A Descriptive Language for Symbol Manipulation*

ROBERT W. FLOYD

Armour Research Foundation of Illinois Institute of Technology, Chicago, Illinois

The algebraic command languages (ALGOL, IT, FORTRAN, UNICODE), although useful in preparing numerical algorithms, have not in the author's opinion proven themselves useful for symbol manipulation algorithms, particularly compilers. List processors, in fact, have been designed primarily to fill this gap. Analogously, the traditional flowchart serves well as a descriptive language for numerical algorithms, but does not lend itself to description of symbol manipulation algorithms in such a way that the intent of the process is clear. It will be the purpose of this paper to present a more suitable notation for description of compilers and other complicated symbol manipulation algorithms.

The algorithms used in formula translation consist principally of the following elements:

(1) A set of linguistic transformations upon the input string, together with conditions determining the applicability of each transformation.

(2) A set of actions, such as the generation of machine language coding, associated with each transformation.

(3) A rule for transfering the attention of the translator from one portion of the input string to another.

The notation presented here greatly simplifies the representation of the first and third elements.

For illustrative purposes, a compilation process for a small subset of ALGOL is described below. The subset consists of assignment statements constructed from identifiers, the five binary arithmetic operators ($\uparrow$, $\times$, $/$, $+$, $-$), the two unary arithmetic operators ($+$, $-$), the replacement operator ($:=$), parentheses, and the library functions of one variable (sin, exp, sqrt, etc.). The assignment statement Σ to be translated is initially taken in the augmented form $\vdash \Delta\Sigma \dashv$, where the characters $\vdash$ and $\dashv$ serve as termination symbols and Δ is a pointer which indicates the portion of the statement where the translator's attention is currently focused.

The following productions and the associated generation rules respectively decompose the original statement in accordance with its structure, and simultaneously create coding to implement the statement. Coding will be represented by ALGOL statements with at most one operator, to avoid reference to particular computers.

* Received February, 1961.

Productions			Generation Rules
(1) $:= \Delta$	$\Rightarrow$	$\leftarrow \Delta$	
(2) $I\,\alpha\,\Delta$	$\Rightarrow$	$I\,\Delta$	ident $\leftarrow$ concatenation (ident, α)
(3) $\lambda\,\Delta$	$\Rightarrow$	$I\,\Delta$	ident $\leftarrow \lambda$
(4) $I\,\sigma\,\Delta$	$\Rightarrow$	$V_i\,\sigma\,\Delta$	$V[i] \leftarrow$ ident, where the array V serves as a symbol table
(5) $I\,\Delta\,\sigma$	$\Rightarrow$	$I\,\sigma\,\Delta$	
(6) $\phi\,\Delta\,\sigma$	$\Rightarrow$	$\phi\,\sigma\,\Delta$	
(7) $(\,\Delta\,\sigma$	$\Rightarrow$	$(\,\sigma\,\Delta$	
(8) $\phi \uparrow \psi\,\sigma\,\Delta$	$\Rightarrow$	$T_j\,\sigma\,\Delta$	code $T_j := \phi \uparrow \psi$; $j \leftarrow j+1$
(9) $\uparrow\,\Delta\,\sigma$	$\Rightarrow$	$\uparrow\,\sigma\,\Delta$	
(10) $\phi \times \psi\,\sigma\,\Delta$	$\Rightarrow$	$T_j\,\sigma\,\Delta$	code $T_j := \phi \times \psi$; $j \leftarrow j+1$
(11) $\phi / \psi\,\sigma\,\Delta$	$\Rightarrow$	$T_j\,\sigma\,\Delta$	code $T_j = \phi / \psi$; $j \leftarrow j+1$
(12) $\mu\,\Delta\,\sigma$	$\Rightarrow$	$\mu\,\sigma\,\Delta$	
(13) $\phi + \psi\,\sigma\,\Delta$	$\Rightarrow$	$T_j\,\sigma\,\Delta$	code $T_j := \phi + \psi$; $j \leftarrow j+1$
(14) $\phi - \psi\,\sigma\,\Delta$	$\Rightarrow$	$T_j\,\sigma\,\Delta$	code $T_j := \phi - \psi$; $j \leftarrow j+1$
(15) $+\,\psi\,\sigma\,\Delta$	$\Rightarrow$	$\psi\,\sigma\,\Delta$	
(16) $-\,\psi\,\sigma\,\Delta$	$\Rightarrow$	$T_j\,\sigma\,\Delta$	code $T_j = -\psi$, $j \leftarrow j+1$
(17) $\pi\,\Delta\,\sigma$	$\Rightarrow$	$\pi\,\sigma\,\Delta$	
(18) $\gamma\,(\,\phi\,)\,\Delta$	$\Rightarrow$	$T_j\,\Delta$	code $T_j := \gamma(\phi)$, $j \leftarrow j+1$
(19) $(\,\phi\,)\,\Delta$	$\Rightarrow$	$\phi\,\Delta$	
(20) $\leftarrow V_i \leftarrow \psi \dashv \Delta$	$\Rightarrow$	$\leftarrow \psi \dashv \Delta$	code $V_i := \psi$
(21) $\vdash V_i \leftarrow \psi \dashv \Delta$	$\Rightarrow$	⟨empty⟩	code $V_i := \psi$; terminate translation
(22) $\Delta\,\sigma$	$\Rightarrow$	$\sigma\,\Delta$	

In the above productions "I" is a symbol introduced by the translator which serves as a placeholder for the first several characters of an identifier; the characters themselves are stored in the word whose name is 'ident'. The arrow '$\leftarrow$' is a placeholder for the character pair ':='. Most of the "productions" are actually production schemes, in which lower case Greek letters serve as metalinguistic variables. The character α may stand for any letter or digit, λ for any letter, μ for either $\times$ or $/$, π for either $+$ or $-$, and σ for any character whatsoever. The compiler is assumed to have the ability to generate a class of names (V_1, V_2, etc.) for identifiers stored in the symbol table, and (T_1, T_2, etc.) for temporary storage locations for partial results; the metalinguistic variable V_i ranges over the former class, T_j over the latter. The variables ϕ and ψ may take on values from either class. The variable γ stands for those identifier names V_i, continually present in the symbol table, which designate library function names (sin, exp, sqrt, etc.).

The production rules may be interpreted in the following way. Every character to the right of the pointer Δ is assumed to be stored on an input medium; the character immediately to the right of Δ, if any, is that which will be brought into the machine on the next read instruction. The terminator may be interpreted as an end-of-file marker for the input device, if the marker itself may be sensed by the central computer. Every character to the left of Δ is assumed to be stored in a push-down list; the character immediately to the left of Δ is that at the head of the list. Then the second production, $I\alpha\Delta \Rightarrow I\Delta$, indicates that if the character 'I' followed by any alphanumeric occurs at the head of the push-down list, the alphanumeric character must be adjoined to those characters already stored in

the word ident, and deleted from the head of the push-down list. A production like $\uparrow \Delta\sigma \Rightarrow \uparrow \sigma\Delta$ signifies that if the character on the head of the list is '$\uparrow$' the next character, σ, on the input medium must be read and stored on the head of the push-down list. Only characters to the left of Δ and a fixed distance from it are eligible for processing; that is, all activity occurs at the head of the push-down list. Productions 6-14, 17, 19, and 22 are roughly equivalent to the scan described by Samelson and Bauer [2].

In general, more than one production might be applicable to a given string; reading from the top, the first applicable production must be applied in each case. This is much simpler than designing mutually exclusive productions. Reference to the example below will clarify the operation of the productions on a typical statement.

Speed of translation might optionally be increased by providing each production with a successor. It has been assumed above that after successful application of a production, attention returns to the first production, then the second, etc., until another is found applicable. The successful application of the mth production, however, may imply that no production before the nth can next be applicable, so that testing by the compiler may be reduced by specifying an immediate transfer to the nth production.

Example

To translate the formula $x := y \uparrow z \times (u + v)$, it is rewritten in the augmented form $\vdash \Delta\, x := y \uparrow z \times (u + v) \dashv$. The table below gives the successive transforms of this string, the numbers of the productions performing the transformation, and the associated actions.

Application of Production	Yields	With Associated Action
22	$\vdash x\, \Delta := y \uparrow z \times (u + v) \dashv$	
3	$\vdash I\, \Delta := y \uparrow z \times (u + v) \dashv$	ident $\leftarrow$ 'x'
5	$\vdash I : \Delta = y \uparrow z \times (u + v) \dashv$	
4	$\vdash V_1\, . \Delta = y \uparrow z \times (u + v) \dashv$	$V[1] \leftarrow$ 'x'
22	$\vdash V_1 := \Delta\, y \uparrow z \times (u + v) \dashv$	
1	$\vdash V_1 \leftarrow \Delta\, y \uparrow z \times (u + v) \dashv$	
22	$\vdash V_1 \leftarrow y\, \Delta \uparrow z \times (u + v) \dashv$	
3, 5, 4	$\vdash V_1 \leftarrow V_2 \uparrow \Delta\, z \times (u + v) \dashv$	ident $\leftarrow$ 'y', $V[2] \leftarrow$ 'y'
9	$\vdash V_1 \leftarrow V_2 \uparrow z\, \Delta \times (u + v) \dashv$	
3, 5, 4	$\vdash V_1 \leftarrow V_2 \uparrow V_3 \times \Delta\, (u + v) \dashv$	ident $\leftarrow$ 'z', $V[3] \leftarrow$ 'z'
8	$\vdash V_1 \leftarrow T_1 \times \Delta\, (u + v) \dashv$	code $T_1 := y \uparrow z$; $j \leftarrow 2$
12	$\vdash V_1 \leftarrow T_1 \times (\Delta\, u + v) \dashv$	
7	$\vdash V_1 \leftarrow T_1 \times (u\, \Delta + v) \dashv$	
3, 5, 4	$\vdash V_1 \leftarrow T_1 \times (V_4 + \Delta\, v) \dashv$	ident $\leftarrow$ 'u', $V[4] \leftarrow$ 'u'
17	$\vdash V_1 \leftarrow T_1 \times (V_4 + v\, \Delta) \dashv$	
3, 5, 4	$\vdash V_1 \leftarrow T_1 \times (V_4 + V_5)\, \Delta \dashv$	ident $\leftarrow$ 'v', $V[5] \leftarrow$ 'v'
13	$\vdash V_1 \leftarrow T_1 \times (T_2)\, \Delta \dashv$	code $T_2 := u + v$; $j \leftarrow 3$
19	$\vdash V_1 \leftarrow T_1 \times T_2\, \Delta \dashv$	
6	$\vdash V_1 \leftarrow T_1 \times T_2 \dashv \Delta$	
10	$\vdash V_1 \leftarrow T_3 \dashv \Delta$	code $T_3 := T_1 \times T_2$; $j \leftarrow 4$
21	⟨empty⟩	code $x := T_3$

The unsuitability of even an extended algebraic language for describing the operation of a compiler may be seen by translating the first three productions into an extended ALGOL, in which variables may take on characters as values. Assume that the push-down list is the array S, where i is the current list index. Then the first three productions are translated to the rather clumsy form:

```
    if S[i − 1] = ':' ∧ S[i] = '=' then begin i := i − 1;  S[i] := '←' end
else if S[i − 1] = 'I' ∧ alphanumeric (S[i]) then
        begin ident := concatenation (ident, S[i]);  i := i − 1 end
else if letter (S[i]) then begin ident := S[i];  S[i] := 'I' end, etc
```

The examples in the text and appendices are adapted from an ALGOL translator which the author is programming for the UNIVAC 1105 in the notation described. It appears that a translator can be described compactly and precisely in this notation, and that it is not difficult to translate the productions into symbolic assembly language.

APPENDIX A

An algorithm for a bi-directional scan to produce efficient coding was described by the author [1] in flowchart form. It is rewritten below, in productions which specify their successors in the event of successful application. Two attention markers, Δ and ∇, are used for the sake of clarity to differentiate between the two modes of scanning. The scan encodes all assignment statements which do not contain **if** clauses or functions of two or more variables. For details of the design of the scan, see [1]. Metalinguistic variables have the same meaning as in the first set of productions. The additional variables δ, C_j, and Q_l stand respectively for a digit, the name of a constant, and the name of a computed address. The variables ϕ and ψ may take on as values V_i, C_j, T_k, and Q_l. M is a placeholder for a string of digits, N for a decimal number, and F for ${}_{10}$ followed by an optional plus or minus sign and any number of digits.

	Production			Successor	Generation Rule
(1)	$:=\Delta$	$\Rightarrow$	$\leftarrow\Delta$	(22)	
(2)	$I\alpha\Delta$	$\Rightarrow$	$I\Delta$	(22)	ident ← concatenation (ident, α)
(3)	$\lambda\Delta$	$\Rightarrow$	$I\Delta$	(22)	ident ← λ
(4)	$I\sigma\Delta$	$\Rightarrow$	$V_i\sigma\Delta$	(20)	$V[i]$ ← ident, where array V is a symbol table
(5)	$M\delta\Delta$	$\Rightarrow$	$M\Delta$	(22)	const ← 10 × const + value (δ), where δ is a digit
(6)	$N\delta\Delta$	$\Rightarrow$	$N\Delta$	(22)	const ← 10 × const + value (δ); dctr ← dctr − 1
(7)	$M.\Delta$	$\Rightarrow$	$N\Delta$	(22)	dctr ← 0
(8)	${}_{10}+\Delta$	$\Rightarrow$	$F\Delta$	(22)	exp ← 0; sign ← 1
(9)	${}_{10}-\Delta$	$\Rightarrow$	$F\Delta$	(22)	exp ← 0; sign ← −1
(10)	${}_{10}\delta\Delta$	$\Rightarrow$	$F\Delta$	(22)	exp ← value (δ); sign ← 1
(11)	$F\delta\Delta$	$\Rightarrow$	$F\Delta$	(22)	exp ← 10 × exp + value (δ)
(12)	${}_{10}\Delta\sigma$	$\Rightarrow$	${}_{10}\sigma\Delta$	(8)	

	Production			Successor	Generation Rule
(13)	$M\sigma\Delta$	$\Rightarrow$	$C_j\sigma\Delta$	(20)	$C[y] \leftarrow$ const
(14)	$N\sigma\Delta$	$\Rightarrow$	$C_j\sigma\Delta$	(20)	$C[y] \leftarrow$ const $\times$ 10 $\uparrow$ dctr
(15)	$MF\sigma\Delta$	$\Rightarrow$	$N\sigma\Delta$	(14)	dctr $\leftarrow$ sign $\times$ exp
(16)	$NF\sigma\Delta$	$\Rightarrow$	$N\sigma\Delta$	(14)	dctr $\leftarrow$ dctr + sign $\times$ exp
(17)	$F\sigma\Delta$	$\Rightarrow$	$MF\sigma\Delta$	(15)	const $\leftarrow$ 1
(18)	$\delta\Delta$	$\Rightarrow$	$M\delta\Delta$	(5)	const $\leftarrow$ 0
(19)	Δ	$\Rightarrow$	$M.\Delta$	(7)	const $\leftarrow$ 0
(20)	$;\Delta$	$\Rightarrow$	∇ ;	(23)	
(21)	$]\Delta$	$\Rightarrow$	$\nabla]$	(23)	
(22)	$\Delta\sigma$	$\Rightarrow$	$\sigma\Delta$	(1)	
(23)	$\nabla\gamma(\phi)$	$\Rightarrow$	∇T_k	(28)	code 'fnct', ϕ , γ
(24)	$\nabla\sigma(\phi)$	$\Rightarrow$	$\nabla\sigma\phi$	(31)	
(25)	$\nabla V_i[\phi]$	$\Rightarrow$	$Q_i\Delta$	(22)	code ',' , ϕ, 0; code '[', V_i , 0
(26)	$\sigma\nabla)$	$\Rightarrow$	$\nabla\sigma)$	(26)	
(27)	$\sigma\nabla]$	$\Rightarrow$	$\nabla\sigma]$	(26)	
(28)	$\sigma\nabla\phi$	$\Rightarrow$	$\nabla\sigma\phi$	(31)	
(29)	$\nabla\sigma + \phi$	$\Rightarrow$	$\nabla\sigma\phi$	(31)	
(30)	$\nabla\sigma - \phi$	$\Rightarrow$	$\nabla\sigma T_k$	(31)	code 'minus', ϕ, 0
(31)	$\sigma\nabla \uparrow$	$\Rightarrow$	$\nabla\sigma \uparrow$	(26)	
(32)	$\nabla\sigma\phi \uparrow \psi$	$\Rightarrow$	$\nabla\sigma T_k$	(32)	code '$\uparrow$', ϕ, ψ
(33)	$\sigma\nabla\mu$	$\Rightarrow$	$\nabla\sigma\mu$	(26)	
(34)	$\nabla\sigma\phi\mu\psi$	$\Rightarrow$	$\nabla\sigma T_k$	(34)	code μ, ϕ, ψ
(35)	$\sigma\nabla\pi$	$\Rightarrow$	$\nabla\sigma\pi$	(26)	
(36)	$\nabla\sigma\phi\pi\psi$	$\Rightarrow$	$\nabla\sigma T_k$	(36)	code π, ϕ, ψ
(37)	∇ , ϕ]	$\Rightarrow$	$\nabla]$	(27)	code ',', ϕ, 0
(38)	$\nabla \leftarrow \psi \leftarrow \phi$	$\Rightarrow$	$\nabla \leftarrow \phi$	(40)	code '$\rightarrow$', ϕ, ψ
(39)	$\nabla\sigma\psi \leftarrow \phi$	$\Rightarrow$	$\sigma\Delta$	(STOP)	code '$\rightarrow$', ϕ, ψ
(40)	$\sigma\nabla$	$\Rightarrow$	$\nabla\sigma$	(23)	

APPENDIX B

An ALGOL compiler requires not only a formula translator but also a method for generating transfers of control and their associated coding. The following productions represent one possible approach to the translation of the control words of ALGOL to symbolic assembly language.

Metalinguistic variable	May stand for
β	any boolean expression
τ	';' or **'end'**
ω	';', **'end'**, or **'else'**
ϕ	any **for** clause
λ	any label
σ	any character
J	'*C*' or '*U*'
Σ	any basic statement

Placeholder	May replace
C_n	an **if** clause (conditional jump)
U_n	⟨**if** clause⟩⟨statement⟩ **else** (unconditional jump)
F_n	a **for** clause (subroutine call)
L_n	the nth label generated by the translator

	Production	Generation Rules
(1)	if β **then** $\Delta \Longrightarrow C_n\,\Delta$	Code a conditional jump to L_n if β is false $n \leftarrow n + 1$.
(2)	$J_i\tau\Delta \Longrightarrow \tau\Delta$	Assign label L_i to the next line of coding generated.
(3)	**begin**; $\Delta \Longrightarrow$ **begin** Δ	
(4)	**begin end** $\Delta \Longrightarrow \Delta$	
(5)	$\Sigma\omega\Delta \Longrightarrow \omega\Delta$	Generate coding for Σ.
(6)	C_i **else** $\Delta \Longrightarrow U_n\Delta$	Generate coding for unconditional jump to L_n ; $n \leftarrow n + 1$, assign label L_i to next line of coding
(7)	$\phi\Delta \Longrightarrow F_n\Delta$	Generate coding governing a loop, with exit instruction assumed labeled L_n ; $n \leftarrow n + 1$.
(8)	$F_i\omega\Delta \Longrightarrow \omega\Delta$	Generate unconditional jump (labeled L_i) to an illegal address
(9)	$\lambda{:}\Delta \Longrightarrow \Delta$	Assign label λ to the next line of coding
(10)	$\vdash \Delta \dashv \Longrightarrow$ ⟨empty⟩	Terminate compilation.
(11)	$\Delta\sigma \Longrightarrow \sigma\Delta$	

For example the (augmented) statement

$$\vdash \Delta\ \textbf{begin}\ \Sigma_1\ ;\ \textbf{if}\ \beta_1\ \textbf{then begin}\ \Sigma_2\ ;\ \ \Sigma_3\ \textbf{end else}\ \Sigma_4\ \textbf{end} \dashv$$

is translated through the following steps:

Application of production	Yields	With generated coding
11, 11, 11	⊢ **begin** Σ_1 ; Δ **if** etc.	
5	⊢ **begin**; Δ **if** etc	code for Σ_1
3	⊢ **begin** Δ **if** etc.	
11, 11, 11	⊢ **begin if** β_1 **then** Δ **begin** etc.	
1	⊢ **begin** $C_1\Delta$ **begin** etc.	jump to L_1 if β_1 is false
11, 11, 11	⊢ **begin** C_1 **begin** Σ_2 ; $\Delta\ \Sigma_3$ etc.	
5	⊢ **begin** C_1 **begin**; $\Delta\Sigma_2$ etc.	code for Σ_2
3	⊢ **begin** C_1 **begin** $\Delta\Sigma_3$ etc.	
11, 11	⊢ **begin** C_1 **begin** Σ_3 **end** Δ **else** etc	
5	⊢ **begin** C_1 **begin end** Δ **else** etc.	code for Σ_3
4	⊢ **begin** $C_1\ \Delta$ **else** Σ_4 **end** ⊣	
11	⊢ **begin** C_1 **else** $\Delta\ \Sigma_4$ **end** ⊣	
6	⊢ **begin** $U_2\Delta\Sigma_4$**end** ⊣	jump to L_2 ; assign label L_1
11, 11	⊢ **begin** $U_2\Sigma_4$ **end** Δ ⊣	
5	⊢ **begin** U_2 **end** Δ ⊣	code for Σ_4
2	⊢ **begin end** Δ ⊣	assign label L_2
4	⊢ Δ ⊣	
10	⟨empty⟩	

REFERENCES

1 FLOYD, R. W. An algorithm for coding efficient arithmetic operations *Comm. ACM 4* (Jan. 1961), 42–51.

2 SAMELSON, K.; AND BAUER, F. L. Sequential formula translation. *Comm. ACM 3* (Feb 1960), 76–83.