Algorithmic Languages, de Bakker/van Vliet (eds.)

From VW-grammar to ALEPH

D. Grune

Mathematical Centre, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

This paper gives an exposition of the designing of ALEPH. ALEPH (acronym for A Language Encouraging Program Hierarchy) is a programming language developed at the Mathematical Centre; it is unusual in that it originates from the world of grammars rather than from the world of programming languages. It has the interesting property that it is large enough not to be dismissed as a toy language and small enough to keep the task of designing it intellectually manageable.

An account of the design of ALEPH is interesting not only because of its results, a language with a very simple but powerful flow-of-control in which the uninitialized-variable problem is solved and in which side effects are under full control, but also because the way in which these results are obtained lies open to examination.

1. Introduction

ALEPH (acronym for *A L*anguage *E*ncouraging *P*rogram *H*ierarchy) [6] is a programming language developed at the Mathematical Centre; it is unusual in that it originates from the world of grammars rather than from the world of programming languages. It has the interesting property that it is large enough not to be dismissed as a toy language and small enough to keep the task of designing it intellectually manageable (although barely so).

Therefore an account of the design of ALEPH is interesting not only because of its results, a language with a very simple but powerful flow-of-control in which the uninitialized-variable problem is solved and in which side effects are under full control, but also because of the fact that the way in which these results are obtained lies open to examination.

In this paper we shall give an exposition of the designing of ALEPH. Little is known about design rules for programming languages. In essence design rules serve to reduce the intellectual complexity of a task. Traditional

means are: imposing a structure, divide-and-conquer, defining interfaces, etc. Hardly any of these applies to the design of programming languages. The most successful principle is still orthogonality, which also has its problems. It does not allow the designer to distinguish between the cheap and the expensive, and its consistent application is difficult.

1.1. *Vocabulary*

Our discussion leads us from VW-grammars through affix grammars to ALEPH and conventional programming languages. A VW-grammar (2.1) can be seen as a recipe for generating an (infinite) grammar capable of generating the context-sensitive language we want. An affix-grammar (2.4) can be seen as a parametrized context-free grammar where the context is stored in the parameters (affixes).

Different terminology is (traditionally) used in these different fields, and it may be helpful for the reader to refer to Table 1.

Table 1

VW-grammars	Affix-grammars	ALEPH	Conventional programming languages
grammar	grammar	program	program
–	initial symbol	root	–
hyper-rule	rule	rule	procedure
–	primitive predicate	external rule	built-in function
	left-hand-side, LHS	rule head	procedure heading
	right-hand-side, RHS	rule body	procedure body
may produce empty	may produce ε	always succeeds	always yields **true**
is a blind alley	produces ω	fails	yields **false**
hypernotion	affix expression	affix form, rule call	call
metarule	affix rule	–	data type
metanotion	affix	affix	parameter
	bound affix	formal affix	formal parameter
	free affix	local affix	local parameter

2. Turning a VW-grammar into a Programming Language

2.1. *VW-grammars*

A VW-grammar (named after its originator, A. van Wijngaarden [9, 10]) is a special type of context-sensitive (CS) grammar which has many properties of a context-free (CF) grammar. It is based on the observation that we can use a CF grammar to describe a CS language, provided that this grammar has infinitely many production rules; every actual production of a desired sentence in the CS language, however, needs only a finite number of them. In essence a VW-grammar is a recipe for generating such an infinity of CF production rules. For an extensive explanation see [2].

A VW-grammar has the following main constituents:

- the metarules, a collection of (interrelated) CF grammars, each producing a language for a specific metanotion,
- the hyper-rules, a collection of templates from which to form (an infinity of) CF production rules.

A CF production rule is derived from a hyper-rule by replacing consistently each of the metanotions it contains by a terminal production of that metanotion. For an example see TCGl below.

2.2. *Two-colour grammars*

Let me now introduce the notion of a 'two-colour' VW-grammar. We start from a VW-grammar R, which produces sequences of symbols in red. We then take a second VW-grammar P, which shares part or all of its metarules with R and which produces its symbols in blue (or in a different alphabet if you will). We now combine the two grammars and insert hypernotions of P in hyperalternatives of rules of R: the resulting grammar produces sentences in mixed red and blue text.

If it now so happens that a hypernotion of P shares one or more metanotions with its neighbours that belonged to R, then the production of blue text is controlled by the same choice of metanotion substitutions as that of the red text, and the red and blue pieces of text will become correlated.

As an example we shall now rewrite grammar Q from [2, p. 64] as a two-colour grammar.

TCG1:

```
N   :: N n;  .
ABC :: a; b; c.

text: red N a, blue N b, blue N c.

red N ABC:
    red symbol ABC, red N1 ABC, where rd N1 plus one is N;
    where rd N is zero.
red symbol ABC: red letter ABC symbol.
where rd N plus one is N n: where true.
where rd    is zero: where true.

blue N ABC:
    where bl N is zero;
    blue symbol ABC, where bl N1 is N minus one, blue N1 ABC.
blue symbol ABC: blue letter ABC symbol.
where bl N is N n minus one: where true.
where bl    is zero: where true.

where true:  .
```

A possible production is (with N = nnn in 'text'):

red-a red-a red-a blue-b blue-b blue-b blue-c blue-c blue-c.

2.3. *A top-down parser*

It is well known that a CF grammar can be turned into a recognizer for the language it produces. In the case of an unrestricted CF grammar such a recognizer has to do extensive backtracking, which is painful in terms of space and time, but if enough restrictions are put on the CF grammar, neat recognizers result. Specifically, the LL(1) restriction leads to an efficient top-down parser, which, as a program, has virtually the same form as the original grammar.

This suggests that it may be possible to consider the red part of the two-colour grammar TCG1 (which, in a sense, *is* LL(1)) as a top-down parser for the red text, while at the same time retaining the producing nature of the blue part. If we do this, we are led to consider the occurrences of metanotions in hypernotions as parameters. We shall not worry at the

moment about the exact parameter-passing mechanism; for the time being it can be thought of as 'call-by-name'. This brings us to the following grammar/program:

P1:

text: read N a, print N b, print N c.

read N ABC:
    read symbol ABC, read N1 ABC, where rd N1 plus one is N;
    where rd N is zero.
read symbol ABC: absorb letter ABC.
where rd N1 plus one is N: set N to N1 plus one.
where rd N is zero: set N to zero.

print N ABC:
    where pt N is zero;
    print symbol ABC, where pt N1 is N minus one, print N1 ABC.
print symbol ABC: produce letter ABC.
where pt N1 is N minus one: set N1 to N minus one.
where pt N is zero: is N zero.

When we read this with the firm conviction that it is a program, semantics begins to attach itself to various constructs. To perform 'text', read N a's, then print N b's, then print N c's. To read N ABC's, we have the choice between two alternatives which we shall try in order. We attempt to read a symbol ABC, and if we succeed we read N1 ABC's and set N to N1 plus one; otherwise (if we cannot read a symbol ABC) we set N to zero. In this same vein we can understand the rest of the program, which prints N b's and N c's.

At this point the reader will have gathered that we have cheated. The above example was rigged so that its interpretation as a program suggested itself. A general VW-grammar does not exhibit such a nice structure, and the parsing problem cannot in general be solved. There is, however, a type of CS grammar related to VW-grammars for which the parsing problem *can* be solved: the affix grammars.

2.4. *Affix grammars*

Affix grammars are defined by Koster [7]; this definition is slightly

corrected and explained well in [1]. Koster shows that if an affix grammar is 'well-formed' (see below) it is possible to construct a parser for the language it generates. Most constituents of a VW-grammar also exist in an affix-grammar. For a list of correspondences see Table 1. The principal differences between affix grammars and VW-grammars are:

- a hypernotion consists of a characteristic name, its 'handle', followed by one or more metanotions, called 'affixes', and
- context conditions are enforced by special rules called 'primitive predicates'; they can be thought of as affix checkers.

A 'primitive predicate' is similar to a (normal) rule in that it has affixes; but rather than producing its output by specifying affix forms and terminal symbols, it contains a total recursive function T which, depending on the affixes, will produce either 'empty' (ε) or the forbidden symbol (ω). We shall call T the 'test' of the primitive predicate.

The well-formedness criterion requires (among other things) that all occurrences of affixes be divided into two groups, the 'derived' (δ) and the 'inherited' (ι) affixes, in such a way that they can properly be interpreted as output and input parameters, respectively. Moreover, for each primitive predicate with derived affixes D, inherited affixes I and test T, a total recursive function must be given which will calculate D from I such that $T(I, D)$ succeeds (i.e., produces ε); this requirement marks the transition from a specification language to an algorithmic language.

We shall now show an affix-grammar equivalent to TCG1 (some comment is given between {{ and }}):

AG1:

```
⟨{{V[n]:}} (text, red, red symbol, blue, blue symbol),
 {{V[t]:}} (red-a, red-b, red-c, blue-a, blue-b, blue-c),
 {{A[n]:}} (N, N1, ABC, ABC1),
 {{A[t]:}} (n, a, b, c),
 {{Q:}} (where rd plus one is, where rd is zero, where is,
         where bl is minus one, where bl is zero
        ),
 {{E:}} text,
 {{R:}} (N: N n; .      ,
         N1: N.         ,
         ABC: a; b; c. ,
         ABC1: ABC.
        ),
```

{{S:}} (⟨text, 0, ϕ, ϕ, ϕ⟩,
⟨red, 2, (δ, ι), (N, ABC), ϕ⟩,
⟨red symbol, 1, (ι), (ABC), ϕ⟩,
⟨where rd plus one is, 2, (ι, δ), (N, N1),
$\lambda x\ \lambda y$: ($x+1=y\rightarrow\varepsilon$, $x+1\neq y\rightarrow\omega$)⟩,
⟨where rd is zero, 1, (δ), (N),
λx: ($x=0\rightarrow\varepsilon$, $x\neq 0\rightarrow\omega$)⟩,
⟨where is, 2, (ABC, ABC1), (ι, ι),
$\lambda x\ \lambda y$: ($x=y\rightarrow\varepsilon$, $x\neq y\rightarrow\omega$)⟩,
⟨blue, 2, (ι, ι), (N, ABC), ϕ⟩,
⟨blue symbol, 1, (ι), (ABC), ϕ⟩,
⟨where bl is minus one, 2, (ι, δ), (N, N1),
$\lambda x\ \lambda y$: ($x=y-1\rightarrow\varepsilon$, $x\neq y-1\rightarrow\omega$)⟩,
⟨where bl is zero, 1, (ι), (N),
λx: ($x=0\rightarrow\varepsilon$, $x\neq 0\rightarrow\omega$)⟩
),
{{P:}} (text: red + N + a, blue + N + b, blue + N + c.
red + N + ABC:
red symbol + ABC, red + N1 + ABC,
where rd plus one is + N1 + N;
where rd is zero + *N*.
red symbol + ABC:
where is + ABC + a, red-a;
where is + ABC + b, red-b;
where is + ABC + c, red-c.
blue + N + ABC:
where bl is zero + N;
blue symbol + ABC, where bl is minus one + N1 + N,
blue + N1 + ABC.
blue symbol + ABC:
where is + ABC + a, blue-a;
where is + ABC + b, blue-b;
where is + ABC + c, blue-c.
)
⟩

To satisfy the well-formedness requirement this text must be augmented by a list of functions, one for each primitive predicate, that calculate the

derived affixes from the inherited ones. Since lambda-notation does not allow output-parameters, these functions cannot be written down here. They correspond to the "set N to ···" in P1.

3. From Affix Grammar to ALEPH

Although the affix grammar AG1 can be converted easily into a program, it will be clear that affix grammars are still a far cry from a usable programming language. We have 'primitive predicates' which form a kind of language inside the language. The global flow-of-control may be obvious but details about the local flow-of-control (i.e., inside a rule) have to be decided. The exact nature of affixes is open to negotiation. The affix rules describe data structures, but their form will depend on decisions about the affixes.

There are of course many ways to approach these problems. One such approach has led to the Compiler Description Language CDL, designed by Koster [8], and its successor CDL2 [4]. We shall follow here a different way which leads to ALEPH.

Like in CDL we shall restrict ourselves to top-down (recursive descent) parsers, since they lead more easily to programming languages than bottom-up parsers. Bottom-up parsers for affix grammars have been constructed by Crowe [3] and Böhm [1].

3.1. *Global flow-of-control*

The global flow-of-control relies completely on rules calling rules (recursively); since there is only one level of rules and rules cannot occur as parameters (nor be assigned to 'rule variables'), the program is a directed graph; the starting point is the **root**. This has the great advantage that many properties of the program can be decided mechanically (recursion check, automatic cross-referencing). On the other hand it means that the rule-calling and affix-passing mechanism will be used heavily and that efficiency will be an important factor in the design of both.

3.2. *Finding a place for the primitive predicates*

We shall incorporate the ι/δ affix information in the rule heads; an ι-affix (input affix) is marked by a *pre*fixed $\rangle$, a δ-affix (output affix) by a

*post*fixed ⟩. We shall postpone the decision about the affix-passing mechanism to Section 4.1.

The number of primitive predicates can often be greatly reduced by describing their effect (producing ε or ω) in hyper-rules. Many full-size examples of this technique can be found in [10, Ch. 7] and in [5]. This suggests the possibility of using a fixed set of metarules for every grammar, i.e., to supply a fixed set of data-types in the programming language. These data-types are then supported by a predefined set of predicates on them, the 'externals'.

The RHS of a rule may contain both affix forms and terminal symbols; we shall simplify this situation by introducing two rules, 'absorb + ABC' and 'produce + ABC'. 'Absorb + ABC' looks at the next character in the input stream; if it is equal to ABC, 'absorb' absorbs it and succeeds; otherwise it fails. 'Produce + ABC' produces the character ABC. They replace the absorption and production mechanism implied in the functioning of a two-colour grammar.

Our program now has the form (character constants are quoted with /'s):

```
P2:

root text.

external set to plus one + N⟩ + ⟩N1 = 'INCR',
    set + ⟩N + N1⟩ = 'SET',
    set to minus one + N⟩ + ⟩N1 = 'DECR',
    equal + ⟩N + ⟩N1 = 'EQUAL'.

text: read + N + /a/, print + N + /b/, print + N + /c/.

read + N⟩ + ⟩ABC:
    read symbol + ABC, read + N1 + ABC, where rd plus one is + N1 + N;
    where rd is zero + N.
read symbol + ⟩ABC: absorb + ABC.
where rd plus one is + ⟩N1 + ⟩N: set to plus one + N + N1.
where rd is zero + N⟩: set + 0 + N.

print + ⟩N + ⟩ABC:
    where pt is zero + N;
    print symbol + ABC, where pt is minus one + N1 + N,
      print + N1 + ABC.
```

print symbol +)ABC: produce + ABC.
where pt is minus one + N1) +)N: set to minus one + N + N1.
where pt is zero +)N: equal + N + 0.

end

Note that characteristic strings have been supplied in the **external** declarations, which enable the compiler to find the proper routines outside the program.

3.3. *Local flow-of-control*

Local flow-of-control is the flow-of-control inside a rule once it is called due to global flow-of-control rules. Since global flow-of-control is trivial, we shall use simply 'flow-of-control' for 'local flow-of-control'.

The parsing problem for affix grammars can be solved by a general top-down parser [7, par. 8], at the expense of extensive back-tracking. Now ALEPH is intended for the writing of production soft-ware; here any back-track problems should be solved once at the writing desk, rather than over and over again when the program is run. A traditional way to avoid back-tracking is to require the grammar to be of type LL(1).

What does it mean for an affix grammar to be LL(1)? It should be borne in mind that the LL(1)-property is important only because it allows simple flow-of-control rules for a backtrack-free deterministic parser. We shall therefore take these rules as a starting point:

LL(1) rules:
- call the initial rule; iff it succeeds, the input belongs to the language;
- a rule is 'called' by trying the alternatives in its RHS for applicability and calling an applicable alternative (there can only be one such alternative);
- an alternative is 'applicable' iff its first rule call succeeds;
- an alternative is 'called' by calling its rules in textual order as long as these rule calls succeed;
- an alternative 'succeeds' iff all of its rule calls succeed;
- a rule call 'succeeds' iff the rule called has an applicable alternative that succeeds.

Moreover we have an error condition:
- if any applicable alternative fails, the input does not belong to the generated language (i.e., if an alternative is applicable it is the correct one).

We want to take over these rules as much as possible. After some experimentation we have come to the following flow-of-control rules:

ALEPH rules:
- execute the affix form in the **root**; it must succeed;
- an affix form is 'executed' by trying the alternatives in the RHS of its rule for applicability and executing the first applicable alternative;
- an alternative is 'applicable' iff its first affix form succeeds;
- an alternative is 'executed' by executing its affix forms in textual order as long as these affix forms succeed;
- an alternative 'succeeds' iff all of its affix forms succeed;
- an affix form 'succeeds' iff the rule called has an applicable alternative that succeeds.

These flow-of-control rules allow us to view the first affix form as an 'entrance key': you enter the first alternative to which you have the right key. Once you enter this alternative no others can be reached any more. An important consequence is that there is only one way to reach a given affix form. This leads immediately to the Central Theorem of ALEPH:

Central Theorem. *When the* Nth *affix form in the* Mth *alternative is reached, the entrance keys of alternatives* 1 *through* $M-1$ *have failed, and affix forms* 1 *through* $N-1$ *in this alternative have succeeded.*

This Central Theorem is a great help in deriving assertions (see below).

We still have to investigate the error condition inherited from the LL(1) flow-of-control rules; we shall postpone this until Section 3.5.

The above rules are (almost) all the flow-of-control ALEPH has: there are no **case**-, **while**-, **do**-, **repeat**-, **until**-, or **exit**-clauses. Rather than emphasizing repetition, ALEPH emphasizes decomposition: each problem is decomposed into several alternatives with entrance keys and each alternative is decomposed into a sequence of sub-problems (which may, of course, be congruent to the original problem). In short, every problem is attacked by recursive descent.

Often a problem that requires a complicated application of the traditional **if**'s and **while**'s can be formulated simply in ALEPH. A good example is searching a list for a given name; the search process stops in one of two ways; the list is empty, or we found the name. We want to do different things in both cases. Here we would need a multi-exit loop or a global toggle; or we would have to perform the same test twice. In ALEPH we simply state the alternatives and tell what to do:

```
find name + >name + >list + entry>:
    is empty + list, insert + name + list + entry;
    is name on top + name + list, top of + list + entry;
    next of + list + list1, find name + name + list1 + entry.
```

3.4. *Success/failure*

We have assumed in the above that any rule can fail (but we have not based any conclusions on that). It soon becomes clear, however, that some rules cannot fail, e.g., because a rule produces ε regardless of the values of its affixes.

The Central Theorem shows us immediately that if any alternative but the last one in a rule has an entrance key that cannot fail, part of the RHS is inaccessible.

3.5. *Side effects*

It is the error condition for LL(1)-parsing in Section 3.3 that allows us to avoid back-tracking, in the following way. When a rule call fails, it has only called other rules that failed. Now since the only terminal rule is 'absorb', and since 'absorb' has no side effect when it fails (Section 3.2), no rule call that fails will have had side effects (by induction). So nothing is modified on failure, and no back-track is necessary. This is the 'No cure – no pay' principle: you may order something, but if you don't get it, you don't pay.

We would certainly like to carry this nice feature of LL(1) parsing over into our programming language. This is done trivially by forbidding any applicable alternative to fail (either statically or dynamically). But we can do better than this.

Where a CF grammar only has rules (which have side effects on success), we have rules (which also have side effects on success) *and* primitive predicates (which never have side effects). Moreover, some of our rules derive entirely from primitive predicates (see Section 3.2). So in ALEPH a successful affix form does not necessarily imply side effects.

Consequently it is perfectly safe to allow failure of an applicable alternative, provided no affix form with side effects has yet succeeded in the alternative.

Under this regime the 'No cure – no pay' principle holds:

If an affix form fails, it has had no side effects.

In Section 3.4 we have divided the rules into two groups, those that can fail and those that can't. Now we have a second division, in those that can have side effects (on success) and those that can't. These divisions are independent, so four classes (rule types) result:

	can fail	cannot fail
can have side effects	**predicate**	**action**
cannot have side effects	**question**	**function**

This classification allows us to give a proper place to 'absorb' and 'produce': their rule types are **external predicate** and **external action**, respectively.

In principle the compiler could assess these properties, but it is much more useful to have the programmer specify his intentions (opinions) and have the compiler check them. The non-trivial redundancy obtained is exploited for error detection.

Our program is now (affixes are written in small letters):

P3:

root text.

external function set to plus one + n⟩ + ⟩n1 = 'INCR',
function set + ⟩n + n1⟩ = 'SET',
function set to minus one + n⟩ + ⟩n1 = 'DECR',
question equal + ⟩n + ⟩n1 = 'EQUAL',
predicate absorb + ⟩abc = 'ABS',
action produce + ⟩abc = 'PROD'.

action text: read + n + /a/, print + n + /b/, print + n + /c/.

action read + n⟩ + ⟩abc:
read symbol + abc, read + n1 + abc, where rd plus one is + n1 + n;
where rd is zero + n.
predicate read symbol + ⟩abc: absorb + abc.
function where rd plus one is + ⟩n1 + ⟩n: set to plus one + n + n1.
function where rd is zero + n⟩: set + 0 + n.

```
action print + )n + )abc:
        where pt is zero + n;
        print symbol + abc, where pt is minus one + n1 + n,
          print + n1 + abc.
action print symbol + )abc: produce + abc.
function where pt is minus one + n1) + )n: set to minus one + n + n1.
question where pt is zero + )n: equal + n + 0.

end
```

We see the impact the rule type classification has on the program: for each rule it is locally clear what to expect of it in terms of flow-of-control. The consistency of the indications is checked by the compiler; we have here strong type checking, not for data types but for rule types.

As with strong type checking on data the errors detected originate from inconsistencies on behalf of the programmer. Suppose there is a rule *'xyz'* which has ε as one of its alternatives and which is used for testing the presence of an *'xyz'*. Now, if *'xyz'* is declared as a **predicate**, the empty alternative will cause an error message, and if it is declared as an **action**, its use as a test will be noticed.

4. Affixes

Rules in an affix grammar can have bound affixes (those that occur in the LHS and in the RHS) and free affixes (that occur in the RHS only). In ALEPH these correspond to formal and local affixes, or 'formals' and 'locals'. There are 'input' and 'output' formals; an input formal has a value upon entry to the rule an output formal must have received a value when the rule ends.

Of course it is necessary that all input affixes of an affix form have obtained a value when the affix form is executed. Now, since

- the Central Theorem states that there is only one path from rule entrance to a given affix form, and the Central Theorem gives that path;
- the initial states of all formals and locals at rule entrance are known from the LHS; and
- for each affix form A on the path the effect on the affixes passed to it is known from the LHS of A,

the compiler can ascertain in an efficient way that never the value of an affix will be used before that affix has received a value. No run-time checking is necessary. A similar test can ensure that an output formal will always receive a value.

The details of this test depend on the affix-passing mechanism.

4.1. *The affix-passing mechanism*

The affix-passing mechanism has to obey two conditions: the value of an inherited affix must be available inside the rule, and the value obtained by a derived affix inside the rule must be made available to the caller.

If we do not allow the value of an affix to be changed (once it has obtained a value), then the story ends here: all affix-passing mechanisms that conform to the above conditions are indistinguishable (except, perhaps, as to efficiency).

Little is known, however, about the possibility of programming with initializable constants only, and we felt that variables are indispensable. This decision has led to an interesting extension of the 'No cure – no pay' principle to local variables.

Since rules need the possibility to change values of affixes of calling rules, it seems that we need at least call-by-reference (or a more general mechanism). Call-by-reference, however, can surprise the programmer painfully with invisible aliases, as in:

```
action produce a or b+p⟩+q⟩:
        set+p+/a/, set+q+/b/, produce+p.
```

where a call 'produce a or b+x+x' produces /b/. Moreover, back-track rears its ugly head again when a rule fails after having changed the value of an (output) affix.

On the other hand it is clear that call-by-value is insufficient.

A good in-between is found in 'copy-restore': upon rule entry all input affixes are copied to a local work space, and upon rule exit all output affixes are restored from that local work space. If we now suppress the restoring if the rule fails ('copy-*maybe*-restore'), no effects on affixes will propagate upwards upon failure, and a failing rule will never spoil information: the 'No cure – no pay' principle also holds for affixes.

Under these circumstances we can easily introduce 'inout-affixes', which

must have a value upon entrance and which return the (possibly changed) value; notation: +⟩tag⟩.

The copy-maybe-restore mechanism allows us to view the (formal and local) affixes as local variables, some of which are already initialized upon rule entrance and some of will be returned to the caller if and when the rule succeeds. This mechanism is easy to explain and efficient to implement. It aids programming in that it supplies automatic back-tracking on local variables.

The introduction of variables allows the following shorter form of our program:

P4:

```
root text.

external function increment by one+⟩n⟩ = 'INCR',
    function set+⟩n+n1⟩ = 'SET',
    function decrement by one+⟩n⟩ = 'DECR',
    question equal+⟩n+⟩n1 = 'EQUAL',
    predicate absorb+⟩abc = 'ABS',
    action produce+⟩abc = 'PROD'.

action text−n:      $ a 'local'
    read+n+/a/, print+n+/b/, print+n+/c/.

action read+n⟩+⟩abc:
    read symbol+abc, read+n+abc, where rd plus one+n;
    where rd is zero+n.
predicate read symbol+⟩abc: absorb+abc.
function where rd plus one+⟩n⟩: increment by one+n.
function where rd is zero+n⟩: set+0+n.

action print+⟩n+⟩abc:
    where pt is zero+n;
    print symbol+abc, where pt minus one+n, print+n+abc.
action print symbol+⟩abc: produce+abc.
function where pt minus one+⟩n⟩: decrement by one+n.
question where pt is zero+⟩n: equal+n+0.

end
```

5. Other Features

Program P4 is correct ALEPH and, given suitable external routines INCR ··· PROD, it will run. However, a number of externals have been predefined in ALEPH; there are other data types besides the integers used here; there are abbreviations for right-recursive rule calls; and there are other features. All these allow the program to be simplified. For lack of space we shall not treat them here. Details can be found in the ALEPH Manual [6].

6. Conclusion

We have shown that by drawing heavily on the analogy between grammars and programs, and between parsing and problem solving, a practical language can be designed that has some properties not generally found in programming languages.

Among these properties are:

- a simple and effective flow-of-control based solely on selection, decomposition and procedure calling;
- a Central Theorem which states in simple terms the conditions that apply when a given construct is reached;
- an efficient compile-time check on the initialization of variables;
- a firm and compiler-checkable concept of side effects.

References

[1] A.P.W. Böhm, Affixgrammatica's, afstudeerverslag (Affix Grammars, MSc. Thesis), TH Delft (1974) in Dutch.

[2] J.C. Cleaveland and R.C. Uzgalis, Grammars for Programming Languages (Elsevier, Amsterdam, 1977).

[3] D. Crowe, Generating parsers for affix grammars, Comm. ACM 15 (1972) 728–734.

[4] J.P. Dehottay, H. Feuerhahn, C.H.A. Koster and H.M. Stahl, Syntaktische Beschreibung von CDL2, Forschungsbericht Technische Universität Berlin (1976).

[5] R. Glandorf, D. Grune and J. Verhagen, A W-grammar of ALEPH, IW 100/78, Mathematical Centre, Amsterdam (1978).

[6] D. Grune, R. Bosch and L.G.L.T. Meertens, ALEPH Manual, IW 17/75, Mathematical Centre, Amsterdam (1975) (third printing).

[7] C.H.A. Koster, Affix grammars, in: J.E.L. Peck (Ed.), ALGOL 68 Implementation (North-Holland, Amsterdam, 1971) p. 95.
[8] C.H.A. Koster, A compiler compiler, MR 127/71, Mathematical Centre, Amsterdam (1971).
[9] A. van Wijngaarden, Orthogonal design and description of a formal language, MR 76, Mathematical Centre, Amsterdam (1965).
[10] A. van Wijngaarden et al. (Eds.), Revised report on the algorithmic language ALGOL 68, Acta Inform. 5 (1975) 1–236.