KYBERNETIKA — VOLUME 9 (1973), NUMBER 4

SMAL – The Symbol Manipulation Language

JÁN CHOVANEC, ADA CHUDÁ, EDUARD KOSTOLANSKÝ, DUŠAN ONDRUŠ

This paper deals with the problem-oriented programming language SMAL (Symbol Manipulation Language) and its implementation on the GIER computer.

0. INTRODUCTION

The SMAL language is a modification of the language defined in [1]. Experience gained in implementing and programming by SNOBOL-1 influenced the definition of the syntax and semantics of the SMAL language. The essential difference lies mainly in the choice of basic symbols, especially delimiters. In choosing them we followed the idea that the program written in SMAL-language should be as near as possible to the natural language and that a SMAL-written program might appear as a string.

The language SMAL is destined to describe algorithmic processes connected with the solution of problems of mechanical translation, mathematical linguistics, the simulation of the process of human thinking, etc.

1. THE STRUCTURE OF THE LANGUAGE

The basic notion applied to the description of algorithmic processes referred to in the introduction, is the *string expression*. Its components are: *string*, *string variables* and *delimiters*.

In order to be able to express the algorithmic process, *jump statements* and the feasibility of setting up *logical values* are added by means of which the repetition of a certain part of the process or its branching can be carried out.

The individual stages of the algorithmic process are expressed by means of *program statements*. The implementation of the algorithmic process sometimes requires reference by one program statement to another. Thus program statements may be fitted out by *labels.*

The *program* is the sequence of program statements. The effect of the program, i.e. the process passing off while carrying the program out, may be derived from the program by means of the syntactical analysis of the program and by application of the corresponding semantics of syntactical units. Subsequently we shall define the syntax and semantics of the language SMAL.

Note. Although the basis of the SMAL language is made up of operations applied in the language described in [1], the designation SMAL was chosen for this modification for the following reasons:

According to the information available [2, 3] the individual versions of the SNOBOL language (hitherto four) do not use the delimiters chosen by us. Since we have no closer knowledge of either of the four versions of the SNOBOL language we find it reasonable not to denominate our language as one of the modification of SNOBOL to avoid the presumption that it actually represents one of the versions.

2. SYNTAX AND SEMANTICS OF THE SMAL LANGUAGE

To define the syntax of the SMAL language we have made use of the reverse Backus-Naur form RBNF [4]. This was done due to the fact that the brackets ⟨and⟩ better play up the fact that basic symbols introduced in these brackets may be looked upon as individual symbols.

Let us concisely note the differences between BNF and RBNF.

BNF	RBNF
⟨metalinguistic variable⟩	metalinguistic variable
basic symbol	⟨basic symbol⟩
	• concatenation

2.1. Basic symbols

The SMAL language is made up of the following basic symbols:

basic symbol ::= alphabet symbol | logical value | delimiter | internal variable

alphabet symbol ::= letter | digit | Algol-60 delimiter | blind

letter ::= ⟨*a*⟩|⟨*b*⟩|⟨*c*⟩|⟨*d*⟩|⟨*e*⟩|⟨*f*⟩|⟨*g*⟩|⟨*h*⟩|⟨*i*⟩|⟨*j*⟩|⟨*k*⟩|⟨*l*⟩|⟨*m*⟩|⟨*n*⟩|⟨*o*⟩|⟨*p*⟩|⟨*q*⟩
⟨*r*⟩|⟨*s*⟩|⟨*t*⟩|⟨*u*⟩|⟨*v*⟩|⟨*w*⟩|⟨*x*⟩|⟨*y*⟩|⟨*z*⟩
⟨*A*⟩|⟨*B*⟩|⟨*C*⟩|⟨*D*⟩|⟨*E*⟩|⟨*F*⟩|⟨*G*⟩|⟨*H*⟩|⟨*I*⟩|⟨*J*⟩|⟨*K*⟩|⟨*L*⟩|⟨*M*⟩|⟨*N*⟩|⟨*O*⟩|
⟨*P*⟩|⟨*Q*⟩|⟨*R*⟩|⟨*S*⟩|⟨*T*⟩|⟨*U*⟩|⟨*V*⟩|⟨*W*⟩|⟨*X*⟩|⟨*Y*⟩|⟨*Z*⟩

digit ::= ⟨*0*⟩|⟨*1*⟩|⟨*2*⟩|⟨*3*⟩|⟨*4*⟩|⟨*5*⟩|⟨*6*⟩|⟨*7*⟩|⟨*8*⟩|⟨*9*⟩

The elements of the alphabet include ALGOL-60 delimiters [5] so that the ALGOL program may appear as a string.

The proper strings are formed from the elements of the alphabet. The set of elements of the alphabet may be suitably narrowed or broadened by new recognizable elements.

Identifiers of variables and labels are formed of characters and digits.

blind ::= ⟨∘⟩

The symbol blind has no functional significance and serves only as a typohraphical adjustment.

logical value ::= ⟨*TRUE*⟩|⟨*FALSE*⟩

Logical values have an obvious engrained significance.

delimiter ::= operator | separator | bracket

operator ::= ⟨*AND*⟩|⟨*PUT*⟩|⟨*REPLACE*⟩|⟨*SEARCH*⟩|⟨*FOR*⟩|⟨*DO*⟩
sequential operator

sequential operator ::= ⟨*GO*∘*TO*⟩|⟨*GO*∘*BY*⟩|⟨*ELSE*∘*TO*⟩|⟨*ELSE*∘*BY*⟩|⟨*IF*⟩|

separator ::= ⟨*INTO*⟩|⟨*LENGTH*⟩|⟨*IN*⟩|⟨*FROM*⟩|⟨*TO*⟩|⟨*OF*⟩|⟨*TOO*⟩|
basic separator

basic separator ::= new line | space
new line ::= ⟨
⟩
space ::= space . space |⟨ ⟩|⟨∘⟩
bracket ::= ⟨*START*⟩|⟨*FINISH*⟩|⟨⟦⟩|⟨⟧⟩

Delimiters are used in designing string expressions, assignment statements and jump statements. Their closer meaning will be discussed in the relevant section.

Defined symbols for space are interchangeable.

Brackets *START* and *FINISH* define the beginning and termination of the program text and the function of brackets ⟦and⟧ will be described in section 2.2.

To make it possible for arbitrary text to be put into the program text in terms of comment and to divide a statement into several rows (2.4) the separator *TOO* is introduced.

The equivalent of the sequence of basic symbols

⟨*TOO*⟩ . space . arbitrary sequence of basic symbols not involving new line . new line, is space

internal variable ::= ⟨*EFFECT*⟩

The internal variable may acquire arbitrary value. The mode of acquiring the logical value will be described in section 2.4.1.

proper string ::= proper string . alphabet symbol | empty

open string ::= open string . open string |⟨⟦⟩ . open string . ⟨⟧⟩ | proper string

string ::= ⟨⟦⟩ . open string . ⟨⟧⟩

To enable the language to handle an arbitrary sequence of basic symbols, string brackets ⟦ and ⟧ are introduced and the value of the string is the open string between these brackets. String length is the number of elements of the alphabet, string brackets and basic separators in the open string.

If S is a string, its length will be denoted as $l(S)$.

The value is a string or a logical value. The values of the string expression and its components will be defined in section 2.3.

Examples. Strings

⟦*Ezo Vlkolinsky*⟧
⟦**begin real** *a*, *b*; *a* := *2 . 3*; *b* := *a* ↑ *3* **end**⟧
⟦*START INTO a PUT*⟦*ab*⟧
FINISH⟧

have values

Ezo Vlkolinsky
begin real *a*, *b*; *a* := *2 . 3*; *b* := *a* ↑ *3* **end**
START INTO a PUT ⟦*ab*⟧
FINISH

and their lengths are

14, 24, 28 .

2.3. Variables. String variables. String expressions

identifier ::= identifier . identifier | letter | digit

Identifiers have no (original) meaning of their own. They serve to denote variables, labels and functions. Identifiers must differ from delimiters.

variable ::= internal variable | identifier

operand ::= string | variable

length designation ::= operand

constant length string variable ::= variable . space . ⟨*LENGTH*⟩ . space . length designation

string variable ::= variable | constant string variable

The variable denotes a value. This value can be used in expressions to create new values and can be changed by the assignment statement.

The constant length string variable denotes a string with an indicated length given after the delimiter *LENGTH*. If the length designation is a string then its value must consist only of digits and is considered a decimal notation of the length. If it is a variable then the length of this variable determines length. The constant length string variable appears only within the construction of pattern (2.4.1). In string expressions only its identifier is used.

Examples. Variable:

program
EFFECT
DATE

Operand:

NAME
⟦*Mr. Smith*⟧

Constant length string variable:

HOP LENGTH ten
BOND LENGTH ⟦*007*⟧

string expression ::= string expression . space . ⟨*AND*⟩ . space . operand | operand

The string expression is a rule for obtaining the string value. This value is obtained by carrying out the operation of concatenation on actual string values. (If V_1 and V_2 are values of strings S_1 and S_2, then the result of operation S_1 *AND* S_2 is the string S_3 whose value is V_1V_2.) The operation of concatenation is associative.

Examples.

⟦*Address:*⟧
CITY
⟦*Address:*⟧ *AND CITY*

2.4. Statements

Units of language having operational significance, are called statements. In the considered language under consideration there occur statements of assignment, jump and function. The program statements are carried out successively one after the other. This successive processing of statements may be interrupted by a jump statement. To be able to define the order of performance of statements they can be furnished with labels.

simple pattern ::= $\langle FROM\rangle$. space . string expression . space . $\langle TO\rangle$. space . string expression . space . $\langle FOR\rangle$. space . string variable | string expression

pattern ::= pattern . space . pattern | $\langle SEARCH\rangle$. space . simple pattern

assignment by pattern ::= $\langle IN\rangle$. space . variable . space . pattern

replace ::= assignement by pattern . space . $\langle REPLACE\rangle$. space . string expression

left part ::= left part . space . left part | $\langle INTO\rangle$. space . variable

simple assignment ::= left part . space . $\langle PUT\rangle$. space . string expression

logical value assignment ::= left part . space . $\langle PUT\rangle$. space . variable

assignment statement ::= assignment by pattern | replace | simple assignment | logical value assignment

Assignment statement are used to assign values to one or several variables. With the individual kinds of assignment statements the assignment process runs as follows:

Simple assignment. The values of the string expression is defined in statement and assigned to all variables in the left part.

Logical values assignment. The value of the variable on the right side of the statement is assigned to all variables in the left part. If this value of the variable is not a logical value the statement becomes a simple assignment statement.

Assignment by pattern. The pattern in the statement can be made up by the sequence of simple patterns or string expressions.

Let the assignment statement in terms of the form

(1) $$IN\ V\ SEARCH\ FROM\ SE_{11}\ TO\ SE_{12}\ FOR\ V_1\ SEARCH\ FROM\ SE_{21}\ TOO\ TO\ SE_{22}\ FOR\ V_2\ \ldots\ SEARCH\ FROM\ SE_{n1}\ TO\ SE_{n2}\ FOR\ V_n$$

where SE_{11}, SE_{12}, SE_{21}, SE_{22}, ... SE_{n1}, SE_{n2} are string, expressions and V_1, V_2, ... V_n are string variables.

Let S be string which is the value of variable V. Let S_{11}, S_{12}, S_{21}, S_{22}, ... S_{n1}, S_{n2} be the values of string expressions SE_{11}, SE_{12}, SE_{21}, SE_{22}, ... SE_{n1}, SE_{n2} respectively

Assignement by pattern is carried out as follows:

Let us consider the following conditions:

1. There exist such strings T_i $(1 \leqq i \leqq n)$ and U_j $(1 \leqq j \leqq n+1)$, that string S is the value of the string expression

(2) $$U_1\ AND\ S_{11}\ AND\ T_1\ AND\ S_{12}\ AND\ U_2\ AND\ S_{21}\ AND\ T_2\ AND\ S_{22}\ AND\ \ldots\ AND\ U_n\ AND\ S_{n1}\ AND\ T_n\ AND\ S_{n2}\ AND\ U_{n+1}$$

whereby the $2n$-tuple of strings $U_1, T_1, U_2, T_2, \ldots, U_n, T_n$ has the smallest length* of all of these $2n$-tuples which may occur in designing (2) so that string S is the value of (2).

2. If in the (1) $(V_{i1}, V_{i2}, \ldots, V_{iz}) \subset (V_1, V_2, \ldots, V_n)$ are constant length string variables with lengths of $L_1, L_2, \ldots, L_z$ and $2n$-tuple of strings $U_1, T_1, U_2, \ldots, U_n, T_n$ is such that

$$l(T_{i1}) = L_1\,, \quad l(T_{i2}) = L_2, \ldots, l(T_{iz}) = L_z$$

where

$$(T_{i1}, T_{i2}, \ldots, T_{iz}) \subset (T_1, T_2, \ldots, T_n)$$

then values $T_1, T_2, \ldots, T_n$ are assigned to the variables $V_1, V_2, \ldots, V_n$ in the pattern.

The assignment is not performed if those conditions are not satisfied. If a simple pattern is a string expression then it is only ascertained whether it appears as substring in string S.

Change. Change is a broadening of assignment by pattern. If in the assignment by pattern statement, being a part of a change a value was assigned to all variables or all occurrences of string expression values were found, the following change is made:

String expressions S_{i1} *AND* T_i *AND* S_{i2}, $i = 1, 2, \ldots, n$,

are replaced by the value of the string expression to the right of *REPLACE.*

A side effect of assignment by pattern is the asquisition of the logical value of the internal variable *EFFECT.* The internal variable *EFFECT* acquires logical value *TRUE*, if to all variables in the pattern were assigned values, i.e. each incidence of string expression values was found. In the opposite case the internal variable *EFFECT* acquires logical value *FALSE.*

Examples. Simple assignment:

INTO variable PUT⟦ ⟧
INTO V1 INTO V2 INTO V3 PUT ⟦*a*⟧ *AND string TOO*
AND ⟦*z*⟧
INTO EFFECT PUT ⟦:=⟧ *AND expression*
INTO VARIABLE PUT EFFECT

Simple pattern:

FROM ⟦**if**⟧ *TO* ⟦**then**⟧ *FOR expression*
FROM ⟦*ab*⟧ *TO VAR AND variable FOR STRING LENGTH* ⟦*3*⟧
⟦*XYZ*⟧
⟦*AB*⟧ *AND INN AND* ⟦*Z*⟧

* Let A be the set of n-tuples of strings. We say that the n-tuple of strings $(A_1, A_2, \ldots, A_n)$ is shorter in length than the n-tuple $(B_1, B_2, \ldots, B_n) \in A$ if for the smallest i, $1 \leq i \leq n$, for which $l(A_i) \neq l(B_i)$ is $l(A_i) < l(B_i)$.

Pattern:

SEARCH FROM name TO address FOR city LENGTH ⟦5⟧
SEARCH a AND b SEARCH FROM m TO s FOR VALUE

Assignment by pattern:

IN STRING SEARCH a AND b

IN program SEARCH FROM ⟦**begin**⟧ *TO* ⟦**end**⟧ *FOR block TOO*
SEARCH FROM ⟦ ⟧ *TO string AND* ⟦;⟧ *FOR var TOO*
LENGTH L TOO
SEARCH DECLARATION

Change:

IN string SEARCH FROM ⟦ ⟧ *TO var FOR V REPLACE* ⟦ ⟧
IN express SEARCH ⟦$a + b$⟧ *REPLACE* ⟦$a - b$⟧ *AND variable*

2.4.2. *Jump statements*

label ::= identifier

destination ::= identifier

go clause ::= ⟨*GO* ∘ *TO*⟩ . space . destination |
⟨*GO* ∘ *BY*⟩ . space . variable

if clause ::= ⟨*IF*⟩ . space . variable . space . logical value

else clause ::= ⟨*ELSE* ∘ *TO*⟩ . space . destination |
⟨*ELSF* ∘ *BY*⟩ . space . variable

unconditional jump ::= go clause

conditional jump ::= go clause . space . if clause |
go clause . space . if clause . space .
else clause

jump statement ::= unconditional jump | conditional jump

The significance of the label is obvious.

Destination is the identifier of the label. If the go clause or if clause involves a variable, then the last value (in a dynamic sense) of the given variable is label pertaining to the jump statement.

The unconditional jump statement has the effect that as the subsequent program statement the having the same label as in the jump statement will be carried out.

The semantics of the conditional jump statement depends on the logical value of the variable featuring in the if clause. If its value complies with the logical value indicated in the if clause then effect of the conditional jump "go clause . if clause" equals the unconditional jump. In the opposite case the statement has no effect whatsoever.

If the conditional jump has the form "go clause . space . if clause . space . else clause .", first the "go clause . if clause" part is evaluated. If this part has the effect of an unconditional jump, then this is carried out and "else clause" is not considered. In the opposite case the unconditional jump is carried out upon the label in the "else clause".

The label, determined by the same identifier, can appear just once in the program (2.4.4). If the destination or value of the variable in the jump statement has not adequate label, then this jump statement has no operational effect.

Examples.

GO ∘ TO LABEL
GO ∘ BY variable
GO ∘ TO L2 IF EFFECT TRUE
GO ∘ TO L3 IF variable FALSE ELSE ∘ BY VARIABLE

2.4.3. *Function statement*

function identifier ::= identifier

function statement ::= ⟨*DO*⟩ . space . function identifier . space .
⟨*OF*⟩ . space . operand

Function identifiers and the effects of function statements are defined by the implementation of SMAL language.

2.4.4. *Program statements. Program*

unlabeled program statement ::= assignment statement . new line |
jump statement . new line |
function statement . new line |
new line

program statement ::= label . space . program statement |
unlabeled program statement

proper program ::= proper program . program statement |
program statement

program ::= basic separator . ⟨*START*⟩ . basic separator .
proper program . ⟨*FINISH*⟩ . basic separator

The semantics of the program statement is given by the semantics of the statement of assignment, jump and function. The program is made up to the sequence of program statements. The effect of the program consists in successively performing the program statements.

3. IMPLEMENTATION

In this part processing system for the program written in the SMAL language is described. Components of this system are the *translating program* (translator) and the *interpreting program* (interpreter). According to the operation of the translator and interpreter we speak of the program translation and program interpretation stage.

The main for choosing this mode of the SMAL language implementation are:

1. The length of the translated program is smaller than in generating a machine code of the translated program.

2. Such mode does not require the design of a pretentious running system needed for the computers with such storage organization as that of the GIER computer on which the SMAL language has been implemented.

3. The system is easily expandable in connection with the possible expansion of the SMAL language.

Translation consists of three passes described under headings 3.1, 3.2 and 3.3. Interpretation is described in chapter 3.4.

In the headings 3.1 and 3.3 we try to give semiformal definition of the L_i and L_f languages.

3.1. First pass

The first pass executes the transformation of the program written in SMAL language into the intermediate language L_i.

sentence in L_i ::= intermediate program . tables

tables ::= constants and label identifiers table .
operands table . labels table

The division of "sentence in L_i" into these four portions ensues from the segmentation of the accessible storage section P as shown in Fig. 1.

intermediate program ::= intermediate program .
intermediate word | intermediate word

intermediate word ::= operand address | delimiter address |
function address

operand address constitutes the connection of intermediate program with "operand table" and with "label table". "Operand address" may be the address of the operand and in that case it refers to the "operand table" or in case of respectively the label address or "destination" it refers to the "label table".

delimiter address is the internal representation of "delimiter".

function address is the address of the position of the function interpretation program.

constants and label identifiers table ::= constants and label identifiers table .
constant or label | constant or label

constant or label ::= constant in L_i | label in L_i

constant in L_i is the internal representation of the constant (string), depending on the actual implementation.

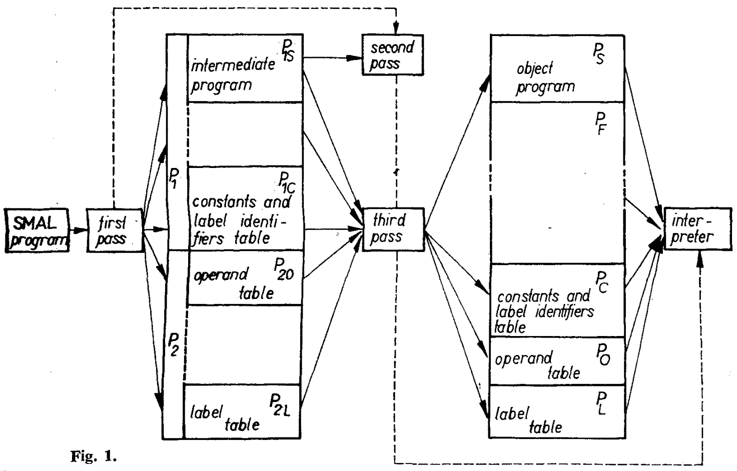

Fig. 1.

label in L_i is the inner representation of the label identifier, identical with the inner representation of "constant in L_i".

operand table ::= operand table . operand item | operand item

operand item ::= variable item | constant item

constant item ::= constant position . length

operand table constitutes a connection between the "intermediate program" and the "constants and label identifiers table" and offers complete information on constants and variables occurring in the program.

variable item is the inner representation of the variable identifier, identical with the inner representation of "constant in L_i".

constant position is the constant position address in the "constants and label identifiers table" the length of which is given by

length—they constitute together the "constant item",

label table ::= label table . label item | label item label item ::= label position . destination address

label table constitutes connection between "intermediate program" and "constants and label identifiers table" and offers complete information on label occurring in the program.

label position is the position address of the inner label representation in the "constants and label identifiers table".

destination address indicates the address of statement designated by the label.

3.2. Second pass

The second pass performs a syntactic check on the program transformed into the "intermediate program". Syntactic checking is done from left to right whereby label, string expression, simple pattern and statement are looked upon as syntactic units.

The construction of the SMAL language allows to check the correctnees of the individual statements syntax independently and the check itself is implemented through a system of subroutines for the individual statements within the framework of which the subroutines for checking the relevant syntactic units are utilised. The following procedures for syntactic check ensues from the abovesaid and from the fact each statement may be provided with a label:

The actual "intermediate word" is the first "intermediate word" in the "intermediate program".

a) ascertain whether "operand address" of the actual "intermediate word" designates label. If it is so, proceed at point d), otherwise

b) ascertain whether "delimiter address" of the actual "intermediate word" determines the statement. If this is so, syntactic check of the statement determined by that address is made. In the opposite case, if the "delimiter address" is *FINISH* the second transition ceases to operate, otherwise

c) the subsequent "intermediate word" is considered as actual and action is resumed at point a);

d) ascertain whether "delimiter address" is the "end" of the statement, if yes, action is resumed at point c), otherwise at point b).

Within the statement the correctness of those syntactic units is checked that are in compliance with the statement syntax.

3.3. Third pass

It ensues from the syntax and semantics of the SMAL language that the most suitable mode of the program interpretation consists in independent performing individual statements by unique interpretation programs.

The third pass performs the transformation of the "intermediate program" in "sentence in L_i" into the "object program", thereby making informal adjustements in the "tables" consisting in shifting sections P_{1C} and P_{2O} so as attain a continuous section. In this way one attains a sentence in the L_f language, which is suitable for interpretation.

object program ::= object program . statement | statement

statement ::= statement address . parameter list

statement address ::= INTO address | IN address | GO address |
DO address

parameter list ::= SEARCH param list | GO param list |
PUT param list

delimiter address in L_f ::= SEARCH address | FOR address |
LENGTH address | GO TO address | GO BY address |
ELSE TO address | ELSE BY address | PUT address |
REPLACE address | TRUE address | FALSE address |
OF address

statement address indicates the position address of the interpretation program implementing the given statement.

parameter list is determined by the type of statement address and its structure is given below.

delimiter address in L_f is the internal representation of "delimiter" in the L_f language.

The transformation of the "intermediate program" into the "object program" is performed as follows:

The actual "intermediate word" is the first "intermediate word" in the "intermediate program". It is ascertained whether the actual "intermediate word" determines the label or the statement. In case the given word determines the label, the "destination address" is completed in the corresponding "label item". If the given word determines the statement, operation continues according to the type of the given statement. Having processed it the subsequent "intermediate word" becomes the actual "intermediate word".

The structure of the statement for each type of "statement address" in the L_f language is the following:

assignment statement ::= INTO address . left part list .
PUT param list

left part list ::= left part list . operand address | operand address

PUT param list ::= PUT address . string expression

string expression ::= string expression . openard address | operand address assignment by pattern ::= IN address . operand address . pattern

pattern ::= pattern . SEARCH param list | SEARCH param list

SEARCH param list ::= SEARCH address . string expression .
FOR address . operand address . string expression |
SEARCH address . string expression . FOR address .
operand address . LENGTH address . operand address .
string expression | SEARCH address . string expression

replace ::= assignment by pattern . REPLACE address . string expression

jump statement ::= GO address . GO param list

GO param list ::= go expression | true expression . go expression |
false expression . go expression |
true expression . go expression . else expression |
false expression . go expression . else expression

go expression ::= GO TO address . operand address | GO BY address .
operand address

true expression ::= TRUE address . operand address

false expression ::= FALSE address . operand address

else expression ::= ELSE TO address . operand address |
ELSE BY address . operand address

function statement ::= DO address . operand address .
OF address . operand address

The semantics of the L_f language is defined by interpretation. After the third pass storage distribution is obvious from Fig. 1.

3.4. Interpretation programs

By performing statements in the SMAL language new values of variables are obtained. Information on the value of the given variable is provided by the variable item which in the course of interpretation assumes the form:

variable item ::= variable position . value specification . length

value specification ::= undefined | logical | empty | string

variable position position address of variable value, whose length it indicates
length

value specification indicates whether a value was assigned to the variable, if so, it defines its type.

3.4.1. *Storage allocation and storing of the variables*

During interpretation the allocation of storage section P is shown in Fig. 1., only section P_F is divided into three parts:

free table	– list of free section in P_w,
P_w	– work area for values of variables
P_f	– free area.

If the variable value is a logical value or an empty string, the complete information on that value is provided by the value specification in the variable item. If an unempty string is assigned to the variable, information on value if provided by the variable item and the string being its value, is stored in section P_w having variable length. By filling section P_w section P_f is shortened and the initial address, where actual P_f begins, is considered as a variable position in creating the value of the new variable.

If the list on the left hand side in the simple assignment includes more variables and their value is an unempty string this is stored in P_w only once and the same variable item is stored at the relavant places in the operand table.

When changing the value of the variable which is part of P_w (the old value is a unempty string) and the same value is not the value of other variables, P_w does not become smaller at once, only the variable item corresponding to the old value is stored in the free table. Thus, in the course of interpretation, section P_w is not homogeneously filled with variable values. If free table is filled, or section P_f has zero length the operation is transferred to program compress. This program according to the free table, carries out a narrowing of P_w, free sections are moved over to P_f, and P_w retains only those parts that are variable values.

Thereby the variable position in the variable item changes so as to determine the positions of the variable values. After the termination of the program compress the free table is empty. If the program compress is called and P_f and the free table are zero, the program is ended by the inability to store the string that has to be the value of some variable.

3.4.2. *The operation of interpretation programs*

Interpretation programs for the implementation of individual statements process the entire statement completely. In addition there must be an organization program, setting the free table to zero initially and storing into all variable items the value specification corresponding to undefined.

Then, according to the statement address in the first word of the object program, it transfers the operation to the corresponding interpretation program that carries out the given statement. After performing the statement, the interpretation program returns into the organizational program and offers it a word that involves the statement address of the subsequent statement.

Simple assignment forms a value that corresponds to the right side and stores the relevant variable item into the operand table at points given by the operand address on the left side.

Assignment by pattern and change defines whether the given pattern occurs in the defined string and creates a table whose items determine the initial and final symbols of substrings $T_1, T_2, \ldots, T_n$ (2.4.1). If an change is to be carried out simultaneously, a table is created, the items of which determine the initial and final symbols of substrings S_{i1} and S_{i2}, respectively (2.4.1) that are to be substituted by the value of the string expression occurring in the change. If the given pattern is included in the defined string and a table of initial and final symbols has been created the relevant assignment are carried out. If the statement also involves change, the value of the string expression is formed in the change and the new variable value is formed from the original string. The corresponding logical value is assigned to variable *EFFECT*.

The jump statement offers the organizational program a word from the final program that includes the statement address of the statement given by label.

Function statement depends on the selection of functions applied at actual realization.

4. GIER – SMAL COMPILER

The GIER – SMAL compiler is a compiler of the language SMAL written for the GIER computer (Regnecentralen, Copenhagen). The features of this computer are described in [6].

Since the implementation of the SMAL language has been generally described in part 3, here we shall concentrate on storage distribution, indication of errors occurring in the translated program and on function statements.

The layout of the GIER computer for which the SMAL compiler is written, has the following types of storages:

operational:	ferrite core memory	1024 words
backing store:	drum	320 tracks per 40 words
	buffer	4096 words
external store	magnetic type	free block structure

One word has 42 bits two of which are without weight and destined only for the marking.

4.1. First pass

Storage section P (3.1) consists of the magnetic drum part (array "free" [7]) constituting P_1 and of the buffer part, forming P_2.

Intermediate word is placed in a cell, 20 bits of which contain the operand address and rest the delimiter address.

Constant labels identifiers and variable identifiers are placed so that one alphabet symbol corresponds to an 8 bit byte, hence one word may maximally include 5 alphabet symbols.

The operand item or label item are placed in one word of the operand table or label table respectively.

Constraints. The label identifier is placed in the label table at the point of label position. In forming the operand table the variable identifiers are placed into the variable item. From this identifier length delimination follows. The first 3 alphabet-symbols are determinant.

Error messages:

compound	complex symbol not pertaining to alphabet symbols
termination	before and after bracket ⟦and⟧ no basic separator follows
⟧improper	bracket is use outside string
length	notation of length missing
+label	labels occured with the same identifier
−label	there is no label for destination
too many identifiers	label table and operand table overlap
program too big	intermediate program and constants overlap

If one of the last two messages turns up the translation process ends in contrast to the others when the process is brought to the finish.

4.2. Second pass

The second pass performs a syntactic check of the program translated by the first pass into the intermediate program, as described in 3.2.

Error messages:

basic	delimiter not defining type of statement
assign	after left part no *PUT* delimiter follows
jump	erroneous structure of jump statement
design	erroneous structure of simple pattern
function	erroneous structure of function statement
−operand	missing operand
+operand	incorrect type of operand
−delimiter	missing delimiter
+delimiter	occurrance of incorrect delimiter

4.3. Third pass

With respect to the placing of sections P_1 and P_2 after the first pass (4.1) it is not necessary to shift sections P_{1C} and P_{2O} as described in 3.3. It follows therefore that in the constant item constant position does not alter during the third pass and in the object program the operand address remains consistent with those in the intermediate program.

4.4. Interpretation

After the third pass P_F is part of the drum, namely the section between object program and constants. Free table is formed in the buffer store. Values of variables are stored into P_F after the object program thus constituting section P_w.

Since the operational storage is small, the interpretation programs must be in the buffer and the organizational program provides for shifting the corresponding interpretational program into the operational store.

Limitation. Since in interpreting assignment by pattern and replace statements tables of initial and terminal substring symbols defining simple patterns are formed (3.4), the number of simple patterns within the pattern is limited. In the GIER-SMAL compiler the maximum number of simple patterns is 20.

Error messages:

kind	variable featuring as operand in the statement has assigned a value of another type than admitted by the semantics of statement;
value	variable featuring in the statement as operand has no value assigned;
−label	the value of the jump statement variable is not consistent with any label in the program.
too long string	compress is called and free table is empty.

4.5. Functions

The function statement is interpreted similarly to any arbitrary statement with the difference that the statement address of function requires further information indicating which function is to be interpreted. This information is provided by an operand address whose value depends on the function identifier.

In the GIER-SMAL compiler the following functions handling input and output units are implemented:

select	input and output units are selected so that the first two alphabet symbols of the variable or string featuring as operand are used as a by-address [6];
input	to the operand that may be only a variable is assigned an open string read from the selected input unit;
lyn	to the operand that may be only a variable is assigned the value of one alphabet symbol from the selected input unit;

output	value of variable or string featuring as operand in the function statement is written on the selected output unit;
length	length of variable or string featuring as operand is written in decimal form on the selected output unit.

These functions may be enlarged by an arbitrary number of further functions.

(Received March 24, 1972.)

REFERENCES

[1] E. Kostolanský: Definícia syntaxe a sémantiky jazyka SNOBOL I. Kybernetika *3* (1967), 3, 3, 253—268.

[2] D. Farber, R. Griswold, I. Polonsky: SNOBOL, A String Manipulation Language. *JACM 11* (January 1964), 1, 21—32.

[3] R. Griswold: String Manipulation and SNOBOL Language. Sommer Schoole Text, Copenhagen, August 1969.

[4] J. Rohl: A Note on Backup Naur Form. The Computer Journal *10* (August 1967), 2, 336—337.

[5] P. Naur et al.: Revised Report on the Algorithmic Language ALGOL 60. Communications of the ACM *6* (January 1963), 1, 1—17.

[6] C. Gram et al.: GIER a Danish Computer of Medium Size. IEEE Trans. of Electronic Computers *EC-12* (December 1963), 5, 629—650.

[7] S. Lauesen et al.: A Manual of HELP 3. A/S Regnecentralen, Copenhagen, 1967.

Ján Chovanec, prom. mat., Ada Chudá, prom. mat., RNDr. Eduard Kostolanský, Dušan Ondruš, prom. mat.; Ústav technickej kybernetiky SAV (Institute of Technical Cybernetics — Slovak Academy of Sciences), Dúbravská cesta 1, 809 31 Bratislava 9. Czechoslovakia.