PL/I

TECHNICAL REPORT

TR 25.087
28 June 1968

ULD
VERSION
II

METHOD AND NOTATION FOR THE FORMAL DEFINITION OF PROGRAMMING LANGUAGES

P. LUCAS
P. LAUER
H. STIGLEITNER

IBM LABORATORY VIENNA

IBM LABORATORY VIENNA, Austria

METHOD AND NOTATION FOR THE FORMAL DEFINITION OF PROGRAMMING LANGUAGES

by

P. LUCAS
P. LAUER
H. STIGLEITNER

ABSTRACT

This document is a tutorial introduction to the method used in the formal definition of programming languages. The method is presented, as far as possible, independently of its application to any particular programming language.

Locator Terms for IBM Subject Index

PL/I
Formal Definition
Syntax
Semantics
21 PROGRAMMING

TR 25.087
28 June 1968

PREFACE

The method for formally defining programming languages presented in this document was developed by the Vienna Laboratory in order to produce a formal definition of PL/I. A first version of this formal definition was made available in the form of two technical reports /1/, /2/ and the method was elaborated in a tutorial style in /3/. The second version of the complete formal definition consists of the technical reports /4/, /5/, /6/, /7/ and /8/ all of which were issued by 28 June 1968.

The initial basis for the development of the method adopted is to be found in publications of J.McCarthy /9/, /10/, C.C.Elgot /11/, and P.Landin /12/. The early ideas of the Vienna group on the method are documented in /13/, /14/.

There has also been an extensive exchange of working papers between the Poughkeepsie Laboratory, the Hursley Laboratory and the Vienna Laboratory. In particular the Hursley group produced a number of relevant technical reports /15/, /16/, /17/ and /18/.

The applicability of the method developed by the Vienna Laboratory in the course of the formal definition of PL/I is, however, not limited to any specific programming language and it is the intent of the authors to present here those features of the method which reflect its generality.

REFERENCES

/1/ PL/I-Definition Group of the Vienna Laboratory: Formal Definition of PL/I (Universal Language Document No. 3).-
IBM Laboratory Vienna, Techn. Report TR 25.071 (Version 1), 30 December 1966.

/2/ ALBER, K.: Syntactical Description of PL/I Text and its Translation into Abstract Normal Form.-
IBM Laboratory Vienna, Techn. Report TR 25.074, 14 April 1967.

/3/ LUCAS, P.: Introduction to the Method Used for the Formal Definition of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.081, 31 October 1967, 28 June 1968 (Revised).

/4/ FLECK, M., NEUHOLD, E.: Formal Definition of the PL/I Compile Time Facilities.-
IBM Laboratory Vienna, Techn. Report TR 25.080, 28 June 1968.

/5/ ALBER, K., OLIVA, P., URSCHLER, G.: Concrete Syntax of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.084, 28 June 1968.

/6/ ALBER, K., OLIVA, P.: Translation of PL/I into Abstract Text.-
IBM Laboratory Vienna, Techn. Report TR 25.086, 28 June 1968.

/7/ LUCAS, P., ALBER, K., BANDAT, K., BEKIC, H., OLIVA, P., WALK, K., ZEISEL, G.: Informal Introduction to the Abstract Syntax and Interpretation of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.083, 28 June 1968.

/8/ WALK, K., ALBER, K., BANDAT, K., BEKIC, H., CHROUST, G., KUDIELKA, V., OLIVA, P., ZEISEL, G.: Abstract Syntax and Interpretation of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.082, 28 June 1968.

/9/ McCARTHY, J.: Towards a Mathematical Science of Computation.-
In: Information Processing 1962 (C.M. POPPLEWELL, Ed.), pp. 21-28; North-Holland Publ. Comp., Amsterdam 1963.

/10/ McCARTHY, J.: A Formal Description of a Subset of ALGOL.-
In: Formal Language Description Languages (T.B. STEEL Jr., Ed.), pp. 1-12, Proc. IFIP Working Conference, Vienna 1964; North-Holland Publ. Comp., Amsterdam 1965.

/11/ ELGOT, C.C., ROBINSON, A.: Random-Access Stored-Program Machines. An Approach to Programming Languages.-
J.ACM 11 (1964) No.4, pp. 365-399.

/12/ LANDIN, P.J.: Correspondence Between ALGOL 60 and Church's Lambda-Notation, Part I.-
Comm.ACM 8 (1965) No.2, pp.89-101.

LANDIN, P.J.: Correspondence Between ALGOL 60 and Church's Lambda-Notation, Part II.-
Comm. ACM 8 (1965) No. 3, pp.158-165.

/13/ BANDAT, K.(Ed.): Tentative Steps Towards a Formal Definition of Semantics of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.056, July 1965.

/14/ LUCAS, P.: On the Formalization of Syntax and Semantics of PL/I.-
IBM Laboratory Vienna, Techn. Report TR 25.060, November 1965.

/15/ BEECH, D., ROWE, R., LARNER, R.A., NICHOLLS, J.E.: Abstract Syntax of PL/I.-
IBM UK Laboratories Hursley, Techn. Note TN 3002, June 1966.

/16/ BEECH, D., NICHOLLS, J.E., ROWE, R.: A PL/I Translator.-
IBM UK Laboratories Hursley, Techn. Note TN 3003, October 1966.

/17/ BEECH, D., ROWE, R., LARNER, R.A., NICHOLLS, J.E.: Concrete Syntax of PL/I.-
IBM UK Laboratories Hursley, Techn. Note TN 3001, November 1966.

/18/ ALLEN, C.D., BEECH, D., NICHOLLS, J.E., ROWE, R.: An Abstract Interpreter of PL/I.-
IBM UK Laboratories Hursley, Techn. Note TN 3004, November 1966.

/19/ NAUR, P.: Revised Report on the Algorithmic Language ALGOL 60.-
The Computer Journal (1963), pp. 349-367.

/20/ LAUER, P.: Abstract Syntax and Interpretation of ALGOL 60.-
IBM Laboratory Vienna, Lab. Report LR 25.6.001, 12 April 1968.

/21/ LAUER, P.: Concrete Representation of Abstract ALGOL-60 Programs.-
IBM Laboratory Vienna, Lab. Report TR 25.6.002, 13 May 1968.

C O N T E N T S

		Page
1.	INTRODUCTION	1-1
1.1	Exposition of the Problem	1-1
1.2	A Brief Description of the Example Programming Language EPL	1-3
1.3	Notational Conventions Presupposed in this Report	1-5
1.3.1	Conditional Expressions	1-5
1.3.2	Equality	1-6
1.3.3	Truth Values, Logical Operators and Quantifiers	1-6
1.3.4	Arithmetic Operators and Relations	1-8
1.3.5	Set Operators, Relations and Notation for Sets	1-9
1.3.6	Functional Composition	1-9
1.3.7	Rules of Precedence	1-10
2.	OBJECTS	2-1
2.1	The General Class of Objects	2-1
2.2	Graphic Representation of Objects	2-3
2.3	The Null Object	2-4
2.4	Composite Selectors	2-5
2.5	The Characteristic Set of an Object	2-6
2.6	The Operator μ	2-10
2.7	Definition of Classes of Objects	2-12
2.8	Further Notational Conventions	2-18
2.8.1	Extension of the Meaning of the μ Operator	2-18
2.8.2	Manipulation of Lists	2-20
2.9	Implicit Definitions	2-21
3.	APPLICATION TO ABSTRACT SYNTAX AND REPRESENTATION OF LANGUAGES	3-1
3.1	Abstract Syntax of EPL	3-2
3.2	The Definition of Concrete Representations	3-6
3.2.1	The Representation System	3-7
3.2.2	A Concrete Representation of Programs of EPL	3-9

Page

3.3 The Correlation of Abstract Syntax and Backus Normal Form 3-12
3.4 A Concrete Syntax for EPL 3-14
3.4.1 The Extended Backus Notation 3-15
3.4.2 A Concrete Syntax 3-17
3.5 The Translation of Concrete Programs to Abstract Programs 3-18
3.5.1 Abstract Representation of Concrete Programs 3-20
3.5.2 The Translator 3-24

4. ABSTRACT MACHINES 4-1

4.1 Introduction 4-1
4.2 The Conventional Concept of Abstract Sequential Machines 4-1
4.3 The Extended Concept of Abstract Machines as Used for the Formal Definition of Programming Languages 4-2
4.4 The Control of the Abstract Machine 4-3
4.4.1 First Survey 4-3
4.4.2 Control Trees 4-5
4.4.3 Defining Control Trees as Objects 4-8
4.4.4 Control Tree Representations 4-10
4.4.5 Instruction Schemata 4-12
4.4.6 The State Transition Function Λ 4-14
4.4.7 Examples 4-15
4.5 Note on Constructs of the State and Some Instructions of the Abstract Machine 4-21
4.5.1 Unique Name Generation 4-21
4.5.2 Representing Functions by Objects 4-22
4.5.3 Realization of Stacks 4-25
4.5.4 Reference to State Components in Instruction Definitions 4-26
4.5.5 The Null Instruction 4-27
4.5.6 Pass Instructions 4-28
4.5.7 Element by Element Evaluation of a List 4-28

5. DEFINING THE INTERPRETATION OF EPL 5-1

5.1 The States of the Interpreting Machine 5-1
5.2 The Interpretation of the Language 5-2
5.3 Intuitive Description Based upon the Formal Definition 5-8
5.3.1 The Components of the State 5-8
5.3.2 Types of Identifiers and Their Dynamic Significance 5-10
5.3.3 Flow of Control 5-12

1. INTRODUCTION

1.1 Exposition of the Problem

The problem to be solved by the method presented in this report is the syntactic and semantic definition of programming languages. The method was developed in view of the definition of PL/I; however, there are a number of principles which are quite general and not restricted to the definition of any particular programming language. This report is an attempt to isolate these more general aspects of the method.

The syntactic definition of a language is, in general, a set of rules which define a set of strings constructed from a given alphabet. In addition the syntax defines a structure for each of these strings. Only strings defined by the syntax are considered to be valid expressions (programs in the case of programming languages) of the language and are subject to interpretation. The structure given to an expression by means of the syntax is of importance in its subsequent interpretation. The syntatic definition of a programming language is usually given by means of productions, i.e., rules allowing the generation of all strings of characters which are considered to be programs of the programming language.

The semantic definition of a language is understood to be a set of rules which allows the interpretation of the expressions specified by the syntax. The method under discussion in this report is based on the definition of an abstract machine which is characterized by the set of its states and its state transition function. A program together with its input data defines an initial state and the subsequent behaviour of the machine is said to define the interpretation of the program for the given input data.

However, it would be cumbersome and unnecessarily confusing for the interpreting abstract machine to operate directly upon the character strings defined by the syntax, i.e., for the syntax analysis to be defined explicitly by the machine. Instead, it is assumed that there is a unique parsing tree according to the syntax for any program and that there is an algorithm which computes the parsing tree. It is part of the definition method here presented to assert exactly this for any language to be defined by the method. It would still be awkward to attach the interpretation directly to the parsing trees since there are in most higher level programming languages conventions which allow the same process to be described in many different ways. The parsing trees are therefore modified in such a way that programs describing the same process which differ only in the use of the notational conventions appear in standard form. In the method described in this

paper the so-called translator performs precisely this task. What remains to be interpreted by the abstract machine are objects which possess a tree structure and represent programs in standard form. The definition of the class of objects which are considered to represent programs (or expressions of a language in general) is called the abstract syntax. The syntactic definition which specifies programs as a set of character strings is called the concrete syntax.

One may, however, consider the objects themselves, as defined by the abstract syntax, to constitute the expressions of the language. In this case there is no concrete syntax specifying what constitutes the language. Then an interpretation of the language can be given directly on the one hand and a possible concrete representation in terms of character strings on the other. Furthermore, one may consider languages which have the same abstract syntax and the same interpretation and which differ only in their concrete representations to be equivalent.

It proved to be convenient to represent the states of the interpreting abstract machine as objects of exactly the same kind as used to represent programs. The set of states the machine can assume may therefore be defined by means of the same devices as used for the abstract syntax of programs.

The method as presented in this report porvides a general class of objects having tree structures. Subclasses of this general class of objects may then be used to represent programs (and have been used to represent programs of specific higher level programming languages, cf. /3/ - /8/, /20/); other subclasses may be used to represent the states of the interpreting machine (and have been used for the states of the machines interpreting the specific programming languages in the reports cited above).

The outline of this report is such that it starts with the most general aspects of the method and then specializes to programming languages.

The above mentioned general class of objects, together with devices for defining subclasses of this class and for manipulating objects is presented in Chapter 2. Only the general properties of objects are dealt with in this chapter and no application to languages or abstract machines is necessarily implied.

The application of the devices, introduced in Chapter 2, to the definition of the syntax and semantics of languages will be demonstrated throughout this report by means of a simple example programming language. An intuitive characterization of the features of this example language is included in this introductory chapter.

In Chapter 3 the notions of abstract and concrete syntax of languages are discussed and the problem of the formulation and the relating of these two types of syntax is dealt with by means of the devices introduced in Chapter 2.

Chapter 4 introduces the notion of abstract machines and the special kind of machine which lies at the basis of the method for interpreting programming languages here being presented. The application of the devices introduced in Chapter 4, is shown by giving a definition of the interpretation for the example programming language in Chapter 5.

An attempt was made in this report not only to show how the definition of a programming language can be constructed but also to indicate how one may possibly discover the consequences of such a definition. The comments to the example in Section 5.3 serve this latter purpose.

1.2 A Brief Description of the Example Programming Language EPL

Throughout the report the definition of a simple programming language (henceforth called EPL) serves as an example for the application of the methods described in the report. The programming language to be described is structured similarly to PL/I (and ALGOL 60); however the number of features covered in EPL is very small when compared with PL/I.

In this section a brief survey of the structure and content of the language is presented in order to give the reader an intuitive idea of EPL which will aid his understanding of the various formal devices used later on to describe syntactic or semantic features of EPL. The terminology used in this survey presupposes some familiarity with PL/I or ALGOL 60.

The data manipulable by the language are truth values and integer values. It is assumed that there are unary and binary operations defined for these data. Which specific operations are available is, however, left open.

Expressions may be built from constants, variables and function designators using the operators.

Constants denote values. Variables are identifiers denoting values. The values denoted by a variable may change dynamically through the execution of assignment statements. The range of a specific variable is restricted either to truth values or to integer values.

A function designator consists of an identifier which is a function identifier, and an argument list which is a (possibly empty) list of identifiers of some type. A function identifier denotes a rule for computing a value depending on arguments. In particular a function identifier denotes a statement and an expression. The value of the function is determined by executing the statement and afterwards evaluating the expression which then yields the value of the function.

An expression is a rule for computing a value. The computation may, besides yielding a value, change the state of the machine in some way (side effects).

There are four kinds of executable statements, namely assignment statements, conditional statements, procedure calls and blocks. An assignment statement consists of a leftpart which is a variable and a rightpart which is an expression. Upon execution the expression is evaluated and the value computed is assigned to the variable which was the leftpart. The variable then denotes the value computed by the expression until further changes.

A conditional statement consists of an expression and two statements: the then-statement and the else-statement. Upon execution the expression is evaluated and if necessary its value is converted to a truth value. If the value is T, the then-statement is executed, otherwise, the else-statement is executed. A procedure call consists of an identifier which is a procedure identifier, and an argument list which is a list of identifiers of some type. A procedure identifier denotes a rule for computing (a statement) depending on the arguments. The argument-passing for function activations and procedure calls is defined in such a way that the parameters are defined to be completely synonymous with the corresponding arguments of the respective activation or call within their scope.

A block consists of a declaration part and a list of statements of any type. The declaration part declares certain identifiers to be either variables (integer or logical) or function identifiers or procedure identifiers. The function or procedure denoted by an identifier is also given in its declaration, i.e. the identifier is associated with a parameter list, statement and expression in the case of functions and with a parameter list and a statement in the case of procedures.

Upon execution of a block the identifiers declared by the declaration part are introduced as names denoting new entities (in accordance with their declaration) and keep this meaning until the end of the execution of the block. The meaning of any identifier, which is identical to a newly declared one, is supressed until the end of the execution of the block. The identifiers declared in the decla-

ration part of a block are called local to that block. The parameters of a function or procedure are treated upon execution of the function or procedure as if they were declared in that function or procedure. This means parameters of a function or procedure are considered to be local identifiers of the respective function or procedure. The meaning of non local identifiers of a function or procedure is frozen after introduction of the new identifiers of the block in which the function or procedure is declared. A program in the language described is a block.

1.3 Notational Conventions Presupposed in this Report

This section is a summary of the notational conventions and symbols taken over from various fields. A major portion of the notation is adopted from LISP (conditional expressions), predicate calculus, arithmetic expressions and relations and set theory with the conventional meaning. All symbols and conventions referred to in this section will be used throughout the document without further comment. This section does not contain the various conventions whose introduction and definition is a major purpose of this paper.

1.3.1 Conditional expressions

Form:

$(p_1 \longrightarrow e_1, p_2 \longrightarrow e_2, \ldots, p_n \longrightarrow e_n)$

p_i expression denoting a truth value

e_i expression denoting some object (the value of e_i)

An alternative form may be used omiting the parentheses and commas:

$$p_1 \longrightarrow e_1$$
$$p_2 \longrightarrow e_2$$
$$\vdots$$
$$p_n \longrightarrow e_n$$

Meaning:

A conditional expression denotes the value of e_i where i is the smallest integer, $1 \leq i \leq n$, for which p_i is true and all preceding p_j, $1 \leq j < i$, are false. If there is no such integer, then the expression has no value.

It is important to note that the left to right order in which the individual conditions p_i are inspected is relevant. If p_i is true then a consequence of the above definition is that the values of the successors of p_i say p_k, $i<k\leq n$, are irrelevant for the valuation of the conditional expression and may even be undefined.

1.3.2 Equality

$=$ equal
$\neq$ not equal
$\overline{\overline{Df}}$ by definition equivalent to

The equality and not equality relations are used with no specific restriction as to the range of arguments.

1.3.3 Truth values, logical operators and quantifiers

1.3.3.1 Truth values

T true
F false

1.3.3.2 Logical operators

$\neg$	not	$\equiv$	equivalence
&	logical and	$\not\equiv$	non equivalence
v	logical or (vel)	$\supset$	implication

The operators have the conventional meaning except for two place operators in cases where one of the operands is undefined. The meaning adopted is best described using conditional expressions:

$(p_1 \,\&\, p_2) \overline{\overline{Df}} (\neg p_1 \longrightarrow F, T \longrightarrow p_2)$
$(p_1 \,v\, p_2) \overline{\overline{Df}} (p_1 \longrightarrow T, T \longrightarrow p_2)$
$(p_1 \equiv p_2) \overline{\overline{Df}} (p_1 \,\&\, p_2) \,v\, (\neg p_1 \,\&\, \neg p_2)$
$(p_1 \not\equiv p_2) \overline{\overline{Df}} \neg(p_1 \equiv p_2)$
$(p_1 \supset p_2) \overline{\overline{Df}} (\neg p_1 \longrightarrow T, T \longrightarrow p_2)$

The following rules for omission of parentheses hold for expressions built from the above operators:

$$p_1 \,\&\, p_2 \,\&\, \ldots \,\&\, p_{n-1} \,\&\, p_n \;\bar{\bar{D}}f\; (p_1 \,\&\, (p_2 \,\&\, \ldots\, (p_{n-1} \,\&\, p_n) \ldots))$$

$$p_1 \vee p_2 \vee \ldots \vee p_{n-1} \vee p_n \;\bar{\bar{D}}f\; (p_1 \vee (p_2 \vee \ldots\, (p_{n-1} \vee p_n) \ldots))$$

$$p_1 \supset p_2 \supset \ldots \supset p_{n-1} \supset p_n \;\bar{\bar{D}}f\; (p_1 \supset (p_2 \supset \ldots\, (p_{n-1} \supset p_n) \ldots))$$

To ease printing the symbols "Et" and "Vel" are used for multiple conjunction and disjunction.

$$\underset{i=1}{\overset{n}{Et}}\; p_i \;\bar{\bar{D}}f\; ((n>0) \longrightarrow p_1 \,\&\, p_2 \,\&\, \ldots \,\&\, p_n,\ (n = 0) \longrightarrow T)$$

$$\underset{i=1}{\overset{n}{Vel}}\; p_i \;\bar{\bar{D}}f\; ((n>0) \longrightarrow p_1 \vee p_2 \vee \ldots \vee p_n,\ (n = 0) \longrightarrow F)$$

1.3.3.3 Quantifiers

$\exists$	existential quantifier
$\forall$	universal quantifier

The above symbols will be used in expressions of the following forms:

$(\exists\, x_1, x_2, \ldots, x_n)(p(x_1, x_2, \ldots, x_n))$

The variables $x_1, x_2, \ldots, x_n$ are called the bound variables [1] of the expression. The expression is true if there exists at least one n-tuple $x_1, x_2, \ldots, x_n$ such that $p(x_1, x_2, \ldots, x_n)$ is true, otherwise the expression is false.

$(\forall\, x_1, x_2, \ldots, x_n)(p(x_1, x_2, \ldots, x_n))$

The variables $x_1, x_2, \ldots, x_n$ are called the bound variables [1] of the expression. The expression is true if for all possible n-tuples $x_1, x_2, \ldots, x_n$ (in the range of the variables) $p(x_1, x_2, \ldots, x_n)$ is true, otherwise the expression is false.

It is important that the range of the bound variable in an expression of the above form should always be defined. This will either explicitly be done by the expression or implicitly by using a convention that associates a range with a specific class of variable names.

---

1) bound variables are variables for which no substitution is allowed.

For convenience, composite constructs containing bound variables as components may be written in place of the bound variable part of the expression, e.g. $(\exists \langle x,y \rangle)\ (p(x,y,\langle x,y \rangle))$.

1.3.3.4 Description

ι iota-operator

The symbol will be used in expressions of the following form:

$$(\iota x)(p(x))$$

The x is called the bound variable of the expression. The expression denotes the value (in the range of x) for which p(x) is true. The expression has no value if no value or more than one value in the range of x has the property p.

1.3.4 Arithmetic operators and relations

1.3.4.1 Operators

- \+ prefix plus, infix plus
- \- prefix minus, infix minus
- * multiplication

1.3.4.2 Relations

- $<$ less
- $\leq$ less or equal
- $=$ equal
- $\neq$ not equal
- $\geq$ greater or equal
- $>$ greater

The relational operators are occasionally used in expressions of the form:

$$e_1\ R_1\ e_2\ R_2\ e_3$$

where e_i is an arithmetic expression and R_i is one of the above relational operators. The meaning is as usual:

$$e_1\ R_1\ e_2\ R_2\ e_3 \overline{\overline{Df}}\ (e_1\ R_1\ e_2)\ \&\ (e_2\ R_2\ e_3)$$

1.3.5 Set operators, relations and notation for sets

1.3.5.1 Set oerators

$\cup$	union
$\cap$	intersection

1.3.5.2 Relations

$\in$	is element of
$\notin$	is not element of
$\subset$	is proper subset
$\subseteq$	is subset or equal
$\supseteq$	is superset or equal
$\supset$	is proper superset

1.3.5.3 Notation for sets

$\{a, b, c, \ldots\}$	The elements a, b, c, ... are the elements of the set,
$\{\ \}$	the empty set
$\{x \mid p(x)\}$	implicit definition of a set

The x is called the bound variable of the expression. The expression denotes the set of all elements such that p(x) is true. As usual, for convenience composite constructs containing bound variables as components will be written in place of the bound variable part of the expression, e.g

$$\{\langle x,y\rangle \mid p(x,y,\langle x,y\rangle)\}$$

the set of pairs <x,y> such that p (x,y,<x,y>).

1.3.6 Functional Composition

$\circ$ functional composition operator

The operator is defined by:

$$(f \circ g)(x_1,\ldots,x_n) \underset{Df}{=} f(g(x_1,\ldots,x_n)$$

f and g may be either simple function names or expressions denoting functions. In the later case, the expression must be parenthesized.

The following rules for omission of parentheses hold:

(1) The functional composition operator binds more strongly than functional application, e.g.:

$$f \circ g(x) = (f \circ g)(x)$$

(2) $f_1 \circ f_2 \circ \ldots \circ f_m\ (x_1, \ldots, x_n) = f_1(f_2(\ldots(f_m(x_1, \ldots, x_n))\ldots))$

(3) $f(x_1)(x_2) \ldots (x_n) = (\ldots((f(x_1))(x_2))\ldots)(x_n)$

1.3.7 Rules of precedence

Parentheses may be omitted according to the following rules of precedence:

Highest precedence:
(binds most strongly)

$\circ$

+, - prefix

$*$

+, - infix

$<, \leq, =, \neq, \geq, >, \subset, \subseteq, \supseteq, \supset, \in, \notin$

$\neg$

$\&$

$\vee$

$\equiv, \not\equiv$

$\supset$

Lowest precedence:
(binds least strongly)

2. OBJECTS

2.1 The General Class of Objects

In the following the general class of objects is introduced in such a way that both the expressions of languages and the states of an important class of abstract machines can be identified with subclasses of objects. In particular, the following definition provides a convenient way of decomposing and manipulating those objects.

An object will in general be composed of components which are themselves objects of similar nature. For convenience the components of an object are uniquely named so that using the names one can refer to the components. The names used to name components of objects are called selectors and it is assumed that there is a countably infinite set S of symbols defined for that purpose. In the following, the symbols s, s_1, s_2,... stand for selectors, the symbols A, A_1, A_2,... stand for arbitrary objects.

For selecting components of given objects an operation is introduced which for a given selector s and a given object A yields the component of A whose name in the formation of A is s if such a component exists at all.

By analogy with functional application the operation is represented by:

$$s(A)$$

and reads "s applied to A". The application of s to A is said to yield the s-component of A.

Because objects will be identified with linguistic expressions, i.e. finite constructs, and with the states of abstract machines whose states are finite constructs, the only interesting class of objects are those having a finite number of components and a finite depth of nesting. The latter means that the successive application of selectors to a given object will, after a finite number of steps, result in an object which does not have any further components, i.e. no selector whatsoever can be meaningfully applied to it. Objects of this kind are called elementary objects and it is assumed that there is a set of symbols EO to represent elementary objects. The symbols eo, eo_1, eo_2,...,eo'_1, eo'_2,... will in the sequel stand for elementary objects. Objects which are not elementary are called composite objects.

A named object is a pair <s:A> where s is a selector and A is an object.

A composite object can now be uniquely described by a finite (and for the moment non empty) set of named objects, either composite or elementary, where the names must be mutually different, i.e. a composite object is described by the set:

$$\{<s_1:A_1>, <s_2:A_2>, \ldots, <s_n:A_n>\}$$

where $n \geq 1$ and $s_i \neq s_j$ for $i \neq j$.

The objects A_i, $1 \leq i \leq n$ are called the immediate components of the object described.

A special form of expression has been chosen to represent composite objects. An object described by the above set of named objects may be represented by:

$$\mu_o(<s_1:A_1>, <s_2:A_2>, \ldots, <s_n:A_n>)$$ [1)]

There is, in general, more than one representation for the same object. In particular the order in which the named components are listed is irrelevant since the set consisting of them describes the object uniquely. One may think of μ_o to be the operator which assembles the named objects listed into a new and composite object.

Example:

Consider the following special assumption about S and EO :

$$S = \{\text{op-code, flag, tag, addr, } \ldots\}$$
$$EO = \{\text{CLA, STO, ADD, } \ldots$$
$$1,2,3, \ldots\}$$

The following object A may then be identified with the corresponding IBM 7040 instruction, namely CLA referring to index register 2, flag 1 and address 350:

$$A = \mu_o(<\text{op-code:CLA}>, <\text{tag:2}>, <\text{flag:1}>, <\text{addr:350}>)$$

1) For the moment it is required that the named objects, listed in the argument list of μ_o have mutually different names. In a succeeding chapter an extension of the meaning of μ_o will be given which among other things drops the restriction, and as a consequence the order of items in the argument list will be relevant.

Application of the respective selectors would yield:

op-code(A) = CLA
tag(A) = 2
flag(A) = 1
addr(A) = 350

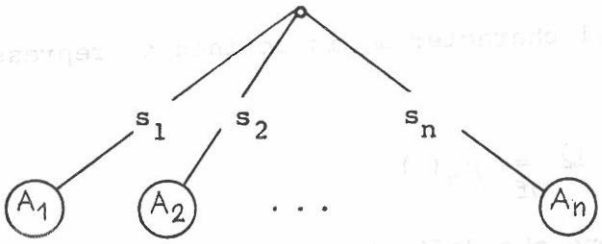
Note that the instruction can be nicely decomposed and that specification of separators between the components and any specific order of the components has been avoided.

2.2 Graphic Representation of Objects

It is sometimes convenient to think of objects as being represented by trees, namely trees with named branches, a finite number of terminal nodes and elementary objects attached to the terminal nodes. In the following a correspondence between the linear representation introduced in the previous section and trees is defined.

(1) An elementary object $eo \in EO$ is represented by the degenerate tree:

(2) A composite object described by $\{<s_1:A_1>, <s_2:A_2>, \ldots, <s_n:A_n>\}$ is represented by :



where for (A1) (A2) ... (An) the respective tree representations have to be inserted.

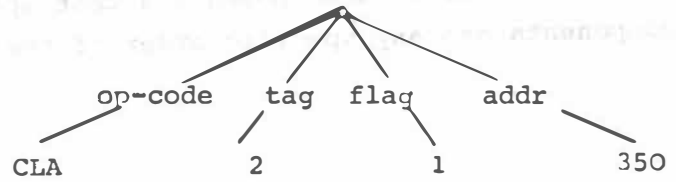
The order in which the branches appear in the tree representation is irrelevant.

Example:

The previous example of the 7040 instruction:

$$\mu_o(\langle op\text{-}code:CLA\rangle, \langle tag:2\rangle, \langle flag:1\rangle, \langle addr:350\rangle)$$

in the tree representation reads:



2.3 The Null Object

So far, the application of a selector to an object is only meaningful if the object has a component whose name is the selector, i.e. $s(\mu_o(\langle s_1:A_1\rangle, \langle s_2:A_2\rangle, \ldots, \langle s_n:A_n\rangle))$ was only meaningful if $s = s_i$ for some i, $1 \leq i \leq n$. For convenience, the definition of the application of selectors to objects will be extended so that any selector $s \in S$ can be meaningfully applied to any object of the general class. For that purpose a special composite object is introduced, namely the null object. The null object is the composite object that has no components, i.e. is described by the null set $\{\}$. Consistent with the rule in 2.1, the null object is represented by:

$$\mu_o( )$$

For convenience the special character Ω is defined to represent the null object:

$$\Omega \underset{Df}{=} \mu_o( )$$

It is now possible to extend the definition of the application of selectors by adding that the application of a selector s to an object which does not contain a component whose name is s yields the null object Ω.

This means in particular:

(a) The application of any selector to an elementary object yields the null object:

$$s(eo) = \Omega$$

(b) For composite objects the application of a selector is defined by:

$$s(\mu_o(\langle s_1:A_1\rangle,\langle s_2:A_2\rangle\ldots\langle s_n:An\rangle)) = \begin{cases} A_i \text{ if there is an } i \text{ such that } s = s_i, \\ \qquad\qquad 1\le i\le n \\ \Omega \text{ otherwise} \end{cases}$$

This means in particular for n = 0:

$$s(\Omega) = \Omega$$

Named objects of the form $\langle s:\Omega\rangle$ are permitted in the argument list of μ_o, however (consistent with rule (b) above) the following identity holds:

$$\mu_o(\langle s_1:A_1\rangle,\langle s_2:A_2\rangle,\ldots,\langle s_n:A_n\rangle,\langle s:\Omega\rangle) = \mu_o(\langle s_1:A_1\rangle,\langle s_2:A_2\rangle,\ldots,\langle s_n:A_n\rangle)$$

We can think of a named object $\langle s:\Omega\rangle$ as being a unit element with respect to the operation μ_o.

The definition of the set describing a composite object needs to be reformulated. A composite object can be uniquely described by a finite set of named non null objects whose names must be mutually different, i.e.:

$$\{\langle s_1:A_1\rangle,\langle s_2:A_2\rangle\ldots\langle s_n:A_n\rangle\}$$

where $n\ge 0$, $s_i \neq s_j$ for $i \neq j$ and $A_i \neq \Omega$ for $1\le i\le n$.

The following proposition about the identity relation between composite objects holds:

$$(\forall s)(s(A_1) = s(A_2)) \equiv A_1 = A_2$$

where A_1, A_2 are composite objects.

2.4 Composite Selectors

By analogy with the operation of functional composition, an operation for selectors is introduced by:

$$s_1\circ s_2\circ\ldots\circ s_n(A) \underset{Df}{=} s_1(s_2(\ldots(s_n(A))\ldots))$$

$s_1\circ s_2\circ\ldots\circ s_n$ is called a composite selector.

In the following $\chi, \chi_1, \chi_2, \ldots, \chi_1', \chi_2', \ldots$ stand for composite selectors.

The identity function I is introduced as the unit element with respect to composition, i.e.:

$$I(A) = A$$
$$I \circ \chi = \chi \circ I = \chi$$

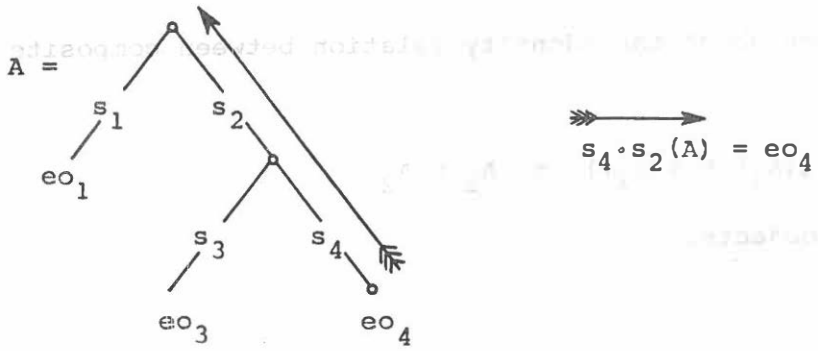
The set of all composite selectors will be called S^*. The introduction of I as a composite selector has a number of advantages. For example, the equivalence stated at the end of chapter 2.3 and reformulated for composite selectors now holds for any object whether it is elementary or composite:

$$(\forall \chi)(\chi(A_1) = \chi(A_2)) \equiv A_1 = A_2$$

Furthermore, as will be shown in the following section, the inclusion of the identity function I makes a fairly elegant treatment of the elementary objects possible.

Note that the direction bottom to top in the tree corresponds to the direction left to right in the composition of selectors.

Example:



2.5 The Characteristic Set of an Object

In section 2.1 a composite object was described by the set of named immediate components which are non null. There was a restriction, namely that the names of different components must be different. An alternative unique description of objects is introduced in the sequel using composite selectors which will prove convenient in many respects.

The characteristic set of an object A is the set of all pairs $<\varkappa:eo>$ such that $\varkappa$ (A) = eo, where $\varkappa$ is a composite selector and eo is an elementary object.

The characteristic set associated with an object determines uniquely the object. Thus one may describe objects by specifying their respective characteristic sets. Each object of the general class has a characteristic set. In particular is $\{\}$ the characteristic set of Ω and $\{<I : eo>\}$ the characteristic set of an elementary object eo.

An example using the tree representation may help the mental operation that has to be performed at this point to be grasped.Consider the object $\mu_o(<s_1:eo_1>,<s_2:\mu_o(<s_3:eo_2>,<s_4:eo_3>)>)$ where eo_1, eo_2, and $eo_3 \in EO$.

The tree representation of this object is:

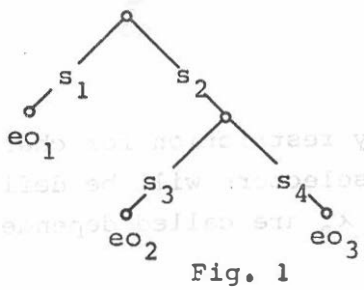


Fig. 1

One now takes this tree apart in the following way:

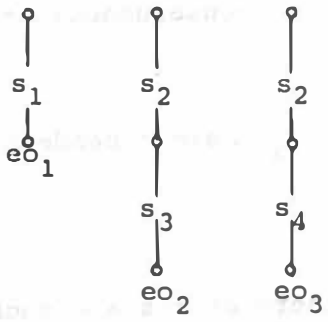
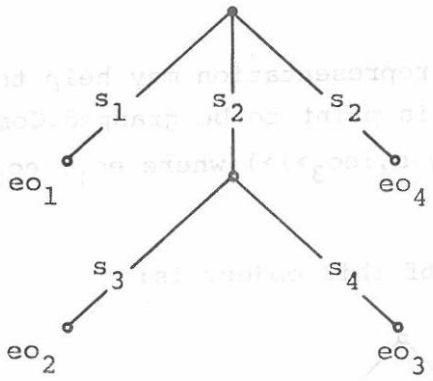


Fig. 2

In other words, one searches for all paths in the trees which lead to an elementary object, i.e. to a terminal node. These paths together with the associated elementary object obviously give the same information as the original tree. In terms of sets, the object that was previously described by the set $\{<s_1:eo_1>, <s_2:A>\}$ where A is described by $\{<s_3:eo_2>,<s_4:eo_3>\}$ is now described by its characteristic set $\{<s_1:eo_1>,<s_3 \circ s_2:eo_2>,<s_4 \circ s_2:eo_3>\}$.

Not every set of pairs where each pair consists of a composite selector and an elementary object is the characteristic set of some object. Consider an attempt to describe an object by its characteristic set $\{\langle s_1:eo_1\rangle, \langle s_3 \circ s_2:eo_2\rangle \langle s_4 \circ s_2:eo_3\rangle, \langle s_2:eo_4\rangle\}$. Going back to the previous description method namely $\{\langle s_1:eo_1\rangle, \langle s_2:A\rangle, \langle s_2:eo_4\rangle\}$, where A is described by $\{\langle s_3:eo_2\rangle, \langle s_4:eo_3\rangle\}$, one may observe that two different components have the same name and thus a rule for the construction of objects is violated. The corresponding tree representation shows the ambiguity more clearly:



Before stating the necessary restriction for characteristic sets, a dependence relation between composite selectors will be defined by means of a predicate dep. Two composite selectors χ_1, χ_2 are called dependent, if one of them is a tail of the other, i.e.:

$$dep(\chi_1,\chi_2) \underset{df}{\equiv} (\exists \chi)(\chi_1 = \chi \circ \chi_2 \vee \chi_2 = \chi \circ \chi_1)$$

For use in some subsequent proofs two consequences of the definition are given.

If χ_1, χ_2 are dependent, then $\chi_1 \circ \tau, \chi_2 \circ \tau$ are dependent, and vice versa:

(*) $dep(\chi_1, \chi_2) \equiv dep(\chi_1 \circ \tau, \chi_2 \circ \tau)$

Proof: Assume $dep(\chi_1,\chi_2)$. Then either there exists a χ such that $\chi_1 = \chi \circ \chi_2$, or there exists a χ such that $\chi_2 = \chi \circ \chi_1$. It follows that either $\chi_1 \circ \tau = \chi \circ \chi_2 \circ \tau$ or $\chi_2 \circ \tau = \chi \circ \chi_1 \circ \tau$, i.e. $dep(\chi_1 \circ \tau, \chi_2 \circ \tau)$.
The proof in the other direction is analogous.

If $\tau \circ \chi_1, \chi_2$ are dependent, then χ_1, χ_2 are dependent:

(**) $dep(\tau \circ \chi_1, \chi_2) \supset dep(\chi_1, \chi_2)$

Proof: Assume $dep(\tau\circ\chi_1,\chi_2)$, i.e. either there exists a χ such that $\tau\circ\chi_1 = \chi\circ\chi_2$, or there exists a χ such that $\chi_2 = \chi\circ\tau\circ\chi_1$.

(1) $\tau\circ\chi_1 = \chi\circ\chi_2$:
Either χ_1 is a tail of χ_2, or χ_2 is a tail of χ_1, i.e. $dep(\chi_1,\chi_2)$.

(2) $\chi_2 = \chi\circ\tau\circ\chi_1$:
χ_1 is a tail of χ_2, i.e. again $dep(\chi_1,\chi_2)$.

Let C be an arbitrary set of pairs of the form $\langle\chi:eo\rangle$. C is the characteristic set of some object, if and only if the selectors occurring as the first elements of the pairs of C are independent of each other. This condition is called the characteristic condition for C and may be formulated as follows:

$$\langle\chi_1:eo_1\rangle,\langle\chi_2:eo_2\rangle \in C \;\&\; \langle\chi_1:eo_1\rangle \neq \langle\chi_2:eo_2\rangle \supset \neg dep(\chi_1,\chi_2)$$

Obviously, $dep(I,\chi)$ holds for any composite selector χ, since $\chi = \chi\circ I$ for all χ. Therefore, only the form $\{\langle I:eo\rangle\}$ is a possible characteristic set using I. The advantage of using characteristic sets to describe objects is that the difference in treatment of elementary and composite objects has disappeared, which means that the case distinctions between elementary and composite objects will similarly disappear in certain places.

The result of the application of a composite selector χ to an object A may now be specified by its characteristic set $C_{\chi(A)}$:

$$C_{\chi(A)} = \{\langle\tau:eo\rangle \mid \langle\tau\circ\chi:eo\rangle \in C_A\}$$

The characteristic condition for $C_{\chi(A)}$ is always fulfilled.

Proof: Assume $\langle\tau_1:eo_1\rangle,\langle\tau_2:eo_2\rangle \in C_{\chi(A)}$ and $\langle\tau_1:eo_1\rangle \neq \langle\tau_2:eo_2\rangle$. Then $\langle\tau_1\circ\chi:eo_1\rangle,\langle\tau_2\circ\chi:eo_2\rangle \in C_A$, and therefore $\neg dep(\tau_1\circ\chi,\tau_2\circ\chi)$. Because of (*) it follows that $\neg dep(\tau_1,\tau_2)$, q.e.d.

2.6 The Operator μ

A rather powerful operation is introduced as the next step of the development. It is a two place operation and the operands are an object A and a pair $<\chi:B>$ where χ is a composite selector and B is an object; the result of the operation is again an object, namely A, where the component to which χ points is replaced by B. The application of the operation is written as follows:

$$\mu(A;<\chi:B>)$$

There are two important special cases to be mentioned. First, if there is no χ component of A, i.e. if $\chi(A) = \Omega$, then the result of the operation is simply A augmented by B which becomes the χ component of the result. Second, if $B = \Omega$ then the result of the operation is just A where the χ component has been deleted.

So far only the intuitive idea of what the operation is to accomplish has been given. The operation will be made precise by specifying the characteristic set of the result for any given A and $<\chi:B>$.

Let C_A, C_B be the characteristic sets of A and B, respectively. The characteristic set $C_{\mu(A;<\chi:B>)}$ of the result will be specified in terms of C_A, C_B and χ as the union of two sets:

$$C_{\mu(A;<\chi:B>)} = \{<\tau:eo> \mid <\tau:eo> \in C_A \;\&\; \neg dep(\chi,\tau)\} \cup \{<\tau\circ\chi:eo> \mid <\tau:eo> \in C_B\}$$

The first set, in the following abbreviated by $C_{A'}$, is the characteristic set of an object which is A with the χ component deleted. The second set, in the following abbreviated by $C_{B'}$, is the characteristic set of an object which has B as χ component and no other components.

The operation μ yields an object for any arbitrarily chosen A and $<\chi:B>$, i.e. does not lead outside the general class of objects, because the characteristic condition for $C_{\mu(A;<\chi:B>)}$ is always fulfilled.

Proof: Assume $<\tau_1:eo_1>, <\tau_2:eo_2> \in C_{\mu(A;<\chi:B>)}$ and $<\tau_1:eo_1> \neq <\tau_2:eo_2>$.
$\neg dep(\tau_1,\tau_2)$ must be shown.

(1) $<\tau_1:eo_1>, <\tau_2:eo_2> \in C_{A'}$:
Since $C_{A'} \subseteq C_A$, $\neg dep(\tau_1,\tau_2)$ follows from the characteristic condition for C_A.

(2) $\langle \tau_1 : eo_1 \rangle, \langle \tau_2 : eo_2 \rangle \in C_B$':

τ_1, τ_2 must have the form $\tau_1' \circ \chi$, $\tau_2' \circ \chi$ respectively, where $\langle \tau_1' : eo_1 \rangle, \langle \tau_2' : eo_2 \rangle \in C_B$. Therefore $\neg dep(\tau_1', \tau_2')$, and because of (*) it follows $\neg dep(\tau_1' \circ \chi, \tau_2' \circ \chi)$, i.e. $\neg dep(\tau_1, \tau_2)$.

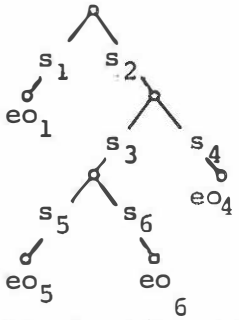
(3) $\langle \tau_1 : eo_1 \rangle \in C_A$', $\langle \tau_2 : eo_2 \rangle \in C_B$':

In this case $\neg dep(\chi, \tau_1)$ holds for τ_1, and τ_2 has the form $\tau_2' \circ \chi$ with $\langle \tau_2' : eo_2 \rangle \in C_B$. From $\neg dep(\chi, \tau_1)$ and (**) it follows that $\neg dep(\tau_2' \circ \chi, \tau_1)$, i.e. again $\neg dep(\tau_1, \tau_2)$.

The following examples illustrate the consequences of the definition.

Let B be a non null object and

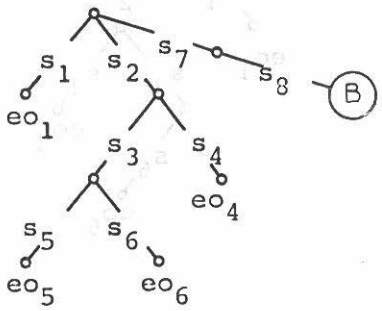
A =



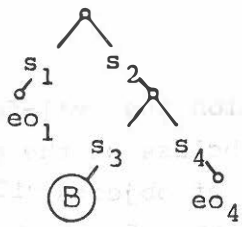
The result of $\mu(A, \langle \chi : B \rangle)$ for various choices of χ is shown in the following:

(1) $\mu(A; \langle s_8 \circ s_7 : B \rangle)$ =

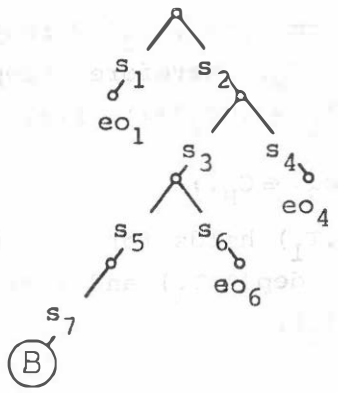
$s_7 \neq s_2$, $s_7 \neq s_1$



(2) $\mu(A; \langle s_3 \circ s_2 : B \rangle)$ =



(3) $\mu(A;\langle s_7 \circ s_5 \circ s_3 \circ s_2 : B\rangle)$ =

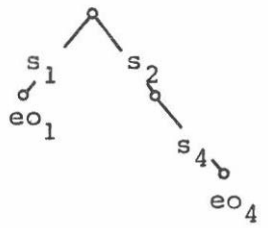


Now, the above three cases will be repeated, but with $B = \Omega$:

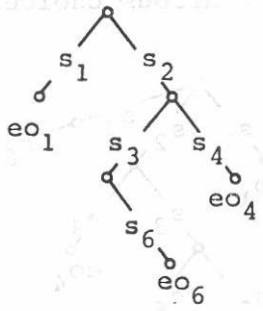
(4) $\mu(A;\langle s_8 \circ s_7 : \Omega\rangle) = A$

$s_7 \neq s_1$, $s_7 \neq s_2$

(5) $\mu(A;\langle s_3 \circ s_2 : \Omega\rangle)$ =



(6) $\mu(A;\langle s_7 \circ s_5 \circ s_3 \circ s_2 : \Omega\rangle)$ =



2.7. Definition of Classes of Objects

As already stated in the introduction the well-formed sentences of specific languages will be identified with a subclass of the general class of objects. The specification of particular classes of objects will be given either in terms of predicate logic or by equivalent devices of set theory. Appropriate notational conventions and abbreviations will be introduced for certain forms of specifications which have turned out to occur quite frequently.

The specification of a certain subclass of objects, to be identified with the well-formed sentences of a language, is called the abstract syntax of that

language according to the use of the term in current literature (e.g. McCarthy /9/). Although the entire apparatus of predicate logic could be used for abstract syntax specifications only a certain number of forms are actually important, i.e. it is sufficient for the purpose to consider only a certain type of syntax specifications.

In order to define subclasses of objects, predicates will be defined which are true exactly for the members of the subclass to be defined. For the present exposition, the names P, P_1, P_2,... will stand for arbitrary predicates and $\hat{P}$, $\hat{P}_1$, $\hat{P}_2$, ... will stand for the subclass of objects determined by the respective predicate.

In particular, the following basic definition schemata are used for the specification of abstract syntax.

(1) There are predicates P which are true for certain subclasses of elementary objects, i.e. $\hat{P} \subset EO$. How these predicates are defined depends on how the set of elementary objects EO is specified.

(2) Given predicates P_1, P_2, ...,P_n a new predicate P may be defined by the disjunction of the given predicates:

$$P \underset{Df}{=} P_1 \vee P_2 \vee \ldots \vee P_n \quad ^{1)}$$

In terms of sets the equation reads:

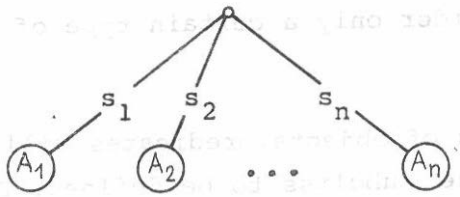
$$\hat{P} \underset{Df}{=} \hat{P}_1 \cup \hat{P}_2 \cup \ldots \cup \hat{P}_n$$

(3) Given n predicates P_1, P_2, ...,P_n and n mutually different selectors s_1, s_2, ...,s_n, $s_i \neq s_j$ for $i \neq j$, a new predicate P may be defined by the following equation:

$$P(A) \underset{Df}{=} (\exists A_1, A_2, \ldots, A_n)(A = \mu_o(\langle s_1 : A_1 \rangle, \langle s_2 : A_2 \rangle, \ldots, \langle s_n : A_n \rangle) \;\&\; \underset{i=1}{\overset{n}{Et}} P_i(A_i))$$

1) The meaning of the logical operator 'v' is extended to apply not only to propositions but also to predicates, i.e. the definition $P(x) \underset{Df}{=} P_1(x) \vee P_2(x) \vee \ldots \vee P_n(x)$ is abbreviated to : $P \underset{Df}{=} P_1 \vee P_2 \vee \ldots \vee P_n$.

The tree representation of objects may help to understand the above equation. The predicate P defined by the equation is true for objects of the following form:



where $A_1, A_2, \ldots, A_n$ are restricted to certain classes, namely: $A_1 \in \hat{P}_1, A_2 \in \hat{P}_2, \ldots, A_n \in \hat{P}_n$.

The above form of definition is rather bulky for actual use, and therefore a special notation for exactly this form is introduced:

$$P = (\langle s_1 : P_1 \rangle, \langle s_2 : P_2 \rangle, \ldots, \langle s_n : P_n \rangle)$$

A set of rules of the above forms, i.e. a set of predicates defined by equations of the forms (1), (2) or (3) may be considered either as a set of rules to produce objects of certain types or alternatively as a set of rules for analyzing given objects of a certain type. It should be mentioned that by the set of rules some predicates may be defined recursively.

In particular, rules of form (3) may be formulated as production rules using the obvious implication:

$$P_1(A_1) \;\&\; P_2(A_2) \;\&\; \ldots \;\&\; P_n(A_n) \supset P(\mu_o(\langle s_1 : A_1 \rangle, \langle s_2 : A_2 \rangle, \ldots, \langle s_n : A_n \rangle))$$

Given objects A_1 such that $P_1(A_1)$ and A_2 such that $P_2(A_2)$ .... and A_n such that $P_n(A_n)$, then
$\mu_o(\langle s_1 : A_1 \rangle, \langle s_2 : A_2 \rangle, \ldots, \langle s_n : A_n \rangle)$ is an object of type P, i.e.
$P(\mu_o(\langle s_1 : A_1 \rangle, \langle s_2 : A_2 \rangle, \ldots, \langle s_n : A_n \rangle))$.

Conversely, if an object of type P is given and P is defined by a rule of form (3), one knows how to analyse the object. In particular, the rule says that the given object must have an s_1 part of type P_1 and an s_2 part of type P_2 ... ... and an s_n part of type P_n. In other words, for analyzing a given object of type P the important implication is:

$$P(A) \supset P_1(s_1(A)) \;\&\; P_2(s_2(A)) \;\&\; \ldots \;\&\; P_n(s_n(A))$$

The classes of objects which can be defined by means of form (1), (2) or (3) have the following property in common. For any such class there is a number N called bound such that no member of the class has more than N immediate components, i.e. the number of components is bounded. It is easy to see that this property holds if one makes the case distinctions according to the three admissible forms:

(1): There are only predicates for elementary objects (which do not have any components) it is therefore sufficient to set N = 0.

(2): $P = P_1 \vee P_2 \vee \ldots \vee P_n$
Under the assumption that the bounds for $P_1, P_2 \ldots, P_n$ are $N_1, N_2, \ldots, N_n$ the bound N for P may be set to
N = maximum of $N_1, N_2, \ldots, N_n$.

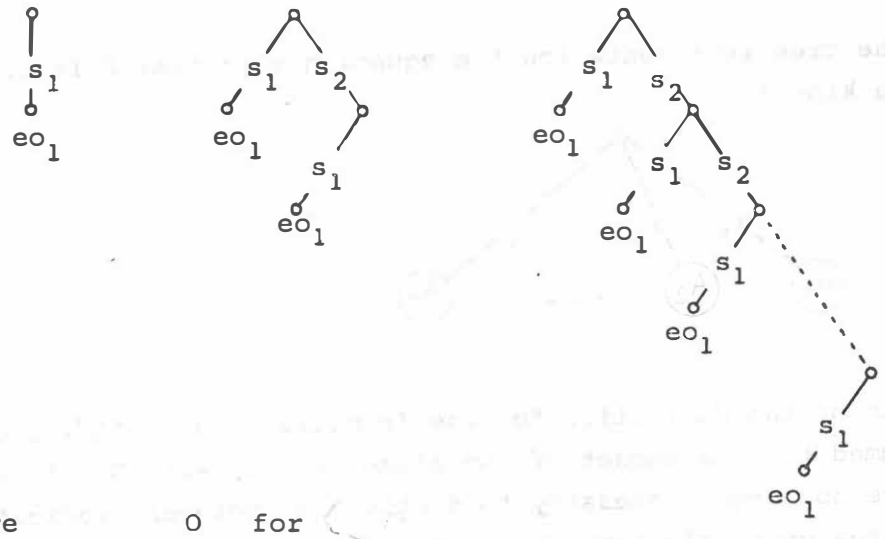
(3): $P = (\langle s_1 : P_1 \rangle, \langle s_2 : P_2 \rangle, \ldots, \langle s_n : P_n \rangle)$
It is obviously sufficient to set N = n.

One should however note that the number of elementary components (terminal nodes of the trees) for the members of a given class is not bounded in general.

Example:

$$\hat{P}_1 = \{eo_1\}, \quad P_2 = (), \quad \text{i.e. } \hat{P}_2 = \{\Omega\},$$
$$P_3 = (\langle s_1 : P_1 \rangle, \langle s_2 : P_3 \rangle) \vee P_2$$

the following are examples for members of P_3:



The bounds are 0 for P_1
0 for P_2
2 for P_3

The number of elementary components is obviously unbounded.

The means so far introduced for defining classes of objects would in principle be sufficient to cover the intended applications. However, the property of bounded number of immediate components for any definable class of objects is sometimes inconvenient, especially for the treatment of lists and arbitrary collections of objects of a certain type. The extensions to the defintion tools described in the following have been introduced to overcome this limitation.

In the previous definition schemata it was sufficient in any special case to talk about finite sets of selectors, i.e. where a set can be given by the list of its members (see definition schema(3)). In order to be able to define classes of objects with an unbounded number of subparts, it is necessary to talk about infinite sets of selectors. For that purpose predicates over selectors are introduced, and Q, Q_1, Q_2,... will stand for these predicates [1].

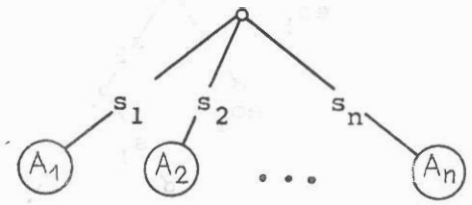
The set of definition schemata is augmented by the following forms:

(4) Given a predicate P_1 over objects and a predicate Q over selectors a new predicate P is defined which applies to all objects which can be built from selectors out of $\hat{Q}$ and objects of $\hat{P}_1$.

More precisely, the definition schema reads:

$$P(A) \underset{Df}{=} (\exists A_1, A_2, \ldots, A_n, s_1, s_2, \ldots, s_n)$$
$$\left(\overset{n}{\underset{i=1}{Et}} (Q(s_i) \;\&\; P_1(A_i)) \;\&\; A = \mu_o(\langle s_1 : A_1 \rangle, \langle s_2 : A_2 \rangle, \ldots, \langle s_n : A_n \rangle) \right)$$

In terms of the tree representation the equation says that P is true for objects of the kind :



1) In the actual use of the definition for the formalization of PL/I the set of selectors F is assumed to be a subset of the elementary objects EO. In this context it is therefore no longer necessary to distinguish between predicates over objects and predicates over selectors. Since this is an additional assumption, it seemed to be advisable to keep the distinction in this introduction.

for an arbitrary choice of n and the constraints:

$$s_i \in \hat{Q} \quad \text{for } 1 \le i \le n$$
$$A_i \in \hat{P}_1 \quad \text{for } 1 \le i \le n$$

A special notation for exactly the above form has been introduced, namely:

$$P = (\{ \langle s:P_1 \rangle \parallel Q(s) \})$$

Definition schemata (3) and (4) may also be used in combined form. The most general case would be:

$$P = (\langle s_1:P_1 \rangle, \ldots, \langle s_n:P_n \rangle, \{\langle s:P_{n+1} \rangle \parallel Q_1(s)\}, \ldots, \{\langle s:P_{n+m} \rangle \parallel Q_m(s)\})$$

with obvious meaning.

For some applications it will be necessary to define objects with an ordered set of immediate subparts, e.g. for the treatment of lists of elements of some kind. The simplest way to satisfy this requirement is to define an order for the selectors or for a subset of the selectors. It is neither necessary nor desirable to make specific assumptions about which symbols are members of $\hat{S}$, i.e. selectors. To avoid such an assumption a function is introduced mapping the natural numbers into the set of selectors. It is of course required that the mapping is one to one. More specifically, the function introduced is called "elem" and its application to an integer is represented by:

elem(i)

It yields a selector for $i \ge 1$ and has the property:

$$\text{elem}(i) \neq \text{elem}(j) \quad \text{for } i \neq j$$

It is now possible to introduce still another definition schema that allows classes of objects to be defined whose members have an arbitrary number of ordered immediate subparts. Such objects are called lists. A special elementary (!) object is introduced, called the null list. The null list will be denoted by:

< >

The following definition schema may now be introduced:

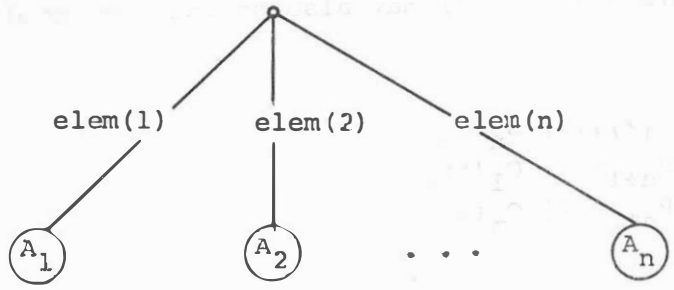
(5) Given a predicate P, a new predicate can be defined by appending the suffix "-list" to the predicate, i.e., the new predicate reads:

P-list

The suffix "-list" is defined by the following equation:

$$\text{P-list}(A) \underset{Df}{\equiv} A = \langle\rangle \vee (\exists A_1, \ldots, A_n)\left(\underset{i=1}{\overset{n}{Et}} P(A_i) \;\&\; \underset{i=1}{\overset{n}{Et}} (A_i \neq \Omega) \;\&\; n \geq 1 \;\&\; A = \mu_o(\langle elem(1):A_1\rangle, \ldots, \langle elem(n):A_n\rangle)\right)$$

In other words, the equation expresses that the predicate P-list holds for the empty list, $\langle\rangle$, and for objects of the kind:



for some $n \geq 1$ and the constraint:

$$P(A_i) \text{ and } A_i \neq \Omega \text{ for } 1 \geq i \geq n$$

2.8 Further Notational Conventions

2.8.1 Extension of the meaning of the μ-operator

The purpose of the extension of the μ-operator is to facilitate the replacement of several components of an object in either specified or unspecified order. One form of the extension will also allow the specification of the set of components to be replaced implicitly.

So far, only the form $\mu(A;\langle\chi:B\rangle)$ has been permitted. The following extensions are now defined:

(1) $\mu(A;\langle\chi_1:B_1\rangle, \langle\chi_2:B_2\rangle, \ldots,\langle\chi_n:B_n\rangle)$

The above form is defined iteratively by the equation:

$$\mu(A;\langle\chi_1:B_1\rangle, \langle\chi_2:B_2\rangle, \ldots,\langle\chi_n:B_n\rangle) \underset{Df}{=} \mu(\mu(A;\langle\chi_1:B_1\rangle);\langle\chi_2:B_2\rangle,\ldots,\langle\chi_n:B_n\rangle)$$

for the case n = 0 the form is defined by:

$$\mu(A;) = A$$

(2) $\mu(A;\{\langle x:B\rangle \mid p(x,B)\})$

The second argument of the above form defines a finite set of pairs $\langle x:B\rangle$, namely the set of pairs for which a certain proposition $p(x,B)$ holds. This form is reducible to the form (1) in the following way: if the elements of the set of pairs are written in any linear order and used in the form (1), then the result is the result of the present form provided (!) that the order of pairs is not significant. If the order is significant, the result is undefined. The form yields for the empty set:

$$\mu(A;\{\}) = A$$

The second argument may also be the union of some implicit defined sets:

$$\mu(A;\{\langle x_1:B_1\rangle \mid p_1(x_1,B_1)\} \cup \ldots \cup \{\langle x_m:B_m\rangle \mid p_m(x_m,B_m)\})$$

(3) $\mu_o(\langle x_1:A_1\rangle, \langle x_2:A_2\rangle, \ldots, \langle x_n:A_n\rangle)$

The restriction for the above form, that no x_i is a tail of x_j for $i \neq j$ and $1 \leq i,j \leq n$, is dropped.

The meaning of μ_o may be redefined by the following equation:

$$\mu_o(\langle x_1:A_1\rangle, \langle x_2:A_2\rangle, \ldots, \langle x_n:A_n\rangle \overline{\overline{Df}}$$
$$\mu(\Omega; \langle x_1:A_1\rangle, \langle x_2:A_2\rangle, \ldots, \langle x_n:A_n\rangle)$$

(4) $\mu_o(\{\langle x:A\rangle \mid p(x,A)\})$

The above form is analogous to (3) and defined by:

$$\mu_o(\{\langle x:A\rangle \mid p(x,A)\}) \overline{\overline{Df}} \mu(\Omega;\{\langle x:A\rangle \mid p(x,A)\})$$

(5) $\delta(A;x_1, x_2, \ldots, x_n)$

The above form deletes the x_i components from A and is defined by the following equation:

$$\delta(A; x_1, x_2, \ldots, x_n) \overline{\overline{Df}} \mu(A; \langle x_1:\Omega\rangle, \langle x_2:\Omega\rangle, \ldots, \langle x_n:\Omega\rangle)$$

(6) $\delta(A;\{x \mid p(x)\})$

The above form is analogous to (5) and defined by:

$$\delta(A;\{x \mid p(x)\}) \overline{\overline{Df}} \mu(A;\{\langle x:\Omega\rangle \mid p(x)\})$$

2.8.2 Manipulation of lists

Lists are an important class of objects. To facilitate the manipulation of lists a number of functions, operators and abbreviations are introduced which correspond closely to the conventional means for the purpose.

First a predicate, is-list, is introduced which holds for any list whatsoever. The predicate may be defined by:

$$\text{is-list}(A) \underset{Df}{=} (\exists P)(P\text{-list}(A))$$

Furthermore, an abbreviation for denoting an element of a list is introduced. Let L be a list, then:

$$\text{elem}(i,L) \underset{Df}{=} \text{elem}(i)(L)$$

The length of a list is defined as the largest index of an element which is not the null object:

$$\begin{aligned}\text{length}(L) = \text{is-list}(L) \longrightarrow\ & (L = \langle\rangle \longrightarrow 0,\\ & T \longrightarrow (\iota i)(\text{elem}(i,L) \neq \Omega \ \&\ \text{elem}(i+1,L) = \Omega))\end{aligned}$$

The following three functions yield when applied to a list, the head (which is the first element of a list if it exists), the last element of a list (if it exists) and the tail of a list (which is the original list except the first element):

$$\text{head}(L) = \text{is-list}(L)\ \&\ (L \neq \langle\rangle) \longrightarrow \text{elem}(1,L)$$

$$\text{last}(L) = \text{is-list}(L)\ \&\ (L \neq \langle\rangle) \longrightarrow \text{elem}(\text{length}(L),L)$$

$$\begin{aligned}\text{tail}(L) = \text{is-list}(L)\ \&\ (L \neq \langle\rangle) \longrightarrow\ & (\text{length}(L) = 1 \longrightarrow \langle\rangle,\\ & \text{length}(L) > 1 \longrightarrow \mu_0(\{\langle \text{elem}(i) : \text{elem}(i+1,L)\rangle \mid\\ & \qquad 1 \leq i \leq (\text{length}(L)-1)\}))\end{aligned}$$

The concatenation of two lists is defined by:

$$\begin{aligned}L_1 \frown L_2 = \text{is-list}(L_1)\ \&\ \text{is-list}(L_2) \longrightarrow\ & \mu(L_1; \{\langle \text{elem}(\text{length}(L_1)+i) : \text{elem}(i,L_2)\rangle \mid\\ & \qquad 1 \leq i \leq \text{length}(L_2)\})\end{aligned}$$

Multiple concatenation is defined by:

$$\underset{i=1}{\overset{n}{\text{CONC}}} L_i = L_1 \frown L_2 \frown \ldots \frown L_n$$

As a convenient form to denote lists one may enumerate the elements within pointed brackets. The form is defined by:

$\langle A_1, A_2, \ldots, A_n \rangle \overline{\text{Df}} \mu_o(\langle \text{elem}(1):A_1 \rangle, \langle \text{elem}(2):A_2 \rangle, \ldots, \langle \text{elem}(n):A_n \rangle)$
for $n \geq 1$, $A_i \neq \Omega$ $(1 \leq i \leq n)$.

An alternative form is: $\underset{i=1}{\overset{n}{\text{LIST}}} A_i \overline{\text{Df}} \langle A_1, A_2, \ldots, A_n \rangle$

2.9 Implicit Definitions

Implicit definition means in the present context that for the definition of a function (and later on also for instruction schemata) a problem is stated rather than an algorithm specified, say by conditional expressions. The reason for definition by stating a problem is that this is sometimes much more intelligible than any special solution to the problem. However, in using implicit definitions one has carefully to ensure that a solution to the problem indeed exists, more precisely that an algorithm exists which solves the problem. In particular, all definitions which contain the ι operator or a set definition of the form $\{x \mid P(x)\}$ are implicit definitions.

The definition of the length of a list as given in section 2.8.2 may serve as an example. The use of the ι operator makes the definition implicit. The problem stated is to find the greatest index i such that the i^{th} element of a given list is not the null object. It is quite obvious that a solution and an algorithm for this problem exist, since a list is a finite sequence of non null objects according to the definition.

The following algorithm would indeed resolve the problem:

$$h(L,i) = (\text{elem}(i+1,L) = \Omega \longrightarrow i,\ T \longrightarrow h(L,i+1))$$
$$\text{length}(L) = (\text{is-list}(L) \longrightarrow h(L,0))$$

So this definition is not longer, it is not as transparent as the implicit definition. In more complicated cases the difference between the two definition methods would become much more apparent especially with regard to their lengths.

3. APPLICATION TO ABSTRACT SYNTAX AND REPRESENTATION OF LANGUAGES

Throughout the rest of this report possible applications of the tools provided by the previous chapters to languages will be suggested by defining the syntax and semantics of the simple programming language EPL described briefly in Section 1.2. The present chapter is concerned more specifically with problems of a syntactical nature and their solution by means of the methods of Chapter 2.

Two basic types of snytax, viz. abstract syntax and concrete syntax, may be considered to be constitutive of the syntactic definition of a language. An abstract syntax is one which only specifies the expressions of the language as to the structures significant for their subsequent interpretation and not as to how they are to be expressed for the purpose of communication either to oneself or others. A concrete syntax is one which specifies the expressions of the language as a set of character strings. Once the syntax of a language has been given, i.e., once it is possible to determine what categories of well-formed expressions a language is to have, one can ask questions as to the possible meanings to be assigned to these expressions. However, depending on the type of syntax taken to constitute the syntactical definition of the language, the solutions to the problems of meaning and representation will differ.

Assume, for the moment, that the syntax of a language has been specified by an abstract syntax. Then the language can easily be interpreted by attaching meaning directly to the expressions as specified by the abstract syntax. However, in such a case, it is of interest to ask not only what the meaning of expressions might be, but also how one might specify their possible concrete representations as character strings. This problem of representation is solved for the programs of EPL, as specified by the abstract syntax, by means of the replacement system given in section 3.2.2. The meaning of the programs of EPL, as specified by the abstract syntax, is defined by means of the interpreter of Chapter 5. The same method was used for giving a formal definition of ALGOL 60 in /20/ and /21/.

If, on the other hand, the syntax of a language had been specified by means of a concrete syntax, then the question of a concrete representation does not arise since the concrete syntax itself defines this representation. The question of the meaning of the expressions of the language still remains to be solved and one might be tempted to think that an interpretation might be given outright by attaching the meaning to the strings of characters producable by the syntax. In the case of the formal definition of the semantics of complex programming languages such direct interpretation has proved to be rather impractical and, hence, there has developed

a tendency to translate the expressions specified by the concrete syntax into some kind of abstract normal form before attempting to define their meaning. In such a case the set of expressions of the language in abstract normal form may be said to constitute an abstract syntax of the language. The problem of specifying the correlation between a given language whose syntax is considered to be constituted by a concrete syntax and the normal form of these expressions as specified by means of an abstract syntax is discussed in the closing sections of this chapter. In particular, the problem of the meaning of programs of EPL as specified by the concrete syntax is solved by first specifying a translator from these expressions to their corresponding abstract normal form as specified by the abstract syntax of EPL and then interpreting these latter expressions by means of the interpreter given in Chapter 5. The same method was used for giving a formal definition of PL/I in /4/ - /8/.

Section 3.3 serves to correlate the notion of abstract syntax, as defined by means of the devices of Chapter 2, with the Backus Normal Form which has been used to define the concrete syntax of a number of programming languages. In Section 3.4.1 the usual Backus Normal Form notation is extended by means of a definitional extension, i.e., certain convenient shorthands are introduced. This extended Backus notation is used in Section 3.4.2 to give the concrete syntax of EPL.

3.1 Abstract Syntax of EPL

One way of specifying an abstract syntax is by means of the methods presented in the previous chapter, i.e. by defining a class of objects. Hence, for the present purposes, an abstract syntax is defined to consist of a specification of the sets EO and S, a specification of predicates by means of the definition schemata described in Chapter 2.7, and a choice of one specific predicate whose corresponding class of objects is identified with the set of expressions of the language to be defined.

In specifying the abstract syntax of EPL the aim is to define a class of objects, which can in a useful way be identified with the programs of EPL (as described intuitively in 1.2), i.e., the objects must mirror the structure of the corresponding programs. Hence, there is essentially no arbitrariness in the choice of the structure of objects, but the choice of the specific selectors, predicate names and symbols for representing elementary objects is arbitrary and only governed by mnemonic considerations. Let the set of objects representing the set of programs that can be formulated in EPL be denoted by $\hat{is\text{-}progr}$.

As a notational convention all predicates have a prefix "is-" and all selectors have a prefix "s-" except possibly identifiers, which are also used as selectors, because their structure has been left completely unspecified. The predicate definitions are labelled (A1), (A2), ... for reference purposes.

is-$\hat{}$id	an infinite set of identifiers;
is-$\hat{}$log	a set of constants denoting the truth values;
is-$\hat{}$int	an infinite set of constants denoting the integer values;
is-un$\hat{}$ary-rt	a set of unary (one-place) operators;
is-bi$\hat{}$nary-rt	a set of binary (two-place) operators;
{INT,LOG}	two attributes used to distinguish integer variables from logical variables.

These sets are assumed to be mutually exclusive. The set EO consists of the union of the sets enumerated above, thus we can also write

EO = is-$\hat{}$id $\cup$ is-$\hat{}$log $\cup$ is-$\hat{}$intg $\cup$ is-un$\hat{}$ary-rt $\cup$ is-bi$\hat{}$nary-rt $\cup$ {INT,LOG}

The set of selectors S necessary for the specification of the abstract syntax of EPL is infinite since the set of identifiers, the members of which are also selectors, is itself infinite. However, the set of selectors that are not identifiers is finite and actually quite small and may, therefore, be enumerated explicitly. Hence, we can define the set S as follows:

S = { s-decl-part, s-st-list, s-param-list, s-st, s-expr, s-left-part,
s-right-part, s-id, s-arg-list, s-op, s-rd, s-rd1, s-rd2,
s-then-st, s-else-st} $\cup$ is-$\hat{}$id.

Note that the set of identifiers is-$\hat{}$id belongs to both EO and S, i.e. the sets EO and S have a common part.

The predicates necessary for the specification of the abstract syntax of the example language are defined using Schemata (2), (3), (4) and (5) from Chapter 2:

(A 1) is-progr = is-block
(A 2) is-block = (<s-decl-part:is-decl-part>,
<s-st-list:is-st-list>)
(A 3) is-decl-part = ({<id:is-attr> || is-id(id)})
(A 4) is-attr = is-var-attr $\vee$ is-proc-attr $\vee$ is-funct-attr
(A 5) is-v$\hat{}$ar-attr = { INT, LOG }

(A 6) is-proc-attr = (<s-param-list:is-id-list>,
<s-st:is-st>)
(A 7) is-funct-attr = (<s-param-list:is-id-list>,
<s-st:is-st>,
<s-expr:is-expr>)
(A 8) is-st = is-assign-st ∨ is-cond-st ∨ is-proc-call ∨ is-block
(A 9) is-assign-st = (<s-left-part:is-var>,
<s-right-part:is-expr>)
(A10) is-expr = is-const ∨ is-var ∨ is-funct-des ∨ is-bin ∨ is-unary
(A11) is-const = is-log ∨ is-int
(A12) is-var = is-id
(A13) is-funct-des = (<s-id:is-id>,<s-arg-list:is-id-list>) [1]
(A14) is-bin = (<s-rd1:is-expr>,<s-rd2:is-expr>,<s-op:is-bin-rt>)
(A15) is-unary = (<s-rd:is-expr>,<s-op:is-unary-rt>)
(A16) is-cond-st = (<s-expr:is-expr>,<s-then-st:is-st>,<s-else-st:is-st>)
(A17) is-proc-call = (<s-id:is-id>,<s-arg-list:is-id-list>) [1] [2]

The programs of EPL are now identified with the members of $\hat{\text{is-progr}}$.

One may observe that in defining the abstract syntax of programs of EPL no specification was necessary as to the order of components or of special punctuation marks.

Some examples of components of members of the class $\hat{\text{is-progr}}$ in tree representation are given below. These components are variables, unary and binary expressions. Their respective tree representations will be accompanied by their usual character string representations using parentheses. For the purpose of these examples let MINUS be a unary operator and MULT and ADD be binary operators.

Key to abbreviations (prefixes and suffixes omitted):

attr	attribute	param	parameter
arg	argument	proc	procedure
bin	binary(two places)	progr	program
cond	conditional	rd	operand
decl-part	declaration part	rt	operator
expr	expression	st	statement
funct	function	var	variable
id	identifier		

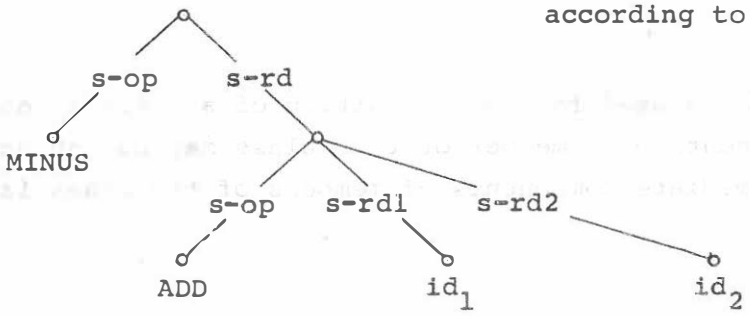
[1] includes single identifiers and hence procedure names and function names.

[2] completely identical to is-funct-des.

Examples

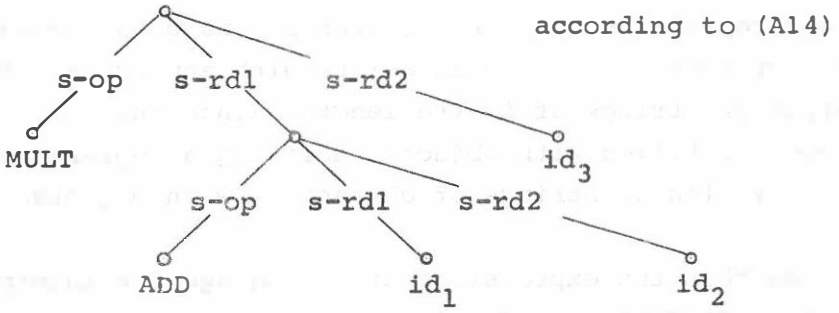
(a) id according to (A10) and (A12).

(b) according to (A14) and (A15)



Usual notation: -(a + b)

according to (A14)



Usual notation: (a + b) . c

It is of importance for later use of the abstract syntax that certain classes, e.g., the subclasses of the class of expressions, be mutually exclusive. As a consequence of this requirement one can ask whether a given expression is a constant, a variable, a binary or a unary expression, in any order without affecting the outcome of the inquiry; in addition, the above series of questions is exhaustive.

Definitions (A2),(A10), (A14) and (A16) are recursive[1], since in each case the definiens refers (indirectly) to the definiendum. On the other hand, these definitions contain alternatives which are not recursive so that the definitions are not circular. E.g., the non-recursive alternatives of (A10) are is-const, is-var and is-funct-des.

---

1) Recursive is used in the sense in which it is used in literature or programming and computers.
In formal logic, however, recursive is used synonymous with computable while the above property is called self-referencial, regressive or recurrent.

The number of immediate components of any element of a class of objects defined only by means of the definition schemata (1), (2) and (3) is of course bounded. For example, the bound for the number of immediate components of any element of is-$\hat{e}$xpr is 3. The number of terminals of the class is-$\hat{e}$xpr is, however, not bounded.

If definition schema (4) is used for the definition of a class of objects, the number of immediate components of a member of this class may be unbounded. For example, the number of immediate components of members of the class is-d$\hat{e}$cl-part is unbounded.

3.2 The Definition of Concrete Representations

To define a concrete representation for the expressions of a language specified by means of an abstract syntax means to associate with any expression of the language one or more character strings of finite length. Since the expressions of languages have so far been identified with objects, defining a representation means more specifically the association of strings of characters with objects.

It is usually assumed that the expressions of a language are interpreted in such a way that, in general, different expressions have different meanings.

A representation is therefore only useful if it is possible to decide whether a given character string is the representation of an expression and if so, the expression must be uniquely determined by its representation. A representation satisfying these conditions is called unambiguous.

In the first part of this section a representation system suitable for defining concrete representations of languages is presented.[1] The central part of the system is a set of conditional replacement schemata which permit the specification of replacement processes leading from abstract expressions of the language to their possible concrete representations.

In the concluding part of the section a concrete representation of EPL will be defined by means of the representation system.

---

1) Such a system, with a few generalizations, has been used in /21/.

3.2.1 The representation system

The first step in specifying the representation system for a language is to give an alphabet of terminal symbols. This alphabet can be conceived as the union of two sets. The elements of the first set represent uniquely the elementary objects of the abstract syntax of the language; this set will be, in general, infinite. The second is a finite set of so called delimiters which are used to reflect the structure of the objects of the language when they are represented as strings. In the present context terminal symbols will be considered as atomic characters.

Let the unique representation of the elementary objects eo by terminal symbols be defined by means of a function rep(eo) and the set of delimiters be denoted by $\mathcal{S}$. Then the set of terminal symbols $\mathcal{T}$ can be defined by:

$$\mathcal{T} = \{ \text{rep}(\text{eo}) \mid \text{eo} \in \text{EO} \} \cup \mathcal{S}$$

A terminal expression is an arbitrary meta-expression[1] denoting a terminal symbol depending on the free variables occurring in the expression. Terminal symbols are considered to be special cases of terminal expressions.

In order to formulate the replacement schemata of the representation system the notions of non-terminal and of non-terminal expression are introduced.

A non-terminal is composed of a non-terminal name and a (possibly empty) list of arguments which are objects. Non-terminals have the following form:

NONTERM[ob_1,...,ob_n]

i.e., non-terminal names are underlined capitalized words and the argument list is enclosed in a pair of brackets (except that the brackets are omitted when the argument list is empty).

A non-terminal expression is a meta-expression denoting a non-terminal depending on the free variables occurring in the expression and is written as follows:

NONTERM[$expr_1$,...,$expr_n$]

where $expr_1$,...,$expr_n$ are arbitrary meta-expressions denoting objects.

---

1) To distinguish the language in which the definitional tools are expressed from the object language to be interpreted, the prefix "meta" will be used for the former. Thus, the notational devices introduced in 1.2 belong to the meta-language.

For each non-terminal name NONTERM there will be one or more corresponding conditional replacement schema of the following general form:

$$p(x_1,\ldots,x_m,t_1,\ldots,t_n):\underline{\text{NONTERM}}[t_1,\ldots,t_n] \Longrightarrow \gamma(x_1,\ldots,x_m,t_1,\ldots,t_n)$$

$t_1,\ldots,t_n$ are the parameters and $x_1,\ldots,x_m$ are the auxiliary variables of the rule. p is a meta-expression, called the condition of the rule, and denotes a truth-value depending on the free variables $x_1,\ldots,x_m,t_1,\ldots,t_n$. γ denotes a string consisting of non-terminal expressions and terminal expressions in the free variables $x_1,\ldots,x_m,t_1,\ldots,t_n$.

A schema of the above type is to be understood in the following sense: For each assignment of specific objects $x_1^o,\ldots,x_m^o,t_1^o,\ldots,t_n^o$ as values of the variables $x_1,\ldots,x_m,t_1,\ldots,t_n$, if the condition p is satisfied, then the non-terminal $\underline{\text{NONTERM}}[t_1^o,\ldots,t_n^o]$ may be replaced at any of its occurrences in a given string by the string $\gamma(x_1^o,\ldots,x_m^o,t_1^o,\ldots,t_n^o)$.

In the special case of a schema not depending on a condition the form is:

$$\underline{\text{NONTERM}}[t_1,\ldots,t_n] \Longrightarrow \gamma(t_1,\ldots,t_n)$$

with obvious meaning.

The representation system is now defined as the quintuple

$$\langle \mathcal{A}, \mathcal{T}, \mathcal{N}, \underline{H}, \mathcal{R} \rangle$$

where $\mathcal{A}$ is the abstract syntax of the language to be represented, $\mathcal{T}$ is the set of symbols, $\mathcal{N}$ is the set of non-terminal names, $\underline{H}$ is the unique non-terminal name out of $\mathcal{N}$ called the head of the system and $\mathcal{R}$ is the set of conditional replacement schemata.

For each abstract expression t^o specified by the abstract syntax, at least one concrete representation of t^o can be found by means of a replacement process. A replacement process is a sequence of string $\gamma_o, \gamma_1,\ldots,\gamma_k$, where γ_o is the non-terminal $\underline{H}[t^o]$ and γ_{i+1} is obtained from γ_i (for $0\le i<k$) by the application of one of the conditional replacement schemata of $\mathcal{R}$. The last string γ_k of the sequence consists solely of terminal symbols and constitutes a concrete representation of t^o.

3.2.2 A concrete representation of programs of EPL

The method used in /21/ and described in the previous section of this chapter will now be demonstrated by means of its application to the abstract syntax of EPL given in 3.1.

The function rep which, when applied to elementary objects, e.g. identifiers, constants, and operators, yields their corresponding terminal symbols, is defined as follows:

rep(t) =

is-id(t) ⟶ rep-id(t)
is-const(t) ⟶ rep-const(t)
(is-bin-rt ∨ is-unary-rt)(t) ⟶ rep-rt(t)
t = INT ⟶ INTEGER
t = LOG ⟶ LOGICAL

Note: The functions rep-id, rep-const and rep-rt are one-to-one. The domain of rep-id is the set of identifiers $\hat{is}$-id and the range is some set of identifiers as specified by some concrete syntax or some implementation. The domain of rep-const is the union of the set $\hat{is}$-log and $\hat{is}$-int and its range is the union of the sets of logical and integer constants as specified by some concrete syntax or some implementation. The domain of rep-rt is the union of the sets $\hat{is}$-bin-rt and $\hat{is}$-unary-rt and its range is the union of the sets of binary and unary operators as specified by some concrete syntax.

The members of the quintuple constituting the replacement system required for the present example will now be defined without much comment so that the manner of application of the method will become apparent to the reader.

The set of non-terminal names $\mathcal{N}$ is defined by means of a tabular enumeration of its elements. Each member of the set will be accompanied by a suggested reading. The first non-terminal name is the head of the system, hence

H = PROGR

(N1)	PROGR	program
(N2)	BLOCK	block
(N3)	DECL	declaration part
(N4)	STL	statement list

cont'd

(N5) ST statement
(N6) EXPR expression
(N7) ID } identifier lists
(N8) ID'

The set $\mathcal{T}$ of terminal symbols is defined by means of the union of the set of delimiters and the range of the function rep.

$$\mathcal{T} = \{\text{BEGIN, END, PROCEDURE, FUNCTION, CALL, IF, THEN, ELSE, RETURNS,}\ (,),;,,,=\} \cup \{\text{rep}(t) \mid t \in EO\}$$

Since the terminal symbols are not underlined they are clearly distinguishable from the underlined non-terminal names when they are written juxtaposed next to each other. The only exceptions are the parentheses, semicolon, comma, and equality, and they are written especially heavy to indicate that they are terminal symbols standing for themselves.

The abstract syntax $\mathcal{A}$ consists of definitions (A1) to (A17) on page 3-7.

The schemas of the set of conditional replacement schemata $\mathcal{R}$ are listed next and labelled (R1), (R2),... for reference purposes.

(R1) PROGR[t] ⟹ BLOCK[t]

(R2) BLOCK[t] ⟹ BEGIN DECL[s-decl-pt(t)]STL[s-st-list(t)]END

(R3) t = Ω :DECL[t] ⟹ λ
is-var-attr∘id(t): DECL[t] ⟹ rep∘id(t) rep(id); DECL [♂(t;id)]
is-proc-attr∘id(t):DECL[t] ⟹ PROCEDURE rep(id) ID[s-param-list∘id(t)];
ST[s-st∘id(t)]; DECL[♂(t;id)]
is-funct-attr∘id(t):DECL[t] ⟹ FUNCTION rep(id) ID[s-param-list∘id(t)];
ST[s-st∘id(t)]
RETURNS EXPR[s-expr∘id(t)]; DECL[♂(t;id)]

(R4) length(t) = 0 : ID[t] ⟹ λ
length(t) > 0 : ID[t] ⟹ (ID'[t])

(R5) length(t) = 1 : ID'[t] ⟹ rep∘head(t)
length(t) > 1 : ID'[t] ⟹ rep∘head(t); ID'[tail(t)]

(R6) length(t) = 1 : STL[t] ⟹ ST[head(t)]
length(t) > 1 : STL[t] ⟹ ST[head(t)] ; STL[tail(t)]

(R7) is-assign-st(t): ST[t] ⟹ rep∘s-left-part(t) := EXPR[s-right-part(t)]
is-cond-st(t) : ST[t] ⟹ IF EXPR[s-expr(t)]THEN ST[s-then-st(t)]
ELSE[ST s-else-st(t)]
is-proc-call(t): ST[t] ⟹ CALL rep∘s-id(t) ID[s-arg-list(t)]
is-block(t) : ST[t] ⟹ BLOCK[t]

(R8) is-const(t) : EXPR[t] ⟹ rep(t)
is-var(t) : EXPR[t] ⟹ rep(t)
is-func-des(t): EXPR[t] ⟹ rep∘s-id(t) ID[s-arg-list(t)]
is-bin(t) : EXPR[t] ⟹ (EXPR[s-rd1(t)]rep∘s-op(t)[EXPR s-rd2(t)])
is-unary(t) : EXPR[t] ⟹ rep∘s-op(t) EXPR[s-rd(t)]

This completes the definition of the replacement system which in turn constitutes a definition of a set of possible representations of members of the set $\hat{\text{is-progr}}$, i.e. of the programs of EPL.

3.3 The Correlation of Abstract Syntax and Backus Normal Form

This chapter is intended for the reader who knows Backus Normal Form and therefore will get some insight into the properties of abstract syntax specifications by considering its relation to syntax specifications in Backus Normal Form, (called concrete syntax for short). The following formulation is used for Backus Normal From specifications. A syntax specification is given by an alphabet of terminal symbols $\mathfrak{T}$, a finite set of rules of a particular form specifying sets of strings over $\mathfrak{T}$ and one selected set of strings which is said to be the set of wellformed expressions of the language to be defined.

The following symbols will stand for:

$\alpha, \alpha_1, \alpha_2, \ldots$	arbitrary elements of the alphabet $\mathfrak{T}$;
$M, M_1, M_2, \ldots$	arbitrary names for sets of strings;
$w, w_1, w_2, \ldots$	arbitrary strings over $\mathfrak{T}$;
$w_1 \frown w_2 \ldots$	concatenation of w_1 and w_2 .

The rules of a syntax specification may assume the following particular forms:

(1) $M = \{\alpha\}$

(2) $M = M_1 \cup M_2 \cup \ldots \cup M_n$

(3) $M = M_1\, M_2 \ldots M_n$
where $M_1\, M_2 \ldots M_n \underset{Df}{=} \{ w_1 \frown w_2 \frown \ldots \frown w_n \mid w_1 \in M_1 \,\&\, w_2 \in M_2 \,\&\, \ldots \,\&\, w_n \in M_n \}$

There are two purposes served by a Backus Normal Form definition. The first purpose is to define a set of strings (well formed expressions) and the second to define a phrase structure for each of these strings. The correlation will be established by reinterpreting the definition schemata of an abstract syntax to serve the same purposes. The only addition to be made to our formalism is the definition of an order for the set S, i.e. an ordering relation $s_1 \leq s_2$ is defined for any pair of selectors s_1, s_2.

Having defined an order for selectors one can immediately associate an order for the composite selectors $\chi = s_n \circ s_{n-1} \circ \ldots \circ s_1$, namely the alphabetical order with s_1 as the most significant position and s_n as the least significant position.

Considering an object characterized by $\{<\chi_1 : e_1>, <\chi_2 : e_2>, \ldots, <\chi_n : e_n>\}$ and $\chi_1 < \chi_2 < \ldots < \chi_n$ the associated string of symbols is defined to be $e_1 \frown e_2 \frown \ldots \frown e_n$, and is called terminal string of the given object. By this device an abstract syntax defines on the one hand a set of strings (over the set of symbols EO) and on the other hand, by the very structure of each object, a structure for each of the terminal strings.

For each syntax specification in Backus Normal Form one can now specify a corresponding abstract syntax.

Let a concrete syntax be given by an alphabet A, a set of names for sets of strings and a set of rules.

A corresponding abstract syntax is then described by:

(a) $EO = T$;

(b) some ordered set of selectors S;

(c) each name M of a set in the concrete syntax is associated with a predicate P_M in the abstract syntax such that if $M_1 \neq M_2$ then $P_{M_1} \neq P_{M_2}$;

(d) rules of the form $M = \{\alpha\}$ are transformed to $\hat{P}_M = \{\alpha\}$;

(e) rules of the form $M = M_1 \cup M_2 \cup \ldots \cup M_n$ are transformed to $P_M = P_{M_1} \vee P_{M_2} \vee \ldots \vee P_{M_n}$;

(f) rules of the form $M = M_1\ M_2 \ldots M_n$ are transformed to $P_M = (<s_1 : P_{M_1}>, <s_2 : P_{M_2}>, \ldots, <s_n : P_{M_n}>)$ where s_1 is the first selector and s_1, s_2 ... s_n are immediate successors using the order given for s;

(g) if M is the head of the concrete syntax then P_M is the head of the abstract syntax.

The set of strings defined by the concrete syntax is identical with the set of terminal strings of the objects defined by a corresponding abstract syntax. Thus all sets of strings, which can be defined using Backus Normal Form can also be defined by using definition schemata (1), (2) and (3) of chapter 2.7. The

phrase structure defined by the concrete syntax will be identical to the phrase structure defined by the corresponding abstract syntax because of the one to one correspondence of the respective rules. The structure of the parsing tree according to the concrete syntax for a given string will, apart from redundant nodes, be the same as the structure of the respective object.

It is therefore possible to conclude that the concrete syntax is ambiguous if there are two different objects defined by the corresponding abstract syntax whose terminal strings are identical. However, the reverse conclusion is not in general true as shown in the example below.

Consider the syntax given by:

$\mathcal{T} = \{x\}$, $M_1 = M_2$, $M_1 = \{x\}$, $M_2 = \{x\}$, head is M_1.

This concrete syntax is clearly ambiguous since there are two derivations for x, namely:

```
M1          M1
|           |
M2          x
|
x
```

The corresponding abstract syntax defines, however, only one object, namely:

```
  °
  x
```

3.4 A Concrete Syntax for EPL

Before proceeding to the statement of a concrete syntax for EPL the well-known Backus Normal Form notation will be extended by adding some convenient shorthands. This extended Backus notation was developed in an attempt to give a clear and readable description of the concrete syntax of PL/I (cf. /5/).

3.4.1 The extended Backus notation

In the following the meaning of the extended forms of Backus notation will be explained by giving the equivalent forms in Backus notation.

In section 3.3 the discussion of Backus Normal Form was carried out in set-theoretic terms. Here, the point of departure will be Backus Normal Form in its usual notation, i.e. we will start with a grammar whose rules have the form

$$V ::= S_1 \mid S_2 \mid \ldots \mid S_n$$

(with the reading: V is to be replaced by one of the alternatives S_1, S_2, ..., S_n.)

In the presentation of these and the following rules, '::=' and '|' are meta-linguistic connectives and

U	denotes arbitrary syntactical units which may be combined to form strings. These units are either terminal symbols or non-terminals.
V	denotes non-terminals.
S_i	denotes arbitrary strings.
T_j	denotes strings different from the empty-string.

The introduction of the meta-linguistic symbols '{', '}', '[', ']', '.' (the last is a fat dot) is determined by the following definitions:

(i) $V ::= S_1\ T_1\ S_2 \mid S_1\ T_2\ S_2 \mid \ldots \mid S_1\ T_n\ S_2$

may be replaced by

$$V ::= S_1 \{ T_1 \mid T_2 \mid \ldots \mid T_n \} S_2$$

and vice versa. (Note: the rule remains valid for the case n = 1.)

Example: variable-declaration ::= { INTEGER | LOGICAL } variable

instead of

variable-declaration ::= INTEGER variable | LOGICAL variable

(ii $V ::= S_1\ S_2 \mid S_1\ T_1\ S_2 \mid \ldots \mid S_1\ T_n\ S_2$

may be replaced by

$V ::= S_1\ [T_1 \mid T_2 \mid \ldots \mid T_n]\ S_2$

and vice versa. (Note: the square brackets differ from the curly brackets in that it is allowed to replace the square brackets and their content by the empty-string.)

Example: function-designator ::= identifier [argument-list]

instead of

function-designator ::= identifier | identifier argument-list

(iii) $V ::= U \mid V\ U$

or

$V ::= U \mid U\ V$

may be replaced by

$V ::= U \bullet\bullet\bullet$

and vice versa. (Note: in the case of the inversion, if U ... occurs in a grammar which has no production rule corresponding to the schema

$V ::= U \bullet\bullet\bullet$,

then this missing rule is to be added to the grammar with a non-terminal V which has not yet been used in the grammar, before the replacement can be performed.)

Example: integer ::= digit •••

instead of

integer ::= digit | integer digit

(iv) $V ::= S_1\ T_1[\ \{\ T_2\ T_1\}\bullet\bullet\bullet]\ S_2$

may be replaced by

$V ::= S_1\{\ T_2 \bullet T_1\}\bullet\bullet\bullet\ S_2$

and vice versa. (Note: instead of '$S_1[\ \{\ T_2 \bullet T_1\ \bullet\bullet\bullet\}\ ]\ S_2$' also '$S_1\ [T_2 \bullet T_1\ \bullet\bullet\bullet]\ S_2$' may be written.)

Example: declaration-list ::= { ;•declaration •••}

instead of

declaration-list ::= declaration [{;declaration}•••]

3.4.2 A concrete syntax

We are now in a position to give a concrete syntax of the example programming language using the above notation. The rules constituting the syntax specification are numbered (C1), (C2), ... for reference purposes and they are here stated without elaborate commentary. The reader should compare this syntax with the intuitive description of the features of the language as stated in the introduction (page 1-3) and with the abstract syntax of section 3.1 to improve his intuitive understanding of the sginficance of the concrete syntax specifications.

(C1) program ::= block
(C2) block ::= BEGIN declaration-list; statement-list END
(C3) declaration-list ::= { ;•declaration •••}
(C4) declaration ::= variable-declaration | procedure-declaration |
function-declaration
(C5) variable-declaration ::= { INTEGER | LOGICAL } variable
(C6) procedure-declaration ::=
PROCEDURE identifier [parameter-list]; statement
(C7) function-declaration ::=
FUNCTION identifier [parameter-list]; statement RETURNS expression
(C8) parameter-list ::= ({,• identifier •••})
(C9) statement-list ::= {;• statement •••}
(C10) statement ::= assignment-statement | conditional-statement |
procedure-call | block
(C11) assignment-statement ::=
identifier = expression

```
(C12) expression ::= constant | variable | function-designator |
                     binary | unary
(C13) constant ::= logical-constant | integer-constant
(C14) variable ::= identifier
(C15) function-designator ::= identifier [ argument-list ]
(C16) argument-list ::= ({,identifier ...})
(C17) binary ::= ( expression binary-operator expression )
(C18) unary ::= unary-operator expression
(C19) conditional-statement ::=
        IF expression THEN statement ELSE statement
(C20) procedure-call ::=
        CALL identifier [argument-list]
```

The set of basic symbols of EPL is defined to be identical with the set $\mathcal{T}$ of section 3.2.2. The range of the function rep, used in the definition of $\mathcal{T}$, is defined as the union of the sets of logical constants, integer constants, identifiers, binary operators and unary operators (cf. (C13), (C14), (C17) and (C18)). These sets are not further specified here.

3.5 The Translation of Concrete Programs to Abstract Programs

The problem is to describe the translation of a concrete program into an abstract program, i.e., the translation of a string producable by the concrete syntax into an object whose structure is described by the abstract syntax.

A program as produced by the concrete syntax is a string of concrete characters, i.e., of members of the alphabet of terminal symbols $\mathcal{T}$ (cf. Section 3.2) of the language under discussion (for EPL, $\mathcal{T}$ is defined on p. 3-10). In order to remain within the range of the methods and concepts for formal definition as explained in Chapter 2, the strings of concrete characters are mapped onto lists of character values, i.e., of abstract elementary objects representing uniquely the concrete characters. Thus, in the remainder of this chapter, a concrete program is a list of character values. The one-to-one mapping between concrete characters and character values can be given by a table, e.g., for EPL, by the following table:

Concrete character	Predicate satisfied by the corresponding character value	
BEGIN	is-BEGIN	These predicates are satisfied uniquely, i.e., for each there exists exactly one elementary object to which it applies.
END	is-END	
PROCEDURE	is-PROC	
FUNCTION	is-FUNCT	
INTEGER	is-INT	
LOGICAL	is-LOG	
CALL	is-CALL	
IF	is-IF	
THEN	is-THEN	
ELSE	is-ELSE	
RETURNS	is-RETURNS	
(	is-LEFT-PAR	
)	is-RIGHT-PAR	
;	is-SEMIC	
,	is-COMMA	
=	is-EQ	
identifier	is-c-id	
integer-constant	is-c-int	
logical-constant	is-c-log	
binary operator	is-c-bin-op	
unary operator	is-c-unary-op	

The translation from concrete programs to abstract programs is performed in two steps by the two functions parse and translate: If txt is a concrete program, the corresponding abstract program is defined as

translate•parse(txt).

The link between these two steps, namely the result of parse and the argument of translate, is a structured object t which is called the abstract representation of txt and may be throught of as the parsing tree of txt.

3.5.1 Abstract representation of concrete programs

In principle one could explicitly define a syntax parser yielding for each concrete program its parsing tree. It is, however, in practice too much of a burden upon the reader to have to read a complete definition of a parser at this point of the formal definition of a programming language so that another way has been sought resulting in an implicit definition of the function parse.

First, the predicate is-c-program characterizing abstract representations is introduced. Next, the function generate mapping abstract representations into concrete programs is defined. Finally, the function parse is implicitly defined as the inverse of generate.

The abstract representation t of a concrete program is an object, satisfying the predicate is-c-program, whose structuring reflects the syntactical structuring of the concrete program and whose elementary components are the character values constituting the concrete program. The predicate is-c-program is defined by a set of predicate definitions obtained by rewriting the production rules of the concrete syntax (cf. Section 3.4.2 and the definitions (AR1) - (AR20) below).

In formulating these predicate definitions certain standard selectors, s_1, s_2 ... are used which are assumed to be mutually different and may be considered as the values s(i) of a selector function s. By means of the selectors s_i objects are formed which have a structure similar to the lists formed by the selectors elem(i) (cf. page 2-16) except that these "s-lists" may have "gaps". In analogy to the function length for lists, a function slength is defined for s-lists as follows:

slength(x) =

$$(\forall i)(\text{is-}\Omega \circ s_i(x)) \longrightarrow 0$$

$$T \longrightarrow (\iota i)(\neg\, \text{is-}\Omega \circ s_i(x) \;\&\; (\forall j)(j > i \supset \text{is-}\Omega \circ s_j(x)))$$

Also an additional definition schema is needed which will be numbered (6) in order to indicate that we are extending the set (1) - (5) of Chapter 2. The short version of the definition is:

(6) is-pred = (<s-sel$_1$:is-pred$_1$>, ...,<s-sel$_n$:is-pred$_n$>,
<s-sel-fct(1):is-pred$_o$>,...)

In this definition sel-fct is a selector function, e.g., elem or s, mapping integer values into selectors. The full definition is:

$$\text{is-pred}(x) =$$

$$(\exists x_1,\ldots,x_n,m,y_1,\ldots,y_m)(m\geq 1 \;\&\; \overset{n}{\underset{i=1}{\&t}}\ \text{is-pred}_i(x_i) \;\&\; \overset{m}{\underset{i=1}{\&t}}\ \text{is-pred}_o(y_i) \;\&$$

$$x = \mu_o(\{\langle \text{s-sel}_1 : x_1\rangle,\ldots,\langle \text{s-sel}_n : x_n\rangle,$$
$$\langle \text{sel-fct}(1) : y_1\rangle,\ldots,\langle \text{sel-fct}(m) : y_m\rangle\})).$$

The short notation is often used with n = 0, i.e.,

$$\text{is-pred} = (\langle \text{sel-fct}(1) : \text{is-pred}_o\rangle,\ldots)$$

which means

$$\text{is-pred}(x) =$$

$$(\exists m,y_1,\ldots,y_m)(m\geq 1 \;\&\; \overset{m}{\underset{i=1}{\&t}}\ \text{is-pred}_o(y_i) \;\&$$

$$x = \mu_o(\langle \text{sel-fct}(1) : y_1\rangle,\ldots,\langle \text{sel-fct}(m) : y_m\rangle)).$$

Using this notation, one could define the notation of is-pred-list by:

$$\text{is-pred-list} = \text{is-}\langle\rangle \vee (\langle \text{elem}(1) : \text{is-pred}\rangle,\ldots).^{1)}$$

An abstract representation of EPL

The following predicate definitions (AR1) - (AR20) can be obtained by a mechanical rewriting rule from the productions (C1) - (C20) in Section 3.4.5, using the table on page 3-19. (For a formulation of this type of rewriting rule see Appendix I in /6/).

---

1) For the definition of the abstract representation of EPL only the selectors s_i are required. In /6/ both s_i and elem(i) are used in order to permit the insertion of blanks in some places and not in others.

```
(AR1)  is-c-progr = is-c-block
(AR2)  is-c-block = (<s1:is-BEGIN>,
                     <s2:is-c-decllist>,
                     <s3:is-SEMIC>,
                     <s4:is-c-stlist>,
                     <s5:is-END>)
(AR3)  is-c-decllist = (<s-del:is-SEMIC>,
                        <s1:is-c-decl>,...)
(AR4)  is-c-decl = is-c-var-decl v is-c-proc-decl v is-c-funct-decl
(AR5)  is-c-var-decl = (<s1:is-INT v is-LOG>,
                        <s2:is-c-var>)
(AR6)  is-proc-decl = (<s1:is-PROC>,
                       <s2:is-c-id>,
                       <s3:is-c-parlist v is-Ω>,
                       <s4:is-SEMIC>,
                       <s5:is-c-st>)
(AR7)  is-funct-decl = (<s1:is-FUNCT>,
                        <s2:is-c-id>,
                        <s3:is-c-par-list v is-Ω>,
                        <s4:is-SEMIC>,
                        <s5:is-c-st>,
                        <s6:is-RETURNS>,
                        <s7:is-c-expr>)
(AR8)  is-c-parlist = (<s1:is-LEFT-PAR>,
                       <s2:(<s-del:is-COMMA>,
                            <s1:is-c-id>,...)>,
                       <s3:is-RIGHT-PAR>)
(AR9)  is-c-stlist = (<s-del:is-SEMIC>,
                      <s1:is-c-st>,...)
(AR10) is-c-st = is-c-assign-st v is-c-cond-st v is-c-proc-call v is-c-block
(AR11) is-c-assign-st = (<s1:s-c-id>,
                         <s2:is-EQ>,
                         <s3:is-c-expr>)
(AR12) is-c-expr = is-c-const v is-c-var v is-c-funct-des v is-c-bin v is-c-unary
(AR13) is-c-const = is-c-log v is-c-int
(AR14) is-c-var = is-c-id
(AR15) is-c-funct-des = (<s1:is-c-id>,
                         <s2:is-c-arglist v is-Ω>)
```

(AR16) is-c-arglist = (<s_1:is-LEFT-PAR>,
<s_2:(<s-del:is-COMMA>,
<s_1:is-c-id>,...)>,
<s_3:is-RIGHT-PAR>)
(AR17) is-c-bin = (<s_1:is-LEFT-PAR>,
<s_2:is-c-expr>,
<s_3:is-c-bin-rt>,
<s_4:is-c-expr>,
<s_5:is-RIGHT-PAR>)
(AR18) is-c-unary = (<s_1:is-c-unary-rt>,
<s_2:is-c-expr>)
(AR19) is-c-cond-st = (<s_1:is-IF>,
<s_2:is-c-expr>,
<s_3:is-THEN>,
<s_4:is-c-st>,
<s_5:is-ELSE>,
<s_6:is-c-st>)
(AR20) is-c-proc-call = (<s_1:is-CALL>,
<s_2:is-c-id>,
<s_3:is-c-arglist ∨ is-Ω>)

The functions generate and parse

In defining the function generate it is assumed that a special selector s-del has been used to select list delimiters (cf. e.g., (AR3) and (AR9)).

The function generate mapping the object t, satisfying the predicate is-c-program, into a set of character value lists is now defined as follows:

generate(t) =
is-Ω(t) $\longrightarrow$ <>
slength(t) = 0 $\longrightarrow$ <t>
T $\longrightarrow$
$$\text{generate}\circ s_1(t) \frown \underset{i=2}{\overset{\text{slength}(t)}{\text{CONC}}} (\text{generate}\circ \text{s-del}(t) \frown \text{generate}\circ s_i(t))$$

The abstract representation of the concrete syntax, which follows in the next section, together with the function generate constitutes a formal definition of an algorithm for generating all concrete programs of the programming language. Hence, they are equivalent to the production rules of the concrete syntax together with the instructions for their use.

Finally, the function parse, which is the inverse of the function generate, is defined as follows:

```
parse(txt) =
    (ιt)(txt = generate(t) & is-c-program(t))
```

Assuming that the concrete syntax is unambiguous, the meaning of this definition is that the function parse transforms a character value list into its parsing tree t, provided the list is a syntactically correct concrete program.

3.5.2 The translator

This section describes the translation from the abstract representation of a concrete program into an abstract program. This translation is performed by the function

```
translate(t)
```

which maps an object t satisfying the predicate is-c-program, described by the abstract representation of the concrete syntax given in the last section, into an object satisfying the predicate is-program, described by the abstract syntax given in Section 3.1, page 3-3. As in the case of the interpretation of an abstract program in Chapter 5, the definition of the translation is here reduced step by step to the translation of the components of the program text t. But there are two essential differences between the concept of the interpreter and the translator.

(a) The translator is specified by a function mapping a concrete program, in its abstract representation, into the corresponding abstract program, while the interpreter is specified by instructions mapping machine states into successor machine states (see Section 5).

(b) The translation of a part of a program of a programming language, of greater complexity than our example (e.g. PL/I), generally depends not only on this program part itself but also on the context in which it occurs within the complete program text, while the interpretation of a part of a program generally depends only on this program part itself (and the current machine state, which may reflect contextual information if necessary).

Even though the second point does not strictly apply to our example it is mirrored in the translator to give the reader an insight into the way in which such contextual information might be treated in a translator of this kind.

Hence, to accomodate the two above mentioned differences the following concepts are applied in defining the translator:

(a) Instead of the current machine state ξ, which is a hidden argument of each instruction of the interpreter (cf. page 5-2), the complete program text t, to be translated, is a hidden argument of each function of the translator, which is not specified explicitly for each function. Throughout this section the letter t denotes this hidden argument, called the text, which is the same object satisfying is-c-program for all programs.

(b) Instead of objects to be translated, which are components of t, generally the selectors selecting them from t are specified as arguments of the functions. These selectors, called "text pointers" or just "pointers", are composed of selectors of the form elem(i) and s(i), i being integer values. They are usually denoted by the letters p, q and r.

The two arguments, the hidden text t and the explicitly specified pointer p, constitute all the necessary information: A part of t, namely p(t) and the context of this part within t.

In a more complicated programming language than that of the example, especially one in which declarations need not be collected in a particular part of the program but may be scattered throughout the program or declared implicitly (as in the case of PL/I), the main job of the translator is the recognition of all declarations in a concrete program and the testing, completing and structuring of their attributes. For the other components of a program and almost entirely in the case of the present example, the translation consists essentially of a one-to-one mapping from the parsing tree into the abstract program. This mapping constructs objects built up with mnemonically named selectors instead of selectors determined only by the ordering in the concrete program.

Lastly one should perhaps mention the kind of tests that may be built into this type of translator. It may be remembered that the function parse rejects all strings which are not programs as specified by the concrete syntax. Similarly, the translator checks for multiple declaration (see (T3)) and for the use of the same parameter in different positions of the same parameter list (see (T5)). In this

way it is possible to check if any context dependent criteria as specified by the definition of the programming language have been violated. This type of error checking is called "static" and may be carried out during translation independent from an interpretation. Errors which can only be checked during interpretation are called "dynamical errors" and they will be discovered by the interpreter only if the program part that contains the error is actually interpreted.

The function translate for EPL

(T1) translate(t) =

$$\text{is-c-progr}(t) \longrightarrow \text{trans-block}(I)$$
$$T \longrightarrow \text{error}$$

(T2) trans-block(p) =

$$\mu_o(\langle \text{s-decl-part}:\text{trans-decllist}(s_2 \circ p)\rangle, \langle \text{s-st-list}:\text{trans-stlist}(s_4 \circ p)\rangle)$$

(T3) trans-decllist(p) =

$$\neg(\exists i,j)(i \neq j \;\&\; s_2 \circ s_i\; p(t) = s_2 \circ s_j \circ p(t) \neq \Omega) \longrightarrow \mu_o(\{\langle \text{id}:\text{trans-decl}(s_1 \circ p)\rangle \mid \text{id} = s_2 \circ s_i \circ p(t) \neq \Omega\})$$
$$T \longrightarrow \text{error}$$

Note: The condition makes sure that there be no multiple declarations.

(T4) trans-decl(p) =

$$\text{is-INT} \circ s_1 \circ p(t) \longrightarrow \text{INT}$$
$$\text{is-LOG} \circ s_1 \circ p(t) \longrightarrow \text{LOG}$$
$$\text{is-PROC} \circ s_1 \circ p(t) \longrightarrow \mu_o(\langle \text{s-param-list}:\text{trans-parlist}(s_3 \circ p)\rangle, \langle \text{s-st}:\text{trans-st}(s_5 \circ p)\rangle)$$
$$\text{is-FUNCT} \circ s_1 \circ p(t) \longrightarrow \mu_o(\langle \text{s-param-list}:\text{trans-parlist}(s_3 \circ p)\rangle, \langle \text{s-st}:\text{trans-st}(s_5 \circ p)\rangle, \langle \text{s-expr}:\text{trans-expr}(s_7 \circ p)\rangle)$$

(T5) trans-parlist(p) =

is-Ω∘p(t) ⟶ ⟨⟩
¬(∃i,j)(i ≠ j & s_i∘s_2∘p(t) = s_j∘s_2∘p(t) ≠ Ω) ⟶
μ_o({⟨elem(i):s_i∘s_2∘p(t)⟩ | 1≤i≤slength∘s_2∘p(t)⟩)
T ⟶ error

Note: The condition prevents that the same parameter occurs more than once in a given parameter list.

(T6) trans-stlist(p) =

μ_o({⟨elem(i):trans-st(s_i∘p)⟩ | 1≤i≤slength∘p(t)})

(T7) trans-st(p) =

is-c-assign-st∘p(t) ⟶
μ_o(⟨s-left-part:s_1∘p(t)⟩,
⟨s-right-part:trans-expr(s_3∘p)⟩)

is-c-cond-st∘p(t) ⟶
μ_o(⟨s-expr:trans-expr(s_2∘p)⟩,
⟨s-then-st:trans-st(s_4∘p)⟩,
⟨s-else-st:trans-st(s_6∘p)⟩)

is-proc-call∘p(t) ⟶
μ_o(⟨s-id:s_2∘p(t)⟩,
⟨s-arg-list:trans-arglist(s_3∘p)⟩)

is-c-block∘p(t) ⟶ trans-block(p)

(T8) trans-arglist(p) =

is-Ω∘p(t) ⟶ ⟨⟩
T ⟶
μ_o({⟨elem(i):s_i∘s_2∘p(t)⟩ | 1≤i≤slength∘s_2∘p(t)})

(T9) trans-expr(p) =

is-c-const∘p(t) ⟶ p(t)
is-c-var∘p(t) ⟶ p(t)
is-c-funct-des∘p(t) ⟶
μ_o(⟨s-id:s_1∘p(t)⟩,
⟨s-arg-list:trans-arglist(s_2∘p)⟩)

cont'd

is-c-bin∘p(t) ⟶

μ_o(<s-rd1:trans-expr(s_2∘p)>,
<s-rd2:trans-expr(s_4∘p)>,
<s-op:s_3∘p(t)>)

is-c-unary∘p(t) ⟶

μ_o(<s-rd:trans-expr(s_2∘p)>,
<s-op:s_1∘p(t)>)

4. ABSTRACT MACHINES

4.1 Introduction

The method adopted for the formal definition of the semantics of a higher level programming language like Algol 60 or PL/I is based on the definition of an abstract machine which is characterized by the set of states it can assume and its state transition function. A specific program in the given language together with its input data defines an initial state of the associated machine, and the subsequent behaviour of the machine is said to define the interpretation of that program for the given input data.

This chapter attempts to describe those aspects of the abstract machine which were felt to be significant independently of the application to a specific programming language. In particular it will be described how the control of the machine works. The control appears to be well-suited for the interpretation of programming languages which possess a nested structure of statements and, furthermore, a certain indeterminance in the sequencing of statements.

4.2 The Conventional Concept of Abstract Sequential Machines[1)]

An abstract sequential machine may be specified by defining the set of states Σ which the machine can assume and a state transition function Λ which for any given state ξ specifies a successor state, i.e. which maps states into states. The criterion for deciding when the machine is said to stop may be given by defining a subset Σ_E of the set of states, the end states of the machine.

For any given initial state ξ_0 one may visualize the behaviour of the machine as running through a succession of states $\xi_0, \xi_1, \ldots, \xi_i, \xi_{i+1}, \ldots$ where $\xi_{i+1} = \Lambda(\xi_i)$. The succession of states, called the computation for the given initial state ξ_0, stops if an end state $\xi_n \in \Sigma_E$ is reached in which case the computation is called terminating. A computation where no end state is ever reached is called non-terminating and is the precise equivalent of a process which loops indefinitely.

---

1) Compare e.g. Elgot /11/.

4.3 The Extended Concept of Abstract Machines as Used for the Formal Definition of Programming Languages

There are a number of cases in the interpretation of a programming language like PL/I or Algol 60 in which the sequence in which certain operations are performed is relevant but left open by definition. Examples for such situations are the evaluation of operands in an expression, the evaluation of expressions occurring in a data attribute, the evaluation of arguments in a procedure or function call etc. The concept of abstract machines has been extended to meet this situation.

The abstract machine underlying the definition of a programming language is defined by a quadruple <EO,S,is-state,Λ>. EO is an infinite set of elementary objects. S is an infinite set of simple selectors. is-state is a predicate which defines the class of objects (out of the general class of objects built from elementary objects of EO with the help of selectors of S, cf. 2.1) representing the states the machine can assume. Λ is a state transition function. Due to the identification of the set of states with a certain class of objects, this set can always be defined by the devices described in section 2.7, "Definition of Classes of Objects". Also, the mappings from given states into successor states can be specified with the help of the μ-operator.

In the sequel ξ, ξ_0, ξ_1,... will stand for arbitrary states of the machine, i.e. for arbitrary elements of $\hat{\text{is-state}}$.

The <u>state transition function</u> Λ [1] specifies for a given state ξ the set of possible successor states, i.e. Λ is a function which maps states into sets of states. This possibility to define more than one successor state for any given state reflects the above mentioned indeterminance of the sequence in which certain operations are performed.

A <u>computation</u> for a given initial state ξ_0 is a sequence of states $\xi_0, \xi_1, \ldots, \xi_i, \xi_{i+1}, \ldots$ such that $\xi_{i+1} \in \Lambda(\xi_i)$. This means that a computation can be produced step by step from left to right by applying the function Λ to the last state in the sequence. The successor state can then be determined by a choice of one element of the result of Λ. Any initial state defines a set of computations according to the possible choices of successor states for any step.

---

1) Called <u>language function</u> in /8/.

A state ξ is called an end state if the state transition function Λ yields the empty set, i.e. $\Lambda(\xi) = \{\}$. There is obviously no possible continuation of a computation if an end state is reached. Such a computation is a finite sequence of states and is called terminating. Computations which are infinite sequences of states are called non-terminating.

The state transition function Λ may be a partial function for the set of possible states which is its domain. Hence there is a third type of computations, namely those which cannot be continued because the function Λ applied to the last state does not have a value.

A program together with its input data defines an initial state. There are initial states whose corresponding set of computations contains elements of all three types described above.

Any state ξ has an immediate component s-c(ξ), called the control part, which satisfies the predicate is-c, i.e.:

is-c(s-c(ξ))

An explanation of the control of the machine, i.e. of the structure and function of the control part, together with the associated notational conventions, is given in section 4.4.

The number of state components and their structure and function depends on the specific language to be defined by the machine. In section 4.5 some general constructs of the state are described.

4.4 The Control of the Abstract Machine[1)]

4.4.1 First survey

The control part of a state ξ of the machine can be visualized as a finite tree where each node of the tree is associated with an instruction. Such a tree is called control tree. The sequence in which the instructions are to be executed is partially given by the convention that only the instructions associated with the terminal nodes of the control tree are candidates for being executed next. The successor state of a given state ξ is determined by choosing one of the terminal nodes of s-c(ξ) and executing the associated instruction.

1) The concept of control described in this section is well suited for one-task machines. For multi-task machines it must be slightly modified.

Consider e.g. that the control tree shown in Fig. 4-1 forms the control part of a certain state Ξ.

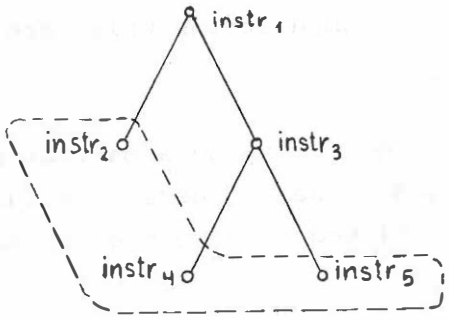


Fig. 4-1

Then the instructions $instr_2$, $instr_4$, and $instr_5$ are the candidates for being executed next.

Let tn(ct) be the set of selectors pointing to terminal nodes of ct. The state transition function Λ is then defined by:

$$\Lambda(\Xi) = \{ \Psi(\Xi, \tau) \mid \tau \in tn(s\text{-}c(\Xi)) \}$$

where $\Psi(\Xi, \tau)$ specifies the successor state according to executing the instruction associated with the terminal node τ.

An instruction is composed of an instruction name and optionally a list of arguments. The notation used for representing instructions is:

$$\underline{in}(arg_1, \ldots, arg_n)$$

Instruction names are underlined words which identify respective instruction definitions, the arguments are objects.

There is no logical restriction as to the type of changes to the state, which the execution of an instruction may cause. In particular, the execution of an instruction may modify the control part. However, it seems to be convenient to get along with only two types of effects which the execution of an instruction might have. The first type of effect, called value-returning, is to delete the instruction being executed from the control tree, to substitute a value specified by the definition of the instruction into argument places of instructions in predecessor nodes of the control tree, and to make some changes in the state, though, in general, no further changes in the control part. The second type of effect, called self-replacing, is such that the instruction being executed replaces itself

in the control tree by another control tree, a process which is very similar to a macro expansion and serves the same purpose. Whether the effect of an instruction is value-returning or self-replacing may, in general, depend on the state ξ in which the instruction is executed.

The remaining sections are organized as follows: First, the structure of control trees will be further specified, and a notation for writing control trees will be introduced (section 4.4.2); this will complete the intuitive description of control trees. Then, a more rigid description will be given, modelling control trees as a certain class of objects (section 4.4.3). Also, control tree representations, i.e. expressions whose values are control trees (and of which the notation introduced in 4.4.2 is a special case), will be introduced, and a rule will be given how they can be interpreted as expressions of conventional shape (section 4.4.4). With these prerequisites, a notation for instruction schemata, i.e. instruction definitions, can be described, together with a rule how an instruction schema can be interpreted as the definition of a state-transforming function (section 4.4.5). The function $\Psi(\xi, \tau)$ can then be defined in terms of these state-transforming functions, and together with the defintion of the function tn(ct), this will complete the definition of the state transition function Λ (section 4.4.6; cf. beginning of this section). Finally, some examples will be given (section 4.4.7).

4.4.2 Control trees

Before going into detailed description of the function and definition of the two types of effects an instruction may cause, the structure of control trees must be further specified.

The argument places of a control tree into which a given instruction has to substitute its value (in the case its effect is value-returning) are indicated in the control tree itself. One may illustrate the situation in the control tree by drawing dotted lines from the node with which the respective instruction is associated to the argument places of predecessor instructions where its value is to be substituted. Those arguments are Ω before substitution. More precisely, it is assumed that a node allows retrieval of not only an instruction but also the necessary information as to where this instruction has to substitute its value. Values may be returned by an instruction to more than one place and over more than one level in the control tree.

A more detailed description of Fig. 4-1 may serve as an example:

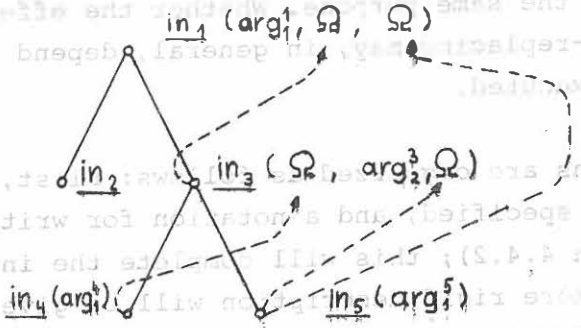


Fig. 4-2

The purpose of the value-return mechanism is to provide a tool for the treatment of intermediate results. Due to the unspecified order in the execution of terminal instructions, a linear stack for holding intermediate results (cf. e.g. Landin /12/) would not be sufficient.

Notation for control trees

The first step towards a notation for control trees is to replace the dotted lines in the control trees by a device which uses dummy names. Instructions which return a value to argument places of predecessor instructions are prefixed with a dummy name followed by a colon. The same dummy name is inserted in those argument places into which the value of the instruction is to be returned.

Using as dummy names a, b, and c, Fig. 4-2 can be rewritten as shown in Fig. 4-3.

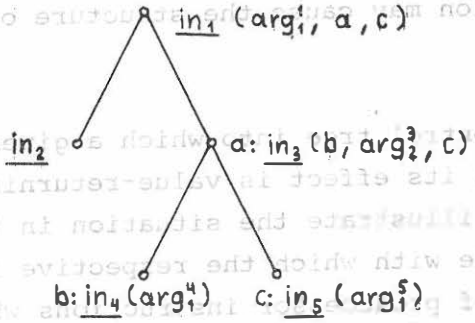


Fig. 4-3

As will turn out later on (cf. 4.4.3 and 4.4.5), dummy names have a completely local meaning within the control tree in which they are specified.

The notation for control trees may now be described as follows: If the control tree consists of the top node only, then the instruction attached to the top node

represents the control tree. If the control tree consists of more than one node, then it is represented by the instruction attached to the top node, followed by semi-colon, followed by the set of control trees which are the immediate components. The notation for the control tree of Fig. 4-3 reads:

$$\underline{in}_1(arg_1^1,a,c);\{\underline{in}_2,a:\underline{in}_3(b,arg_2^3,c);\{b:\underline{in}_4(arg_1^4),c:\underline{in}_5(arg_1^5)\}\}$$

Curly brackets may be omitted if they contain only one instruction. Furthermore, instead of curly brackets, indentation may be used in such a way that instructions which are on the same level in the tree are written starting on identical left margins.

The above example rewritten using indentation reads:

$\underline{in}_1(arg_1^1,a,c);$

  $\underline{in}_2,$

  $a:\underline{in}_3(b,arg_2^3,c);$

    $b:\underline{in}_4(arg_1^4),$

    $c:\underline{in}_5(arg_1^5)$

Fig. 4-4

It is of importance to note that the direction of execution is from bottom to top as indicated in Fig. 4-4.

Extension to the value-return mechanism

As described so far instructions can only return values to argument places of predecessor instructions. However, it occurs quite frequently that intermediate results are to be built from the values of several instructions. For this purpose a device is introduced which allows the return of values from instructions to component positions of arguments of predecessor instructions. In other words, values of instructions may replace components of arguments in predecessor instructions and not solely entire arguments.

If say a is a dummy name at several argument positions in a control tree, then the notation

$\varkappa(a)$

points to the positions of the $\varkappa$ components of the respective argument positions.

Consider the following example:

$\underline{in}_1(a)$; $\{\varkappa_1(a):\underline{in}_2, \varkappa_2(a):\underline{in}_3\}$

Instruction $\underline{in}_2$ returns its value to the $\varkappa_1$-part and instruction $\underline{in}_3$ returns its value to the $\varkappa_2$-part of argument position a of instruction $\underline{in}_1(a)$.

4.4.3 Defining control trees as objects

First, some special sets of objects and selectors used in the following are defined.

Sets of elementary objects:

is-intg	{1,2,...}
is-in	infinite set of instruction names
is-name	infinite set of metavariables
is-sel	set of all selectors (i.e. is-sel = S^*)

The predicate is-ob holds for arbitrary objects.

The predicate is-sel-pair holds for arbitrary pairs of selectors, i.e.:

is-sel-pair = (<elem(1):is-sel>,<elem(2):is-sel>)

s-in, s-al, s-ri, s-sel, s-dum are constant simple selectors.

R is an infinite set of simple selectors, not containing s-in, s-al and s-ri.

R^* is the set of all selectors composed from selectors of R, including I.

Key to abbreviations:

s-in	select instruction name
s-al	select argument list
s-ri	select return information
s-sel	select selector
s-dum	select dummy name

The previously described control trees are now given a precise shape by defining an abstract syntax for them. The value-return mechanism (cf. the dotted lines in Fig. 4-2, p. 4-6) is modelled by kepping for each node a set of pairs of selectors. The first selector of a pair says where the value to be returned comes from, the second, where it has to be returned to (cf. 4.4.6 for the use of the first selector). I.e., a control tree is an object satisfying the following predicate is-ct:

```
is-ct = (<s-in:is-in>,
         <s-al:({<elem(i):is-ob> || is-intg(i)})>,
         <s-ri:is-sel-pair-set>,
          {<r:is-ct> || r ∈ R})
```

The predicate is-c of section 4.3 which holds for the control part s-c(Ξ) of any state Ξ is then defined by:

$$\text{is-c} = \text{is-ct} \vee \text{is-}\Omega$$

Intermediate control trees

In section 4.4.2, a notation has been introduced which uses dummy names to express the value-return mechanism. It is convenient to have a class of objects which correspond more closely to this notation (cf. 4.4.4). These objects are called intermediate control trees and satisfy the predicate is-int-ct:

```
is-int-ct = (<s-in:is-in>,
             <s-al:({<elem(i):is-ob> || is-intg(i)})>,
             <s-ri:(<s-sel:is-sel>,<s-dum:is-name ∨ is-Ω>)>,
              {<r:is-int-ct> || r ∈ R})
```

The control tree corresponding to an intermediate control tree

For the transformation from intermediate control trees to control trees the function tr(ct) will be defined in terms of some other functions.

The function nd(ct) is defined for is-int-ct(ct) (as well as for is-c(ct), see section 4.4.6) and yields the set of selectors pointing to nodes of ct:

$$\text{nd}(ct) = \{ \varkappa \mid \varkappa \in R^* \ \& \ \varkappa(ct) \neq \Omega \}$$

The function arg(ct) is defined for is-int-ct(ct) and yields the set of selectors pointing to arguments of instructions of ct not equal Ω:

$$\text{arg}(ct) = \{\text{elem}(i) \circ \text{s-al} \circ \varkappa \mid \varkappa \in \text{nd}(ct) \ \& \ \text{elem}(i) \circ \text{s-al} \circ \varkappa(ct) \neq \Omega\}$$

The function dum(ct) is defined for is-int-ct(ct) and yields the set of dummy names occurring in ct:

$$dum(ct) = \{ s\text{-}dum \circ s\text{-}ri \circ \varkappa(ct) \mid \varkappa \in nd(ct) \ \& \ s\text{-}dum \circ s\text{-}ri \circ \varkappa(ct) \neq \Omega \}$$

The function ri(ct,ϰ) is defined for is-int-ct(ct), ϰ ∈ nd(ct) and yields the set of pairs of selectors, which will be the return information of the control tree at node ϰ:

$$ri(ct,\varkappa) = \{ \langle \varkappa, \varkappa_1 \circ \varkappa_2 \rangle \mid \varkappa_1 = s\text{-}sel \circ s\text{-}ri \circ \varkappa(ct) \ \& \ \varkappa_2 \in arg(ct) \ \& \ \varkappa_2(ct) = s\text{-}dum \circ s\text{-}ri \circ \varkappa(ct) \}$$

The function tr(ct) can now be defined for is-int-ct(ct) and yields the corresponding control tree:

$$tr(ct) = \mu(ct; \{ \langle \varkappa : \Omega \rangle \mid \varkappa \in arg(ct) \ \& \ \varkappa(ct) \in dum(ct) \} \cup \{ \langle s\text{-}ri \circ \varkappa : ri(ct,\varkappa) \rangle \mid \varkappa \in nd(ct) \})$$

4.4.4 Control tree representations[1)]

A control tree representation is a meta-expression[2)] which, for given values of its free variables, denotes a control tree. It will be abbreviated by ct-rep, or, showing its free variables, by ct-rep($x_1,\ldots,x_n,\xi$). Control tree representations are used within instruction schemata (cf. 4.4.5), and in fact may occur within the metalanguage in any context where unconditional expressions are allowed. A special case of control tree representations has already been introduced in 4.4.2.

The meaning of a control tree representation will be determined in terms of an associated meta-expression of known shape for which it stands. The definition is given in form of a table listing the syntactical categories of the constituents of control tree representations, terminating with the category of control tree representation itself. For each syntactical category abbreviation, form and meaning of its members is specified.

---

1) Called control representations in /8/.

2) Cf. foot-note 1), page 3-7.

syntactical category	abbreviation	form	meaning
metavariable	x	x	x
unconditional meta-expression	expr	arbitrary unconditional meta-expression	meaning of the meta-expression
instruction name	in	in	in
instruction	instr	(1) in (2) in(expr$_1$,..., expr$_m$)	(1) μ_o(<s-in:in>) (2) μ_o(<s-in:in>,<s-al:μ_o(<elem(1):expr$_1$>,..., <elem(m):expr$_m$>)>)
prefixed instruction	pref-instr	(1) instr (2) x:instr (3) expr(x):instr	(1) μ(instr;<s-ri:μ_o(<s-sel:I>)>) (2) μ(instr;<s-ri:μ_o(<s-sel:I>,<s-dum:x>)>) (3) μ(instr;<s-ri:μ_o(<s-sel:expr>,<s-dum:x>)>)
successor	succ	(1) pref-instr (2) pref-instr; succ-set	(1) pref-instr (2) μ(pref-instr;{<sel(x,succ-set):x> | x $\in$ succ-set}) 1)
successor set	succ-set	(1) {succ$_1$,..., succ$_k$} (2) {succ | expr} (3) succ-set$_1$ $\cup$ ... $\cup$ succ-set$_r$	(1) {succ$_1$,...,succ$_k$} (2) {succ | expr} (3) succ-set$_1$ $\cup$ ... $\cup$ succ-set$_r$
control tree representation	ct-rep	succ 2)	tr(succ)

1) x must not occur free in succ-set.

2) The initial pref-instr of succ is restricted to form (1).

In the column "meaning", the transformation of subparts is implied without notice, i.e. the abbreviations instr, pref-instr, etc. occurring there stand for the already transformed subparts. Into what a given text is transformed, may depend not only on the text itself, but also on the syntactical category as member of which the text is considered. So, a prefixed instruction transforms differently from an instruction, even in the special case where it is an instruction.

The function sel(ob,succ-set) occurring in the table defines a one to one mapping from objects of the set succ-set to selectors of R. For the function tr(ct), see 4.4.3.

In the case that a successor set of form (1) occurs in a control tree representation, indentation may be used instead of curly brackets.

4.4.5 Instruction schemata

As mentioned in section 4.4.1, each instruction name identifies an instruction schema which defines the effect of executing an instruction with this name, depending on the arguments of the instruction and the state ξ .

An instruction schema associated with an instruction name *in* has in general the following form:

$\underline{in}(x_1,\ldots,x_n) =$

$p_1(x_1,\ldots,x_n,\xi) \longrightarrow group_1(x_1,\ldots,x_n,\xi)$

.....

$p_m(x_1,\ldots,x_n,\xi) \longrightarrow group_m(x_1,\ldots,x_n,\xi)$

$p_i(x_1,\ldots,x_n,\xi)$, for $1 \leq i \leq m$, are meta-expressions denoting truth values.

$group_i(x_1,\ldots,x_n,\xi)$, for $1 \leq i \leq m$, are called groups and can have one of the following two forms:

(1) *value-returning* alternative of a group:

PASS $:\varepsilon_0(x_1,\ldots,x_n,\xi)$

$\underline{s\text{-}sc_1}:\varepsilon_1(x_1,\ldots,x_n,\xi)$

.....

$\underline{s\text{-}sc_r}:\varepsilon_r(x_1,\ldots,x_n,\xi)$

$\varepsilon_i(x_1,\ldots,x_n,\Xi)$, for $0\le i\le r$, are arbitrary meta-expressions.

s-sc$_i$, for $1\le i\le r$, are simple selectors which point to immediate components of the state (not necessarily all immediate components must be referred to).

If the first line of the group is missing, this is equivalent to:
PASS: Ω

(2) <u>self-replacing</u> alternative of a group:

ct-rep$(x_1,\ldots,x_n,\Xi)$

ct-rep$(x_1,\ldots,x_n,\Xi)$ is a control tree representation.

The special case:

$\underline{in}(x_1,\ldots,x_n)$ =

$T \longrightarrow$ group$(x_1,\ldots,x_n,\Xi)$

may also be written:

$\underline{in}(x_1,\ldots,x_n)$ =

group$(x_1,\ldots,x_n,\Xi)$

Each instruction schema will now be associated with a function definition. The definition of the state transition function Λ then can be given easily in terms of these functions.

The function definition associated with an instruction schema of the previously given general form has the following form:

$\phi_{\underline{in}}(x_1,\ldots,x_n,\Xi,\tau,ri)$ =

$p_1(x_1,\ldots,x_n,\Xi) \longrightarrow$ group$_1^*(x_1,\ldots,x_n,\Xi,\tau,ri)$

. . . . . .

$p_m(x_1,\ldots,x_n,\Xi) \longrightarrow$ group$_m^*(x_1,\ldots,x_n,\Xi,\tau,ri)$

group$_i^*(x_1,\ldots,x_n,\Xi,\tau,ri)$, for $1\le i\le m$, is obtained from group$_i(x_1,\ldots,x_n,\Xi)$ in the following way:

If group$_i(x_1,\ldots,x_n,\xi)$ has form (1) (value-returning alternative), then group$_i^*(x_1,\ldots,x_n,\xi,\tau,ri)$ has the form:

$\mu(\mu(\Xi;\{\langle\chi_2\circ(\tau-\chi_1)\circ\text{s-c}:\varepsilon_o(x_1,\ldots,x_n,\Xi)\rangle \mid \langle\chi_1,\chi_2\rangle\in ri\});$

$\langle\text{s-sc}_1:\varepsilon_1(x_1,\ldots,x_n,\xi)\rangle,\ldots,\langle\text{s-sc}_r:\varepsilon_r(x_1,\ldots,x_n,\xi)\rangle)$

The operation $\varkappa_1 - \varkappa_2$ is defined, if $\varkappa_2$ is head of $\varkappa_1$, and yields the tail of $\varkappa_1$ according to $\varkappa_2$:

$$\varkappa_1 - \varkappa_2 = (\iota\varkappa)(\varkappa_1 = \varkappa_2 \circ \varkappa)$$

If $group_i(x_1,\ldots,x_n,\xi)$ has form (2) (self-replacing alternative), then $group_i^*(x_1,\ldots,x_n,\xi,\tau,ri)$ has the form:

$$\mu(\xi; <\tau \circ s\text{-}c; \mu(ct\text{-}rep(x_1,\ldots,x_n,\xi); <s\text{-}ri:ri>)>)$$

4.4.6 The state transition function $\wedge$

The definition of the state transition function $\wedge$ given in section 4.4.1, i.e.:

$$\wedge(\xi) = \{\Psi(\xi,\tau) \mid \tau \in tn(s\text{-}c(\xi))\}$$

can now be completed by the formal definition of the functions tn(ct) and $\Psi(\xi,\tau)$.

The function tn(ct) is defined for is-c(ct) and yields the set of selectors pointing to terminal nodes of ct. The definition is given in terms of the function nd(ct) defined in section 4.4.3:

$$tn(ct) = \{\tau \mid \tau \in nd(ct) \;\&\; (\forall r)(r \in \mathbb{R} \supset r \circ \tau(ct) = \Omega)\}$$

The function $\Psi(\xi,\tau)$ is defined for $\tau \in tn(s\text{-}c(\xi))$ and yields the successor state for the execution of the instruction at position τ of $s\text{-}c(\xi)$. With the abbreviations:

$\underline{in} = s\text{-}in \circ \tau \circ s\text{-}c(\xi)$
n = number of parameters associated with $\underline{in}$
$al = s\text{-}al \circ \tau \circ s\text{-}c(\xi)$
$ri = s\text{-}ri \circ \tau \circ s\text{-}c(\xi)$

the definition can be given in terms of $\Phi_{\underline{in}}$:

$$\Psi(\xi,\tau) = \Phi_{\underline{in}}(elem(1,al),\ldots,elem(n,al),\delta(\xi;\tau \circ s\text{-}c),\tau,ri)$$

4.4.7 Examples

Example 1:

An instruction schema int-expr shall be defined such that any instruction int-expr(e) evaluates the given expression e and returns its value. Expressions are built from constants denoting values, variables, and some unary and binary operations defined for the set of values.

The abstract syntax for expressions may be specified as follows:

Sets of elementary objects:

is-$\hat{}$const	set of constants
is-$\hat{}$var	set of variables
is-$\hat{}$unary-rt	set of unary operators
is-$\hat{}$bin-rt	set of binary operators

The set of expressions is defined by:

is-expr = is-const ∨ is-var ∨ is-bin ∨ is-unary

is-bin = (<s-rd1:is-expr>,
<s-rd2:is-expr>,
<s-op:is-bin-rt>)

is-unary = (<s-rd:is-expr>,
<s-op:is-unary-rt>)

Furthermore, the following functions and instructions are assumed to be defined:

value(c)	function which yields the value denoted by a given constant c
content(v, ξ)	function which yields the value of a given variable v for the state ξ of the machine

---

Key to abbreviations:

s-rd1	select operand one
s-rd2	select operand two
s-rd	select operand
s-op	select operator

int-bin-op(op,a,b) instruction which returns the value corresponding to the application of the binary operator op to a and b, where a and b are values

int-un-op(op,a) instruction which returns the value corresponding to the application of the unary operator op to a, where a is a value

The instruction schema int-expr may now be defined as follows:

```
int-expr(e) =
   is-bin(e)    ⟶ int-bin-op(s-op(e),a,b);
                     a:int-expr(s-rd1(e)),
                     b:int-expr(s-rd2(e))
   is-unary(e)  ⟶ int-un-op(s-op(e),a);
                     a:int-expr(s-rd(e))
   is-var(e)    ⟶ PASS:content(e,ξ)
   is-const(e)  ⟶ PASS:value(e)
```

Whether an instruction int-expr(e) is value-returning or self-replacing depends in this example only upon the argument e and not upon the state ξ of the machine. The first two alternatives of an instruction int-expr(e) are self-replacing, i.e. a control tree is specified. The last two alternatives are value-returning with no further changes of the state specified. The value returned for the alternative is-var(e) depends on the specific variable and the state ξ of the machine.

The following example is only to illustrate the expansion and reduction of the control thought of as actually working. It is, however, by no means recommended that more complicated examples in a more complicated machine environment should be attempted in a similar manner. In particular, the test whether a given instruction definition works for all cases as intended must be based on a more general argument than just a few test examples. For the example to be worked out in the sequel it is assumed that the values are natural numbers, that the operators for addition (+) and multiplication (*) are available and that x_1, x_2 are variable names. The expressions to be interpreted will be represented in infix notation and enclosed in quotes to distinguish them from the metalanguage. It is assumed that value ('1') = 1, value ('2') = 2 and so on. Furthermore, it is assumed that content (x_1,ξ) = 2 and content (x_2,ξ) = 5 for all states ξ to be considered in the example. Suppose now that the instruction int-expr ('(x1 + (x2*3))') is in some terminal position; the problem is to show how this position expands and contracts upon execution of the above instructions.

(1) int-expr('(x1 + (x2*3))')

(2) If e = '(x1 + (x2*3))' then

is-bin(e),rd1(e) = 'x1',rd2(e) = '(x2*3)' and op(e) = '+'.

Therefore, the instruction expands to:

```
int-bin-op('+',a,b);
   a:int-expr('x1'),
   b:int-expr('(x2*3)')
```

(3) There is now a choice either to execute int-expr('x1') or to expand int-expr('(x2*3)'). Doing the expansion with some suitable changes of the dummy names results in:

```
int-bin-op('+',a,b);
   a:int-expr('x1'),
   b:int-bin-op('*',a1,b1);
        a1:int-expr('x2'),
        b1:int-expr('3')
```

(4) The order of evaluation of all the terminal instructions is now irrevelant for this special example. The execution of all the terminal instructions is therefore done simultaneously:

```
int-bin-op('+',2,b);
   b:int-bin-op('*',5,3)
```

(5) Execution of the only terminal instruction results in:

```
int-bin-op('+',2,15)
```

The above instruction will return the value 17 to the argument places specified for its position in accordance with the assumptions.

Example 2:

For use in the next example an instruction schema pass is defined such that any instruction pass(x) returns x as its value:

```
pass(x) =
   PASS:x
```

There is only one group in the above definition, and this group is a value-returning alternative. No other state transitions than those implied by the value-return mechanism are performed upon execution of pass(x).

Example 3:

Consider a non-empty list of expressions, e-list, and the problem of computing a corresponding list, v-list, where each expression is replaced by its value. The order of evaluation of the individual expressions is to be left unspecified. The problem will be solved by defining an instruction schema int-expr-list, which will, for any specific choice of the expression list e-list, define the control tree which solves the problem for the specific e-list. The control tree will actually upon execution return the result, i.e. the list of values corresponding to e-list.

The instruction schema int-expr-list may be defined as follows:

```
int-expr-list(e-list) =
   pass(v-list);
      {elem(i)(v-list):int-expr(elem(i,e-list)) | 1≤i≤length(e-list)}
```

There is again only one group, but this time it is a self-replacing alternative, i.e. a control tree representation. e-list is the only parameter of the instruction schema, v-list is a dummy name used to designate the argument place where the value list is being constructed, and i is the bound variable of the implicit definition of the successor set of pass(v-list).

Executing the instruction int-expr-list(<e_1,e_2>) with the specific list of expressions <e_1,e_2> as argument, this instruction will be replaced in the control part by the following control tree:

pass(v-list);
elem(1)(v-list):int-expr(e_1),
elem(2)(v-list):int-expr(e_2)

Upon further computation there is the choice of e_1 or e_2 to be evaluated first. The result of evaluating e_i will be substituted in the elem(i)-component of the argument of pass(v-list), which is, of course, initially Ω. The instruction pass(v-list) will eventually pass the list of values so constructed into the argument place specified for its position.

Example 4:

An instruction schema merge is defined such that any instruction merge(x,y,k), where the arguments x and y are objects and the argument k is a set of selectors, returns an object built from x by substituting the $\varkappa$ components of y for $\varkappa \in k$:

merge(x,y,k) =

PASS:μ(x; {<$\varkappa$: $\varkappa$(y)> | $\varkappa \in k$})

Example 5:

The problem is to replace for a given PL/I data attribute, roughly speaking, the expressions which occur in the data attribute (as lower bounds and upper bounds of arrays and string lengths) by their values. A data attribute of PL/I as specified by the abstract syntax of the formal definition is a quite complicated composite object. However, the only properties which are important for the present example are:

(1) that there are components which are expressions and are selected by composite selectors of the forms s-ub$\circ\varkappa$, s-lb$\circ\varkappa$ or s-length$\circ\varkappa$,

(2) that these components are independent of one another, i.e. that no component is part of another one.

Key to abbreviations:

s-lb	select lower bound
s-ub	select upper bound
s-length	select length

The problem may be solved by defining an instruction schema eval-da as follows:

$$\underline{\text{eval-da}}(\text{da}) =$$
$$\underline{\text{merge}}(\text{da}, x, K(\text{da}));$$
$$\{\varkappa(x) : \underline{\text{int-expr}}(\varkappa(\text{da})) \mid \varkappa \in K(\text{da})\}$$
$$\text{where: } K(\text{da}) = \{\varkappa \mid (\exists \varkappa_1)\ (\varkappa = \text{s-ub}\circ\varkappa_1 \vee \varkappa = \text{s-lb}\circ\varkappa_1 \vee \varkappa = \text{s-length}\circ\varkappa_1)\ \&\ \varkappa(\text{da}) \neq \Omega\}$$

The function K(da) yields the set of all selectors pointing to lower bounds, upper bounds and lengths of a data attribute da.

The actual process performed upon computation is to evaluate the expressions for lower bounds, upper bounds, and lengths in some order, to construct an auxiliary object x containing the values of the expressions in the corresponding positions, to merge the given data attribute with the auxiliary object, i.e. to replace the expressions by their values, and finally to return the so modified data attribute.

Consider now the special case of a data attribute da_1 specifying a linear array of floating point numbers. There are two expressions contained in the data attribute which are significant for our example, namely a lower bound and an upper bound. It is assumed that the lower bound is selected by $\text{s-lb}\circ\varkappa_1$ and the upper bound by $\text{s-ub}\circ\varkappa_2$. Therefore the set $K(da_1)$ is:

$$K(da_1) = \{\text{s-lb}\circ\varkappa_1, \text{s-ub}\circ\varkappa_2\}$$

The associated control tree according to the above schema is:

$$\underline{\text{merge}}(da_1, x, \{\text{s-lb}\circ\varkappa_1, \text{s-ub}\circ\varkappa_2\});$$
$$\text{s-lb}\circ\varkappa_1(x) : \underline{\text{int-expr}}(\text{s-lb}\circ\varkappa_1(da_1)),$$
$$\text{s-ub}\circ\varkappa_2(x) : \underline{\text{int-expr}}(\text{s-ub}\circ\varkappa_2(da_1))$$

4.5 Note on Constructs of the State and some Instructions of the Abstract Machine

4.5.1 Unique name generation

It is necessary upon several occasions during the interpretation of a program to name something. For this purpose a mechanism is built into the machine which on request generates a unique name, i.e. a name which is different from all names generated before.

The name generating mechanism may be defined as follows. It is assumed that there is an infinite list of mutually different names $\langle n_1, n_2, n_3, \ldots \rangle$. Furthermore, for any state ξ there is an immediate component s-n(ξ) which is a natural number and points to the unique name of the above list to be used next. The instruction un-name is then used to get hold of the unique name to be used next and to increase the counter:

un-name =
    PASS:$n_{s\text{-}n(\xi)}$
    s-n:s-n(ξ) + 1

There are two reasons for the use of unique names in the state of the machine:

(1) Sharing patterns:

It is frequently the case that certain objects (representing some information) are to be available in two or more parts of the state. If this information is never updated during the interpretation it is sufficient to have copies available in the respective parts of the state. If, however, the information may be updated during the process of interpretation, then the object representing the information is named uniquely and is made available under this name. The following consideration may serve as a simple example of a sharing pattern. Assume that in a certain program two variables x and y are supposed to occupy the same storage, i.e. updating of the value of x means at the same time updating of the value of y and vice versa:

x - value
y - value

Key to abbreviations:

s-n	select name counter
un-name	generate unique name

The mere association of x and y with their values as shown above would not reflect the situation properly. The introduction of one step of indirectness, however, is sufficient in our example to expresss the fact that x and y are supposed to share storage:

```
x - n
y - n        n - value
```

(2) Self-referencing information structures:

Consider an object which in one of its components refers to itself via its name. The process which replaces the name by a copy of the object itself is not terminating, i.e. the resulting structure becomes infinite. The use of a name in those cases is therefore necessary.

As a simple example consider the recursive definition of the function Fact:

$$Fact(n) = (n = 0 \longrightarrow 1,\ T \longrightarrow n*Fact(n-1))$$

The attempt to replace Fact on the right hand side by its definition does not remove the reference to Fact:

$$Fact(n) = (n = 0 \longrightarrow 1,\ T \longrightarrow n*((n-1) = 0 \longrightarrow 1,\ T \longrightarrow (n-1)*Fact(n-2)))$$

4.5.2 Representing functions by objects

Let A be an object defined by the collection of its immediate components, i.e.:

$$A = \mu_o(\langle s_1:A_1\rangle, \langle s_2:A_2\rangle, \ldots, \langle s_n:A_n\rangle)$$

A function f_A may be associated with any such object by:

$$f_A(s_i) = \begin{cases} A_i & \text{if } 1 \le i \le n \\ \Omega & \text{otherwise} \end{cases}$$

i.e. these functions are mappings of selectors of S into objects:

$$f_A : S \longrightarrow \text{Objects}$$

Key to abbreviations:

Fact factorial

All functions associated with objects in the above manner are functions whose domain is the set of selectors and which yield a value $\neq \Omega$ only for a finite set of arguments.

The state of the machine may contain several such mappings represented by the associated objects. The assumption to be made is of course that the domains of such mappings are subsets of the set of selectors.

Two examples may illustrate the application.

(1) The environment component

The version of the environment compnent given in this section will correspond to the needs of the example in chapter 5. The environment component s-env(ξ) of a state ξ is a mapping of the identifiers, which may be referenced in the given state, into unique names. The environment component serves to resolve the scope problem within a block structure as will be explained in the example given in chapter 5. The present problem is to represent environments as objects. For that purpose all identifiers which might occur in a program are assumed to be members of the set of selectors S. If now for a specific environment the identifiers id_1, id_2, ..., id_m are to be mapped into the unique names n_1, n_2, ..., n_m respectively, the mapping is represented by the object ENV:

$$ENV = \mu_o(\langle id_1:n_1\rangle, \langle id_2:n_2\rangle, \ldots, \langle id_m:n_m\rangle)$$

To retrieve the unique name for a given identifier id_i, the identifier is simply applied to the environment, since $id_i(ENV) = n_i$.

Since the environment is the immediate component s-env(ξ) of any state ξ, the question as to which unique name is associated to a given identifier id_i for a given state ξ is answered by:

$$id_i \circ s\text{-}env(\xi)$$

All possible objects which may serve as an environment can be defined by the predicate is-env:

$$is\text{-}env = (\{\langle id:is\text{-}n\rangle \;||\; is\text{-}name(id)\})$$

where: $\hat{is\text{-}n}$ is the set of all unique names and $\hat{is\text{-}name}$ is the set of all possible identifiers.

(2) Directories

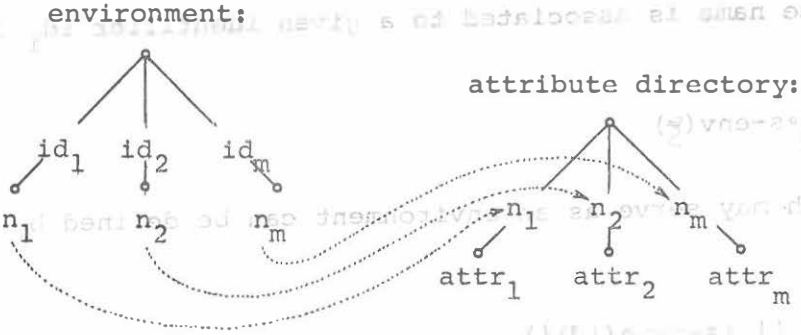
Certain components of states of the machine are directories. By the term directory is understood a mapping of unique names into certain objects. Any pair of a unique name and its associated object is called an entry with respect to the directory. The structure of the object representing a directory is always such that the unique names serve as selectors and yield their associated object when applied to the directory. For example, upon entrance into a block, identifiers declared in that block are first associated with unique names in the environment. For all of those unique names an entry in the so called attribute directory s-at(ξ) is then made associating the unique name with the attributes declared for the corresponding identifier and some additional information.

Given an identifier id and a certain state ξ one may retrieve the attributes of that identifier by:

$$(id \circ s\text{-}env(\xi))(s\text{-}at(\xi))$$

In other words one has first to apply s-env to the state to get the environment; application of the identifier to the environment yields the corresponding unique name which in turn applied to the attribute directory s-at(ξ) yields the desired attributes.

Another principle used in the above example should be noted, namely, in order to associate identifiers with unique names which are furthermore associated with certain attributes, the unique names have been used in a double role, on the one hand as a component of the environment, on the other as a selector in the attribute directory. One may illustrate the situation by means of the following picture:



Key to abbreviations:

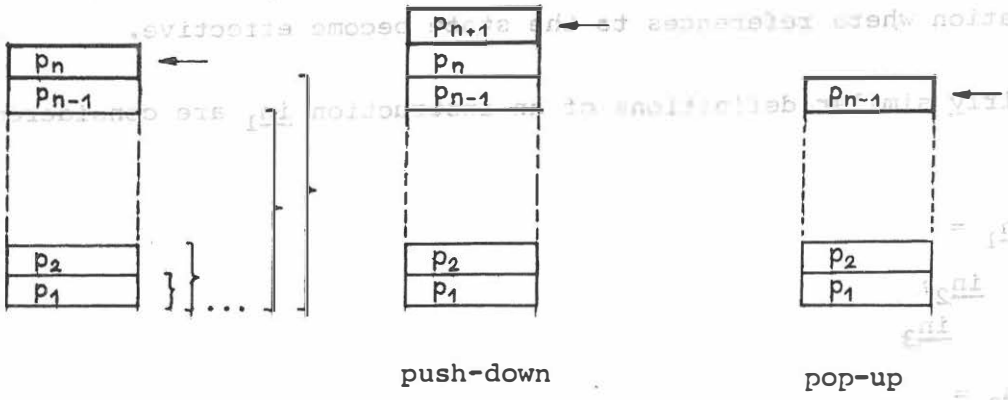
s-at	select attribute directory
s-env	select environment

With the above construct one may introduce as many steps of indirectness as necessary and desired. The construct serves precisely the purpose of indirect addressing known in usual machine programming.

4.5.3 Realization of stacks

Stacks play an important role in the interpretation of programming languages like PL/I because of its nested structure. By the term stack is understood a linear arrangement of elements say $p_1, p_2, \ldots, p_{n-1}, p_n$ where p_n is called the top element. The assumption is that at any point of time only the top element is referred to. The two operations by which a stack is manipulated are: the push down operation which adds a new element p_{n+1} on top of the stack and the pop up operation which deletes the top element from the stack so that p_{n-1} becomes the new top element. This means that it will be useful to let a stack appear as an object having the top element and the rest of the stack as its immediate components.

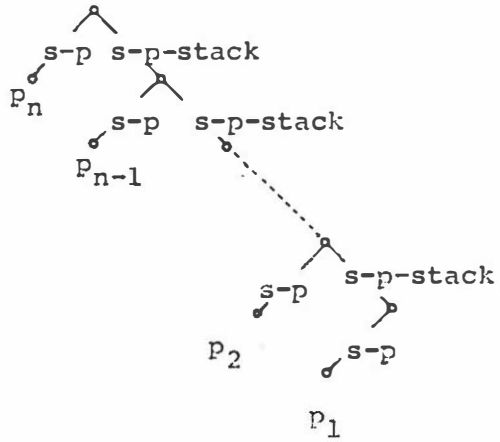
The situation may be visualized by the picture given below.



push-down pop-up

The following schema will be used to represent a stack as an object. The selectors s-p for the top element and s-p-stack for the rest of the stack will be used for the purposes of this section.

ST =



object representing a stack

The stack operations may be formulated as follows:

push down:

$$\mu_o(\langle s\text{-}p\text{:}p_{n+1}\rangle, \langle s\text{-}p\text{-}stack\text{:}ST\rangle)$$

pop up:

$$s\text{-}p\text{-}stack(ST)$$

The members of the class of possible stacks, is-$\hat{p}$-stack, which can be formed from elements of is-$\hat{p}$ may simply be defined by:

is-p-stack = (<s-p:is-p>, <s-p-stack:is-p-stack>) $\vee$ is-Ω

4.5.4 Reference to state components in instruction definitions

It seems worthwhile to mention that the definition of auxiliary instructions is not only a means of introducing abbreviations but may also influence the point in a computation where references to the state become effective.

Two fairly similar definitions of an instruction $\underline{in}_1$ are considered as an example.

(1) $\underline{in}_1$ =

$\underline{in}_2$;

$\underline{in}_3$

$\underline{in}_2$ =

$\underline{in}_4$(s-p(ξ))

where s-p(ξ) refers to some component of the state ξ.

(2) $\underline{in}_1$ =

$\underline{in}_5$(s-p(ξ));

$\underline{in}_3$

$\underline{in}_5$(a) =

$\underline{in}_4$(a)

It is now assumed that in the course of a computation $\underline{in}_1$ is being executed and the problem is to observe the difference of the two definitions of $\underline{in}_1$.

ad (1):

(a) execution of $\underline{in}_1$ means its replacement by the control tree

$$\begin{array}{l}\underline{in}_2\,; \\ \quad \underline{in}_3\end{array}$$

(b) execution of $\underline{in}_3$

(c) execution of $\underline{in}_2$ means its replacement by $\underline{in}_4(s\text{-}p(\xi))$
ξ therefore refers to the state after execution of $\underline{in}_3$ (!)

ad (2):

(a) execution of $\underline{in}_1$ means its replacement by $\underline{in}_5(s\text{-}p(\xi))\,;\ \underline{in}_3$
ξ therefore refers to the state before execution of $\underline{in}_3$ (!)

(b) execution of $\underline{in}_3$

(c) execution of $\underline{in}_5$ which means its replacement by $\underline{in}_4(\ldots)$

This means that in both cases the instructions are executed in the order $\underline{in}_1$, $\underline{in}_3$, ..., $\underline{in}_4$, however, the reference to ξ is a reference to two different states in the above two cases.

4.5.5 The null instruction

The problem is to construct a control tree for instructions $instr_1$, $instr_2$, ..., $instr_n$ to be executed in unspecified order. It is assumed that the instructions do not pass any value or construct a common auxiliary object. To construct a valid control tree for the above problem it is necessary to have an instruction which does nothing else than to delete itself from the control tree. This instruction is called the null instruction and is defined by:

$$\underline{null} = \text{PASS: } \Omega$$

The above problem may now be resolved by:

$$\underline{null};\ \{instr_1,\ instr_2,\ \ldots,\ instr_n\}$$

4.5.6 Pass instructions

The instruction schema pass has already been used in a previous example; its definition is repeated for completeness:

```
pass(x) =
   PASS;x
```

The following problem occurs quite frequently. An auxiliary object x is to be constructed whose components are computed by instructions to be executed in unspecified order . However, it is not the object itself that is to be passed but the result of applying a function f to it.

Since only argument places of an instruction may be occupied by a dummy name in a control tree, the instruction pass is not sufficient to solve this problem. Therefore, a special abbreviation has been introduced to avoid the definition of auxiliary instructions for any special function f:

```
pass-f(x) =
     PASS;f(x)
```

where f may be replaced by any function applicable to x.

The following control tree may now be constructed if $instr_1$, $instr_2$, ..., $instr_n$ compute the $x_1, x_2, \ldots, x_n$ components of the auxiliary object and f_1 is the special function to be applied to it:

$$pass\text{-}f_1(x);\ \{\, x_1(x):instr_1,\ x_2(x):instr_2,\ \ldots,\ x_n(x):instr_n \,\}$$

4.5.7 Element by element evaluation of a list

In section 4.4.10 it has already been shown how a list may be evaluated by evaluating the elements in unspecified order. It will now be shown, how a list may be evaluated element by element in the natural order. The empty list will be included and its evaluation is supposed again to yield the empty list. It is further assumed that the instruction evaluating a specific element el and passing the desired value is eval(el).

The instruction schema which solves the given problem is eval-list and its definition reads:

```
eval-list(list) =
     is-<>(list) ——→ PASS:<>
          T ——→ mk-list(eh, et);
                    et: eval-list(tail(list));
                         eh:eval(head(list))
```

where:

```
mk-list(x,list) =
     PASS:<x>^list
```

Key to abbreviations:

eval-list	evaluate list
is-<>	is-empty list
mk-list	make list
eh	evaluated head
et	evaluated tail
eval	evaluate

The instruction schema which solves the given problem in eval-list is of definition reads:

```
eval-list(list) =
    is-<>(list) ——→ PASS:<>
    T ——→ mk-list(eh, et);
        et: eval-list(tail(list))
        eh:eval(head(list))
```

where:

```
mk-list(x,list) =
    PASS:<x,list>
```

Key to abbreviations:

eval-list	evaluate list
is-<>	is-empty list
mk-list	make list
eh	evaluated head
et	evaluated tail
eval	evaluate

5. DEFINING THE INTERPRETATION OF EPL

This chapter is devoted to the definition of the interpretation of EPL using the notion of abstract machine introduced in the previous chapter. It might be helpful at this point if the reader refers to Section 1.2 in order to refresh his understanding of the general structure and content of EPL and to Section 3.1 for a precise statement of the abstract syntax of EPL.

Sections 5.1 and 5.2 present the formal definition of the semantics of EPL with almost no comments given concerning the formulas. The rest of the chapter is devoted to different comments and the elaboration of consequences of the formal definiton.

5.1 The States of the Interpreting Machine

This section defines the set of states, is-$\hat{}$state, which the interpreting machine can assume. These include the initial state for any given program and the set of end states.

In addition to the sets of elementary objects and the set of selectors specified for the abstract syntax of EPL in Section 3.1, page 3-6, the following sets of elementary objects and selectors are distinguished:

is-$\hat{}$n	infinite set of names (used for the generation of unique names)
{PROC, FUNCT}	two attributes used to distinguish function names and procedure names
{s-env,s-c,s-at,s-dn,s-d,s-n}	selectors for the components of the interpreting machine.

(S1) is-state = (<s-env:is-env>,
<s-c:is-c>,
<s-at:is-at>,
<s-dn:is-dn>,
<s-d:is-d>,
<s-n:is-integer-value>)[1)]

(S2) is-env = ({<id:is-n> || is-id(id)})

(S3) is-c = [2)]

(S4) is-at = ({<n:is-type> || is-n(n)})

(S5) is-$\hat{}$type = { INT, LOG, PROC, FUNCT }

---

1) This is the counter for the unique name generation (see 4.5.1).

2) is-$\hat{}$c is a set of control trees having the properties described in chapter 4., section 4.3.

(S6) is-dn = ({<n:(<s-env:is-env>,
<s-attr:(is-proc-attr ∨ is-funct-attr)>) ∨ is-value> ||
is-n(n)})

(S7) is-d = (<s-env:is-env>,<s-c:is-c>,<s-d:is-d>) ∨ is-Ω

The initial state for any given program t ∈ is-$\hat{\text{progr}}$ is:

$$\mu_0(\langle s\text{-}c:\underline{\text{int-progr}}(t)\rangle,\langle s\text{-}n:1\rangle)$$

The initial state has only a control part which is the instruction int-progr(t). The instruction int-progr is defined in the next section.

States ξ whose control part s-c(ξ) is Ω are end states.

5.2 The Interpretation of the Language

This section defines an instruction schema int-progr whose parameter is a program. The execution of an instruction int-progr(t) specifies, in terms of the abstract machine, the task of the program t.

The following additional notational conventions similar the formal definition of PL/I have been introduced. For better readability abbreviations for the immediate components of a current state ξ have been introduced. The lefthand sides of the following list may always be replaced by the corresponding righthand side.

Key to abbreviations (prefixes are omitted):

env	environment	n	unique name
c	control	d	dump (stack representing the
at	attribute directory		dynamic nesting of block,
dn	denotation directory		procedure and function acti-
id	identifier		vations)

ENV	s-env(ξ)
C	s-c(ξ)
AT	s-at(ξ)
DN	s-dn(ξ)
D	s-d(ξ)

If the case distinctions made in the definition are not exhaustive, i.e. there are cases for which the instruction definition is undefined, then this is indicated by an additional final line in the definition:

T ⟶ error.

The definition of instruction schemata will be given in the following format:

(Ii) DEFINITION OF INSTRUCTION SCHEMA

where: LIST OF ABBR. LOCAL TO THIS DEFINITION
for: RANGES OF ARGUMENTS OF THE SCHEMA DEFINED
Ref.: REFERENCES
Note: ADDITIONAL NOTES

Any of the last four items may be omitted.

The following functions and instructions are not further specified:

convert(v,attr)	the function yields v converted (if necessary) to the type specified by attr which may either be INT or LOG.
int-bin-op(op,a,b)	Instruction which returns the result of applying the operator op to a and b. It is left open whether there is a conversion performed in case the operator is not applicable to operands of type a and b.
int-un-op(op,a)	Instruction which returns the result of applying the operator op to a (for the problem of conversion see above).
value(a)	Function which yields the value given a constant a.

(I1) int-progr(t) = int-block(t)

for : is-progr(t)

(I2) int-block(t) =

s-d:μ_o(<s-env:ENV>,<s-c:C>,<s-d:D>)
s-c:exit;
int-st-list(s-st-list(t));
int-decl-part(s-decl-part(t));
update-env(s-decl-part(t));

for: is-block(t)

(I3) update-env(t) =

null;
{update-id(id,n); n:un-name | id(t) ≠ Ω}

for: is-decl-part(t)
Ref.: null (see 4.4.5), un-name(see 4.4.1)

(I4) update-id(id,n) =

s-env:μ(ENV;<id:n>)

for: is-id(id), is-n(n)

(I5) int-decl-part(t) =

null;
{int-decl(id(ENV),id(t)) | id(t) ≠ Ω}

for: is-decl-part(t)
Ref.: null (see 4.4.5)

(I6) int-decl(n,attr) =

is-var-attr(attr) → s-at:μ(AT;<n:attr>)
is-proc-attr(attr) → s-at:μ(AT;<n:PROC>)
s-dn:μ(DN;<n:μ_o(<s-attr:attr>,<s-env:ENV>)>)
is-funct-attr(attr) → s-at:μ(AT;<n:FUNCT>)
s-dn:μ(DN;<n:μ_o(<s-attr:attr>,<s-env:ENV>)>)

for: is-n(n), is-attr(attr)

(I7) int-st-list(t) =

is-<>(t) ⟶ null

T ⟶ int-st-list(tail(t));
int-st(head(t))

for: is-st-list(t)
Ref.: null (see 4.4.5)

(I8) int-st(t) =

is-assign-st(t) ⟶ int-assign-st(t)

is-cond-st(t) ⟶ int-cond-st(t)

is-proc-call(t) & (at_t = PROC) ⟶ int-proc-call(t)

is-block(t) ⟶ int-block(t)

where: at_t = ((s-id(t))(ENV))(AT)
for: is-st(t)

(I9) int-assign-st(t) =

is-var-attr(n_t(AT)) ⟶
assign(n_t,v);
v:int-expr(s-right-part(t))

T ⟶ error

where: n_t = (s-left-part(t))(ENV)
for: is-assign-st(t)

(I10) assign(n,v) =

s-dn:μ(DN;<n:convert(v,n(AT))>)

for: is-n(n), is-value(v)
Ref.: convert (not specified further)

(I11) int-cond-st(t) =

branch(v,s-then-st(t), s-else-st(t));
v:int-expr(s-expr(t))

for: is-cond-st(t)

(I12) branch(v,st1,st2) =

convert(v,LOG) ⟶ int-st(st1)
¬convert(v,LOG) ⟶ int-st(st2)

for: is-value(v), is-st(st1), is-st(st2)

(I13) int-proc-call(t) =

(length(arg-list$_t$) = length(p-list$_t$)) ⟶

s-env:μ(env$_t$;{<elem(i,p-list$_t$): elem(i,arg-list$_t$)(ENV)> | 1≤i≤length(p-list$_t$)}

s-d:μ_o(<s-env:ENV>,<s-c:C>,<s-d:D>)

s-c:exit;
int-st(st$_t$)

T ⟶ error

where: n_t = (s-id(t))(ENV), p-list$_t$ = s-param-list∘s-attr∘n_t(DN),
env$_t$ = s-env∘n_t(DN), arg-list$_t$ = s-arg-list(t),
st$_t$ = s-st∘s-attr∘n_t(DN)

for: is-proc-call(t)

(I14) exit =

s-env:s-env(D)

s-c :s-c(D)

s-d :s-d(D)

(I15) int-expr(t) =

is-bin(t) ⟶ int-bin-op(s-op(t),a,b,);
a:int-expr(s-rd1(t)),
b:int-expr(s-rd2(t))

is-unary(t) ⟶ int-un-op(s-op(t),a);
a:int-expr(s-rd(t))

is-funct-des(t) & (at$_t$ = FUNCT) ⟶
pass-value(n);
int-funct-call(t,n);
n:un-name

is-var(t) & is-var-attr(n_t(AT)) ⟶ PASS:n_t(DN)

is-const(t) ⟶ PASS:value(t)

T ⟶ error

where: n_t = t(ENV), at_t = ((s-id(t))(ENV))(AT)
for: is-expr(t)
Ref.: value, int-bin-op and int-un-op are not further specified;
un-name (see 4.4.1)

(I16) pass-value(n) =

PASS:n(AT)

(I17) int-funct-call(t,n) =

(length($arg\text{-}list_t$) = length($p\text{-}list_t$)) ⟶

s-env:μ(env_t;{<elem(i,$p\text{-}list_t$):elem(i,$arg\text{-}list_t$)(ENV)> | 1≤i≤length($p\text{-}list_t$)}

s-d:μ_o(<s-env:ENV>,<s-c:C>,<s-d:D>)

s-c:exit;
assign(n,v);
v:int-expr($expr_t$);
int-st(st_t)

T ⟶ error

where: n_t = (s-id(t))(ENV), $p\text{-}list_t$ = s-param-list∘s-attr∘n_t(DN),
env_t = s-env∘n_t(DN), $arg\text{-}list_t$ = s-arg-list(t),
st_t = s-st∘s-attr∘n_t(DN), $expr_t$ = s-expr∘s-attr∘n_t(DN)

for: is-funct-des(t), is-n(n)

Note: the definition is almost identical with (I13)

5.3 Intuitive Description Based upon the Formal Definition

5.3.1 The Components of the State

The Environment and Dump

The environment associates identifiers with unique names. For any state ξ the environment component contains all those identifiers which can possibly be referred to in this state. The associated unique name of any identifier gives access to the meaning of the identifier in the present state via the corresponding entries in the denotation directory and attribute directory. For example, if a given identifier is a variable then the corresponding unique name is associated with the value of the variable by an entry in the denotation directory and with the type of the variable in the attribute directory.

To gain some insight into the significance of the block structure of the language one may inspect all instructions which replace or modify the environment component of the state (see for the occurences of s-env). Initially, the environment is Ω.

Since a program in the given language is a block, the first action is to activate this block /see (I1), (I2)/. Activating a block means putting a copy of the current environment and the current control on top of the dump and installing a new control part. The next instruction executed updates the environment /see (I2), (I3), (I4)/. This means that for any identifier id declared in the declaration part of the block a unique name n is generated /see (I3)/ and a component n with id as selector is built into the environment by means of the μ function. If the identifier is already present then the corresponding component is overwritten by the new n. Exit from the block /see (I2), (I14)/ the environment and the control of the top element of the dump are reinstalled as the current environment and control and the dump is popped up.

During the execution of a block any further activation of a block and also any call of a procedure /see (I13)/ and any call of a function /see (I17)/ causes a change of the environments. However, in all of the above cases, a copy of the environment is put on top of the dump before it is changed and it is reinstalled upon return. In other words, there is an environment established for any activation of a block, procedure or function and the environment remains constant for this activation of the block, procedure or function.

Interpretation of a block /see (I2)/ means updating of the environment and afterwards interpreting the declaration part. In the interpretation of procedure or function declarations, the environment becomes part of the denotation entered into the denotation directory for the procedure or function name /see (I6)/. Upon activation of a procedure or function the environment which is part of the denotation is installed and updated, i.e. the environment which was valid at the time of the declaration of the procedure, /see (I13), (I17)/. In other words, the meaning of the global variables which occur in a procedure or function declaration is frozen at the time when the declaration is interpreted.

The above technique may be generalized as follows. Given some piece of text whose meaning depends on the meaning of the references to certain identifiers in it, the meaning of this text may be determined by associating the text with an environment containing the respective identifiers.

The argument passing performed upon any procedure or function call consists simply in associating the parameters with the unique names of the arguments /see (I13), (I17)/, and thus making the parameters "synonymous" with the arguments.

An identifier which is not declared yields Ω when applied to the environment. Any application of Ω to either the denotation directory or attribute directory is undefined because Ω is not a valid selector.

The Denotation Directory

The denotation directory associates unique names n with a denotation dn. A specific entry into the directory will be symbolized by:

n — dn.

What the denotation for a specific case is, depends on the type of the name. In the present example, the following cases occur:

variables:	n — value
functions:	n — (param-list,st,expr),env [1]
procedures:	n — (param-list,st),env [1]

1) The parentheses and commas indicate the structure of the object. The precise definition of the objects can be taken from the abstract syntax of the state /see (S6)/.

Entries are made upon activation of a block in the case of functions and procedures and upon assignment in the case of variables. The entries for functions and procedures are never changed or deleted. The values associated with a variable name may change upon execution of an assignment to that variable /see (I6), (I10)/. The value of a function is also returned via an auxiliary entry to the denotation directory /see (I15), (I17)/.

The Attribute Directory

The attribute directory associates unique names with the type of these names. The following cases occur in the present example.

interger variables	:	n —— INT
logical variables	:	n —— LOG
procedures	:	n —— PROC
functions	:	n —— FUNCT

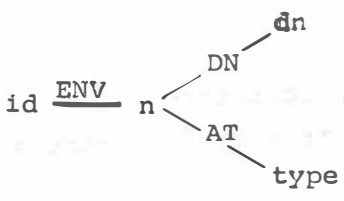
The abstract syntax does not reflect the fact that identifiers of a certain type may only be referenced in a context compatible with their type. This fact is, however, expressed in the interpreter by suitable checks. The check is made in (I8) for type procedure, for type function and type variable in (I15).

Entries into the attribute directory are only made upon activation of a block and never deleted or changed.

5.3.2 Types of Identifiers and their Dynamic Significance

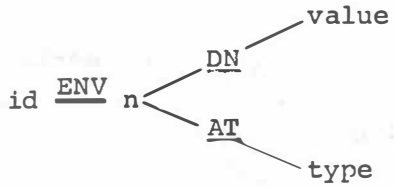
This section answers the question as to which types of identifiers exist in the language and what is the meaning associated dynamically with identifiers. The possible sharing patterns will be discussed at the end of this section.

As explained previously, any identifier, id, that can be referred to at a certain state ξ is associated with a unique name n via the environment component. This unique name is then furthermore associated with a denotation dn in the denotation directory and with a type in the attribute directory. This situation can be symbolized as follows:



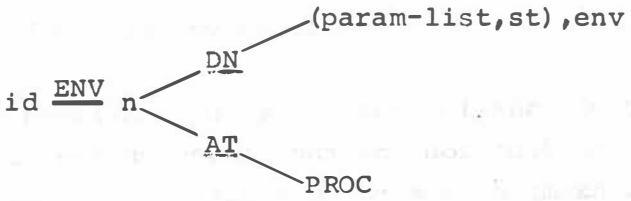
The following cases occur in the present example:

variables:

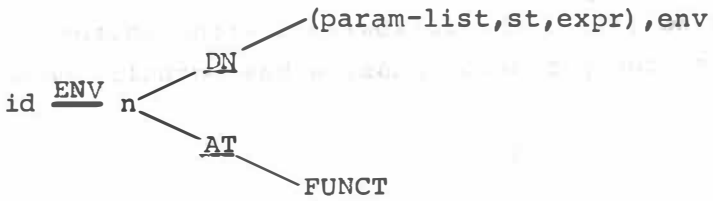


where: type is either INT or LOG

procedures:



functions:



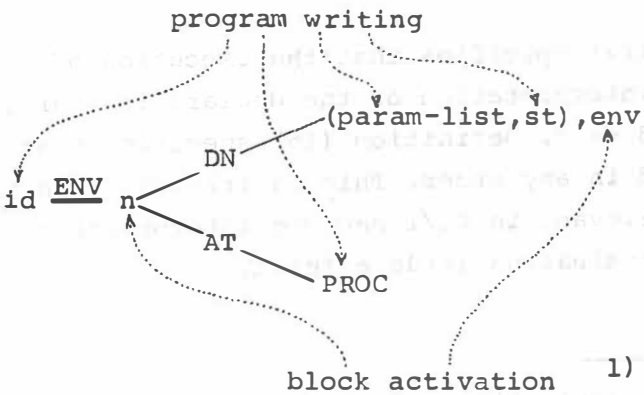
The following two important questions may be distinguished:

(1) When are the entities of the above information structures created?

(2) When are the diverse associations established and disolved?

The above two questions are quite informative when applied to the formal definition of PL/I. The answers for the present example are comparatively simple. The case of procedures is selected as illustration:

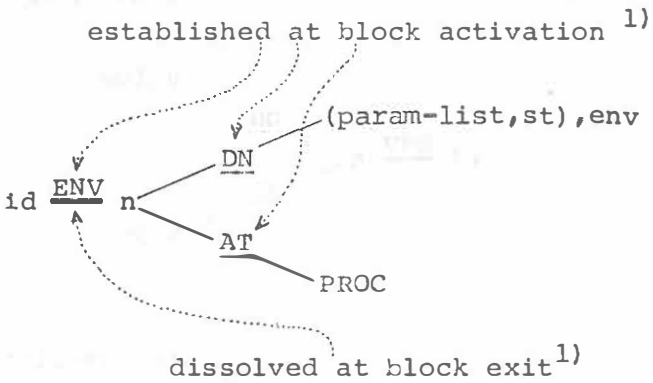
(1):



1)

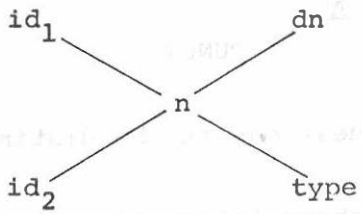
1) The terms block activation and block exit refer to the block in which the declaration of id occurs.

(2) :



There is only one sharing pattern to be mentioned, namely between an argument of a procedure or function and the corresponding parameter. Again, the situation in the present example is extremely simple as compared to PL/I.

Consider a procedure call where an argument id_1 is passed to a parameter id_2. According to (I13), the following sharing pattern via the unique name of id_1 will exist after the parameter passing has actually been performed:



5.3.3 Flow of Control

In the previous chapter, the dynamic significance of identifiers has been isolated. This section will draw attention to another aspect of the language, namely the order in which the actions specified by a program are taken.

Definition (I2) specifies that the execution of a block means updating of the environment, interpretation of the declaration part, interpretation of the statement list and exit. Definition (I5) specifies that the individual declarations may be interpreted in any order. This is irrelevant in the present case; it becomes, however, relevant in PL/I because interpretation of declarations may involve expression evaluation (side effects).

---

1) The terms block activation and block exit refer to the block in which the declaration of id occurs.

Interpretation of the statement list means, as specified in (I7), interpreting the list element by element in the given order.

Next, one has to consider the individual statements. Starting with the assignment statement definition, (I9) specifies that the expression has to be evaluated and the resulting value is assigned to the variable on the lefthand side of the assignment. The operands of an expression may be evaluated in any order according to definition (I15). One should note that there are, however, expressions where the choice of an order is relevant to its meaning. Upon the activation of a function in the course of expression evaluation one should note that the current control is put into the dump and a new control is installed according to (I17). This means that the evaluation of other operands is temporarily stopped until the exit of the function. A simple example may illustrate the situation.

Consider an expression:

$$f_1(x) + f_2(y) * f_3(z).$$

The operands of this expression may be evaluated in any order, e.g. $f_3(z)$, $f_1(x)$, $f_2(y)$. The action of the evaluation of any two of the functions may not, however, be interleaved. The interpretation of the conditional statement is trivial and given in (I11).

The interpretation of a procedure call causes the control again to be dumped. In the case of procedures this is, however, irrelevant.