A more formal treatment of a less simple example.

For obvious reasons, most programming experiments that have been carried out in the exploration of formal techniques, dealt with simple, algebraic examples. For equally obvious reasons, the examples shown in tutorial texts on this subject are mostly of the same nature. (There has been a time when all of Computing Science seemed to boil down to massaging Euclid's Algorithm for the greatest common divisor!) This paper is primarily directed at remedying this situation.

* * *

Our ultimate goal is to develop a program that will transform expressions from infix notation to postfix notation. The subject matter to be manipulated by our program are therefore not integers, but strings of characters that may, or may not belong to certain syntactic categories. For variables of type "character string" we have to have at our disposal the analogon of high-school algebra (such as $(a > b \text{ and } c > d) \Rightarrow a + c > b + d$, etc.) that sufficed for the well-known numerical examples. Before embarking on our problem proper, we shall first introduce the necessary formal apparatus and the notation needed for its description.

We assume our syntax given in BNF. Let $<$ pqr $>$ denote a syntactical category. We shall then express the fact that a string named K belongs to the syntactical category $<$ pqr $>$ by

$$pqr(K) \quad .$$

For strings (named K , L , ...) and characters (named y, z , ...) we shall denote concatenation by juxtaposition, e.g. KL , Ky , Ky; etc. If L may be any string and y may be any character, any non-empty string may be denoted by yL or Ly .

With any syntactic category $<$ pqr $>$ we may associate the syntactic category $<$ bopqr $>$ --"begin of a $<$ pqr $>$"-- consisting of all the strings that either are a $<$ pqr $>$ or can be extended at the right-hand side so as to become a $<$ pqr $>$ or both. According to that definition the statement that the syntactic category $<$ pqr $>$ is not empty --i.e. contains, as most useful syntactic categories, at least one string-- is equivalent with the predicate

$$bopqr(\text{ empty string}) \quad .$$

The formal definition of the predicate bopqr in terms of pqr --with K and L denoting arbitrary strings-- is

$$bopqr(K) \Leftarrow (\underline{E}\ L:\ pqr(KL)) \qquad (1)$$

Separating the case that L is empty and the case that L is not empty, we can rewrite (1) as

$$bopqr(K) \Leftarrow (pqr(K)\ \underline{or}\ (\underline{E}\ yL:\ pqr(KyL)))$$

which, thanks to (1), can be reduced to

$$bopqr(K) \Leftarrow (pqr(K)\ \underline{or}\ (\underline{E}\ y:\ bopqr(Ky))) \qquad (2)$$

from which we immediately derive

$$(bopqr(K)\ \underline{and}\ (\underline{A}\ y:\ \underline{non}\ bopqr(Ky))) \Rightarrow pqr(K) \qquad (3)$$

From (1) we derive further

$$\begin{aligned} bopqr(Ky) &\Leftarrow (\underline{E}\ L:\ pqr(KyL)) \\ &= (\underline{E}\ yL:\ pqr(KyL)) \\ &\Rightarrow bopqr(K) \quad . \end{aligned}$$

From this result

$$bopqr(Ky) \Rightarrow bopqr(K) \qquad (4)$$

follows that $< bopqr > = < bobopqr >$.

Because $pqr(K) \Rightarrow (\underline{E}\ L:\ pqr(KL))$ --L = the emptystring does the job-- a further consequence of (1) is

$$pqr(K) \Rightarrow bopqr(K) \qquad (5)$$

From our informal description of what we intended the notion "begin of" to mean, the above is all intuitively obvious, and by now the reader may wonder what all the fuss is about. The point is that we need such formulae as soon as we wish to give a more rigorous treatment of a parser.

* * *

We intend to develop a mechanism called "sentsearch" that is intended to recognize strings from the syntactical category $< sent >$. More precisely, we assume that the input string can be scanned in the order from left to right and reserve the identifier "x" for the next visible character of the input string. If the input string starts with "a + b ...." , then we have initially

x = "a" ; after the execution of "move" the relation x = "+" will hold. Besides assigning a new value to x , the primitive "move" can be viewed as also appending the old value of x to the right to "the strings of characters moved over" or "the string of characters read" or "the string of characters that are no longer visible."

Let S be the string of characters "moved over" by an activation of sentsearch .

Note 1. When developing the body of sentsearch we may assume that a local so-called "ghost variable" S is initialized at the beginning as the empty string, that each call on "move" is implicitly preceded by "S:= Sx" , and that upon termination S is handed back as a "ghost function value" to the calling environment. (End of Note 1.)

In the case that the input sequence does not start with a < sent > , we want S to be the sequence that is insufficient to establish this fact, while Sx is long enough to make this conclusion. That is, upon termination

bosent(S) and non bosent(Sx)

will hold. The first term expresses that not too much has been moved over, the second term expresses that enough has been moved over. In the case that the input string does start with a < sent > , we wish S to be equal to that < sent > and assume our syntax for < sent > --about which nothing has been given yet-- to satisfy

$$\text{sent}(L) \Rightarrow \underline{\text{non}}\ (\underline{\text{E}}\ y:\ \text{bosent}(Ly)) \qquad (6)$$

Whether or not a < sent > has been found is to be recorded in the global boolean c --short for "correct"-- and our complete specification of sentsearch is that it has to establish Rs(S, x, c) , where Rs(S, x, c) is given by

$$\text{bosent}(S)\ \underline{\text{and}}\ \underline{\text{non}}\ \text{bosent}(Sx)\ \underline{\text{and}}\ c = \text{sent}(S) \qquad (7)$$

Note 2. The consequence of assumption (6) is that when the input string starts with a < sent > and the analysis has progressed to S equal to that < sent > , the term non bosent(Sx) is true for all possible values of x , i.e. sentsearch can then terminate without inspecting the next visible character. The end of a < sent > is assumed to be detectable without looking beyond it. (End of Note 2.)

We now give the syntax for < sent > :

< sent > ::= < exp > ; (8)

From this we have to derive the syntax for the syntactical category < bosent > :

< bosent > ::= < sent > | < boexp > (9)

Each < bosent > can be derived by taking a < sent > and removing at the right-hand side zero or more characters from it. Removal of zero characters gives the first alternative, removal of one or more charachters from "< exp > ;" boils down --because the semicolon is a single character-- to the removal of zero or more characters from < exp > : but that is by definition the syntactic category called < boexp > . Hence (9). The two alternatives are mutually exclusive, for we have for any string L :

boexp(L) => non sent(L) (10)

This can be proved by deriving a contradiction from boexp(L) and sent(L) . From boexp(L) follows --according to (2)--

exp(L) or (E y: boexp(Ly))

We deal with both term separately:

exp(L) => (on account of (8))

sent(L;) => (on account of (5))

bosent(L;) => (E y: bosent(Ly)) ;

the second term gives

(E y: boexp(Ly)) => (on account of (9))

(E y: bosent(Ly)) .

As both terms of the disjunction imply the same, we conclude that also

boexp(L) => (E y: bosent(Ly)) .

According to (6), however,

sent(L) => non (E y: bosent(Ly)) .

The desired contradiction has been established and (10) has been proved.

Syntax rule (8) strongly suggests that the body of sentsearch should start with a call of expsearch. In order to design sentsearch in terms of expsearch we only need to know the net effect of expsearch and we propose in analogy to (7) that --when E is the string of characters moved over by expsearch-- the primitive expsearch will establish Re(E, x, c) , where Re(E, x, c) is given by

boexp(E) and non boexp(Ex) and c = exp(E) (11)

Designing sentsearch in terms of expsearch means that we would like to have theorems, such that from the truth of a relation of the form Re the truth of relations of the form Rs can be concluded. There are three such theorems.

Theorem 1. (Re(L, x, c) and non c) => Rs(L, x, c)

Proof. Assumed:

0.	Re(L, x, c) and non c	
	Derived:	
1.	boexp(L)	with (11) from 0
2.	bosent(L)	with (9) from 1
3.	c = exp(L)	with (11) from 0
4.	non c	from 0
5.	non exp(L)	from 3 and 4
6.	non sent(Lx)	with (8) from 5
7.	non boexp(Lx)	with (11) from 0
8.	non bosent(Lx)	with (9) from 6 and 7
9.	non sent(L)	with (10) from 1
10.	c = sent(L)	from 4 and 9
11.	Rs(L, x, c)	with (7) from 2, 8 and 10
		(End of Proof of Theorem 1.)

Theorem 2. (Re(L, x, c) and c and non semi(x)) => Rs(L, x, false)

Proof. Assumed:

0.	Re(L, x, c) and c and non semi(x)	
	Derived:	
1.	boexp(L)	with (11) from 0
2.	bosent(L)	with (9) from 1
3.	non semi(x)	from 0
4.	non sent(Lx)	with (8) from 3
5.	non boexp(Lx)	with (11) from 0
6.	non bosent(Lx)	with (9) from 4 and 5
7.	false = sent(L)	with (10) from 1
8.	Rs(L, x, false)	with (7) from 2, 6 and 7
		(End of Proof of Theorem 2.)

Theorem 3. (Re(L, x, c) and c and semi(x)) => Rs(Lx, y, c)

Proof. Assumed:

0.	Re(L, x, c) and c and semi(x)	
	Derived:	
1.	c = exp(L)	with (11) from 0
2.	c	from 0
3.	exp(L)	from 1 and 2
4.	semi(x)	from 0
5.	sent(Lx)	with (8) from 3 and 4
6.	c = sent(Lx)	from 2 and ~~X~~5
7.	bosent(Lx)	with (5) from 5
8.	non bosent(Lxy)	with (6) from 5
9.	Rs(Lx, y, c)	with (7) from 7, 8 and 6.
		(End of Proof of Theorem 3.)

And now a possible body of sentsearch is evident, when we realize that its call on expsearch implies for the ghost variable S the assignment "S:= SE"

```
proc sentsearch: { S = empty string}
                 expsearch {Re(S, x, c)};
                 if non c → skip
                 [] c and non semi(x) → c:= false
                 [] c and semi(x) → move
                 fi {Rs(S, x, c)}
corp
```

Note 3. Instead of Theorems 1 and 2 we could have discovered

Theorem 1'. (Re(L, x, c) and non c) => Rs(L, x, false)

Theorem 2'. (Re(L, x, c) and non semi(x)) => Rs(L, x, false).

This would have directed us towards the design of the body

```
proc sentsearch: expsearch;
                 if non c or non semi(x) → c:= false
                  [] c and semi(x) → move
                 fi
corp
```

which, thanks to de Morgan's Theorem, has no aborting alternative construct.

(End of note 3.)

We now consider for < exp > the following syntax

$$< \text{exp} > ::= < \text{adder} > < \text{term} > \qquad (12)$$

$$< \text{adder} > ::= \{ < \text{term} > < \text{adop} > \} \qquad (13)$$

$$< \text{adop} > ::= + \mid - \qquad (14)$$

where the braces indicate a succession of zero or more instances of the enclosed. Because each instance of the syntactic category < adop > is a single character, we derive

$$< \text{boexp} > ::= < \text{adder} > < \text{boterm} > \qquad (15)$$

from which follows $(\text{adder}(L) \text{ and } \text{boterm}(K)) \Rightarrow \text{boexp}(LK)$ (16)

But this gives us no way of proving that a string is not of the syntactic category < boexp > . In particular, the conclusion

$(\text{adder}(L) \text{ and non } \text{boterm}(K)) \Rightarrow \text{non } \text{boexp}(LK)$ is not justified.

We must make --in analogy to (6)-- an assumption about < term > and < adop > , and we assume $(\text{term}(L) \text{ and } \text{adop}(y)) \Rightarrow \text{non } \text{boterm}(Ly)$ (17)

This means, to start with, that with term(L), term(L'), adop(y), and adop(y'), we can conlude from $LyS = L'y'S'$, that $L = L'$ and $y = y'$. In other words, for every < boexp > that starts with an instance of < term >< adop > , that instance is uniquely defined. By removing it from the front end , we are still left with a string from the syntactic category < boexp > , and therefore we are allowed to conclude

$(\text{adder}(L) \text{ and non } \text{boexp}(K)) \Rightarrow \text{non } \text{boexp}(LK)$ (18)

This does not solve our problems yet, because, in order to use (18) in order to prove non boexp(LK) , we still have to prove non boexp(K) , be it only for a possibly shorter string K . We can do it, however, for a string related to the syntactic category < term > , as we can prove

$(\text{boterm}(L) \text{ and non } \text{boterm}(Ly) \text{ and } \text{boexp}(Ly)) \Rightarrow (\text{term}(L) \text{ and } \text{adop}(y))$ (19)

The nonempty string Ly , satisfying boexp(Ly) can have one of three different forms:

1) < term >< adop >< nonempty boexp >

This would imply, that L itself is of the form

< term >< adop >< boexp >

which, on account of its first two elements and (17) is incompatible with

boterm(L)

2) < term > < adop >

Because all instances of < adop > are single characters, this case implies indeed term(L) and adop(y)

3) < boterm >

This case is incompatible with non boterm(Ly).

Hence, formula (19) has been proved.

Similarly, we should ask ourselves how to prove that some string is not an element of the syntactic category < exp > . From (12) we can derive

$$(\text{adder}(L) \text{ and } \text{term}(K)) \Rightarrow \text{exp}(LK) \qquad (20)$$

but, again, the conclusion

(adder(L) and non term(K)) => non exp(LK) is not justified,

only --similar to (18)--

$$(\text{adder}(L) \text{ and non } \text{exp}(K)) \Rightarrow \text{non } \text{exp}(LK) \qquad (21)$$

Analogous to (19) we have

$$(\text{boterm}(L) \text{ and } \text{exp}(L)) \Rightarrow \text{term}(L) \qquad (22)$$

The term exp(L) tells us that the string L can have one of two different forms:

1) < term >

This case indeed implies term(L)

2) < nonempty adder > < term >

On account of (17) --and also (4)-- this case is exluded by boterm(L).

Hence formula (22) has been proved.

Finally we can conclude that

$$(\text{exp}(L) \text{ and } \text{adop}(y)) \Rightarrow \text{adder}(Ly) \qquad (23)$$

The left-hand side tells us on account of (12) that Ly is of the form

< adder > < term > < adop >

and therefore (13) alows us to conclude adder(Ly) , and (23) has been proved.

Syntax rules (12) and (13) strongly suggest that the body of expsearch should call --possibly repeatedly-- a new primitive termsearch. In order to design expsearch in terms of termsearch we only need to know the net effect

of termsearch and we propose --in analogy to (7) and (11)-- that, when T is defined as the string of characters moved over by termsearch , the primitive termsearch will establish Rt(T, x, c) , where Rt(T, x, c) is given by

boterm(T) and non boterm(Tx) and c = term(T) (24)

Designing expsearch in terms of termsearch means that we would like to have theorems allowing us to draw conclusions from the truth of a relation of the form Rt .

Theorem 4. (adder(L) and Rt(T, x, c) and c and adop(x)) => adder(LTx)

Proof. Assumed:

0.	adder(L) and Rt(T, x, c) and c and adop(x)	
	Derived:	
1.	c = term(T)	with (24) from 0
2.	c	from 0
3.	term(T)	from 1 and 2
4.	adder(L)	from 0
5.	exp(LT)	with (20) from 3 and 4
6.	adop(x)	from 0
7	adder(LTx)	with (23) from 5 and 6
		(End of Proof of Theorem 4.)

Theorem 5. (adder(L) and Rt(T, x, c) and non c) => Re(LT, x, c)

Proof. Assumed:

0.	adder(L) and Rt(T, x, c) and non c	
	Derived:	
1.	c = term(T)	with (24) from 0
2.	non c	from 0
3.	non term(T)	from 1 and 2
4.	boterm(T)	with (24) from 0
5.	non boterm(Tx)	with (24) from 0
6.	non boexp(Tx)	with (19) from 3, 4, and 5
7.	adder(L)	from 0
8.	non boexp(LTx)	with (18) from 6 and 7
9.	boexp(LT)	with (16) from 4 and 7
10.	non exp(T)	with (22) from 3 and 4
11.	non exp(LT)	with (21) from 7 and 10
12.	c = exp(LT)	from 2 and 11

13.	Re(LT, x, c)	with (11) from 8, 9, and 12
		(End of Proof of Theorem 5.)

Theorem 6. (adder(L) and Rt(T, x, c) and non adop(x)) => Re(LT, x, c)

Proof. Assumed:

0.	adder(L) and Rt(T, x, c) and non adop(x)	
	Derived:	
1.	boterm(T)	with (24) from 0
2.	adder(L)	from 0
3.	boexp(LT)	with (16) from 1 and 2
4.	non boterm(Tx)	with (24) from 0
5.	non adop(x)	from 0
6.	non boexp(Tx)	with (19) from 1, 4, and 5
7.	non boexp(LTx)	with (18) from 2 and 6
8.	c = term(T)	with (24) from 0
9.	c => term(T)	from 8
10.	c => exp(LT)	with (20) from 2 and 9
11.	non c => non term(T)	from 8
12.	non c => non exp(T)	with (22) from 1 and 11
13.	non c => non exp(LT)	with (21) from 2 and 12
14.	c = exp(LT)	from 10 and 13
15.	Re(LT, x, c)	with (11) from 3, 7, and 14
		(End of Proof of Theorem 6.)

A corollary of Theorems 5 and 6 is

(adder(L) and Rt(T, x, c) and non(c and adop(x))) => Re(LT, x, c) .

A possible body for expsearch is by now pretty obvious when we realize that its calls on termsearch imply for its ghost variable E the assignment E:= ET (as "move" implies E:= Ex). In the post-assertions for calls on termsearch the relation E = LT has been given in order to define L in terms of E and T.

```
proc expsearch: {adder(E) because E = empty string}
                termsearch {E = LT and adder(L) and Rt(T, x, c)};
                do c and adop(x) → {adder(Ex)}
                      move {adder(E)};
                      termsearch {E = LT and adder(L) and Rt(T, x, c)}
                od {Re(E, x, c)}
corp
```

We now consider for < term > the following syntax

$$< \text{term} > ::= < \text{plier} > < \text{prim} > \tag{25}$$
$$< \text{plier} > ::= \{ < \text{prim} > < \text{mult} > \} \tag{26}$$
$$< \text{mult} > ::= * \tag{27}$$

and assume about < prim > and < mult >

$$(\text{prim}(L) \text{ and } \text{mult}(y)) \Rightarrow \text{non } \text{boprim}(Ly) \tag{28}$$

Formulae (25), (26), (27), and (28) are similar to (12), (13), (14), and (17) respectively, and all our conclusions since then carry over. With P as the string of characters moved over by a primitive primsearch that establishes --in analogy to (24)-- Rp(P, x, c) , where Rp(P, x, c) is given by

$$\text{boprim}(P) \text{ and non } \text{boprim}(Px) \text{ and } c = \text{prim}(P) \tag{29}$$

we can write immediately (!)

```
proc termsearch: {plier(T) because T = empty string}
                 primsearch {T = LP and plier(L) and Rp(P, x, c)};
                 do c and mult(x) → {plier(Tx)}
                       move {plier(T)};
                       primsearch {T = LP and plier(L) and Rp(P, x, c)}
                 od {Rt(T, x, c)}
corp
```

It is time to "close" our syntax:

$$< \text{prim} > ::= < \text{iden} > \mid < \text{paren} > \tag{30}$$
$$< \text{iden} > ::= \{ < \text{letter} > \} < \text{letter} > \tag{31}$$
$$< \text{paren} > ::= < \text{open} > < \text{exp} > < \text{close} > \tag{32}$$
$$< \text{open} > ::= (\tag{33}$$
$$< \text{close} > ::=) \tag{34}$$
$$< \text{letter} > ::= a \mid b \mid c \mid d \mid e \mid f \tag{35}$$

The important conclusions from (35) are:

1) that the syntactic category < letter > is nonempty

2) that all instances of the syntactic category < letter > are all single characters

3) that these characters differ from the six previously introduced

characters.

From the nonemptiness of the syntactic category < letter > we draw the same conclusion for < iden > , hence for < prim > , hence for < term > hence for < exp > , and hence for < sent > . In particular we shall need to fer to

boprim(empty string) (36)

From (30) we derive

< boprim > ::= < boiden > | < boparen > (37)

From (31) and (32) respectively, we derive

(boiden(y) = letter(y)) and non iden(empty string) (38)

(boparen(y) = open(y)) and non paren(empty string) (39)

and hence

boprim(y) = (letter(y) or open(y)) (40)

non prim(empty string) (41)

From (31) we derive

< boiden > ::= { < letter > } (42)

and, because instances of < letter > are single characters

non letter(y) => non boiden(Ly) (43)

From (32) we derive

< boparen > ::= empty string | < open >< boexp > | < paren > (44)

The three alternatives for < boparen > are mutually exclusive: for the first one versus the two others, it is obvious. For the last two I can prove the mutual exclusion only by using the technique of the bracket count.

Lemma 1. exp(L) implies that the number of instances of < open > in L equals the number of instances of < close > in L .

Lemma 2. boexp(L) implies that the number of instances of < open > in L equals at least the number of instances of < close > in L .

Lemma 1 follows from the fact that in the original syntax --i.e. without the "begin-of"-derivations-- the only rule using < open > or < close > , viz. (32), introduces them pairwise. Lemma 2 follows from the observation that in this only introduction, the instance of < open > precedes that of < close >

(Presumably official syntactic theory has more formal proofs for these two Lemmata; I am fully convinced of their correctness by the preceding four lines of argument.)

The last two alternatives of (44) are mutually exclusive, because from Lemma 2 we can conclude that in a string of the form $<\text{open}><\text{boexp}>$ the number of instances of $<\text{open}>$ exceeds the number of instances of $<\text{close}>$, while in a string of the form $<\text{paren}>$ these numbers are equal on account of Lemma 1. In other words:

$$(\text{open}(y) \text{ and } \text{boexp}(L)) \Rightarrow \text{non paren}(yL) \qquad (45)$$

or, equivalently

$$\text{paren}(yL) \Rightarrow (\text{open}(y) \text{ and non boexp}(L)) \qquad (45')$$

Expressed in terms of paren and boparen only, also holds

$$\text{paren}(L) \Rightarrow \text{non}(E\ z:\ \text{boparen}(Lz)) \qquad (46)$$

This formula can be derived by deriving a contradiction from the truth of the left-hand side and the falsity of the right-hand side. From paren(L) and (39) we conclude that L is nonempty, and we may write L = yK, such that, on account of (45'), we deduce

$$\text{open}(y) \text{ and non boexp}(K)$$

On the other hand, $(E\ z:\ \text{boparen}(yKz))$ is, according to (1), equivalent to

$$(E\ z,M:\ \text{paren}(yKzM))$$

or

$$(E\ M,z:\ \text{paren}(yKMz))$$

Rule (32) then allows us to conclude

$$\text{open}(y) \text{ and } (E\ M:\ \text{exp}(KM)) \text{ and } (E\ z:\ \text{close}(z)) \ .$$

The second term is equivalent to boexp(K), we have the contradiction we were looking for, and hence, (46) has been proved.

Theorem 7. (L = empty string and non(letter(x) or open(x))) => Rp(L, x, false)

Proof. Assumed:

0. L = empty string and non(letter(x) or open(x))

Derived:

1. L = empty string — from 0
2. boprim(L) — with (36) from 1
3. non(letter(x) or open(x)) — from 0

4.	non boprim(x)	with (40) from 3
5.	x = Lx	from 1
6.	non boprim(Lx)	from 4 and 5
7.	false = prim(L)	with (41) from 1
8.	Rp(L, x, false)	with (29) from 2, 6, and 7
		(End of Proof of Theorem 7.)

Theorem 8. (iden(yL) and letter(x)) => iden(yLx)

Proof. Evident from (31)

Theorem 9. (iden(yL) and non letter(x)) => Rp(yL, x, true)

Proof. Assumed:

0.	iden(yL) and non letter(x)	
	Derived:	
1.	iden(yL)	from 0
2.	boiden(yL)	with (5) from 1
3.	boprim(yL)	with (37) from 2
4.	boiden(y)	with (4) from 2
5.	letter(y)	with (38) from 4
6.	non open(y)	from 5
7.	non boparen(y)	with (39) from 6
8.	non boparen(yLx)	with (4) from 7
9.	non letter(x)	from 0
10.	non boiden(yLx)	with (43) from 9
11.	non boprim(yLx)	with (37) from 8 and 10
12.	true = prim(yL)	with (30) from 1
13.	Rp(yL, x, true)	with (29) from 3, 11, and 12
		(End of Proof of Theorem 9.)

See Note 4 on page EWD550 - 15

Theorem 10. (open(y) and Re(E, x, c) and c and close(x)) => Rp(yEx, z, c)

Proof. Assumed:

0.	open(y) and Re(E, x, c) and c and close(x)	
	Derived:	
1.	c = exp(E)	with (11) from 0
2.	c	from 0
3.	exp(E)	from 1 and 2
4.	open(y)	from 0
5.	close(x)	from 0
6.	paren(yEx)	with (32) from 3, 4, 5

7.	prim(yEx)	with (30) from 6
8.	boprim (yEx)	with (5) from 7
9.	non boparen(yExz)	with (46) from 6
10.	non letter(y)	from 4
11.	non boiden(y)	with (38) from 10
12.	non boiden(yExz)	with (4) from 11
13.	non boprim(yExz)	with (37) from 9 and 12
14.	c = prim(yEx)	from 2 and 7
15.	Rp(yEx, z, c)	with (29) from 8, 13, and 14
		(End of Proof of Theorem 10.)

Theorem 11. (open(y) and Re(E, x, c) and non c) => Rp(yE, x, c)

Proof. Assumed:

0.	open(y) and Re(E, x, c) and non c	
	Derived:	
1.	boexp(E)	with (11) from 0
2.	open(y)	from 0
3.	boparen(yE)	with (44) from 1 and 2
4.	boprim(yE)	with (37) from 3
5.	non letter(y)	from 2
6.	non boiden(y)	with (38) from 5
7.	non boiden(yEx)	with (4) from 6
8.	non boexp(Ex)	with (11) from 0
9.	c = exp(E)	with (11) from 0
10.	non c	from 0
11.	non exp(E)	from 9 and 10
12.	non paren(yEx)	with (32) from (2 and) 11
13.	non boparen(yEx)	with (44) from 8 and 12
14.	non boprim(yEx)	with (37) from 7 and 13
15.	non boiden(yE)	with (4) from 6
16.	non iden(yE)	with (5) from 15
17.	non paren(yE)	with (45) from 1 and 2
18.	non prim(yE)	with (30) from 16 and 17
19.	c = prim(yE)	from 10 and 18
20.	Rp(yE, x, c)	with (29) from 4, 14 and 19
		(End of Proof of Theorem 11.)

Theorem 12. (open(y) and Re(E, x, c) and non close(x)) => Rp(yE, x, false)

Proof. Assumed:

0.	open(y) and Re(E, x, c) and non close(x)	
	Derived:	
1.	boexp(E)	with (11) from 0
2.	open(y)	from 0
3.	boparen(yE)	with (44) from 1 and 2
4.	boprim(yE)	with (37) from 3
5.	non letter(y)	from 2
6.	non boiden(y)	with (38) from 5
7.	non boiden(yEx)	with (4) from 6
8.	non boexp(Ex)	with (11) from 0
9.	non close(x)	from 0
10.	non paren(yEx)	with (32) from 9
11.	non boparen(yEx)	with (44) from 8 and 10
12.	non boprim(yEx)	with (37) from 7 and 11
13.	non boiden(yE)	with (4) from 6
14.	non iden(yE)	with (5) from 13
15.	non paren(yE)	with (45) from 1 and 2
16.	false = prim(yE)	with (30) from 14 and 15
17.	Rp(yE, x, false)	with (29) from 4, 12, and 16

Note 4. In proofs 9 through 12, I refer a number of times to formula (4), but it is not really that one that is needed, but the obvious generalization

bopqr(KL) => bopqr(K) ;

sometimes it is used in the inverted, but equivalent, form

non bopqr(K) => non bopqr(KL) .

Furthermore I offer my apologies for the great similarity between the proofs of Theorem 11 and Theorem 12. The total text could have been shortened by first stating a Lemma 3 that captures the intersection of the two proofs. It is just too expensive to change in this respect this document, which is not intended to be submitted for publication. (End of Note 4.)

With Theorems 7 through 12 we have prepared the way for the following design of a body for primsearch .

```
proc primsearch : {P = empty string}
        if non(letter(x) or open(x)) → {Rp(P, x, false)}
                c:= false {Rp(P, x, c)}
         [] letter(x) → move {P = yL and iden(P)};
                do letter(x) → {P = yL and iden(Px)}
                        move {P = yL and iden(P)}
                od {Rp(P, x, true};
                c:= true {Rp(P, x, c)}
         [] open(x) → move {P = y and open(y)};
                expsearch {P = yE and open(y) and Re(E, x, c)};
                if c and close(x) → {Rp(Px, z, c)}
                        move {Rp(P, x, c)}
                 [] non c → skip {Rp(P, x, c)}
                 [] non close(x) → {Rp(P, x, false}
                        c:= false {Rp(P, x, c)}
                fi {Rp(P, x, c)}
        fi {Rp(P, x, c)}
corp
```

Now our syntax has been "closed" by (30) through (35), we can at last fulfill our obligation of proving what up till now have been assumptions, viz.

$$\text{sent}(L) \Rightarrow \underline{\text{non}}\ (\underline{\text{E}}\ y{:}\ \text{bosent}(Ly)) \tag{6}$$

$$(\text{term}(L)\ \underline{\text{and}}\ \text{adop}(y)) \Rightarrow \underline{\text{non}}\ \text{boterm}(Ly) \tag{17}$$

$$(\text{prim}(L)\ \underline{\text{and}}\ \text{mult}(y)) \Rightarrow \underline{\text{non}}\ \text{boprim}(Ly) \tag{28}$$

Relation (6) follows from the fact that bosent(Ly) implies boexp(L), and from our syntax for < exp >, < term >, < prim >, and < iden >, this implies that L does not contain a semicolon; sent(L) implies according to (8) that L does contain a semicolon. This is the contradiction that follows from the assumption that (6) does not hold; hence (6) has been proved. In order to prove (17) --under the assumption of (28)!-- we observe that with

< term > ::= < plier > < prim >

< boterm > ::= < plier > < boprim >

the negation of (17)

term(L) and adop(y) and boterm(Ly)

would imply that < prim > < adop > could be of the form < boprim >. It

therefore suffices to prove that

(prim(L) and op(y)) => non boprim(Ly) with
< op > ::= < adop > | < mult > .

This last implication can be proved by deriving a contradiction from its negation:

prim(L) and op(y) and boprim(Ly) ;

it can be done using Lemma 1 and Lemma 2, and I gladly leave this detail to the reader.

* * *

In view of the lenth of this report, the transformation from infix to postfix notation --on page EWD550 - 0 announced as "our ultimate goal"!-- will be postponed and left for some later document.

History.

Nearly three years ago I wrote a seven-page report, EWD375 "A non algebraic example of a constructive correctness proof." in which (essentially) the same problem has been tackled as here. Last January, while I was lecturing in La Jolla, Jack Mazola urged me to show a more complicated example; I tried to reconstruct the argument of EWD375 on the spot and failed.

Last February, when I was home again, I reread EWD375 and it left me greatly dissatisfied. I remembered that EWD375 had been a cause for great enthusiasm when it was written, and I could not understand that enthusiasm anymore. I found EWD375 very hard to read and hardly convincing: what three years ago I had considered as "a proof" now struck me at best as "helpful heuristics". (A strange experience to be nearly ashamed of what had been a source of pride only a few years ago!)

It was now clear why, last January in La Jolla, I was unable to give on the spot a formal treatment of the syntax analyzer: it was not just a failure of memory, it was also a profound change in my standards of rigor (undoubtedly also caused by the fact that over the last few years I burned a few fingers!). I decided to forget EWD375 and to start again from scratch. This document is the result of that effort.

It has been surprisingly hard to write. After the first six pages had been written --I had only dealt with sentsearch-- there has been a long pause before I gathered the strength and the courage to tackle expsearch, and for a few weeks I put the unfinished document away. To undertake the treatment of primsearch proved to be another hurdle.

What the final document does not show is that the notation eventually used for the assertions, the theorems and the proofs is the result of many experiments. Before we invented, for instance, the trick to use the predicate pqr(K) to denote that the string K belongs to the syntactic category < pqr > all our formulae became unwieldy; so they did, as long as we indicated concatenation of strings with an explicit operator instead of --as eventually-- just by juxtaposition. I hesitated, when I wrote --as on the middle of page EWD550 - 3-- sent(L;) because I saw problems coming by the time that I had to write such predicates for strings containing unmatched parentheses; the trick of introducing < open > and < close > solved that problem. Instead of (8) I should have written

< sent > ::= < exp > < semi >
< semi > ::= ;

Again, at the time of writing, also this report has been a source of great excitement. This is somewhat amazing as it does not contain a single deep thought! Is it, because we now still remember how much more beautiful it is than all the rejected efforts? I wonder how I shall feel about it in a few years time!

Acknowledgement.

I am greatly indebted to W.H.J.Feijen, M.Rem, A.J.Martin and C.S.Scholten, whose encouragement and active participation have been absolutely essential. And I am grateful to Jack Mazola for prividing me with the incentive.

19th of March 1976
Burroughs
Plataanstraat 5
NUENEN - 4565
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow