FORMAL DEVELOPMENT OF CORRECT ALGORITHMS: AN EXAMPLE BASED ON EARLEY'S RECOGNISER

C B Jones
Programming Technology Dept
Product Test Laboratory
IBM Laboratories Ltd
Hursley Park
WINCHESTER, Hants

ABSTRACT

This paper contains the formal development of a correct algorithm from an implicit definition of the task to be performed. Each step of the development can be accompanied by a proof of its correctness. As well as ensuring the correctness of the final program, the structured development gives considerable insight into the algorithm and possible alternatives. The example used is a simplified form of the recognition algorithm due to Earley.

0 INTRODUCTION

The published literature on proving programs correct has tended to confine itself to rather simple algorithms. Apart from the obvious difficulties of long papers, part of the reason for avoiding proofs of larger programs may be that they are almost invariably incorrect! In order to avoid writing a plausible algorithm and then constructing an equally plausible proof which misses the errors, it is necessary for the proof construction to be extremely detailed. The combination of large programs and detailed proofs makes the whole picture difficult to grasp: any errors that are present become elusive.

Is there an alternative ? This paper supports an affirmative answer by going through a formal development of a reasonable-sized algorithm. The steps of development represent a, hopefully, natural evolution from the problem specification to the final program. Each step is stated in a formal notation which makes it possible to construct the proof.† Although the overall size may appear large, the structure of the development makes it easy to absorb.

At each stage of development certain assumptions are made which may range from implicit function definitions to properties of data types and their operations. The same stage will, in turn, be further defining functions or adding properties of data types. A proof can be constructed that, under the new set of assumptions, this stage of development fulfils the earlier assumptions. The new assumptions provide, of course, the specification for the next stage. Thus, at each level, it is only necessary to prove that the hypotheses of the preceding level hold: not to prove again earlier results. Moreover, the development is not a chain of equivalences. It is a completely structured development in which it is possible to think, and prove theorems, about one set of problems at a time.

An example of this development taken from the paper begins with an implicit definition of a set (i.e. an object with no order); at the next stage operations are given whose closure on an initial set is shown to satisfy the implicit definition; a mapping of the set onto a list is then given as the next stage and the problems of mimicking the closure are considered.

Although the author ventures some general comments in the next section, this paper does not purport to present a worked out method of formal development. The hope is to encourage further work on a range of examples. Such work should throw light on some of the problems referred to in the discussion of Section 10. In particular there are some areas requiring a much more thorough approach. However, if this is handled correctly further developments should be less difficult than that given below because they can appeal to general theorems.

The body of the paper (Sections 3 to 7 and 9), consists of the development of the algorithm. The example chosen was a recogniser using a simplified form of the ideas of Earley. The reader is asked to refer to ref 1 for a description since such a readable presentation would make detailed exposition, in the current paper, pointless. Suffice it to say, that the algorithm carries out all possible top down parses in parallel, keeping track of them in sets of states. (The simplification made for the current paper was to omit Earley's H_k look-ahead.)

As a result of a draft of this paper the question of modifying developed programs was posed: an attempt to address this question is made in Section 8 with an, albeit rather simple, modification.

† The proofs, which are omitted from the current version of the paper for reasons of space, are to be published in a report available from the author.

1 GENERAL COMMENTS

As already mentioned, this section is far from presenting a complete general method for Formal Development. It is confined to making some observations resulting from the development contained below.

The most pervasive rule in Formal Development appears to be to define as little as possible at each step. Thus only enough new properties are introduced to give some point to the step. Addition of too much, firstly, clouds the structure of the proof and, secondly, may be regretted. The process is addition of properties at each step, removal would require backing up to the point of introduction. An example of the addition of properties is the use of (finite) sets as data types until the main lines of the algorithm are clear, at which time the additional properties of ordering are considered.

The actual mode of development used by the author was at each step to sketch a solution using "understood" notation; then to go back and, in proving that step formally, write down exactly what properties were required of the notation. This provides a detailed specification for the next level.

There is an important division in the subsequent development between the areas with which the development is concerned. Sections 4 and 5 are exclusively related to the Problem domain, the subsequent development is related more to the Data and Language domains.

Comments on the process of modification are left to Section 8.

2 NOTATION

This section reviews the notation used below. Whilst an attempt has been made to minimize the quantity of non-standard notation, use has been made of parts of, so called, VDL (see ref 2 for a fuller treatment). Those parts adopted are defined below in a way which, hopefully, makes this paper self contained.

	normal function notation: conditional expressions, where clauses, etc.	standard meanings
(2-1)	$f:A\to B$	f is total over A
	$f_a:B\to C$	specialisation of $f:A \times B\to C$ to $a\in A$
	$\in,\notin,\subseteq,\cup,\cap,\phi,\ \{x\mid p(x)\}$	normal set operations
	$\beta(P) = \{s\mid s\subseteq P\}$	the power set (i.e. set of all subsets)
	$N = \{1,2,3,\ldots\}$	natural numbers
	$N^0 = \{0,1,2,\ldots\}$	integers
	$N^n = \{1,2,\ldots,n\}$	natural numbers up to n
	$\sim,\vee,\wedge,\supset,\equiv,\exists$	normal logical connectives
	$(\exists\ x)_{x\in X}(p(x))$	bounded quantification
	$(\exists!x)(p(x))$	there exists exactly one x, i.e. $(\exists x)(p(x)) \wedge (p(x) \wedge p(y) \supset x=y)$
	$(\iota x)(p(x))$	that unique x, under above hypothesis
(2-2)	a & b	the conditional expression form of "and" (i.e. false if first operand yields false even if second is undefined)

Certain sets of elementary objects are given (e.g. N). Composite objects can be viewed as finite tree structures with elementary objects at the terminal nodes. Branches are named with, so called, selectors. No two branches emanating from the same node may have the same selector name. Subtrees of a given object are called components . As well as naming branches, selectors are used to select components from given objects.

`s(x)` — the component of x named s

(2-3) `is-p = (<s-1:is-p-1>,<s-2:is-p-2>,...,<s-n:is-p-n>)` — predicate defining a class of objects each with n sub-components etc.

(2-4) `P = {x|is-p(x)}` — set of objects satisfying is-p

(2-5)
```
s-1:P->P-1
s-2:P->P-2
   :
   :
s-n:->P-n
```
selectors of 2-3 viewed as functions

`s-2(s-1(x))` — iterated selection

Selectors are used in the sequel only when such a component exists. The convention of using the prefixes "is-" and "s-", for predicates and selectors respectively, is followed. Certain "obvious extensions" of strict VDL are employed, e.g.

(2-6) `is-p = is-(<s-1:is-p-1>,<s-2:is-p-2>)`

(2-7) `L = [el-1,el-2,...,el-n]` — lists can be considered as objects whose components are named by a subset of the natural numbers (N^n).

(2-8) $\ell(L) = n$ — length of list

(2-9) $1 \leq i \leq \ell(L): L_i$ = el-i — selection shown by conventional subscript notation

`is-p-list` — class of all finite lists whose elements satisfy is-p

(2-10) ℓ:P-LIST->N^o

(2-11) N^n:P-LIST->P

(2-12) is-p-list^o — used for lists which index from zero

(2-13) is-p-set — denotes a class of finite sets whose members satisfy is-p

(2-14) $\in$:P β(P)->{T,F} — "is a member of" written as an infix predicate symbol

(2-15) $x \in \{e|p(e)\} \equiv p(x)$ — implicit set definition

(2-16) $\cup$:S×S-> S — set union, such that $x \in (A \cup B) \equiv (x \in A \vee x \in B)$

(2-17) elements of is-p-set cannot be "duplicates"

3 PROBLEM DEFINITION

This section gives the definition of the task to be performed by the algorithm developed in subsequent sections. In outline: the class of grammars is defined in 3-1 to 3-4 in the familiar terminal/non-terminal way; the class of strings which can be produced from a given grammar is defined in 3-5 and 3-6; a function "root" is introduced in 3-9 which is assumed to provide the non-terminal from which acceptable strings must be derivable. The recognition task, for given string and grammar, can now be defined in 3-10 using the above notions. A useful Lemma on productions is given in 3-7.

The class of grammars is defined:

(3-1) is-grammar = is-rule-set

which is assumed to have properties 2-13, 2-14

(3-2) is-rule = (<s-lhs:is-nt> , <s-rhs:is-el-list>)

properties 2-5 and 2-10, 2-11 are assumed

$\bar{r}$ will be used as an abbreviation for ℓ(s-rhs(r))

(3-3) is-el = is-nt $\vee$ is-t

(3-4) NT $\cap$ T = ϕ non-terminal and terminal sets are disjoint

The following production symbols are defined:

for is-el-list(α), is-el-list(β), is-grammar(G):

(3-5) $\alpha\ A\ \beta \underset{G}{\Rightarrow} \alpha\ \gamma\ \beta \equiv (\exists_{r \in G} r)$ (s-lhs(r) = A $\wedge$ s - rhs(r)=γ)

(3-6) $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta \equiv (\exists m)_{0 \le m} (\exists \alpha_0, \alpha_1, \ldots, \alpha_m)(\alpha = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_m = \beta))$

In the sequel, the grammar can be omitted from the production symbol with no danger of confusion.

The following Lemma can be proven from 3-5, 3-6.

(3-7) Lemma $\gamma \overset{*}{\Rightarrow} \tau \supset (\exists t_1, \ldots, t_{\ell(\gamma)})(1 \le t_1 \le t_2 \le \ldots \le t_{\ell(\gamma)} = \ell(\tau) \wedge$

$$\gamma_1 \overset{*}{\Rightarrow} \tau_1 \cdots \tau_{t_1} \wedge$$
$$\vdots$$
$$\gamma_{\ell(\gamma)} \overset{*}{\Rightarrow} \tau_{t_{\ell(\gamma)-1}+1} \cdots \tau_{t_{\ell(\gamma)}})$$

The "specification" of the algorithm to be developed, say rec, can now be stated as follows:

(3-8) rec : T-LIST $\times$ GRAMMAR -> {YES, NO}

(3-9) such that for is-t-list assuming properties 2-10,2-11

is-grammar(G) see 3-1

root : GRAMMAR -> NT a predefined function

(3-10) rec (X,G) = YES iff root(G) $\overset{*}{\Rightarrow}$ X

NO otherwise

Given that 3-8 requires that rec be total (see 2-1), the second part of 3-10 is not proved explicitly.

4 EARLEY'S STATE SETS

The reader is assumed to be familiar with the concepts of states and state sets as described by Earley in ref 1. (The simplification in the current paper drops the fourth element of Earley's states). Collections of state sets correspond to objects satisfying is-S (see 4-2) from which individual state sets can be retrieved using the function "state-set". The only properties required of the data object containing state-sets (see 4-3) are the applicability of the predicate "is member of" and the lack of duplicate members. Objects containing states are defined by 4-4 to be triples whose contents will be referred to via given selectors or using the specified abbreviation. More properties of the storage will be added as the algorithm evolves, those given so far are enough to show the first ideas about the algorithm.

It is convenient to assume that there is only one rule of the grammar of which root (G) is the left hand side, this restriction is made in 4-1 since it has no real effect on the class of valid grammars:

(4-1) $(\exists!r)\ (s\text{-}lhs(r) = root(G))$
$r \in G$

The object containing all state-sets satisfies:

(4-2) is-S

which is characterised by the existence of two functions:

len : S -> N
state-set : N^n x S -> STATE-SET
where N^n is the set {0,1,...,len(S)}

(4-3) is-stateset = is-state-set

of which 2-14 and 2-17 are required to hold.

(4-4) is-state = (<s-rule:is-rule> , <s-j:is-n> , <s-f:is-n>)

of which 2-5 is required to hold.

The abbreviation:

s = <r,j,f> is used for the object for which s-rule (s) = r
s-j (s) = j
s-f (s) = f

States will be used to record partial parses. The presence of a particular state, say <r,j,f>, in the ith state set will mean that a string can be derived from the grammar, G, which consists of the first f characters of X (the string to be recognised) followed by left hand side of r, and that the first j elements of the right hand side of r can produce, from G, the string $X_{f+1}...X_i$. This property is stated formally in 4-7 below and represents an upper bound for valid state-sets: any state present must possess this property. In order to permit the "rec" algorithm to decide recognition, not all of these states are essential so 4-6 and 4-8 set a lower bound on valid state-sets: the former specifies a start element which must be present; the latter specifies that if <r,j,f> is in the ith state set and the next (i.e. j+1 th) element of the right hand side of r can produce $X_{i+1}...X_m$ then <r,j+1,f> must be a member of the mth state-set.

It is fundamental to the style of development proposed by this paper that the freedom, to choose which set between the lower and upper bound will be generated, is left for the time being. The properties given permit the development of the first steps of the algorithm (see 4-5) and the proof that, under the assumptions on the creation of S, it fulfils the task described in Section 3. This freedom is used to show the correctness of an optimisation modification in Section 8 and could be used to validate many different algorithms including the use of Earley style look-ahead. Furthermore, the proof of the actual construction of state sets constructed in Section 5 has been made simpler by only having to prove that these looser properties hold.

The definition of a function, alleged to satisfy 3-8, 3-10, can now be given:

(4-5) rec (X,G) = $\langle rr,\bar{\bar{r}},0\rangle \in$ state-set $(\ell(x),S)$ -> YES

T -> NO

where is-S (S)
and len (S) = $\ell(x)$
and $rr = (\iota r)_{r\in G}(s\text{-}lhs(r) = root(G))$

The correctness of this function relies on the following assumptions about S. (These assumptions will, of course, form the definition for further development.) Some base element must exist in S:

(4-6) $\langle rr,0,0\rangle \in$ state-set $(0,S)$

Any element of a state set must, at least, satisfy the properties:

(4-7) $0\le i\le\ell(X)$ & $\langle r,j,f\rangle \in$ state-set $(i,S) \supset r \in G \wedge$
$(\exists\alpha$ is-el-list$(\alpha))(root(G) \overset{*}{\Rightarrow} X_1 \ldots X_f\ s\text{-}lhs(r)\alpha) \wedge$
$(j>0 \supset s\text{-}rhs(r)_1 \ldots s\text{-}rhs(r)_j \overset{*}{\Rightarrow} X_{f+1}\ldots X_i) \wedge$
$j\le\bar{r} \wedge 0\le f\le i$

Certain elements must be in the state-sets:

(4-8) $0\le i<\ell(x)$ & $\langle r,j,f\rangle \in$ state-set$(i,S) \wedge j \neq \bar{r}$ & $(\exists m_{i+1\le m\le\ell(X)})(s\text{-}rhs(r)_{j+1} \overset{*}{\Rightarrow} X_{i+1}\ldots X_m)$

$\supset \langle r,j+1,f\rangle \in$ state-set(m,S)

(4-9) S must be created in a finite amount of time.

Property 4-9 above is given to ensure that any subsequent algorithm generates state-sets in a useful way and permits the proof (see 4-13) that 4-5 is total. After a trivial Lemma (4-10) on the finiteness of state-sets, the proofs that both implications of the first part of 3-10 hold are given in 4-11 and 4-12. Of course, these proofs all rely on the assumptions that S satisfies 4-6 to 4-9.

(4-10) Lemma for $0\le i\le$len(S) : state-set (i,S) is finite.

proof follows from finiteness of G and $\bar{r}$.

(4-11) Theorem rec (X,G) = YES $\supset$ root(G) $\overset{*}{\Rightarrow}$ X

proof follows from 4-7.

(4-12) Theorem root(G) $\overset{*}{\Rightarrow}$ X $\supset$ rec (X,G) = YES

proof follows from 4-6 and 4-8 using 3-7.

(4-13) Theorem rec is total

proof follows from 4-10 and 4-9 and check of domains/ranges.

5 CREATING STATE SETS

Section 4 introduced the concept of state sets and some essential properties thereof, but offered no way of creating them. In fact (s-rhs(rr)) entries, presumably created by magic, would have been sufficient to fulfil the properties stated. This section shows how state-sets can be constructed using the main ideas (i.e. prediction, completion, scanning) of Earley's method. In order to facilitate construction of the required elements, many other states are generated. It is interesting to consider the parallel with proof construction where an induction hypothesis stronger than the theorem statement is used in order to prove the latter. First, the definition of rr which was in the last section a local convention, is adopted for the sequel.

(5-0) $rr = (\iota r)_{r \in G}(s\text{-lhs}(r) = root(G))$ see 4-1

Earley's operations of prediction etc, are, of course, described in terms of lists of states. As will be clear shortly, many of the interesting points about these operations can be brought out using sets of states. Again it is found that deferring part of the detail, in this case the special ordering problems imposed by using lists, will structure and clarify the proofs. It will be necessary to use implicit set notation for creation of state sets so now the restriction:

(5-1) Property 2-15 holds for is-stateset is added

Using sets of states which are cumulative, prompts the idea of using the notion of the closure of a set under an operation as generating function. Intuitively the closure of an operation on a set is the minimum set containing the base set and having the property that applying the operation to any element creates only elements already in the set. We shall use the following properties:

for op : P -> β(P) i.e. from elements of P to sets of such elements

(5-2) closure : (op) $\times$ β(P) -> β(P) i.e. from sets of elements to sets of elements

such that for $S \subseteq P$:

$$x \in (\text{closure of op on } S) \equiv x \in S \vee (\exists_{y \in s}\, y)(x \in \text{closure of op on } (op(y)))$$

Notice that closure is total for suitable op. It should be clear that closure is monotonic. That is:

(5-3) $S_1 \subseteq S_2 \supset (\text{closure of op on } S_1) \subseteq (\text{closure of op on } S_2)$

Using closure it is now possible to show how the basic operations (i.e. prediction etc) are used to create the statesets. The base element is inserted by 5-4 which also shows that prediction is the only applicable operation on the 0th stateset. The creation, for all other statesets, of a kernel set of states resulting from the scanning operation on the preceding stateset is specified in 5-5. Both the prediction and completion operations are employed on all statesets but the last, where the former is not required. The closure is created from the kernel set described. The importance of the reliance by the completer on S (strictly on a part of S) is returned to in the next section.

(5-4)† state-set $(0,S)$ = closure of $\text{predict}_{0,X,G}$ on $\{<r,0,0>\}$

(5-5) for $1 \le i \le \ell(X)$: state-set $(i,S)^s$ = scan (i,X,G,state-set(i-1,S))

(5-6) for $1 \le i \le \ell(X)$:
state-set (i,S) = closure of $\left\{ \begin{matrix} \text{predict}_{i,X,G} \\ \text{complete}_{S,i,X,G} \end{matrix} \right\}$ on state-set $(i,S)^s$

(5-7) state-set $(\ell(X),S)$ = closure of $\text{complete}_{S,i,X,G}$ on state-set $(\ell(X),S)^s$

† Notice use of function specialisation, see Section 2.

The definitions of the basic operations of prediction, scanning and completion can now be given. For reasons discussed in Section 9, it is important to guard against predicates becoming undefined by careless use of "$\wedge$" which is not normally defined for three valued logic. (see ref 4)

(5-8) predict (i,X,G,<r,j,f>) = $\{<s,0,i> \mid j \neq \bar{r}\ \&\ \text{is-nt}(\text{s-rhs}(r)_{j+1})\ \&\ s \in G\ \&\ \text{s-lhs}(s) = \text{s-rhs}(r)_{j+1}$

(5-9) scan (i,X,G,ss) = $\{<r,j+1,f> \mid <r,j,f> \in ss \wedge j \neq \bar{r}\ \&\ \text{is-t}(\text{s-rhs}(r)_{j+1})\ \&\ X_i = \text{s-rhs}(r)_{j+1}\}$

(5-10) complete (S,i,X,G,<r,j,f>) =

$\{<s,m+1,g> \mid j=\bar{r}\ \&\ <s,m,g> \in \text{state-set}(f,S)\ \&\ m \neq \bar{s}\ \&\ \text{is-nt}(\text{s-rhs}(s)_{m+1})\ \&\ \text{s-rhs}(s)_{m+1} = \text{s-lhs}(r)\}$

In order to resolve any possible difficulties deriving from too heavy a reliance on formalism the example used in ref 1 is now presented in the terms of the current paper. (Notice the addition of R to satisfy 4-1)

G = {r1, r2, r3, r4, r5, r6} where

r1 = <<s-lhs : R>,<s-rhs : [E]>>
r2 = <<s-lhs : E>,<s-rhs : [T]>>
r3 = <<s-lhs : E>,<s-rhs : [E,+,T]>>
r4 = <<s-lhs : T>,<s-rhs : [P]>>
r5 = <<s-lhs : T>,<s-rhs : [T,*,P]>>
r6 = <<s-lhs : P>,<s-rhs : [a]>>

such that is-nt(R), is-nt(E), is-nt(T), is-nt(P), is-t(+), is-t(*), is-t(a), rr = r1

Consider the recognition of:

X = [a,+,a]

state-set (0,S) = closure of predict on {<r1,0,0>}
= closure of predict on { ,<r2,0,0>}
= closure of predict on { ,<r4,0,0>,<r5,0,0>}
= closure of predict on { ,<r6,0,0>}
= {<r1,0,0>,<r2,0,0>,<r4,0,0>,<r5,0,0>,<r6,0,0>}

state-set (1,S)^s = scan (state-set (0,S))
= {<r6,1,0>}

state-set (1,S) = closure of predict/complete on state-set(1,S)^s
= closure of predict/complete on {<r6,1,0>,<r4,1,0>,<r5,1,0>}
= closure of predict/complete on { ,<r2,1,0>,<r3,1,0>}
= closure of predict/complete on { ,<r1,1,0>}
= {<r6,1,0>,<r4,1,0>,<r5,1,0>,<r2,1,0>,<r3,1,0>,<r1,1,0>}

state-set(2,S)^s = scan(state-set(1,S))
= {<r3,2,0>}

state-set(2,S) = closure of predict/complete on state-set (2,S)^s
= closure of predict/complete on {<r3,2,0>,<r4,0,2>,<r5,0,2>}
= closure of predict/complete on { ,<r6,0,2>}
= {<r3,2,0>,<r4,0,2>,<r5,0,2>,<r6,0,2>}

state-set(3,S) = scan (state-set(2,S))
= {<r6,1,2>}

state-set(3,S)^s = closure of complete on state-set(3,S)^s
= closure of complete on {<r6,1,2>,<r4,1,2>}
= closure of complete on { ,<r3,3,0>,<r5,1,2>}
= closure of complete on { ,<r1,1,0>}
= {<r6,1,2>,<r4,1,2>,<r3,3,0>,<r5,1,2>,<r1,1,0>}

Referring to 4-5 gives:

rec (X,G) = YES

The algorithm represents a way of generating some set of states between those specified by 4-8 and 4-7. The justification† consists of proving this containment (see 5-12, 5-13) and establishing 4-6 and 4-9 (see 5-11, 5-14 respectively). Notice that this is all! It is not necessary to prove again any of the results dealt with in Section 3.

(5-11) Theorem $\langle rr,0,0\rangle \in$ state-set $(0,S)$

proof follows from 5-4, 5-3.

(5.12) Theorem

$$0\leq i\leq \ell(X) \;\&\; \langle r,j,f\rangle \in \text{state-set}\,(i,S) \supset r \in G \;\wedge\; (\exists \alpha \text{ is-el-list}(\alpha))(\text{root}(G) \overset{*}{\Rightarrow} X_1 \ldots X_f\ \text{s-lhs}(r)\alpha) \;\wedge\; (j>0 \supset \text{s-rhs}(r)_1 \ldots \text{s-rhs}(r)_j \overset{*}{\Rightarrow} X_{f+1}\ldots X_i) \;\wedge\; j\leq \bar{r} \;\wedge\; 0\leq f\leq i$$

proof: shows that the set satisfying the above is a fixed point of 5-4 to 5-7 (i.e. that given such a set the operations 5-8, 5-9, 5-10 create only elements of that set).

(5-13) Theorem

$$0\leq i<\ell(X) \;\&\; \langle r,j,f\rangle \in \text{state-set}(i,S) \;\wedge\; j\neq \bar{r} \;\&\; (\exists m\ i+1\leq m\leq \ell(X))(\text{s-rhs}(r)_{j+1} \overset{*}{\Rightarrow} X_{i+1}\ldots X_m) \supset \langle r,j+1,f\rangle \in \text{state-set}(m,S)$$

proof by induction on n(the number of => steps in $\overset{*}{\Rightarrow}$ see 3-5, 3-6, using 5-8, 5-9, 5-10.

(5-14) Theorem S is created in finite time

proof follows from the closure and finiteness (4-10)

6 MAPPING STATE SETS ONTO LISTS

So far state sets have been assumed to be set-like objects and they have been extended by closure. Whilst it would be possible to model this state of affairs in a conventional store it would be much more convenient if we could show how the sets can be mapped onto lists. The lists will obviously have extra properties, in particular an ordered property. If closure can now be replaced by a new, more efficient, operation which relies on this ordering, an increase in efficiency results. This section makes precisely this mapping. Subject to a restriction on grammars (see 6-9) the extension is performed by a single scan over the lists. It is, in the author's opinion, worthwhile understanding the cause of the restriction. It would for instance be a relatively simple task, at this level, to avoid introducing the restriction by utilizing a more sophisticated scan.

So far the only properties assumed of state-sets are those required by 4-3 and 5-1. That is:

$\in$: STATE×STATESET -> {T,F}
x $\in$ {state | p(state)} $\equiv$ p(x)
statesets do not contain duplicates

This section describes how these objects can be mapped onto lists. Thus:

(6-1) is-stateset = is-state-list assuming properties 2-10 and 2-11.

† It should be possible to construct a direct proof that a set given by 4-7, with equivalence replacing implication, is the minimum fixed point (see ref 3) of 5-4 to 5-10. The current author's attempts have foundered on the ordering problems caused by the completer.

Now the membership and implicit set definitions are reinterpreted as follows:

(6-2) $e \in s$ becomes $(\exists i_{1\le i\le \ell(s)})(s_i = e)$ but will be abbreviated $e \underline{\in} s$

(6-3) $\{s \mid p(s)\}$ becomes create-state-list(p) but will be abbreviated $\underline{\{}s\rfloor p(s)\underline{\}}$ with the following properties:

let $\ell(\underline{\{}s\rfloor p(s)\underline{\}}) = n$

$$p(x) \supset (\exists !j_{1\le j\le n})(\underline{\{}s\rfloor p(s)\underline{\}})_j = x)$$

$$1\le j\le n \supset p(\underline{\{}s\rfloor p(s)\underline{\}})_j$$

A new operation will be required to manipulate state-sets.

(6-4) combine $(list_1, list_2)$ which will be written $\underline{\cup}$

with the following properties:

$$s \underline{\in} (list_1 \ \underline{\cup}\ list_2) \equiv s \underline{\in} list_1 \vee s \underline{\in} list_2$$

$$(\exists j\ k_{1\le j,k\le \ell(list_1 \underline{\cup} list_2)})((list_1 \ \underline{\cup}\ list_2)_j = (list_1 \ \underline{\cup}\ list_2)_k \supset j = k)$$

$$1\le i\le \ell(list_1) \supset (list_1 \ \underline{\cup}\ list_2)_i = (list_1)_i$$

The properties required of state-sets can now be verified:

(6-5) Lemma

a) $\underline{\in}$: STATE × STATESET -> {T,F}

b) $x \underline{\in} \underline{\{}state \rfloor p(state)\underline{\}} \equiv p(x)$

c) statesets contain no duplicates

Operating on lists makes it desirable to replace the closure operations (see 5-2) with linear scans. It will be found necessary to introduce a restriction on grammars in order to fulfil property 5-2.

The construction of is now given by:

(6-6) $S = S^z$ <u>where</u> next (z+1) = <u>end</u>

(6-7) next (1) = (0,1)
for i>1:

next (i) = ℓ(state-set(j,S^{i-1})) = k -> j≠ℓ(X) & state-set(j+1,S^{i-1})≠[] ->(j+1,1)
T -> <u>end</u>
T -> (j,k+1)
<u>where</u> (j,k) = next (i-1)

(6-8) $\text{state-set}(0,S^0) = [\langle rr,0,0\rangle]$

for $0<i<\ell(X)$: $\text{state-set}(i,S^0) = []$

for $k\geq 0$:

$j \neq \bar{r} \wedge i<\ell(X)$ & $\text{is-nt}(\text{s-rhs}(r)_{j+1})$ $\supset$

$\text{state-set}(i,S^{k+1}) = \text{state-set}(i,S^k) \cup \{s \mid s\in \text{predict}(i,X,G,\langle r,j,f\rangle)\}$

$j \neq \bar{r} \wedge i<\ell(X)$ & $\text{is-t}(\text{s-rhs}(r)_{j+1})$ & $\text{s-rhs}(r)_{j+1} = X_{i+1}$ $\supset$

$\text{state-set}(i+1,S^{k+1}) = \text{state-set}(i+1,S^k) \cup \{s \mid s\in \text{scan}(i+1,X,G,\langle r,j,f\rangle)\}$

$j = \bar{r} \supset \text{state-set}(i,S^{k+1}) = \text{state-set}(i,S^k) \cup \{s \mid s\in \text{complete}(S^k,i,X,G,\langle r,j,f\rangle)\}$

otherwise $\text{state-set}(p,S^{k+1}) = \text{state-set}(p,S^k)$

where $\langle r,j,f\rangle = \text{state-set}(i,S^k)_n$

where $(i,n) = \text{next}(k+1)$

The "algorithm" as shown would not work for all possible grammars! In particular if the grammar includes rules which generate the empty string, the sequential pass over state-sets coupled with the fact that although created many times only one copy of a state appears in a state-set, prevents the completer creating the full closure.

Consider $G = \{r1,r2\}$ where $r1 = \langle\langle \text{s-lhs} : R\rangle,\langle \text{s-rhs} : [a,E,E,a]\rangle\rangle$

$r2 = \langle\langle \text{s-lhs} : E\rangle,\langle \text{s-rhs} : []\rangle\rangle$

$x = [a,a]$

$\text{state-set}(0,S) = [\langle r1,0,0\rangle]$
$\text{state-set}(1,S) = [\langle r1,1,0\rangle,\langle r2,0,1\rangle,\langle r1,2,0\rangle]$
and no more elements will be appended to the list.

The following restriction on grammars is made:

(6-9) $r \in G \supset \bar{r} > 0$

It should be clear that Lemma 3-7 now becomes:

(6-10) $\gamma \overset{*}{\Rightarrow} \tau \supset (\exists t_1,\ldots,t_{\ell(\gamma)})(1\leq t_1<t_2<\ldots<t_{\ell(\gamma)} = \ell(\tau) \wedge \gamma_1 \overset{*}{\Rightarrow} \tau_1\ldots\tau_{t_1} \wedge$

$\vdots$

$\gamma_{\ell(\gamma)} \overset{*}{\Rightarrow} \tau_{t_{\ell(\gamma)-1}+1}\ldots\tau_{t_{\ell(\gamma)}})$

The proof that the state-sets of Section 5 and the state lists of this section contain the same elements (assuming 6-9) is now given in 6-11. Since closure was considered total (see 5-2) it is also necessary to show that the scanning which replaced it is total. Notice that, once again, the theorems revolve around only those things which have been changed, and the earlier theorems still apply because properties are only enhanced.

(6-11) Theorem $s \in \text{state-set}(i,S^z)$ see 6-8

iff

$s \in \text{state-set}(i,S)$ see 5-4,5-5,5-6,5-7

proof follows from the inspection of each state by next; mutual disjoint conditions; and 6-10 for completer.

Because 5-2 claimed completeness for closure use must also prove -

(6-12) Theorem The creation of S given in 6-6 to 6-8 is total

proof follows from inspection of domains/bounds

7 STRUCTURE FOR STORE

The preceding section represented a solution to the main parts of the recognition task. However, for a number of reasons it is not a program which can be run on a computer. The principal point is that the algorithm of Section 6 uses a storage object which is not representable in any known language, for example the grammar is just "referred to" whenever required. It might be possible to sit at a console and supply rules whenever needed, but a more useful program will result if the first step is to read in both grammar and string to be recognised in such a way that subsequent references are simple. The particular storage structure given below has been adopted from an actual program written in the conventional way and designed with a view to efficiency. This decision was taken so as to avoid the danger of choosing a data structure particularly suited to the proof but which put limitations on the potential efficiency of the final algorithm.

The input of grammar and string will be assumed to be carried out by routines called INGR and INSTR respectively. The grammar is stored in the s-RULES, s-NONT, s-RULED, s-STR and s-STRCHAR; the string in the s-INPUT components of the store given in 7-1 to 7-7 below. What were, in the algorithm references to the original arguments, now become references to these storage components and the reinterpretations to be made are specified in 7-9 to 7-17.

The object is-S is also refined in this section and now occupies the s-S and s-STATE parts of 7-1 to 7-7.

The store is still presented as a VDL style object and a further transition to PL/I Data Structures will be made later. The decision to introduce this step is perhaps questionable, but is probably justified since the object of 7-1 could be realised in other languages. The only properties required are those in 7-8 and although "bunches" were used with PL/I based storage in mind, the only objection to an array style implementation with numeric "selectors" would be that all elements would have to be of the same length.

Store is considered as an object satisfying:

```
(7-1)    is-ξ = (<s-S : is-state-ptr-list°>,
                 <s-STATE : is-STATE-bunch>,
                 <s-INPUT : is-t-list>,
                 <s-RULES : is-(<s-TYPE : is-T ∨ is-N>,
                                <s-N : is-n>) - list>,
                 <s-NONT : is-ruled-ptr-list>,
                 <s-RULED : is-RULED-bunch>,
                 <s-STR : is-(<s-START : is-n>)-list>,
                 <s-STRCHAR : is-char-list>)
         where -
(7-2)    is-state-ptr = is-sel
(7-3)    is-STATE-bunch = {<κ : state> || is-sel(κ) ∧ is-STATE(state)}
(7-4)    is-STATE = (<s-SIZE : is-n>,
                    <s-N : is-n>,
                    <s-INFO : is-(<s-RULE  : is-(<s-RPTR : is-ruled-ptr>,
                                                 <s-SSC : is-n>)>,
                                  <s-RULPOS : is-n>,
                                  <s-STRPOS : is-n>)-list>)
(7-5)    is-ruled-ptr = is-sel
(7-6)    is-RULED-bunch = {<κ: ruled> || is-sel(κ) ∧ is-RULED(ruled)
(7-7)    is-RULED = (<s-NEXT : is-n>,
                    <s-RULED : is-(<s-START : is-n>,
                                   <s-LENGT : is-n>)-list>)

(7-8)    In all object of 7-1 to 7-7:          selectors are assumed to have property 2-5
                                               lists are assumed to have properties 2-10, 2-11
```

Using the above data object as storage the algorithm can now use the following ways of referring to its arguments:

(7-9) $\ell(X)$ becomes ℓ(s-INPUT(ξ))

(7-10) X_i becomes s-INPUT(ξ)$_i$

Now, INGR defines a relation between rules of is-grammar and objects satisfying is-(<s-RPTR : is-ruled-ptr>,<s-SSC : is-n>) which is written:

$$\left.\begin{matrix} I(r,(p,i)) \\ \bar{r} \\ \ell(\text{s-rhs}(r)) \end{matrix}\right\} \quad \text{becomes} \quad \text{s-LENGT}\ (\text{s-RULE}(p(\text{s-RULED}(\xi)))_i)$$

$$\text{s-rhs}(r)_j \quad \text{becomes} \quad \text{s-RULES}(\xi)_{\text{s-START}\ (\text{s-RULED}(p(\text{s-RULED}(\xi)))_i)+j-1}$$

for convenience, let $I(\text{s-rhs}(r)_j)$ be an abbreviation for $\text{s-RULES}(\xi)_{\text{s-START}(\text{s-RULE}(p(\text{s-RULED}(\xi)))_i)+j-1}$

$$\text{is-t}(\text{s-rhs}(r)_j) \quad \text{becomes} \quad \text{s-TYPE}(I(\text{s-rhs}(r)_j)) = T$$

$$\text{is-nt}(\text{s-rhs}(r)_j) \quad \text{becomes} \quad \text{s-TYPE}(I(\text{s-rhs}(r)_j)) = N$$

for $\text{is-t}(\text{s-rhs}(r)_j)$:

$$\text{s-rhs}(r)_j \quad \text{becomes} \quad \text{s-STRCHAR}(\xi)_{\text{s-N}\ (I(\text{s-rhs}(r)_j))}$$

for $\text{is-nt}(\text{s-rhs}(r)_j)$:

$$\text{s-rhs}(r)_j = \text{s-lhs}(s) \quad \text{becomes} \quad \text{s-NONT}(\xi)_{\text{s-N}(I(\text{s-rhs}(r)_j))} = q \quad \underline{\text{where}}\ I(s,(q,k))$$

$$r \in G \quad \text{becomes} \quad \text{is-RULED}(p(\text{s-RULED}(\xi))) \wedge i<\text{s-NEXT}(p(\text{s-RULED}(\xi)))$$

Now, it is necessary to show that the reinterpretations of these functions still satisfy the stated properties (3-8 and 3-1 to 3-4).

(7-18) Lemma

ℓ:	T-LIST -> N^o	see 7-9,3-8,2-10
N^n:	T-LIST -> T	see 7-10,3-8,2-11

Both results follow immediately from 7-1, 7-8, 2-10, 2-11.

(7-19) Lemma

a)	s-RULES,s-NONT,s-RULED,s-STR,s-STRCHAR are finite	see 3-1,2-13
b)	ϵ: (RULED-PTR × N) × (RULES,...,STRCHAR) -> {T,F}	see 7-17,3-1,2-14
c)	ℓ: (RULED-PTR × N) × (RULES,...,STRCHAR) -> N	see 7-11,3-2,2-10
d)	N^n: (RULED-PTR × N) × (RULES,...,STRCHAR) ->	see 7-15,7-16,3-2, 2-11
e)	$\sim(\text{is-nt}(\text{s-rhs}(r)_j) \wedge \text{is-t}(\text{s-rhs}(r)_j))$	see 7-13,7-14,3-3, 3-4

proofs follow from definitions and 7-8.

However, it is not possible to simply change the problem definition (see 3-10 and 3-5, 3-6,) which is stated in terms of the original operations (e.g. $r \in G$). It was clear from the beginning that routines like INSTR and INGR would be required and it would have been possible to use different operations in the algorithm and note the equivalences which must hold between the original and algorithm notations. This was not done since it would have made the subsequent writing more tedious. In making the change to our re-interpreted operations we must now, carefully, check what properties are required and use them as a specification of the INSTR and INGR routines.

(7-20) Constraints on INSTR -

$$\ell(X) = \ell(\text{s-INPUT}(\xi))$$

$$X_i = \text{s-INPUT}(\xi)_i$$

(7-21) Constraints on INGR -

let $I(r,(p,i))$

$r \in G \equiv$ is-RULED(p(s-RULED(ξ))) ∧ i <s-NEXT(p(s-RULED(ξ)))

$\bar{r} =$ s-LENGT (s-RULE(p(s-RULED(ξ)))$_i$)

is-t(s-rhs(r)$_j$) ≡ s-TYPE (I(s-rhs(r)$_j$)) = T

is-nt(s-rhs(r)$_j$) ≡ s-TYPE (I(s-rhs(r)$_j$)) = N

for is-t(s-rhs(r)$_j$):

s-rhs(r)$_j$ ≡ s-STRCHAR(ξ)$_{\text{s-N }(I(\text{s-rhs(r)}_j))}$

for is-nt(s-rhs(r)$_j$):

s-rhs(r)$_j$ = s-lhs(s) ≡ s-NONT(ξ)$_{\text{s-N }(I(\text{s-rhs(r)}_j))}$ = q

where I(s,(q,k))

The storage structure given in 7-1 to 7-4 also gives us an interpretation of the collection of state-sets (see 4-2).

(7-22) len(S) becomes ℓ^0 (s-S(ξ))

(7-23) state-set(i,S) becomes s-S(ξ)$_i$(s-STATE(ξ))

(7-24) for is-state(s): s-rule(s) becomes s-RULE(s)

(7-25) s-j(s) becomes s-RULEPOS(s)

(7-26) s-f(s) becomes s-STRPOS(s)

It is, of course, necessary to check that the required properties still hold:

(7-27) Lemma ℓ^0, state-set,s-RULE,s-RULPOS,s-STRPOS behave as in 4-2,4-4

proofs all follow from 7-1,7-2,7-3,7-4,7-8.

The resulting reinterpretations of is-stateset are:

(7-28) ℓ: STATE-SET -> N becomes s-N(STATE)

(7-29) N^n: STATE-SET -> STATE becomes s-INFO(STATE)$_i$

(7-30) Lemma

The obvious equivalences hold.

Follows from 7-1,7-4,7-8

Notice that $\subseteq$ and $\cup$ (see 6-2 and 6-4) are automatically correct.

It would now be possible to rewrite the "algorithm" of Section 6 so that it uses the data structures of 7-1 to 7-7, i.e. using the various reinterpretations set out above.

8 A MODIFICATION

Changes of specification and new ideas to, for example, improve efficiency are facts of life: if Formal Development is to be an accepted way of constructing programs its influence on subsequent changes must be understood. It is the opinion of the current author that, if generality and abstractness have been the keynotes of the development, the difficulty of installing changes should be in realistic relation to how drastic that change is. Certainly more work will be required than with a "quick patch", but the same gains of certainty as with Formal Development should make the investment worthwhile.

The modification made in this section is an optimisation to reduce the number of irrelevant states created and scanned. It is, again, taken from a conventionally built version of the program. The concept is to append to rules an inevitable terminal character (i.e. a character with which any derivation of the rule must start), if such exists. This is specified formally in 8-2. During prediction this look ahead character, if it exists, is compared to the next character of the string: if they are unequal no state for this rule need be generated. Any such state would, after some number of further predictions, have been shown to be a "blind alley" by the scanner. This test will be performed by the predicate "cond" whose essential property is stated in 8-3: an, obvious, realisation is given in 9-13.

The effect of this change is now traced beginning with the definition of is-rule (3-2) which is extended to include the pre-computed look ahead symbols:

(8-1) is-rule = (<s-lhs : is-nt>,
<s-rhs : is-el-list>,
<s-lk : is-t>)

Further, the restriction on the look ahead character is:

(8-2) for r ∈ G : s-lk(r) ≠ ϭ ⊃ (s-rhs(r) $\overset{*}{\Rightarrow}$ α ∧ is-t(α_1) ⊃ α_1 = s-lk(r))

Now a check of the development process is made to determine the extent of the changes.

Consider the problem definition (Section 3) - there is no change to be made since the whole point is to recognise the same strings, but to do it more efficiently.

Consider the outline of Earley's state sets (Section 4) - there will be no change since the cut down state sets will still fall between those satisfying equations 4-7 and 4-8.

Consider the creation of State sets (Section 5) - A predicate "cond" is introduced which is assumed to satisfy the property:

(8-3) for r ∈ G : s-rhs(r) $\overset{*}{\Rightarrow}$ α ∧ is-t(α_1) ⊃ cond(r,α_1)

cond: RULE × T -> {T,F}

Then 5-8 is modified to employ cond:

(8-4) predict (i,X,G,<r,j,f>) = {<s,0,i>| j ≠ $\bar{r}$ & is-nt(s-rhs(r)$_{j+1}$) & s ∈ G & s-lhs(s) =
s-rhs(r)$_{j+1}$ ∧ cond(s,X_{i+1})}

The proof that the state-sets still satisfy 4-6 to 4-9 is modified as follows:

Theorem 5-11	unchanged
Theorem 5-12	unaffected, because the set satisfying 8-4 is clearly a subset of that satisfying 5-8.
Theorem 5-13	slight change related to cond

Consider the mapping of state sets to lists (Section 6) - no change to be made.

Consider the structuring for store (Section 7) - the reinterpretation of 8-1 (see 7-17) includes:

(8-5) s-lk (r) becomes s-STRCHAR(ξ)$_{s\text{-}N\ (I(s\text{-}rhs(r)_0))}$

and the additional constraint on INGR (see 7-21) -

(8-6) s-lk (r) = s-STRCHAR(ξ)$_{s\text{-}N\ (I(s\text{-}rhs(r)_0))}$

Notice that this simple change to the input properties would require a fairly drastic change to INGR in that space must now be left for all the look ahead characters in s-RULES.

9 CODING IN PL/I

This section discusses the PL/I version (see APPENDIX 1) of the algorithm contained in Section 7 (plus 5-8,5-9,5-10) as modified by Section 8. The transition is not shown in full detail since this would require a significant portion of the theory of PL/I to be formalised. This is not only outside the scope of the current paper, it would also be inappropriate since it should be part of a larger, more general, endeavour.

First a number of constructs of PL/I are related to the properties that have been used above; then a number of specific results are established.

PL/I data properties -

(9-1) Objects (as in 2-6) can be replaced by PL/I structures and the named selectors can be mimicked by qualified naming. Notice that this property only holds with selectors which cannot yield the null object (see ref 2).

(9-2) Where the selectors are arbitrary (i.e. their names are not explicitly given) the objects can be replaced by a collection of allocations of a based variable. In this case the selectors are replaced by pointers. (N.B. in the representation of STATE-SETS given, the REFER option has also been used to obtain the possibility to dynamically vary the maximum size of a state set).

(9-3) Lists (as in 2-10, 2-11) can be replaced by PL/I arrays, these present upper bound problems which make the algorithm non-total. Referencing is via subscripting and the current length is usually stored in another variable.

(9-4) As an alternative, lists of characters can be replaced by PL/I variable character strings. In this case selection is via the 'SUBSTR' built-in function, and length can be found via the LENGTH built-in function.

(9-5) Integers are represented by FIXED BINARY numbers of length 15. This again introduces a, difficult to specify, restriction on the domain of the algorithm.

PL/I statement properties -

(9-6) Conditional Expressions can be replaced by PL/I statements of the form -

```
IF   p1      THEN  ...
     :
     :
ELSE IF pn   THEN  ...
```

(9-7) With objects which extend monotonically (e.g. state sets) assignment can be used with no loss of information.

(9-8) The use of where clauses can be replaced by computing the required value and assigning it to a variable which is referred to wherever the name is used (this can be thought of as "assignment as an abbreviation").

(9-9) PL/I DO loops are used to specify the sequencing of operations. In particular nested DO loops iterating through the collection of state sets and through the members of a state set are used to mirror the function of "next" (see 7-33). Notice that the correctness of this relies on the fact that PL/I re-evaluates the while clause of a DO loop at each iteration. Thus a DO BY TO construct would fail!

(9-10) The conditional expression form of and (see 2-2) is translated so that the second operand is not evaluated if the first is false.

Special Results -

(9-11) Lemma $ss' = ss \cup \{<s>\}$ yields same result as a procedure whose argument is a state and which changes the state-set by side effect.

```
DOcv = 1  BY 1 TO ℓ(ss);
   IF  ss(cv) = <s> THEN RETURN ;
END
ss(ℓ(ss)+1) = <s> ;
```

(9-12) Lemma The effect of adding a set defined implicitly can be obtained by an ordered search, calling the procedure of 9-10 each time a candidate for addition is located.

(9-13) Lemma

cond (r,t) ≡ s-lk(r) = ↑ ∨ t = s-lk(r) satisfies 8-3

proof follows from 8-2 and 9-12.

The program invokes procedures called INGR and INSTR to read grammar and string into store. (These routines were in fact taken from the conventionally built program.) A procedure IPTEST is then invoked to check restrictions 4-1,6-9,7-20 and 7-21,8-2. Clearly the relation to the input grammar cannot be checked directly and the expedient of printing the grammar as it "must have looked" is adopted. Finally, the parsing proper is performed.

10 DISCUSSION

This section attempts to discuss some of the open questions. It is the hope that work on Formal Development will be applied to a variety of examples in an attempt to resolve these and other questions.

Breaking the development down into steps has given a clearer picture of the algorithm and its correctness: is the work justified ? Making a comparison to the more widely used "write then attempt proof" there are few portions of the above work which are wasted. (One example is the necessity to set up the same inductive form for proofs at two or more steps.) In view of the difficulty of composing a complete algorithm correctly from the beginning, this cost is not great. The proofs also tend to follow the intuative program ideas more closely. Comparisons to methods not involving formal proofs are more difficult, but the opinion of the current author is that without the discipline of proof such methods will migrate poorly from their original environment.

The comment was made, by an experienced programmer during a presentation of this work, that the much maligned tendency to start coding too early may be a symptom of wanting to write something formal but lacking any alternative language in which to state the partially thought out ideas!

The development given here was aided by the existence and knowledge of an actual working version of the program. In an actual development much more backtracking would be required: it is certainly not claimed that the formal development approach would give rise to inspirations like Earley's state sets. However, the formal approach would certainly offer a framework in which to search - only further work can resolve this point.

One of the principal problems is the level of formality required in the proofs. The developed algorithm was keypunched and run on an Algol grammar. Although no errors have been uncovered this does not establish that the above level of proof would be formal enough to provide security against errors. There is certainly an important distinction to be made between assumptions on the data which are extremely dangerous (being the source of many conventional programming errors), and formality in the deductions which is not attempted. In particular, the extremely attractive, notion of machine checkable proofs may not be practical because of the tedium of formalising the deductive steps. A particular problem of this distinction is the failure in the development to separate variables of the problem and language (e.g. control variables) domains.

There are many general problems which become apparent in the above development: is there a clear set of ways of using storage/assignment (see 9-7,9-8) ? Are there better ways of specifying order than the usual DO loops ? Questions like this could provide valuable feedback to language design from a problem, rather than the usual machine, angle.

ACKNOWLEDGEMENTS

The author gratefully acknowledges that the idea of levels of an algorithm was suggested by P Lucas. The current paper represents a reasonably complex example and the author's own view of how to add the proofs. Earley's paper provided not only the ideas of the algorithm but also the basis for some of the proofs of the first sections. The author is also grateful for useful discussions with C D Allen, D Chapman, and P D Wright for providing the original running program. Although different in important respects the author acknowledges the stimulus of having read the works of Dijkstra, Hoare and Mills relating to development of programs.

REFERENCES

1. J EARLEY. "An Efficient Context-free parsing Algorithm". Comm ACM Vol 13 No 2, Feb 1970

2. P LUCAS and K WALK. "On the formal Description of PL/I" Annual Review in Automatic Programming Vol 6 Part 3, 1969

3. D PARK. "Fixpoint Induction and Proofs of Program Properties". Machine Intelligence 5, 1969

4. Z MANNA and J McCARTY. "Properties of Programs and Partial Function Logic" Stanford AI Memo 100

```
EARLY:  PROC       OPTIONS(MAIN);
DCL NULL BUILTIN;
        DCL   INPUT CHAR(5000) VARYING INIT('') EXTERNAL;
        DCL  1 NONT(500) EXTERNAL PTR,
             1 STR(250) EXTERNAL,
                  2 START FIXED BIN(15),
               STRCHAR CHAR(4000) VARYING INIT('') EXTERNAL,
             1 RULES(4000) EXTERNAL,
                  2 TYPE CHAR(1),
                  2 N FIXED BIN(15),
             1 RULED BASED(Q),
                  2 MAXNRULES FIXED BIN(15),
                  2 NEXT     FIXED BIN (15),
                  2 RULE(NRULES REFER (MAXNRULES)),
                       3 START FIXED BIN(15),
                       3 LENGT  FIXED BIN(15);

        DCL IPGR    ENTRY EXTERNAL;
        DCL IPSTR   ENTRY EXTERNAL;
        DCL IPTEST  ENTRY EXTERNAL;

CALL IPGR   ;
CALL IPSTR  ;
CALL IPTEST ;

 /* N.B. IPSTR HAS SQUEZED ALL BLANKS TO AVOID REQUIREMENT TO CODE
         SAME IN ALGOL GRAMMAR  */

BEGIN;

DCL S(0:LENGTH(INPUT)) PTR;
DCL 1 STATE BASED (SI) ,
       2 SIZE FIXED BIN(15),
       2 N    FIXED BIN(15),
       2 INFO (STATE_SIZE REFER (SIZE)),
          3 RULE,
             4 RPTR PTR,
             4 SSC FIXED BIN(15),
          3 RULPOS FIXED BIN(15),
          3 STRPOS FIXED BIN(15);
DCL STATE_SIZE FIXED BIN(15) INIT(70);

DCL I FIXED BIN(15);       /* STATE SET INDEX ,MAJOR LOOP            */
DCL N FIXED BIN(15);       /* STATE INDEX     ,2ND LOOP              */
DCL P1 PTR;                /* PTR OF S-RULE OF STATE(N,STATE-SET(I)) */
DCL I1 FIXED BIN(15);      /* INT OF S-RULE OF STATE(N,STATE-SET(I)) */
DCL J  FIXED BIN(15);      /* RULPOS        OF STATE(N,STATE-SET(I)) */
DCL F  FIXED BIN(15);      /* STRPOS        OF STATE(N,STATE-SET(I)) */
DCL R_ FIXED BIN(15);      /* LEN-PHS(P1,I1)                         */

DCL 1 RHS_EL,              /* RHS_EL(J+1,(P1,I1))                    */
       2 TYPE CHAR(1),
       2 N FIXED BIN(15);

DCL P2 PTR;                /* PTR PT OF RULES S.T.
                                 RHS_NT_EQ_LHS((P1,I1),J+1,(P2,?))  */
DCL K FIXED BIN(15);       /* PREDICTOR'S RULE INDEX                 */

DCL LK CHAR(1);            /* LK_EL(P2,K)                            */

DCL KS FIXED BIN(15);      /* STATE INDEX WITHIN S(F)                */

DCL P3 PTR;                /* PTR OF S-RULE (STATE(KS,STATE-SET(F))  */
DCL I2 FIXED BIN(15);      /* INT OF S-RULE (STATE(KS,STATE-SET(F))  */
DCL L  FIXED BIN(15);      /* S-RULPOS      (STATE(KS,STATE-SET(F))  */
DCL G  FIXED BIN(15);      /* S-STRPOS      (STATE(KS,STATE-SET(F))  */

DCL 1 SF_RHS_EL,           /* RHS_EL(P3,I2)                          */
       2 TYPE CHAR(1),
       2 N FIXED BIN(15);
```

```
DCL ADD_STATE ENTRY(PTR,FIXED BIN(15),FIXED BIN(15),FIXED BIN(15),
                                                    FIXED BIN(15));

DCL SI PTR;                 /* CONTAINS S(I) : F COMP RESTR          */
DCL SF PTR;                 /* CONTAINS S(F) : F COMP RESTR          */
DCL NONT1PTR PTR;           /* CONTAINS NONT(1).PTR : F COMP RESTR   */

   ADD_STATE:
   PROC (PTR,INT,J,F,I);
         /* ADDS ((PTR,INT),J,F) TO STATE-SET(I) ,UNLESS THERE */

   DCL PTR PTR;
   DCL INT FIXED BIN(15);
   DCL J   FIXED BIN(15);
   DCL F   FIXED BIN(15);
   DCL I   FIXED BIN(15);

   DCL K FIXED BIN(15);   /* STATE INDEX WITHIN STATE_SET(I)        */

   DCL SI PTR;            /* CONTAINS S(I)  : F COMP RESTR */

      SI = S(I);
      DO K = 1B BY 1B TO SI -> STATE.N ;
         IF (SI -> STATE.INFO(K).RULE.RPTR = PTR )
           &(SI -> STATE.INFO(K).RULE.SSC  = INT )
           &(SI -> STATE.INFO(K).RULPOS    = J )
           &(SI -> STATE.INFO(K).STRPOS    = F ) THEN
            RETURN;                          /* STATE ALREADY THERE */
      END;

      IF SI -> STATE.N  = SI -> STATE.SIZE THEN
         DO;                                 /* STATE SET FULL */
            PUT EDIT(' R4 FAILED ')(SKIP,A);
            STOP;
         END;
                                             ELSE
         DO;                                 /* ADD */
         SI -> STATE.N = SI -> STATE.N + 1B;
         SI -> STATE.INFO(SI -> STATE.N).RULE.RPTR = PTR;
         SI -> STATE.INFO(SI -> STATE.N).RULE.SSC  = INT;
         SI -> STATE.INFO(SI -> STATE.N).RULPOS    = J;
         SI -> STATE.INFO(SI -> STATE.N).STRPOS    = F;
         END;
   END;

   DO I = 0B BY 1B WHILE(I <= LENGTH(INPUT));
      ALLOCATE STATE;            /* SETS  SI */
      S(I) = SI;
      SI -> STATE.N = 0B;
   END;

   SI = S(0B);
   SI -> STATE.N = 1B;
   SI -> STATE.INFO(1B).RULE.RPTR = NONT(1B);
   SI -> STATE.INFO(1B).RULE.SSC  = 1B;
   SI -> STATE.INFO(1B).RULPOS    = 0B;
   SI -> STATE.INFO(1B).STRPOS    = 0B;

         /* INITIAL S S.T. : IS-S(S)
                             L(STATE_SET(0))=1
                             STATE(1,STATE_SET(0)) =
                                    <(R)(LAS(R)=ROOT(G)),0,0>
                             1 <= I <= L(INPUT) IMP
                                       L(STATE_SET(I,S))= 0 */
```

```
   DO I = 0B BY 1B WHILE(I<= LENGTH(INPUT));
      SI = S(I);
      DO N = 1B BY 1B WHILE(N<= SI -> STATE.N);

         P1 = SI -> STATE.INFO(N).RULE.RPTR;
         I1 = SI -> STATE.INFO(N).RULE.SSC;
         J  = SI -> STATE.INFO(N).RULPOS;
         F  = SI -> STATE.INFO(N).STRPOS;
         R_ = P1  -> RULED.RULE(I1).LENGT;

         IF(J ¬= R_) &(I ¬= LENGTH(INPUT)) THEN
            DO;
               RHS_EL = RULES(P1 -> RULED.RULE(I1).START + J ) ;
               IF RHS_EL.TYPE = 'N' THEN
                  DO;                                          /* PREDICTOR */
                     P2 = NONT(RHS_EL.N);
                     DO K = 1B BY 1B TO P2 -> RULED.NEXT -1B;
                        LK = SUBSTR(STRCHAR,STR(RULES(P2->RULED.RULE(K).START-1B).N)
                                                                    .START,1);
                        IF(LK=' ')|(LK=SUBSTR(INPUT,I+1B,1))THEN
                           CALL ADD_STATE(P2,K,0B,I,I);
                     END;
                  END;

               ELSE IF(RHS_EL.TYPE = 'T') THEN
                  IF SUBSTR(STRCHAR,STR(RHS_EL.N).START,1)  = SUBSTR(INPUT,I+1,1) THEN
                     CALL ADD_STATE(P1,I1,J+1B,F,I+1B);        /* SCANNER */
            END;

         ELSE IF J = R_ THEN
            DO;                                                /* COMPLETEP */
               SF = S(F);
               DO KS = 1B BY 1B TO SF -> STATE.N;
                  P3 = SF -> STATE.INFO(KS).RULE.RPTR;
                  I2 = SF -> STATE.INFO(KS).RULE.SSC;
                  L  = SF -> STATE.INFO(KS).RULPOS;
                  G  = SF -> STATE.INFO(KS).STRPOS;

                  IF L ¬= P3->RULED.RULE(I2).LENGT THEN
                     DO;
                        SF_RHS_EL = RULES(P3->RULED.RULE(I2).START+L);
                        IF SF_RHS_EL.TYPE = 'N' THEN
                           IF NONT(SF_RHS_EL.N) = P1 THEN
                              CALL ADD_STATE(P3,I2,L+1B,G,I);
                     END;
               END;
            END;
      END;
   END;


        /* CHECK FOR END_STATE IN S(L(X))        */

   SI = S(LENGTH(INPUT));
   NONT1PTR = NONT(1B);

   DO N = 1B BY 1B TO SI -> STATE.N;
      IF (SI -> STATE.INFO(N).RULE.RPTR = NONT(1B))
        &(SI -> STATE.INFO(N).RULE.SSC  = 1B )
        &(SI -> STATE.INFO(N).RULPOS    = NONT1PTR -> RULED.RULE(1B).LENGT)
        &(SI -> STATE.INFO(N).STRPOS    = 0B  ) THEN

         DO;
         PUT EDIT (' YES' ) (SKIP ,A); RETURN ;
         END;
   END;

   PUT EDIT (' NO')(SKIP,A); RETURN ;
END;
END;
```