprits will not only be removable, but it will only take a few deletions to produce a culprit-free grammar. This fact, plus the proven usefulness of the production language, makes the algorithm a powerful aid in compiler writing.

Practical Considerations. In any practical applications, such as a program to write productions from BNF, there are many heuristics which can be added to make both the algorithm and the productions more efficient than they might have seemed so far. Metacharacters may be included in the BNF and productions, which stand for any one of a specified set of terminal characters. These are imperative for type (2) 1-productions. It would also be helpful to combine the 0-productions for non-terminals having similar sets of initial terminal characters. A very useful tool would be an algorithm to separate the BNF into relatively independent subsets and apply the algorithm to each of these separately. The culprit deletion routine should also be provided with a means of choosing the most profitable order for deleting strings.

Example: The following is an illustration of how the algorithm works on a small algebraic language.

B is < block >	AE is < arithmetic expression >
H is < head >	T is < term >
D is < declaration >	F is < factor >
TL is < type list >	P is < primary >
S is < statement >	i is identifier

B ::= H end

H ::= begin | begin D | H ; S

D ::= real TL | D ; real TL

TL ::= i | TL , i

S ::= i ← AE

AE ::= T | ± T | AE ± T

T ::= F | T */ F

F ::= P | F ↑ P

P ::= i | (AE)

"BEGIN D" forms a culprit with "D; real TL," and the two "i"'s form a culprit. After deleting "begin D" and the first "i", the changed lines of the grammar are:

B ::= H end | begin D end

H ::= begin | H ; S | begin D ; S

D ::= real TL | D ; real TL | real i | D ; real i

TL ::= TL, i | i , i

The algorithm then constructs the following productions:

B0	begin	|	|	*BG1
	σ	|	|	ERROR EXIT
BG1	begin real	|	|	D0
	begin σ	| →	H σ |	H1

	σ	|	|	ERROR EXIT
D0	real	|	|	*RL1
	σ	|	|	ERROR EXIT
H1	H end	| →	B |	*B1
	H ;	|	|	*S0
	σ	|	|	ERROR EXIT
RL1	D; real i	|	|	*i1
	real i	|	|	*i1
	σ	|	|	ERROR EXIT
B1	⊢ B ⊣	| →	|	SUCCESS EXIT
	σ	|	|	ERROR EXIT
S0	i	|	|	*i1
	σ	|	|	ERROR EXIT
i1	i ,	|	|	* , 1
	i ←	|	|	* ← 1
	D ; real i σ	| →	D σ |	D1
	real i σ	| →	D σ |	D1
	TL , i σ	| →	TL σ |	TL1
	i , i σ	| →	TL σ |	TL1
	i σ	| →	P σ |	P1
	σ	|	|	ERROR EXIT
, 1	TL , i	| →	TL |	*TL1
	i , i	| →	TL |	*TL1
	σ	|	|	ERROR EXIT
← 1	i ← ±	|	|	AE0
	i ← (	|	|	AE0

	i ← i	|		|	AEO
	σ	|		|	ERROR EXIT
D1	begin D end	|	→ B	|	*B1
	D ;	|		|	; 1
	σ	|		|	ERROR EXIT
TL1	D ; real TL σ	|	→ D σ	|	D1
	real TL σ	|	→ D σ	|	D1
	TL ,	|		|	* , 1
	σ	|		|	ERROR EXIT
P1	F ↑ P σ	|	→ F σ	|	F1
	P σ	|	→ F σ	|	F1
	σ	|		|	ERROR EXIT
AEO	±	|		|	*T0
	i	|	→ P	|	*P1
	(	|		|	*AEO
	σ	|		|	ERROR EXIT
T0	i	|	→ P	|	*P1
	(	|		|	*AEO
	σ	|		|	ERROR EXIT
;1	D ; real	|		|	*RL1
	H ; i	|		|	S0
	begin D ; i	|		|	S0
	σ	|		|	ERROR EXIT
S1	begin D ; S σ	|	→ H σ	|	H1
	H ; S σ	|	→ H σ	|	H1
	σ	|		|	ERROR EXIT

F1		F ↑			*P0
	T */	F σ	→	T σ	T1
		F σ	→	T σ	T1
		σ			ERROR EXIT
P0		i	→	P	*P1
		(			*AE0
		σ			ERROR EXIT
T1		T */			*F0
	AE ±	T σ	→	AE σ	AE1
	±	T σ	→	AE σ	AE1
		T σ	→	AE σ	AE1
		σ			ERROR EXIT
F0		i	→	P	*P1
		(			*AE0
		σ			ERROR EXIT
AE1		(AE)	→	P	*P1
		AE ±			*T0
	i ← AE	σ	→	S σ	S1
		σ			ERROR EXIT

Outline of Proof. We want to prove that in parsing any legal sentence, we must leave by the success exit. We define proper conditions for entering a set of 0- or 1-productions and prove by induction that these exist when entering all sets of productions. Then we show that if proper conditions hold when entering a set of productions, one of the productions must be applied, so we can't leave by an error exit. This coupled with the fact that we must exit eventually, proves the theorem.

Definition: The parse time T is initialized at 0 when R0 is first entered and is incremented by one each time a set of productions is entered.

Definition: An alternative γ is a correct parse at a certain parse time if we have part of γ in the syntax stack and we know that instances of the rest must follow immediately in any legal sentence.

Definition: The syntax stack corresponds (m,n) to an alternative if the substack of m elements found below the top n elements of the syntax stack is the same as the first m elements of the alternative which is a correct parse at the time. "Corresponds (m,1)" will be abbreviated "corresponds."

Conditions: (1) Let α be an alternative with definition G. Then previous conditions hold if, for all α and for all previous parse times $T \neq 1$ in which the syntax stack corresponded (1,1) to α, the syntax stack also corresponded (m,2) to an alternative β which had G or C as its (m + 1) symbol (where G begins some C-derivative). (2) Proper conditions for entering a set of 0-productions C0 hold if either $T = 0$ or the syntax stack corresponds to an alternative which has C as its (m + 1) character. (3) Proper conditions for entering a set of 1-productions hold if either the syntax stack $= \vdash D \dashv$ or it corresponds to an alternative.

Definition: An initial string of a non-terminal C is an alternative which is a head of some C derivative.

Theorem 1. In parsing a legal sentence, if proper and previous conditions hold when entering a set of 0-productions C0 at time T, then they will hold at T + 1.

Proof: There are two possibilities: (1) C is the root and we are entering R0 with $\vdash$ and a member of I(R) in the syntax stack. (2) The syntax stack corresponds to an alternative which has C as its (m + 1) element. In this case the top character in the syntax stack must be a member of I(C). So, in either case,

since each 0-production stack contains one of the members of I(C), the syntax stack must match one of the 0-productions.

Case 1. If it matches a 0-production of type (1), with top stack element a, then we transform the a in the syntax stack to its definition G. We then scan the next character X and enter G1 with the syntax stack = either ⊢GX or ... GX. Now, because of the method of search in constructing 0-productions, G must be either C or the first character in one or more initial strings of C.

Case 1A. If G is C, and C is the root, then the syntax stack will be ⊢RX, and X must be ⊣. Therefore, proper conditions hold in this case.

Case 1B. Otherwise, if G is C, then the syntax stack will correspond (m+1,1) to the same alternative which it did on entering C0. This is because we have just replaced the previous top element with C (the (m + 1) character in the alternative) and added the next character. So proper conditions hold.

Case 1C. If G is the first character in one or more initial strings of C, then the syntax stack will correspond (1,1) to one of them, so proper conditions hold again. Now we must show that previous conditions still hold after this (1,1) correspondence. The definition of this string is either C or the first-symbol in a C-derivative. On entering C0, by proper conditions the syntax stack corresponded (m,1) to an alternative which had C as its (m + 1) symbol. Since then we have just added a character to the syntax stack, so now it corresponds (m,2) as we want.

Case 2. If it matches a 0-production of type (2), with top stack element a, then there are one or more initial strings of C which begin with a. We scan the next character and enter a1. The syntax stack will correspond (1,1) to one of them, so proper conditions are satisfied. Previous conditions also hold for exactly the same reasons as in the last case.

Case 3. If it matches a 0-production of type (3) with top stack element a, then the alternative α from which this production was constructed is the only initial string of C which starts with a. We scan the next character and go to H0. The syntax stack must correspond (1,1) to α and we know H is its second character so proper conditions are again satisfied. Previous conditions hold also, by the same reasoning as before.

Theorem 2. In parsing a legal sentence, if all culprits have been removed, and if proper conditions exist when entering a set of 1-productions, b1, so that the syntax stack corresponds to an alternative β, then a production which was constructed from β will be applied to the syntax stack.

Proof: Let the syntax stack be ... α b c |. Then β is α b... where c is a legal successor to the αb in β. If a production of type (3), (4), or (5) was constructed from β alone, its stack will be α b c and it will apply. If a type (2) production was constructed from β, it means that β is α b H... where c$\in$I(H), and since one type (2) production is constructed for each member of I(H), one of them must apply. If a type (4) production was constructed from combining β and other alternatives, it means the β is one of the two above cases, and the production is γ b c where γ is a substack of α; so the production will apply. If a type (1) production was constructed from β, then its stack is αbc and it will apply. Thus we have shown that one of the productions constructed from the alternative which corresponds to the syntax stack will always be applied, unless there is another production which precludes it. Let us examine an arbitrary production under b1 to which the stack can be applied to see if there can exist any production which precludes it.

Case 1. If the production has a terminal at the top of its stack, (is of the form αbc) then it came from one or more alternatives of the form ...αbc... .

It can't be precluded by productions with σ's at the top, because they will be after it (except for the special case, see Case 2B(2).). It can't be precluded by a production unless it has a stack of which αbc is a substack, because productions whose stacks are substacks of αbc will be after it, and others could not match the same syntax stack. So, let this other production be $\beta\alpha bc$. We now ask, Can this other production be applied to the same syntax stack to which αbc can? This means, find the longest alternative from which αbc was constructed $\gamma\alpha bc\ldots$. Is γ a tail of β, and is $\beta - \gamma$ a legal predecessor to $\gamma\alpha bc$? This is obviously only true if the alternatives form a head culprit, but in this case, by the algorithm, only one production will be constructed here, so this is impossible.

Case 2. If the production has a σ at the top (is of the form $\alpha b\sigma$), then it came from a string of the form αb.

Case 2A. We will first consider preclusion by another production with σ at the top. The alternative of this production must form a tail culprit with αb. As before, we need only consider productions which have a stack of which $\alpha b\sigma$ is a substack, and as before, the only way there can be preclusion is if the alternatives also form a head culprit. However, this makes them culprits, and this is impossible.

Case 2B. We now consider preclusion by a production with a terminal c at the top (βbc). Since both productions must apply to the same stack, c must be a legal successor to αb. This means that the alternatives form a tail culprit. In this case, since the terminal production always goes before the σ production despite their stacks, we must examine productions with stacks both longer and shorter than $\alpha b\sigma$. (1) If α is a substack of β, then as before the strings must form a head culprit for preclusion to exist. Since we have also shown that they

form a tail culprit, this is impossible. (2) If β is a substack of α, then βbc will preclude $\alpha b\sigma$ even if they don't form a head culprit. But they do form a head pseudo-culprit, so their order will be reversed. Now we must ask, can $\alpha b\sigma$ preclude βbc. And as before, the answer now is: Only if their alternatives form a head culprit, and again this makes them culprits.

So we have shown for all cases that preclusion cannot occur.

Theorem 3. In parsing a legal sentence, if proper and previous conditions hold when entering a set of 1-productions X1 at time T, then they will hold at T + 1.

Proof: If the syntax stack = ⊢ D ⊣ , then we will be entering D1, and the production will be ⊢ D ⊣ | → | SUCCESS EXIT So we will exit. Otherwise, the syntax stack corresponds to an alternative, and by Theorem 2, the productions which is applied to the syntax stack was constructed from this alternative.

Case 1. If a production of type (1) is applied, with stack $\alpha X\sigma$, we know that it was constructed from just one alternative $\beta = \alpha X$, so we replace αX with its definition G and transfer to G1. By previous conditions, when the syntax stack corresponded (1,1) to β it also corresponded (m,2) to another alternative which had G or C (where some C-derivative starts with G) as its (m+1) character.

Case 1A. If G was the (m+1) character in the alternative, the syntax stack will now correspond (m+1,1) to that same alternative becuase we have just replaced β in the stack by G and added another character.

Case 1B. If C was the (m+1) character in the alternatives, the stack will correspond (1,1) to an initial string of C which begin with G, so proper conditions hold again. Previous conditions hold here because they held before and we now have back the same m elements of the syntax stack below the top two.

If a production of type (3) is applied, we can use exactly the same reasoning to prove the theorem except that we replace αbc with G, and we must scan a character before going to G1.

Case 2. If a production of type (2) is applied, with stack αXc, we know that it was constructed from just one alternative β = αXH... where c is a member of I(H). So we enter H0 knowing that the syntax stack still corresponds to β with the same m, and H is the next character in β. So proper conditions hold. The same argument goes for type (5) productions.

Case 3. If a production of type (4) is applied with stack αXc, it was constructed from one or more alternatives of the form ...αXc... or ...αXH... (where c ∈ I (H)), one of which is a correct parse. So we scan the next character and go to c1.

Case 3A. If ...aXc... is a correct parse, the syntax stack will now correspond (m+1,1) to it, and proper conditions will hold.

Case 3B. If ...aXH... is a correct parse, then the syntax stack must now correspond (1,1) to one of the initial strings of H which begins with c, and proper conditions hold again. Previous conditions hold because the syntax stack now corresponds (m,2) to ...αXH....

Therefore, the Theorem is satisfied for all cases.

Theorem 4. A finite sentence will be scanned by the algorithm in a finite number of steps.

Proof: Let us examine the links of the different types of productions that can be constructed. The only types of productions which do not scan a new character in their links are 1-productions of types (1) and (2). Productions of type (2), however, link to a 0-production, and all 0-productions scan a character in their links. So the only possible troublemaker is a type (1) 1-production.

We want to show that only a finite number of these productions can link to each other before linking to another type of production, and thus scanning a character.

Each type (1) 1-production transforms one or more characters in the syntax stack into a non-terminal. The productions which transform more than one element of the syntax stack are making it smaller. The productions which transform just one character into a non-terminal are just redefining this character.

We cannot have more than r redefinitions of a character (where r is the number of non-terminals in the grammar) because if we did, a certain non-terminal would be redefined as itself. This cycle could then be repeated a different number of times in different derivations of the same sentence, making the grammar ambiguous. So after these r redefinitions there must be a production which reduces the size of the syntax stack. Therefore, if s is the number of elements in the syntax stack at the time, there can be at most s(r+1) type (1) 1-productions before we must link to another type and at most one more before we scan another character. So, any finite source string must be scanned in a finite number of steps.

Theorem 5. Given a culprit-free grammar, the algorithm constructs productions which are a recognizer for the grammar.

Proof: (1) First we assume that we are parsing a legal sentence and prove that we must leave by the success exit.

Theorem 5 says that we must eventually scan the whole sentence so that we must scan ⊣. There is only one production which contains ⊣; this is the success exit. So it must match this, an error exit, or a σ-production. But a σ-production cannot scan a new character or remove the ⊣, so it must eventually match an exit in parsing all finite sentences.

In parsing a legal sentence, proper conditions hold by definition at T = 0; theorems 1 and 3 show that if proper and previous conditions hold when entering

a set of productions, at parse time T they will hold at T + 1. So by induction, proper conditions hold for all sets. We have already shown in the proofs of the previous theorems that if proper conditions hold upon entering a set of productions, one of the productions of the set must be applied, so we can't leave by an error exit. Since we must exit, it must be by the success exit for any legal sentence.

(2) We now assume that we leave by the success exit, and prove that the sentence we have parsed must have been legal. This is equivalent to proving that all illegal sentences must go out error exits.

If we have left by the the success exit, it means that we had ⊢ D ⊣ in the syntax stack. The termination symbols show that this is all that is in the stack and that the sentence has been exhausted. Thus all characters in the sentence have been transformed into non-terminals which have eventually all been transformed into D. Therefore the sentence is legal.

Therefore, (1) and (2) together show that the productions are a recognizer for the grammar.

References

[1] Floyd, R. "Bounded Context Syntactic Analysis", Comm. ACM, February, 1964.

[2] Standish, T. "Generating Productions From a Restricted Class of BNF Grammars". Unpublished paper, Carnegie Institute of Technology Computation Center.

[3] Evans, A. "An Algol 60 Compiler", National ACM Conference, Denver, 1963.

[4] Floyd, R. "A Descriptive Language For Symbol Manipulation", Journal ACM, October 1961.

[5] Feldman, J. "A Formal Semantics For Computer-Oriented Languages", Doctoral Thesis, Carnegie Institute of Technology, 1964.