Use of Design Criteria

for Intermediate Languages

by
Ole Lehrmann Madsen,
Bent Bruun Kristensen
and
Jørgen Staunstrup

DAIMI PB-59
August 1976.

Use of Design Criteria for Intermediate Languages

by

Ole Lehrmann Madsen,
Bent Bruun Kristensen, and
Jørgen Staunstrup

Abstract:
A number of design criteria for intermediate languages are proposed. The intermediate language is viewed as an interface between the (syntax) analysis and the synthesis (code generation). To illustrate the ideas we propose an intermediate language for an existing high level language. The translation from the source language to the intermediate language is defined formally by an affix translation grammar.

TABLE OF CONTENTS

Introduction.

Over the past few years a number of intermediate languages or abstract machines have been proposed in order to serve various purposes in the implementation of high level languages. In this paper we shall try to evaluate some of these proposals and set up a number of design criteria for such intermediate languages.

In a recent paper [10] it is argued that it is unlikely that the same intermediate form will be suited both for immediate interpretation and for code generation. We shall emphasize the code generation aspect in order to make our intermediate language a powerful tool for the implementation of efficient high level language processors.

To illustrate our ideas we present a proposal for an intermediate language for the systems programming language Platon. Platon has been designed, implemented, and used by The Regional EDP Center, University of Aarhus. The language bears some resemblance to Pascal [14], and it is specially suited for describing a dynamic system of concurrent processes and the communication between them. Platon is described in detail in [8,9].

Section 1 summarizes our views on traditional compiler architecture. In section 2 our design criteria and their justification are presented. In section 3 a few well known proposals for intermediate languages are considered, using our design criteria. In section 4 our proposal for an intermediate language is presented. Finally the appendix contains a formal definition of the translation from Platon source programs to the intermediate form. This is given in the form of an affix translation grammar. The full understanding of section 4 requires some knowledge of Platon and the appendix requires in addition some knowledge of affix translation grammars. Throughout the whole paper it is assumed that the reader is familiar with compiler techniques, especially code generation.

1 Compiler Organization.

The aim of this work is to investigate one way of simplifying the compilation of programs written in high level languages. It is the authors' belief that many transformations can be simplified by dividing them up into a number of sub-transformations. (Well known examples are the construction of an executable program from an abstract one, sorting a table by sorting pieces first, and proving a mathematical theorem by proving a number of lemmas first.) We advocate a similar approach for the compilation process i.e. rather than doing the single logical phases (lexical analysis, syntax analysis, code generation, etc) intermixed in a single pass, these should be distributed on a number of passes. This idea is not new, compilers organized in this way are usually called multi-pass compilers, a succesful example is [11]. Our distinction between pass and phase is the following:

A *pass* is a scan of the whole program and a transformation of it.
A *phase* is a collection of manipulations which are logically connected.

The distribution of the various phases of a compiler on the passes has strong influence on its performance. Below we will describe how and why we suggest the distribution to be done. We emphasize that the division should fulfil the following requirements:

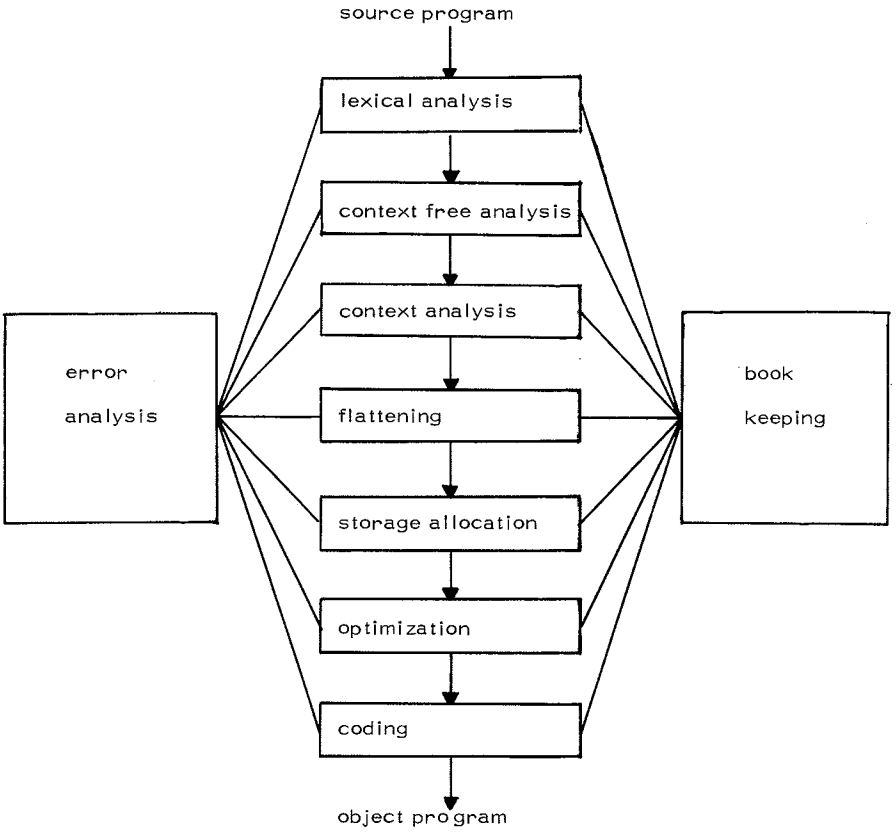
- The total amount of work used in constructing a compiler must not be greater than for a one-pass compiler.

- It should be possible to automate the single phases of the compiler. Thus we obtain more correct language processors, because the compiler, the source language, and the target language are more precisely described.

- The size (core store requirement) of the compiler should smaller than that of a traditional compiler and thus making it useable on smaller machines.

- The phases which are not dependent on the target machine should be portable.

The following diagram shows what we consider to be the distinguishable phases of a traditional compiler:



The *lexical analysis* phase transforms the source program (a sequence of characters) into a sequence of tokens (symbols).

The *context free analysis* phase arranges the tokens into a syntax-tree. This phase is often referred to as parsing.

The *context analysis* phase performs the rest of the analysis which can be statically ascertained. This phase is often referred to as static semantics.

The *flattening* phase transforms the tree structure into some intermediate linear form.

The *storage allocation* phase is the process of assigning storage space to the data items of the program.

Optimization is a general term for a number of manipulations which are done to reduce the execution time and the size of the object program.

The *coding* phase is the translation of the intermediate form into the machine code of the target machine.

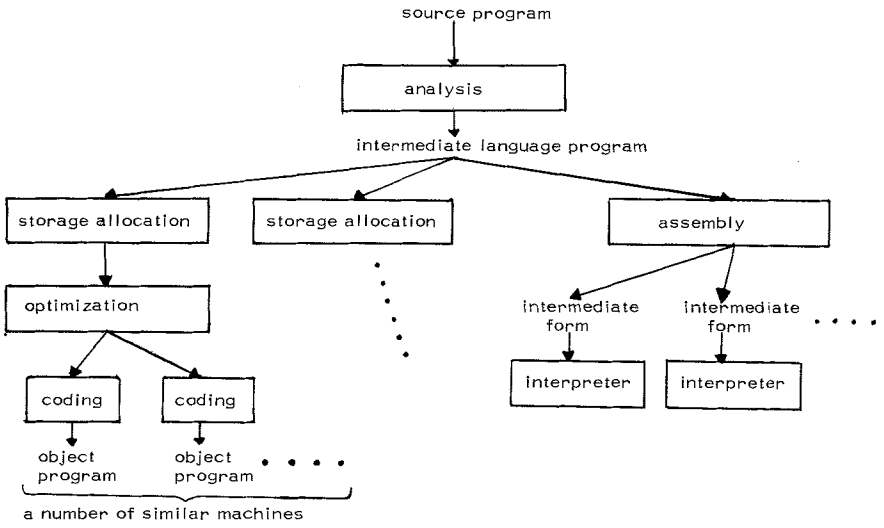
The *bookkeeping* phase administers information retrieved from the program.

The *error analysis* phase handles possible errors reported by the various phases.

We will illustrate how a real programming language can be implemented by designing an intermediate language as a platform for the first partition. We will describe how this intermediate language is designed to meet the goals listed above.

This intermediate language is the interface between the first and the second parts of the compiler.

In this paper we have concentrated on the first part and its interface to the rest of the compiler, but we think it will be possible to treat the other part in a similar fashion. The first part includes lexical analysis, context free analysis, context analysis, and flattening. In the sequel it will be referred to as the *analysis-phase.* These phases belong together because they are all concerned with analysis of the source program structure, furthermore they constitute a machine independent and automatable unit. We imagine that the intermediate language is primarily used for code generation. But after a (simple) transformation (an assembly including storage allocation) it should be possible to interpret the intermediate language. Our approach can be illustrated by the following diagram.


source program
analysis
intermediate language program
storage allocation
storage allocation
assembly
optimization
intermediate form
intermediate form
coding
coding
interpreter
interpreter
object program
object program
a number of similar machines

2 Design Criteria.

In this section we will try to give a more explicit formulation of the design criteria for our intermediate form.

2.1 Source Program Structure.

No information which can be useful in later phases should be thrown away:

- The control structure must be maintained.

For example, repetition statements must not be converted into an equivalent form using explicit jumps, making confusion with other jumps possible.

- The operand structure must be maintained.

For example, the denotation of an array element must not be converted into a sequence of arithmetic instructions, making it indistinguishable from other arithmetic expressions. Furthermore, compile-time evaluation of expressions with constant operands must not be performed in this phase. We do not argue that such optimizations should not be done, but that these should not be done in the analysis phase.

In order to enable later phases to refer to declared quantities by their names:

- The symbolic names and the declarative information must be preserved.

The declarative information can be kept in two ways: either by keeping the symbol table (or a transformation of it); or by generating explicit declare instructions.

2.2 Machine Independence.

In order that the the first phase of the compiler be portable it is important that it is machine independent. This implies that it must be independent of both the machine which executes the first phase and the target machine. Therefore the following requirements must be made:

- No storage allocation should be done.

- No machine dependent expressions should be evaluated.

- The intermediate language must be in symbolic form.

2.3 Compiler Structure.

It is very important to find the optimal point for the split between the analysis and the subsequent phases. Both doing too little and doing too much in the analysis phase will impair the desired reduction in the total costs.

As shown in the diagram above our suggestion is that the analysis-phase does all the syntax analysis (including scope- and type-checking). But for many languages the context dependent syntax must be divided into two: in the following referred to as "machine dependent syntax" and "machine independent syntax". This distinction is often necessary when the interpretation of a symbol is dependent on the target machine. As an example the following construct from Pascal can be mentioned:

A subrange specification consists of a lower and an upper bound separated by .. e.g. a..b. Furthermore the lower bound must be less than the upper bound, consequently the subrange specification '0'..'b' is only legal on some machines (namely those where the digits precede the letters in the character set). Since this is a static property it should be included in the syntax check. On the other hand the checking is machine dependent. Consequently we will classify this construct among those with a machine dependent syntax.
We conclude that the following requirement must be made:

- All machine independent syntactic checks have been done.

In order to ligthen the analysis and the subsequent phases it must be required that:

- The intermediate language must have a simple and uniform syntax,

and that:

- The retrieved information must be directed to the context where it is needed.

As already mentioned the individual compiler phases must be kept small and simple. This is of course also the case with the analysis phase.

- The analysis phase must be kept small and simple.

Furthermore it is possible to specify the transformations made by the analysis phase formally. Thereby it is possible to automate the construction of this part of the compiler. This point is elaborated in the appendix.

- The construction of the analysis part should be automated.

At this point it must be emphasized that we imagine the intermediate language to be very dependent on the source language. We do *not* suggest a common intermediate language for all (or a number of) source languages. Furthermore we do not consider our intermediate form to be an abstract machine (with registers, storage structure etc.) although of course any language can be considered as defining an abstract machine.

3 Comparison with other Intermediate Forms.

Much work has been done recently on developing intermediate languages or abstract machines to be used for the implementation of high level languages. Such intermediate languages have mostly been defined as an aid in the bootstrapping of self-compiling compilers (e.g. O-code [3], and P-code [5]). Therefore most of these intermediate languages are designed for straightforward interpretation. Since bootstrapping is not the subject of this paper, we will not try to judge these languages from this standpoint (which is a bit unfair, since they have been rather successful for this purpose). Instead we will estimate how they meet the goals discussed in the previous section. Most intermediate language designers claim that the first interpreter can be replaced by an efficient code generator, when the initial bootstrap-phase is finished. It must therefore be reasonable to analyze how these intermediate languages are suited for code generation. We have examined the proposals made in the following reports [1,2,3,4,5,6,7].
The list of languages discussed is by no means exhaustive.

Among these IMP [6] is the language which is closest to our design criteria, e.g. symbolic names are kept and no storage allocation is performed. On the other hand IMP has a very scrolled syntax. Furthermore IMP is a stack-machine which makes efficient code-generation difficult.

In O-code and P-code the control structure of the source language program has been lost. This is because all repetition and conditional statements have been converted into explicit jumps. Hereby it becomes difficult to generate efficient code, not to mention optimizations (see for example [13 sec 17.4]). Similarly these languages throw away the operand structure i.e. it becomes difficult for the code-generator to distinguish between the various kinds of operands: arrays, parameters, simple variables etc.

In order to make the intermediate form suited for interpretation, storage allocation is performed, this is, for example, the case both in O-code and P-code. This makes efficient storage utilization on the target machine very difficult.

In the intermediate languages considered all syntactic and declarative checks have been done. But many of the compilers for these languages have done "too much", for example storage allocation and some optimization. Furthermore most of these languages have a rather complicated structure e.g. many different formats and tricky features. This leads to compilers of the same size as a one-pass compiler (doing the entire compilation). For example the Zurich P-code compiler is a Pascal program of 3500 lines, and the original Pascal

compiler for the CDC6400 is 4100 lines.

Most of the compilers we have analyzed generate symbolic output and therefore movement to other machines should be possible. But various machine dependent quantities have been calculated. Usually literals have been evaluated i.e. the *symbols* 2, 5.14, and "ab" are converted into integer, real, and string *values* respectively. Normally the conversion of integers is harmless, but the evaluation of booleans, reals, and strings is usually irrepairable. This is of course only a problem if the program is going to be executed on a machine which is different from the one which performs the compilation. To be completely transportable the literals must be kept in symbolic form.

The most serious problem with all the intermediate languages we have seen is that they do not lead to the desired reduction in the total compilation costs. As already mentioned a badly designed intermediate language (interface) may very well result in a compiler where each phase is of the same size as a full one-pass compiler.

4 An Intermediate Form for Platon.

4.1 Choice of an Intermediate Form.

To illustrate the effect of our design criteria we present a proposal for an intermediate form for the language Platon.

First we discuss our choice of the type of an intermediate form; secondly, given the type, we work out the details of the intermediate form for the particular language Platon. Designing an intermediate form meeting the stated design criteria may lead to several distinct proposals, as it includes many arbitrary decisions.

The following general types of intermediate forms have served as the basis for the selection :

- an abstract machine
  (stack machine, general register machine),
- an expression or polish string (prefix,postfix),
- tuples (triples, quadruples),
- a tree.

For all these types it is possible to meet the stated criterion concerning the source program structure.

Since an abstract machine implies that some kind of storage allocation has been done it is difficult to obtain a machine-independent and efficient implementation using this type of intermediate form.

All the listed types of intermediate forms can meet the goal concerning machine independent syntactic checking. Through a discussion of the suitability of the types in both the analysis phase and the subsequent phases, we determine the type to be preferred :

- The tree structure is a suitable form for the analysis phase and is often also desirable for the subsequent phases. One big coherent tree structure is impractical; a sequence of independent minor trees is to be preferred. However, a tree structure has to be linearised (and later restored) at least once in the translation process, which makes the tree a rather troublesome type.

- The postfix expression implies a simple analysis phase (a postfix simple syntax-directed translation), whereas the prefix expression may be more complicated (requiring a generalized translation scheme). In the subsequent phases the prefix expression may be preferable to the postfix expression (a

simplified pseudo-evaluation). However, both types are linearised trees without explicit references. This means that an establishment of the underlying tree requires at least a pseudo-evaluation using a stack.

- A sequence of tuples can be considered as a representation of a sequence of trees. We will distinguish between *postfix tuples* and *prefix tuples*. In prefix tuples the tuple representing the root precede the tuples representing the sub-trees (top-down). In postfix tuples the order of the root and the sub-trees is reversed (bottom-up). As with the postfix expressions the use of postfix tuples leads to a simple analysis phase. In the subsequent phases the prefix tuples seems preferable because of the immediate access to the root. However, in order to utilize an underlying tree we need access to the entire tree. Consequently the postfix tuples are equally powerful if the root is somehow easy to retrieve.

We conclude that the following mixture of prefix and postfix tuples is a reasonable compromise: each construct is translated into a sequence of postfix tuples initiated by a tuple giving the reference to the root.

Working out the particular intermediate form, the requirements for a simple and uniform syntax is complied with as follows :
All tuples have the form:

(operator,argument*1*,...,argument*n*)

with a symbolic operator and with an argument being either a tuple number, a symbolic name, or a symbolic constant. An operator is restricted to have an arbitrary but fixed number of arguments.

It follows that our intermediate form can be interpreted as a sequence of trees represented as postfix tuples, where each tree is headed by the root represented as a prefix tuple. Each tree corresponds roughly to an expression.

The distribution of the tuple number of the root of the underlying tree as an argument to the tuples where it is needed, is one way of directing the retrieved information to the appropriate context. Furthermore in the nested sequencing structures the nesting depth is given as an argument to the tuples associated with the structure.

The analysis schema in section 4.3 shows how each language construct is transformed. The precise set of tuples constituting the intermediate form and the use of the various tuples may be found in this schema.

The "meaning" of the intermediate language instructions is not given explicitly, since there is a one to one correspondence with the equivalent Platon constructs. Whenever this correspondence is not obvious a comment is included giving an informal prose definition of the meaning.

The notion of constant expressions in Platon gives rise to constructs of which the syntax is machine-dependent (The result of a Platon operator is machine-dependent). For instance the lower and upper bounds of an array declaration may be a constant expression. This implies that it may not be possible to check whether or not the lower bound is less than the upper bound. In such cases the intermediate form will contain explicit operations for checking this.

A more serious problem is assignment and parameter substitution of structured variables containing array components. Two variables are of the same type if their structure is the same. This is not possible to check if the structure contains array types using constant expressions. To be consistent with our design criteria we then should generate instructions to do such type checking at code generation time. The syntax of a programming language should in our opinion be as little machine-dependent as possible. To postpone a major part of the type checking of Platon to code generation time seems to be a mistake and we have decided on not to do so. The result is that two array types may only be identical if they use constants as bounds or are declared in the same declaration.

4.2 The Symbol Table.

4.2.1 The Symbol Table in the Analysis Schema.

The analysis schema in section 4.3 includes the construction of a symbol table. For this purpose a recursive data structure is used. When a symbol table construct appears in the analysis schema we imagine that this data structure (regarded as a set of partial symbol tables) is updated appropriately. A precise description of the recursive data structure may be found in the appendix (A.2.2.2).

4.2.2 The Appearance of the Symbol Table.

As mentioned above the declarative information may be maintained either by keeping the symbol table or by generating explicit declare instructions. In the first case a reconstruction of the table is reduced to a copying of the table. In the case of declare instructions the single instructions have to be interpreted.

Our proposal is a symbolic table, except for the parenthetic instructions : symboltable, endsymboltable, fieldlist, endfieldlist, paramlist, endparamlist.

The declarative information is accessed through the following tuples :

(process,NAME,SYMBOLTABLE) ,

(procedure,NAME,SYMBOLTABLE), or
(function,NAME,SYMBOLTABLE) .

SYMBOLTABLE refers to a sequence of tuples of the form :
(symboltable,ND)
sequence of declarations of the form (←,NAME,DENOTATION)
(endsymboltable,ND)
The sequence of declarations may be intermixed with other kinds of declarations (listed below) and may refer to other declarations in other declaration sequences.

DENOTATION refers to a tuple of the form :
(const,TUPLE OPERAND,KIND,STRUCTURE) ,
(type,KIND,STRUCTURE) ,
(var,KIND,STRUCTURE) ,
(field,KIND,STRUCTURE) ,
(formalconst,KIND,STRUCTURE) ,
(formalvar,KIND,STRUCTURE) ,
(procedure,PARAMETER PART) ,
(function,PARAMETER PART) ,
(localfunction,PARAMETER PART) ,
(externalprocedure,PARAMETER PART) ,
(externalfunction,PARAMETER PART) ,
(formalprocedure,PARAMETER PART) ,
(formalfunction,PARAMETER PART) ,
(process,PARAMETER PART), or
(paramlist,PARAMETER PART) .

KIND may be either active, passive, or reference.

STRUCTURE may refer to a tuple of the form .
(word),
(byte),
(semaphore),
(shadow),
(reference),
(pool,INFO,TUPLE OPERAND) ,
(array,TUPLE OPERAND,TUPLE OPERAND,KIND,STRUCTURE), or
(record,FIELDLIST) .

INFO may refer to a tuple of the form :
(noinfo), or
(info,KIND,STRUCTURE) .

FIELDLIST refers to a sequence of tuples of the form :
(fieldlist,ND)
sequence of declarations of the form : (←,NAME,I)
I: (field,KIND,STRUCTURE)
(endfieldlist,ND)
The sequence of declarations may be intermixed with other kinds of declarations and may refer to other declarations in other declaration sequences.

PARAMETER PART refers to a tuple of the form :
(list,PARAMLIST), or
(recursive,NAME) .

PARAMLIST refers to a sequence of tuples of the form :
(paramlist,ND)
sequence of declarations of the form : (←,NAME,PARAMETER)
(endparamlist,ND)

PARAMETER refers to a tuple of the form :
(formalconst,STRUCTURE) ,
(formalvar,KIND,STRUCTURE) ,
(formalprocedure,PARAMETER PART), or
(formalfunction,PARAMETER PART) .
The sequence of declarations may be intermixed with other kinds of declarations and may refer to other declarations in other declaration sequences.

TUPLE OPERAND may be one of the following constructors:
t(CONST) denoting a tuple reference,
c(CONST) denoting a Platon constant or an integer, or
v(NAME) denoting a declared Platon identifier.

4.3 The analysis Schema.

In the schema the result of the analysis is given for each language construct.

4.3.1 Notation.

The following extensive but straightforward notational conventions are used :

Given a construct X, we let **A**(X) denote the *analysis* of X. **A**(X) may consist of both a sequence of tuples and a partial symbol table. The constructs having a *trivial* analysis of the form **A**(X) = **A**(X*1*)...**A**(X*n*), where X = X*1*...X*n*, are not

included in the schema.

The elements of the constructs may appear in abbreviated form and indices are used to distinguish between distinct occurrences of an element in a construct.

To simplify the denotation of **A**(X) for a construct X = X*1*...X*k*...X*n* the following conventions are used :

a tuple is enclosed in brackets (and) and implies a generation of the tuple.

a symbol table construct is enclosed in brackets { and } and implies an insertion in the symbol table. However, *no* tuples are generated.

NAME, CONST, and CHAR denote a simple, straightforward linkage to the lexical analyser.

←: denotes that the item following the colon is part of the result of the analysis of the construct. This result may be a tuple number, a partial symbol table, a symbolic name, or a symbolic constant. In this way the evaluation of a construct may return a value in the same way as a function may yield a result. Several ←: may occur in the same construct. The result is the union of the values (partial symbol tables) appearing after the ←:.

S(NAME) denotes a symbol table look-up. The partial symbol table associated with NAME is returned as the resulting value.

T(tuple no) denotes a linearized copy of the partial symbol table created in the construct associated with tuple no. For a description of the abstract symbol table see appendix A. The linearization is described in section 4.2. The reference to the symbol table returned as the value of **T** is called the *entry* of the symbol table.

I: ,J: ,etc (possibly indexed) in front of a tuple or **A**(X*k*) denotes the number of the tuple or the tuple number returned from **A**(X*k*), respectively.

A: ,B: ,etc (possibly indexed) in front of a symbol table construct or **A**(X*k*) denote that construct or the partial symbol table returned from **A**(X*k*).

ND denotes the actual nesting depth of statements, or declarations.

K denotes the kind of a variable (active, passive, or reference). The kind has significance in the formal section only.

e denotes an empty symbol table, field list, or parameterlist.

identifiers written in small letters denote intermediate language operators and source language symbols.

identifiers written in capital letters denote source language constructs

comments are enclosed in (* and *) .

To illustrate our notation we explain two examples in detail.

Example 1

STATEMENTS (S)

while E do S

	(while,ND,I)
I:	**A**(E)
	(whiledo,ND,I)
	A(S)
	(endwhile,ND,I)

This example shows the analysis of a construct, the while statement (written : while E do S). The while statement belongs to the class of STATEMENTS (abbreviated to S). The analysis (i.e. the result of the translation) is given in the right hand column: First a tuple is generated which consists of the operator while, the operand ND (the nesting depth of the while statement), and the operand I (the tuple number returned as the result of the analysis of E). Subsequently the analysis of E (**A**(E)), a tuple with operator whiledo, the analysis of S (**A**(S)), and a tuple with operator endwhile follow.

Example 2

TYPE DECLARATION

NAME := TS

A:	**A**(TS)
←:	{NAME→type(A)}

This example shows the analysis of a construct, the type declaration (written : NAME := TS , where TS is an abbreviation of TYPE SPECIFICATION). The analysis consists of the analysis of TS (**A**(TS)) which returns a partial symbol table as result denoted A. This result is used in a symbol table construct which associates NAME with type(A). This symbol table construct is returned as the result of the analysis.

4.3.2 The Schema for Platon.

PROGRAM

PROCESS DECLARATION

(program)
A(PROCESS DECLARATION)
(endprogram)

CONSTANT DECLARATION

NAME := E

I: **A**(E)
←: {NAME→const(I,(K,T))}

(* T denotes the type of E *)

(NAME*1*,...,NAME*n*)

←: {NAME*1*→const(init*1*,(K,word))}
:
:
←: {NAME*n*→const(init*n*,(K,word))}

(* init*1*, init2,...,init*n* denote a consecutive but implementation dependent sequence of values *)

TYPE DECLARATION

NAME := TS

A: **A**(TS)
←: {NAME→type(A)}

TYPE SPECIFICATION (TS)

NAME

←: **S**(NAME)

---

word

←: (passive,word)

---

byte

←: (passive,byte)

---

shadow

←: (active,shadow)

---

reference

←: (reference,reference)

---

semaphore

←: (active,semaphore)

```
record
NAME11,...,NAME1n1 : TS1
:
:
NAMEm1,...,NAMEmnm : TSm
end

      A1: A(TS1)
     B11: {NAME11→field(A1)}
          :
          :
   B1n1: {NAME1n1→field(A1)}
          :
          :
      Am: A(TSm)
     Bm1: {NAMEm1→field(Am)}
          :
          :
   Bmnm: {NAMEmnm→field(Am)}
       ←: {(K,record(B11 U ... U Bmnm))}
---------- ----------------------------------------------------------

array
( EL1 .. EU1
:
:
, ELn .. EUn
) of TS

      I1: A(EL1)
      J1: A(EU1)
          (arraybounds,I1,J1)
          :
          :
      In: A(ELn)
      Jn: A(EUn)
          (arraybounds,In,Jn)

      An: A(TS)
   An-1: {(K,array(In,Jn,An))}
          :
          :
       ←: {(K,array(I1,J1,A1))}
```

(* arraybounds denotes a machine dependent syntax check: the index range must be positive *)

---

sharedset E of empty

I: **A**(E)
←: {(K,pool(noinfo,I))}

---

sharedset E of TS

I: **A**(E)
A: **A**(TS)
←: {(K,pool(info(A),I))}

VARIABLE DECLARATION

NAME*1*,...,NAME*n* : TS

A: **A**(TS)
←: {NAME*1*→var(A)}
:
:
←: {NAME*n*→var(A)}

PROCESS DECLARATION

process NAME FORMAL PARAMETERS ; PROCESS BODY ;

(process,NAME,I)
I: **T**(B)
A: **A**(FORMAL PARAMETERS)
←: {NAME→process(A)}
B: **A**(PROCESS BODY)
(endprocess,NAME)

---

process NAME ; PROCESS BODY ;

(process,NAME,I)
I: **T**(B)
←: {NAME→process(*e*)}
B: **A**(PROCESS BODY)

(endprocess,NAME)

PROCEDURE DECLARATION

procedure NAME*1* : NAME*2* ; BODY ;

(procedure,NAME*1*,I)
I: **T**(B)
A: **S**(NAME*2*)
←: {NAME*1*→procedure(A)}
B: **A**(BODY)
(endprocedure,NAME*1*)

---

procedure NAME FORMAL PARAMETERS ; BODY ;

(procedure,NAME,I)
I: **T**(B)
A: **A**(FORMAL PARAMETERS)
←: {NAME→procedure(A)}
B: **A**(BODY)
(endprocedure,NAME)

---

procedure NAME ; BODY ;

(procedure,NAME,I)
I: **T**(B)
←: {NAME→procedure(*e*)}
B: **A**(BODY)
(endprocedure,NAME)

FUNCTION DECLARATION

function NAME*1* ; NAME*2* ; BODY ;

(function,NAME,I)
I: **T**(B)
A: **S**(NAME*2*)
←: {NAME*1*→function(A)}
B: **A**(BODY)
(endfunction,NAME)

---

function NAME FORMAL PARAMETERS ; BODY ;

(function,NAME,I)
I: **T**(B)
A: **A**(FORMAL PARAMETERS)
←: {NAME→function(A)}
B: **A**(BODY)
(endfunction,NAME)

---

function NAME ; BODY ;

(function,NAME,I)
I: **T**(B)
←: {NAME→function(*e*)}
B: **A**(BODY)
(endfunction,NAME)

LOCK DECLARATION

NAME := TS

A: **A**(TS)
←: {NAME→var(A)}

PARAMLIST DECLARATION

NAME := FORMAL PARAMETERS

A: **A**(FORMAL PARAMETERS)
←: {NAME→paramlist(A)}

FORMAL PARAMETERS

(PARAMETER DESCRIPTION*1*;...;PARAMETER DESCRIPTION*n*)

←: **A**(PARAMETER DESCRIPTION*1*)
:
:
←: **A**(PARAMETER DESCRIPTION*n*)

PARAMETER DESCRIPTION

NAME*1*,...,NAME*n* : TS

A: **A**(TS)
←: {NAME*1*→formalconst(A)}
:
:
←: {NAME*n*→formalconst(A)}

---

const NAME*1*,...,NAME*n* : TS

A: **A**(TS)
←: {NAME*1*→formalconst(A)}
:
:
←: {NAME*n*→formalconst(A)}

---

var NAME*1*,...,NAME*n* : TS

A: **A**(TS)
←: {NAME*1*→formalvar(A)}
:
:
←: {NAME*n*→formalvar(A)}

---

procedure NAME*1*,...,NAME*n* : NAME

A: **S**(NAME)
←: {NAME*1*→formalprocedure(A)}
:
:
←: {NAME*n*→formalprocedure(A)}

---

procedure NAME*1*,...,NAME*n*

←: {NAME*1*→formalprocedure(*e*)}
:
:
←: {NAME*n*→formalprocedure(*e*)}

---

function NAME*1*,...NAME*n* : NAME

A: **S**(NAME)
←: {NAME*1*→formalfunction(A)}
:
:
←: {NAME*n*→formalfunction(A)}

---

function NAME*1*,...,NAME*n*

←: {NAME*1*→formalfunction(*e*)}
:
:
←: {NAME*n*→formalfunction(*e*)}

PROCESS BODY

GLOBAL DECLARATIONS ; COMPOUND STATEMENT

←: **A**(GLOBAL DECLARATIONS)
A(COMPOUND STATEMENT)

(* GLOBAL DECLARATIONS is not specified in further detail. Its result is the union of the partial symbol tables, which are returned as the result of the analysis of the various declaration parts *)

---

internal

(internal)

BODY

LOCAL DECLARATIONS ; COMPOUND STATEMENT

←: **A**(LOCAL DECLARATIONS)
A(COMPOUND STATEMENT)

(* LOCAL DECLARATIONS is not specified in further detail. It consists of a sequence of partial symbol tables, which are returned as the result of the analysis of the various declaration parts *)

---

external

(external)

(* There is a slight inaccurancy in the description of external procedures, because they should appear in the symbol table as externalprocedure and not as procedure. This is corrected in the formal definition in the appendix *)

VARIABLE DECLARATION WITH INITLIST

VARIABLE DECLARATION :=
INITLIST ELEMENT*1*,...INITLIST ELEMENT*n*

A: **A**(VARIABLE DECLARATION)
(initlist,I)
I: **T**(A)
A(INITLIST ELEMENT*1*)
:
:
A(INITLIST ELEMENT*n*)
(endinitlist)

INITLIST ELEMENT

E*r* * E*v*

I: **A**(E*r*)
J: **A**(E*v*)
(repetition,I,J)

E

J: **A**(E)
(repetition,1,J)

(* repetition includes a machine dependent syntax check : the type of the initial value must be the same as the type of the variable being initialized *)

STATEMENTS (S)

repeat S*1*,...,S*n* until E

```
        (repeat,ND,I)
        A(S1)
        :
        :
        A(Sn)
    I:  A(E)
        (endrepeat,ND,I)
```

---

while E do S

```
        (while,ND,I)
    I:  A(E)
        (whiledo,ND,I)
        A(S)
        (endwhile,ND,I)
```

---

if E then S

```
        (if,ND,I)
    I:  A(E)
        (ifthen,ND,I)
        A(S)
        (endif,ND,I)
```

---

if E then S*1* else S*2*

```
        (if,ND,I)
    I:  A(E)
        (ifthen,ND,I)
        A(S1)
        (ifelse,ND,I)
        A(S2)
        (endif,ND,I)
```

---

with VD do S

```
        (with,ND,I)
    I:  A(VD)
```

```
          (withdo,ND,I)
          A(S)
          (endwith,ND,I)
---------- ------------------------------------------------------------

lock VD to LOCK DECLARATION in S

          (lock,ND,I)
       I: A(VD)
          (lockto,ND,I)
   ←:A: A(LOCK DECLARATION)
       I: T(A)
          (lockin,ND,I,J)
          A(S)
          (endlock,ND,I)
---------- ------------------------------------------------------------

VD1 := VD2 := ... VDn := E

      I1: A(VD1)
      I2: A(VD2)
          :
          :
      In: A(VDn)
      Jn: A(E)
   Jn-1: (:=,In,Jn)
          :
          :
      J1: (:=,I2,J2)
          (:=,I1,J1)
          (* In the case that VD is the name of a function the operator
          functionresult is generated instead of := *)
---------- ------------------------------------------------------------

exit

          (exit)
---------- ------------------------------------------------------------

halt

          (halt)
---------- ------------------------------------------------------------

NAME ( ACTUAL PARAMETER1,...,ACTUAL PARAMETERn )

   ←:J0: (procedurecall,NAME,Jn+1)
```

```
        I1: A(ACTUAL PARAMETER1)
        J1: (parameter,I1,J0)
            :
            :
        In: A(ACTUAL PARAMETERn)
        Jn: (parameter,In,Jn-1)
    Jn+1: (endprocedurecall,Jn)

            (* The analysis of an actual parameter is not specified in further detail
            *)
```

```
case E of
E11,...,E1n1 : S1 ;
:
:
Em1,...,Emnm : Sm ;
end

            (case,ND,I)
        I:  A(E)
            (caseof,ND,I)
      I11:  A(E11)
            (caselabel,ND,I11)
            :
            :
    I1n1:   A(E1n1)
            (caselabel,ND,I1n1)
            A(S1)
            (caseelement,ND)
            :
            :
     Im1:   A(Em1)
            (caselabel,ND,Im1)
            :
            :
   Imnm:    A(Emnm)
            (caselabel,ND,Imnm)
            A(Sn)
            (caseelement,ND)
            (endcase,ND,I)

            (* at most one Eij must be the default symbol in which case a special
            tuple is introduced: (default,ND) *)
```

(* caselabel includes a machine dependent syntax check since no two caselabels on the same level of nesting may have the same value *)

VARIABLE DENOTATION (VD)

NAME

←: NAME

VD . NAME

I: **A**(VD)
←: (field,I,NAME)

VD (E*1* , E*2* ,..., E*n*)

J*1*: **A**(VD)
I*1*: **A**(E*1*)
J*2*: (index,J*1*,I*1*,I*1*,u*1*)
I*2*: **A**(E*2*)
J*3*: (index,J*2*,I*2*,I*2*,u*2*)
:
:
I*n*: **A**(E*n*)
←: (index,J*n*,I*n*,I*n*,u*n*)

(* I*i*, u*i* denote lower and upper bound, respectively of the i'th index of the array *)

EXPRESSION (E)

FACTOR*0* dyadic*1* FACTOR*1* ... dyadic*n* FACTOR*n*

J*1*: **A**(FACTOR*0*)
I*1*: **A**(FACTOR*1*)
J*2*: (dyadic*1*,J*1*,I*1*)
:
:
I*n*: **A**(FACTOR*n*)

←: (dyadic*n*,J*n*,I*n*)

(* dyadic*i* denote any of the dyadic operators : +, −, and, or, xor, ls, rs, mask, <, <=, =, >=, >, <>, oddpar, extract, extend *)

FACTOR

(E)

←: **A**(E)

---

monadic FACTOR

I: **A**(FACTOR)
←: (monadic,I)

(* monadic denotes any of the monadic operators : neg, not *)

NAME (ACTUAL PARAMETER*1*,...,ACTUAL PARAMETER*n*)

←:J*0*: (functioncall,NAME,J*n+1*)
I*1*: **A**(ACTUAL PARAMETER*1*)
J*1*: (parameter,I*1*,J*0*)
:
:
I*n*: **A**(ACTUAL PARAMETER*n*)
J*n*: (parameter,I*n*,J*n−1*)
J*n+1*: (endfunctioncall,J*n*)

(* The analysis of an actual parameter is not specified in further detail *)

---

true

←: 'true'

---

false

←: 'false'

CONST

←: CONST

CONST*1* ↑ CONST*2*

←: (radix,CONST*1*,CONST*2*)

'CHAR'

←: CHAR

NAME

←: t

(* NAME must be the name of a formal constant; t denotes the constant expression associated with that formal constant *)

PREDEFINED PROCEDURES

alloc VD*1* from VD*2*

J0: (alloc,*J2*)
I1: **A**(VD*1*)
J1: (parameter,*I1*,*J0*)
I2: **A**(VD*2*)
J2: (endalloc,*I2*,*J1*)

---

alloc VD*1* form VD*2* with VD*3*

J0: (alloc,*J3*)
I1: **A**(VD*1*)
J1: (parameter,*I1*,*J0*)
I2: **A**(VD*2*)
J2: (parameter,*I2*,*J1*)
I3: **A**(VD*3*)
J3: (endalloc,*I3*,*J2*)

---

return VD

J0: (return,*J1*)
I: **A**(VD)
J1: (endreturn,I,*J0*)

---

signal VD*1* to VD*2*

J0: (signal,*J2*)
I1: **A**(VD*1*)
J1: (parameter,*I1*,*J0*)
I2: **A**(VD*2*)
J2: (endsignal,*I2*,*J1*)

---

wait VD*1* from VD*2*

J0: (wait,*J2*)
I1: **A**(VD*1*)
J1: (parameter,*I1*,*J0*)
I2: **A**(VD*2*)
J2: (endwait,*I2*,*J1*)

---

setcode E in VD

J0: (setcode,*J2*)
I1: **A**(E)
J1: (parameter,*I1*,*J0*)
I2: **A**(VD)
J2: (endsetcode,*I2*,*J1*)

---

readcode VD*1* in VD*2*

J0: (readcode,*J2*)
I1: **A**(VD*1*)
J1: (parameter,*I1*,*J0*)
I2: **A**(VD*2*)
J2: (endreadcode,*I2*,*J1*)

---

load VD from E report PROCEDURE NAME

J0: (load,*J3*)
I1: **A**(VD)
J1: (parameter,*I1*,*J0*)
I2: **A**(E)
J2: (parameter,*I2*,*J1*)
I3: **A**(PROCEDURE NAME)
J3: (endload,*I3*,*J2*)

---

create VD like PROCESS CALL with E report PROCEDURE NAME

J0: (create,*J4*)
I1: **A**(VD)
J1: (parameter,*I1*,*J0*)
I2: **A**(PROCESS CALL)
J2: (parameter,*I2*,*J1*)
I3: **A**(E)
J3: (parameter,*I3*,*J2*)
I4: **A**(PROCEDURE NAME)
J4: (endcreate,*I4*,*J3*)

---

start VD report PROCEDURE NAME

J0: (start,*J2*)
I1: **A**(VD)
J1: (parameter,*I1*,*J0*)
I2: **A**(PROCEDURE NAME)
J2: (endstart,*I2*,*J1*)

---

stop VD report PROCEDURE NAME

J0: (stop,*J2*)
I1: **A**(VD)
J1: (parameter,*I1*,*J0*)
I2: **A**(PROCEDURE NAME)
J2: (endstop,*I2*,*J1*)

---

remove VD report PROCEDURE NAME

J0: (remove,*J2*)
I1: **A**(VD)
J1: (parameter,*I1*,*J0*)
I2: **A**(PROCEDURE NAME)
J2: (endremove,*I2*,*J1*)

---

unload VD report PROCEDURE NAME

J0: (unload,*J2*)
I1: **A**(VD)
J1: (parameter,*I1*,*J0*)
I2: **A**(PROCEDURE NAME)
J2: (endunload,*I2*,*J1*)

PROCESS CALL

NAME (E*1*, ... ,E*n*)

J0: (processcall,NAME,J*n+1*)
I*1*: **A**(E*1*)
J*1*: (parameter,I*1*,J*0*)
:
:
I*n*: **A**(E*n*)
J*n*: (parameter,I*n*,J*n−1*)
J*n+1*: (endprocesscall,J*n*)

PROCEDURE NAME

NAME
←: NAME

---

5 Conclusion.

We have stated and given a justification for a number of design criteria for an intermediate language used as a vehicle for the efficient implementation of a high level language. As an example of the use of these design criteria we have proposed an intermediate language for Platon. Furthermore, we have supported the claim made in [12], namely that the analysis phase can be automized, by giving an affix translation grammar defining the translation. We think it should be possible to generalize the remaining phases of a compiler in a similar fashion.

While working out the formal definition of the translation we came across a number of syntactic irregularities in Platon. Experience with the use of the language has shown that these "irregular constructs" are sources of error when programming in Platon. An example of such a construct is the type-specification of parameters and variables. These specifications have different syntax although they are the same concept from the programmers point of view. We can therefore support the often stated advice, that a formal definition is a very important (if not indispensable) tool in the design, definition, and implementation of a programming language.

References.

[1]: Kristensen B.B., Madsen O.L., and Jensen B.B.
A Pascal Environment Machine (P-code).
Department of Computer Science, University of Aarhus,
Aarhus, Denmark, DAIMI PB-28, 1974.

[2]: Wirth N.
Pascal-s: A Subset and its Implementation.
Eidgenossische Technische Hochschule,
Zurich, Switzerland, 1975.

[3]: Richards M.
The Portability of the BCPL Compiler.
Software - Practice and Experience, Vol.1, 1971.

[4]: Pasko H.J.
A Pseudo-machine for Code-generation.
University of Toronto,
Toronto, Canada, CSRG-30, 1973.

[5]: Nori K.V., Ammann V., Jensen K., and Nageli H.H.
The Pascal <p> Compiler: Implementation Notes.
Eidgenossische Technische Hochschule,
Zurich, Switzerland, 1974.

[6]: Robertson P.
Compiler Intermediate Code (IMP).
Edinburgh, private communication, 1975.

[7]: Brinch Hansen P
Concurrent Pascal Machine.
California Institute of Technology,
Pasadena, Information Science Report, 1975.

[8]: Sørensen S.M. and Staunstrup J.
Platon. Reference Manual.
RECAU. The Regional EDP Center, University of Aarhus,
Aarhus, Denmark, R-75-58, 1975.

[9]: Staunstrup J., and Sørensen S.M.
Platon. A High Level Language for Systems Programming.
in Minicomputer Software.
North Holland , 1976.

[10]: Kornerup P., Kristensen B.B., and Madsen O.L.
Interpretation and Code Generation Based on
Hypothetical Machines. Department of Computer Science,
University of Aarhus, Aarhus, Denmark, in preparation, 1976

[11]: Naur P.
The design of the GIER Algol Compiler.
BIT 3, 2-3, p.124-140 and 145-166, 1963

[12]: Madsen O.L.
On the Use of Attribute Grammars in a
Practical Translator Writing System.
University of Aarhus, Aarhus, Denmark, Master Thesis, 1975

[13]: Gries D.
Compiler Construction for Digital Computers.
New York, John Wiley and sons, 1971.

[14]: Wirth N.
The Programming Language Pascal.
Acta Informatica 1, p.35-63, 1973.

[15]: Knuth D.E.
Examples of Formal Semantics.
in Lecture Notes in Mathematics 188.
Springer Verlag, 1971.

[16]: Lewis P.M., Rosenkrantz D.J., and Stearns R.E.
Attributed Translations.
Journal of Computing and System Sciences 9.
p.279-307, 1974.

[17]: Watt D.A.
Analysis Oriented Two-Level Grammars.
University of Glasgow, Glasgow, Scotland
Ph.D. Thesis, 1974

[18]: Hoare C.A.R.
Notes on Data Structuring.
in Structured Programming, Academic Press, 1972.

[19]: Hoare C.A.R.
Recursive Data Structures.
Stanford Intelligence Laboratory Memo.
Stan-CS-73-400, 1973.

APPENDIX A. Formal Definition of the Translation.

A1 Affix Translation Grammars.

In this appendix we give a formal definition of the syntax of Platon (including context dependencies) and of the translation from Platon to the intermediate form defined in section 4. We use the notion of an affix translation grammar as defined in [12]. Affix translation grammars are a combination of attributed translation grammars [16] and extended affix grammars [17]. It is not in the scope of this paper to give a complete introduction to affix translation grammars and we just give a concentrated presentation.

We first define a *translation grammar* which is a context free grammar in which the set of terminal symbols is partitioned into a set of *input symbols* and a set of *action symbols*. The strings generated by the grammar are called *activity sequences*. An activity sequence may be split into an *input part* and an *action part*. The input part is obtained by deleting all action symbols. Likewise the action part is obtained by deleting all input symbols. The set of all such pairs of action and input parts is called the *syntax directed translation* defined by that translation grammar.

An *affix translation grammar* (ATG) is a translation grammar in which each symbol (input, nonterminal, or action symbol) has a fixed (possible zero) number of *affix-positions* (or positions). Each position is associated with a set of values called the *domain* of the position. The positions are used to carry attribute values of the corresponding language construct. Each position is classified as either *inherited* or *synthesized*. Inherited positions carry information about the context of the symbol. Synthesized positions carry information about the phrase derived from the symbol in the given context.

A *production rule* has the form

$$\mathbf{Z}_0 = \mathbf{Z}_1\ \mathbf{Z}_2\ \ldots\ \mathbf{Z}n$$

where each $\mathbf{Z}i$ (i=0,1,...,n) has the form

$$\mathbf{X}i[\mathbf{E}_1,\mathbf{E}_2,\ldots,\mathbf{E}ik]$$

where $\mathbf{X}i$ (i=1,2,...,n) is an input, nonterminal, or action symbol with ik positions. $\mathbf{X}_0$ is always a nonterminal. $\mathbf{E}j$ (j=1,2...,jk) is an expression possibly containing *variables*. Each variable has an associated domain. By replacing each variable by a value in its domain, and evaluating the expression, the result must be a value in the domain of the position.

A production rule acts as a generator for a set of context free like production rules in the following way: identical variables in the rule are replaced by identical values and all the expressions are evaluated. The resulting rule is called an

attributed production rule. In this way an ATG may be used to generate a translation grammar in a straightforward manner. The result of a derivation in this translation grammar is called an *attributed activity sequence* which consists of an input-part and an action-part. The set of all input-parts and action-parts is called the attributed translation defined by the ATG.

A symbol with affix positions may be interpreted as a procedure with parameters. The inherited positions correspond to input parameters and the synthesized positions correspond to output parameters. In a production rule the symbols on the right-hand side correspond to calls and the symbol on the left-hand side is the one to be defined. The *applied positions*: the output parameters of the left-hand side and the input parameters of the right-hand side are defined in terms of the *defining positions*: the input parameters of the left-hand side and the output parameters of the right-hand side. The values of the variables are determined by the expressions in the defining positions and they are used in the expressions in the applied positions. A value received in a defining position has to match the expression at that position. In general this implies that a number of equations have to be solved in order to determine the values of the variables in the defining positions. In practice it suffices to use simple expressions in the defining positions such as constants, variables, or constructors [18] built from constants and variables. If a variable does not appear in a defining position its value is arbitrary.

The use of an expression which is not just a variable makes it possible to differentiate the rules into several cases.

A2 The Definition of Platon.

A2.1 Notational Conventions.

Nonterminals and variables will be represented by sequences of upper case Latin letters and hyphens. Input symbols will be represented by sequences of lower case Latin letters or special characters in bold face. Action symbols will be represented by sequences of lower case letters in italics.

Inherited positions will be marked with a ↓. Synthesized positions will be marked with a ↑.

On the right-hand sides we use the operators * (zero or more repetions), + (one or more repetitions), ? (optional) and {,} (grouping). In a *-clause an inherited position containing an expression of the form $E\downarrow^1 F\downarrow$ is related to a synthesized position containing the expression $F\uparrow E\uparrow^0$:

$$Z = a\ \{b\ X[...,E\downarrow^1 F\downarrow,...,F\uparrow E\uparrow^0,...]\ c\}^*\ d$$

with E and F being variables. E is supposed to appear in a defining position of Z, *a*, or *d*. If n>0 instances of the clause appear, then the expressions are interpreted as follows:

$$Z = abX[...,E\downarrow,...,F_1\uparrow,...]cbX[...,F_1\downarrow,...,F_2\uparrow...]c$$
$$...b[...,Fn-1\downarrow,...,F\uparrow...]cd$$

with $F_1,F_2,...,Fn-1$ being new variables. If no instance of the clause appear, then F is defined to be the value of E. In a +−clause similar expressions may appear, however no rule for zero instances is necessary.

A2.2 Domains.

Domain definitions.

Let B be a domain and A a name, then A=B defines the domain A to be B. We allow domain definitions to be recursive.

A2.2.1 Domain Types.

Besides certain base domains we use the following kinds of domains:

Cartesian product.

Let $T_1,T_2,...,Tn$ be domains, then $(T_1,T_2,...,Tn)$ denotes the Cartesian product.

If ti is a Ti (i=1,2,...,n), then $(t_1,t_2,...,tn)$ is a $(T_1,T_2,...,Tn)$.

(ti) will be abbreviated to ti.

Discriminated Unions.

Let $T_1,T_2,...,Tn$ be domains, and $g_1,g_2,...,gn$ be distinct names, then

$(g_1(T_1) \mid g_2(T_2) \mid ... \mid gn(Tn))$ is a discriminated union with selectors $g_1,g_2,...,gn$.

If $Tk=Tk+1=...=Tl=T$, then we may write $(g_1(T_1) \mid ... \mid gk,gk+1,...,gl(T) \mid ... \mid gn(Tn))$.

If ti is a Ti (i **in** [1,n]), then $gi(ti)$ is a $(g_1(T_1) \mid g_2(T_2) \mid ... \mid gn(Tn))$.

If $Ti=\emptyset$, then $gi(Ti)$ is abbreviated gi.

Functions.

Let A and B be domains, then {A→B} is a function from A to B.

Let A,B,C be domains, f,g be a {A→B}, a,a′ be an A, b a B, and c be a C, then dom(f) is a subset of A

if a **in** dom(f) **then** f(a) is a B (function application),
e is a {A→B} (the empty function),
{a→b} is {A→B} (a function defined in one point),
if the intersection of dom(f) and dom(g) is empty **then** f**U**g is a {A→B},
(denotes the disjoint union of f and g),
f\g is a {A→B} (denotes the function f overriden by g),
and f**X**c is a {A→B**X**C} (denotes the function f extended with the value c in all defined points).

f = f**U***e* = *e***U**f = f*e* = *e*\f.

dom(*e*)=Ø, dom({a→b})={a}, dom(f**U**g)=dom(f)**U**dom(g),
dom(f\g)=dom(f)**U**dom(g), dom(f**X**c)=dom(f).

e(a)=undefined
f**U**{a→b}(a')=**if** a=a' **then** b **else** f(a')
f\{a→b}(a')=**if** a=a' **then** b **else** f(a')
f**X**c(a)=(f(a),c)

Cartesian product and discriminated union are from Hoare [18,19]. Functions from Knuth [15].

A2.2.2 Domain Definitions.

SYMBOLTABLE={NAME→DENOTATION}
A collection of declared names and their denotations which are visible at a given place in a Platon program.

DENOTATION=(const(TUPLE-OPERAND,TYPE)
| type,var,field,formalconst,formalvar(TYPE)
| procedure,function,localfunction,externalprocedure
, externalfunction,formalprocedure,formalfunction
, process,paramlist(PARAMETER-PART))
Denotation has a case for each possible meaning of a Platon identifier. Localfunction is a function which must be assigned a value at the given place. The rest of the cases correspond to Platon declarations. TUPLE-OPERAND of const is the value of the constant.

TYPE=(KIND,STRUCTURE)

KIND=(active | passive | reference)
Besides the structure of a type its use depends on the basic types of which it is built. This is determined by the KIND part of TYPE:
passive: The STRUCTURE part of TYPE is word, byte, or any structure built from these two basic types by using the array or record structure.
reference: As for passive, but at least one component is reference.

active: A structure which is neither passive nor reference.

STRUCTURE=(word | byte | reference | semaphore | shadow
| pool(POOL-INF,TUPLE-OPERAND)
| array(TUPLE-OPERAND,TUPLE-OPERAND,TYPE)
| record(FIELD-LIST))

Structure has a case for each possible Platon type. Pool is a sharedset variable. TUPLE-OPERAND of pool is the size of the variable. The two TUPLE-OPERANDs of array are the lower and upper bounds of the array.

POOL-INF=(noinfo | info(TYPE))

Distinguish between a **sharedset of empty** and a **sharedset of** TYPE.

FIELD-LIST={NAME→field(TYPE)}

Similar to SYMBOL-TABLE but only applicable to names declared in records.

PARAMETER-PART=(I(PARAMETER-LIST) | recursive(NAME))

A parameter part may be either a list of specified parameters (I) or the name of a declared parameter list (recursive). Recursive is used when the name of a parameter list appears recursively in its own declaration. This means that the parameter part of a formal procedure/function may have the form: recursive(NAME).

PARAMETER-LIST={NAME→PARAMETER}

PARAMETER={formalconst,formalvar(TYPE)
| formalprocedure,formalfunction(PARAMETER-PART))

TUPLE-OPERAND=(t(CONST) | c(CONST) | v(NAME))

A TUPLE-OPERAND may be a tuple number(t), a Platon constant(c), or a Platon variable(v).

TUPLE-OPERATOR= the set of operators as defined in section 4.

OPERATOR=(+ | − | **and** | **or** | **xor** | **ls** | **rs** | **mask**
| < | <= | = | => | > | <> | **oddpar**)

NAME and CONST are the set of strings denoting Platon identifiers and constants respectively.

CHAR is the set of Platon characters.

The above domains are those which will appear in the intermediate form.

In the affix definition we further use:

CONTEXT={NAME→(EXTENDED-DENOTATION,TUPLE-OPERAND)}

EXTENDED-DENOTATION=DENOTATION extended with:
(forwardprocedure,forwardfunction(PARAMETER-PART))

CONTEXT and EXTENDED-DENOTATION are similar to SYMBOL-TABLE and DENOTATION but are extended to handle with statements and forward declarations.

EXP-KIND=(const | var)
Denotes whether an expression is constant or not.

If a domain T is a subset of a domain T', then variables and expressions of type T are also of type T'. If a variable or expression is of type T', then it may be used as a T if its actual value is a T. FIELD-LIST and PARAMETER-LIST are subsets of SYMBOL-TABLE. PARAMETER is a subset of DENOTATION which again is a subset of EXTENDED-DENOTATION. SYMBOL-TABLE is a subset of CONTEXT. OPERATOR is a subset of TUPLE-OPERATOR.

A2.3 Variables.

The following is a list of the used variables and their domains:

C,L: CONTEXT; P: PARAMETER-PART; N: NAME; K: KIND;

S: STRUCTURE; D: DENOTATION; PL: PARAMETER-LIST;

TO: TUPLE-OPERAND; V: CONST; T: TYPE; F: FIELD-LIST;

ND: INTEGER; EK: EXP-KIND; OP: OPERATOR;

CH: CHAR; PI: POOL-INF;

The variables are also used with subscripts.

A2.4 Symbols.

The following is a list of symbols together with the domains of their affix-positions:

Input symbols:

id[NAME↑]

const[CONST↑]

char[CHAR↑]

operator[OPERATOR↑]

The positions of **id**, **const**, **char**, and **operator** may be interpreted as an interface to a lexical analyser.

Action symbols:

tuple[TUPLE-OPERATOR↓,TUPLE-OPERAND$_1$↓,...,TUPLE-OPERANDn↓,
TUPLE-OPERAND↑]

Note: we allow *tuple* to have a variable number of positions. The last position (also optional) may be interpreted as an interface to some action routines and it is supposed to return the number of the tuple being generated.

table[SYMBOLTABLE↓,TUPLE-OPERAND↑]

Table is supposed to generate the linear sequence of tuples corresponding to the symboltable of the first position (see section 4.2). The second position is supposed to return the number of the tuple being the entry of the symboltable.

Nonterminal symbols:

Before each symbol is given the number of the production which defines the symbol.

44 ACTUAL-PARAMETERS[CONTEXT↓,PARAMETER-LIST↓,TUPLE-OPERAND↓,
PARAMETER-LIST↑,TUPLE-OPERAND↑]
52 ARRAY-VAR[CONTEXT↓,TUPLE-OPERAND↑,STRUCTURE↑]
38 ASSIGNMENT-STATEMENT[CONTEXT↓,STRUCTURE↑,TUPLE-OPERAND↑]
13 BODY[CONTEXT↓,CONTEXT↓,CONTEXT↑]
41 CASE-ELM[CONTEXT↓,CONTEXT↓,INTEGER↓,STRUCTURE↓]
40 CASE-EXP[CONTEXT↓,TUPLE-OPERAND↑,EXP-KIND↑,STRUCTURE↑]
42 CASE-LABEL[CONTEXT↓,INTEGER↓,STRUCTURE↓]
36 COMPOUND-STATEMENT[CONTEXT↓,CONTEXT↓,INTEGER↓]
19 CONST-DCL[CONTEXT↓,CONTEXT↓,CONTEXT↑]
18 CONST-DCL-PART[CONTEXT↓,CONTEXT↓,CONTEXT↑]
20 CONST-ID[CONTEXT↓,INTEGER↓,CONTEXT↑,INTEGER↑]
12 DECL-ID[CONTEXT↓,NAME↑,DENOTATION↑]
48 DYADIC[STRUCTURE↓,STRUCTURE↓,STRUCTURE↑,TUPLE-OPERATOR↑]
47 EXP[CONTEXT↓,TUPLE-OPERAND↑,EXP-KIND↑,STRUCTURE↑]
27 FIELD-LIST[CONTEXT↓,FIELD-LIST↓,KIND↓
FIELD-LIST↑,KIND↑]
28 FIELD-TYPE[CONTEXT↓,KIND↓,KIND↑,TYPE↑]
53 FUNCTION-CALL[CONTEXT↓,TUPLE-OPERAND↑]
15 FUNCTION-DCL[CONTEXT↓,CONTEXT↓,CONTEXT↑]
54 FUNCTION-ID[CONTEXT↓,NAME↑,PARAMETER-PART↑]
4 GLOBAL-DCL[CONTEXT↓,CONTEXT↑]
30 GLOBAL-VAR-DCL[CONTEXT↓,CONTEXT↑]
29 GLOBAL-VAR-DCL-PART[CONTEXT↓,CONTEXT↑]
11 ID-LIST[CONTEXT↓,DENOTATION↓,CONTEXT↑]
26 INDEX[CONTEXT↓,TYPE↓,TYPE↑]
35 INITELEMENT[CONTEXT↓]
33 INITLIST[CONTEXT↓,CONTEXT↓,TYPE↓]
25 INX-LIST[CONTEXT↓,TYPE↓,TYPE↑]
39 LEFTSIDE[CONTEXT↓,TUPLE-OPERAND↑,DENOTATION↑]
14 LOCAL-DCL[CONTEXT↓,CONTEXT↓,CONTEXT↑]

32 LOCAL-VAR-DCL[CONTEXT↓,CONTEXT↓,CONTEXT↑]
31 LOCAL-VAR-DCL-PART[CONTEXT↓,CONTEXT↓,CONTEXT↑]
50 MONADIC[TUPLE-OPERATOR↑]
24 NAMED-TYPE[CONTEXT↓,TYPE↑]
45 PARAM[CONTEXT↓,PARAMETER-LIST↓,TUPLE-OPERAND↑,PARAMETER-LIST↑]
10 PARAM-DCL[CONTEXT↓,PARAMETER-LIST↓,PARAMETER-LIST↑]
9 PARAM-LIST[CONTEXT↓,PARAMETER-LIST↑]
17 PARAMLIST-DCL[CONTEXT↓,CONTEXT↓,CONTEXT↑]
16 PARAMLIST-DCL-PART[CONTEXT↓,CONTEXT↓,CONTEXT↑]
8 PARAM-PART[CONTEXT↓,PARAMETER-LIST↑]
55 PREDEFINED-PROCEDURE-CALL[CONTEXT↓]
7 PROCEDURE-DCL[CONTEXT↓,CONTEXT↓,CONTEXT↑]
43 PROCEDURE-ID[CONTEXT↓,NAME↑,PARAMETER-PART↑]
3 PROCESS-BODY[CONTEXT↓,CONTEXT↑]
56 PROCESS-CALL[CONTEXT↓,TUPLE-OPERAND↑]
2 PROCESS-DCL[CONTEXT↓,CONTEXT↑]
6 PROCESS-PARAM-DCL[PARAMETER-LIST↓,PARAMETER-LIST↑]
5 PROCESS-PARAM-PART[PARAMETER-LIST↑]
46 RECURSIVE-PARAM[CONTEXT↓,NAME↓,DENOTATION↑]
37 STATEMENT[CONTEXT↓,CONTEXT↓,INTEGER↓]
49 TERM[CONTEXT↓,TUPLE-OPERAND↑,EXP-KIND↑,STRUCTURE↑]
23 TYPE[CONTEXT↓,TYPE↑]
22 TYPE-DCL[CONTEXT↓,CONTEXT↓,CONTEXT↑]
21 TYPE-DCL-PART[CONTEXT↓,CONTEXT↓,CONTEXT↑]
51 VAR[CONTEXT↓,TUPLE-OPERAND↑,TYPE↑]

A2.5 Production Rules.

We start by explaining the meaning of the commonly used positions. The remaining positions of the nonterminals are explained at their definitions.

The first position (inherited) of PROCESS-DCL is the set of names and their denotations declared locally, up to but excluding the construct. The second position (synthesized) is the names of the first position extended with the name of the declared process, i.e. the set of names declared locally, up to and including the construct. This relationship between an inherited and synthesized position appears in a similar way in the following symbols:
between the first and second position of PROCESS-BODY, PROCESS-PARAM-DCL, GLOBAL-VAR-DCL-PART, and GLOBAL-DCL, between the second and third position of PROC-DCL, FUNC-DCL, PARAMLIST-DCL, PARAM-LIST, PARAM-DCL, LOCAL-DCL, LOCAL-VAR-DCL, CONST-DCL-PART, CONST-DCL, TYPE-DCL-PART, LOCAL-VAR-DCL-PART, TYPE-DCL, and BODY, between the first and third position of ID-LIST, and CONST-ID, between the second and fourth position of of FIELD-LIST.

The first position of each of PROC-DCL, PARAM-DCL, BODY, LOCAL-DCL,

FUNC-DCL, CONST-DCL-PART, CONST-DCL, TYPE-DCL-PART, TYPE-DCL, LOCAL-VAR-DCL-PART, COMPOUND-STATEMENT, STATEMENT is the set of global names which are visible in that construct and their denotation.

The second position of each of BODY, COMPOUND-STATEMENT, STATEMENT is the set of local names which are visible in that construct and their denotation.

The first position of each of PARAM-PART, PARAM-LIST, DECL-IDENT, TYPE, NAMED-TYPE, INX-LIST, INDEX, FIELD-LIST, INITLIST, INITELM, ASSIGNMENT-STATEMENT, LEFTSIDE, CASE-EXP, CASE-LABEL, PROC-ID, ACTUAL-PARAMETERS, PARAM, RECURSIVE-PARAM, EXP, TERM, VAR, INDEX-TAIL is the set of names (global and local) which are visible in that construct and their denotation.

(1) PROGRAM =
(1.1) *tuple*[program↓] PROCESS-DCL[*e*↓,L↑] *tuple*[endprogram↓]

(2) PROCESS-DCL[L↓,L**U**{N→process(I(PL))}↑] =
(2.1) **process id**[N↑] *tuple*[process↓,v(N)↓,TO↓]
table[L_1↓,TO↑]
PROCESS-PARAM-PART[PL↑] ;
PROCESS-BODY[PL↓,L_1↑] ;
tuple[endprocess↓,v(N)↓]

(3) PROCESS-BODY[L↓,L_1↑] =
(3.1) {GLOBAL-DCL[L↓^{1}L_1↓,L_1↑L^0↑]}*
COMPOUND-STATEMENT[*e*↓,L_1↓,0↓]

(3) PROCESS-BODY[L↓,L↑] =
(3.2) **internal** *tuple*[internal↓]

(4) GLOBAL-DCL[L↓,L_1↑] =
(4.1) PROCESS-DCL[L↓,L_1↑]
(4.2) | PROCEDURE-DCL[L↓,L↓,L_1↑]
(4.3) | FUNCTION-DCL[L↓,L↓,L_1↑]
(4.4) | PARAMLIST-DCL-PART[L↓,L↓,L_1↑]
(4.5) | CONST-DCL-PART[L↓,L↓,L_1↑]
(4.6) | TYPE-DCL-PART[L↓,L↓,L_1↑]
(4.7) | GLOBAL-VAR-DCL-PART[L↓,L_1↑]

(5) PROCESS-PARAM-PART[PL↑] =
(5.1) **(** PROCESS-PARAM-DCL[*e*↓,PL_1↑]
{; PROCESS-PARAM-DCL[PL_1↓^{1}PL↓,PL↑PL_1↑^{0}]}* **)**

(6) PROCESS-PARAM-DCL[PL↓,PL_1↑] =
(6.1) {**const**}? ID-LIST[PL↓,formalconst(passive,word)↓,PL_1↑]

```
            : word
(6.2)   | {const}? ID-LIST[PL↓,formalconst(passive,byte)↓,PL1↑]
            : byte
(6.3)   | { var | const | EMPTY }
            ID-LIST[PL↓,formalvar(active,semaphore)↓,PL1↑]
            : semaphore

(7)     PROCEDURE-DCL[C↓,L↓,,LU{N→procedure(I(PL))}↑]=
(7.1)       procedure id[N↑] tuple[procedure↓,v(N)↓,TO↓]
              table[L1↓,TO↑]
              PARAM-PART[C\L↓,PL↑]
              BODY[C↓,PLU{N→procedure(I(PL))}↓,L1↑] ;
              tuple[endprocedure↓,v(N)↓]

(7)     PROCEDURE-DCL[C↓,L↓,L\{N→procedure(I(PL))}↑]=
(7.2)       procedure DECL-ID[L↓,N↑,forwardprocedure(I(PL))↑]
              tuple[procedure↓,v(N)↓,TO↓]
              table[L1↓,TO↑]
              BODY[C↓,PLU{N→procedure(I(PL))}↓,L1↑] ;
              tuple[endprocedure↓,v(N)↓]

(7)     PROCEDURE-DCL[C↓,L↓,LU{N→forwardprocedure(I(PL))}↑] =
(7.3)       procedure id[N↑]
              PARAM-PART[C\L↓,PL↑] ; forward ;
        (*Productions 7.2 and 7.3 handles forward declarations of procedures.
        Production 7.3 is the forward declaration consisting of the name and
        the parameter list. In production 7.2 the body of a forward procedure
        is declared. The denotation of the name of the procedure is changed
        from forwardprodcedure to procedure by overriding L in production 7.2.
        In production 2.1 (resp. 7.1,7.2,15.1, and 15.2) the second position of
          PROCESS-BODY (resp. BODY) is a CONTEXT. The first position
          of table is a SYMBOL-TABLE. This means that the value of L1
          being defined in PROCESS-BODY must be a SYMBOL-TABLE. This
          exhibits procedures and functions being declared forward without
          a succeeding declaration.*)

(7)     PROCEDURE-DCL[C↓,L↓,LU{N→externalprocedure(I(PL))}↑] =
(7.4)       procedure id[N↑]
              tuple[procedure↓,v(N)↓,TO↓] table[PL↓,TO↑]
              PARAM-PART[C\L↓,PL↑] ; external ;
              tuple[external↓] tuple[endprocedure↓,v(N)↓]

(8)     PARAM-PART[C↓,e↑] =
(8.1)       EMPTY
        (*The last position of PROCESS-PARAM-PART, PARAM-PART and
          PARAM-LIST is the formal parameters declared in that construct*)
```

(8) PARAM-PART[C↓,PL↑] =
(8.2) : DECL-ID[C↓,N↑,paramlist(I(PL))↑]
(8.3) | PARAM-LIST[C↓,PL↑]

(9) PARAM-LIST[C↓,PL↑] =
(9.1) **(** PARAM-DCL[C↓,*e*↓,PL_1↑]
{ **;** PARAM-DCL[C↓,PL_{i-1}↓,PL↓,PL↑PL_i↑^{0}] }* **)**

(10) PARAM-DCL[C↓,PL↓,PL_1↑] =
(10.1) {**const**}? ID-LIST[PL↓,formalconst(K,S)↓,PL_1↑]
: NAMED-TYPE[C↓,(K,S)↑]
(10.2) | **var** ID-LIST[PL↓,formalvar(K,S)↓,PL_1↑]
: NAMED-TYPE[C↓,(K,S)↑]
(10.3) | **procedure** ID-LIST[PL↓,formalprocedure(P)↓,PL_1↑]
: DECL-ID[C↓,N↑,paramlist(P)↑]
(10.4) | **procedure** ID-LIST[PL↓,formalprocedure(*e*)↓,PL_1↑]
(10.5) | **function** ID-LIST[PL↓,formalfunction(P)↓,PL_1↑]
: DECL-ID[C↓,N↑,paramlist(P)↑]
(10.6) | **function** ID-LIST[PL↓,formalfunction(*e*)↓,PL_1↑]

(11) ID-LIST[L↓,D↓,L**U**{N→D}↑] =
(11.1) **id**[N↑]

(11) ID-LIST[L↓,D↓,L_1**U**{N→D}↑] =
(11.2) ID-LIST[L↓,D↓,L_1↑] **, id**[N↑]
(*The second position of ID-LIST is the denotation
of the names declared in the list *)

(12) DECL-ID[C↓,N↑,C(N)↑] =
(12.1) **id**[N↑]
(*The second position of DECL-ID is a name, which must
exist in the given context(C). The denotation of the
name is passed in the third position *)

(13) BODY[C↓,L↓,L_1↑] =
(13.1) { LOCAL-DCL[C↓,L↓^{1}L_1↓,L_1↑L↑^{0}] }*
COMPOUND-STATEMENT[C↓,L_1↓,0↓]

(14) LOCAL-DCL[C↓,L↓,L_1↑] =
(14.1) PROCEDURE-DCL[C↓,L↓,L_1↑]
(14.2) | FUNCTION-DCL[C↓,L↓,L_1↑]
(14.3) | PARAMLIST-DCL-PART[C↓,L↓,L_1↑]
(14.4) | CONST-DCL-PART[C↓,L↓,L_1↑]
(14.5) | TYPE-DCL-PART[C↓,L↓,L_1↑]
(14.6) | LOCAL-VAR-DCL-PART[C↓,L↓,L_1↑]

(15) FUNCTION-DCL[C↓,L↓,L**U**{N→function(I(PL))}↑] =
(15.1) **function id**[N↑] *tuple*[function↓,v(N)↓,TO↓]

table[L_1↓,TO↑]
PARAM-PART[C\L↓,PL↑] ;
BODY[C↓,PL**U**{N→localfunction(I(PL))}↓,L_1↑] ;
tuple[endfunction↓,v(N)↓]

(15) FUNCTION-DCL[C↓,L↓,L\{N→function(I(PL))}↑] =
(15.2) **function** DECL-ID[L↓,N↑,forwardfunction(I(PL))↑]
tuple[function↓,v(N)↓,TO↓]
table[L_1↓,TO↑]
BODY[C↓,PL**U**{N→localfunction(I(PL))}↓,L_1↑]
tuple[endfunction↓,v(N)↓]

(15) FUNCTION-DCL[C↓,L↓,L**U**{N→forwardfunction(I(PL))↑] =
(15.3) **function id**[N↑]
PARAM-PART[C\L↓,PL↑] **; forward ;**

(15) FUNCTION-DCL[C↓,L↓,L**U**{N→externalfunction(I(PL))}↑] =
(15.4) **function id**[N↑]
tuple[function↓,v(N)↓,TO↓] *table*[PL↓,TO↑]
PARAM-PART[C\L↓,PL↑] **; external ;**
tuple[external↓] *tuple*[endfunction↓,v(N)↓]

(16) PARAMLIST-DCL-PART[C↓,L↓,L_1↑] =
(16.1) **paramlist** { PARAMLIST-DCL[C↓,L↓$^1L_1$↓,L_1↑] ;}+

(17) PARAMLIST-DCL[C↓,L↓,L**U**{N→paramlist(I(PL))}↑] =
(17.1) **id**[N↑] **:=**
PARAM-LIST[C\(L**U**{N→paramlist(recursive(N))})↓,PL↑]
(*If a formal procedure/function uses N inside PARAM-LIST, then its parameter part will get the form: recursive(N).*)

(18) CONST-DCL-PART[C↓,L↓,L_1↑] =
(18.1) **const** { CONST-DCL[C↓,L↓$^1L_1$↓,L_1↑] ; }+

(19) CONST-DCL[C↓,L↓,L**U**{N→const(TO,(passive,S))}↑] =
(19.1) **id**[N↑] **:=** EXP[C\L↓,TO↑,const↑,S↑]

(19) CONST-DCL[C↓,L↓,L_2↑] =
(19.2) **(** CONST-ID[L↓,init↓,L_1↑,V_1↑]
{, CONST-ID[L_1↓$^1L_2$↓,V_1↓$^1V_2$↓,L_2↑L_1↑^{0},V_2↑] }* **)**

(20) CONST-ID[L↓,V↓,L**U**{N→const(c(chr(V),(passive,word))}↑,V+1↑] =
(20.1) **id**[N↑]
(*The second position of CONST-ID is the value to be assigned to the const declared in the construct. The third position returns the next value. init is an arbitrary value. chr converts an integer to a string denoting the integer *)

(21) TYPE-DCL-PART[C↓,L↓,L_1↑] =

(21.1) **type** { TYPE-DCL[C↓,L↓^{1}L$_{1}$↓,L$_{1}$↑] ; }+

(22) TYPE-DCL[C↓,L↓,L**U**{N→type(T)}↑] =
(22.1) **id**[N↑] := TYPE[C\L↓,T↑]

(23) TYPE[C↓,T↑] =
(23.1) NAMED-TYPE[C↓,T↑]
(*the second pos. of TYPE and NAMED-TYPE is the represented type*)

(23) TYPE[C↓,T$_{1}$↑] =
(23.2) **array (**INX-LIST[C↓,T↓,T$_{1}$↑] **) of** TYPE[C↓,T↑]

(23) TYPE[C↓,(K$_{1}$,record(F$_{1}$))↑] =
(23.3) **record** FIELD-LIST[C↓,*e*↓,passive↓,F↑,K↑]
{; FIELD-LIST[C↓,F↓^{1}F$_{1}$↓,K↓^{1}K$_{1}$↓,F$_{1}$↑F↑^{0},K$_{1}$↑K↑^{0}] }*
{;}? **end**

(23) TYPE[C↓,(active,pool(info(passive,S),TO))↑] =
(23.4) **sharedset** EXP[C↓,TO↑,const↑,word↑]
of TYPE[C↓,(passive,S)↑]

(23) TYPE[C↓,(active,pool(noinfo,TO))↑] =
(23.5) **sharedset** EXP[C↓,TO↑,const↑,word↑]
of empty

(24) NAMED-TYPE[C↓,T↑] =
(24.1) DECL-ID[C↓,N↑,type(T)↑]

(24) NAMED-TYPE[C↓,(passive,word)↑] =
(24.2) **word**

(24) NAMED-TYPE[C↓,(passive,byte)↑] =
(24.3) **byte**

(24) NAMED-TYPE[C↓,(reference,reference)↑] =
(24.4) **reference**

(24) NAMED-TYPE[C↓,(active,shadow)↑] =
(24.5) **shadow**

(24) NAMED-TYPE[C↓,(active,semaphore)↑] =
(24.6) **semaphore**

(25) INX-LIST[C↓,T↓,T$_{1}$↑] =
(25.1) INDEX[C↓,T↓,T$_{1}$↑]

(25) INX-LIST[C↓,T↓,T$_{2}$↑] =
(25.2) INX-LIST[C↓,T$_{1}$↓,T$_{2}$↑] , INDEX[C↓,T↓,T$_{1}$↑]

(26) INDEX[C↓,(K,S)↓,(K,array(TO$_{1}$,TO$_{2}$,(K,S)))↑] =
(26.1) EXP[C↓,TO$_{1}$↑,const↑,word↑] ..
EXP[C↓,TO$_{2}$↑,const↑,word↑] *tuple*[arraybounds↓,TO$_{1}$↓,TO$_{2}$↓]

(*The second position of INX-LIST (and INDEX) is the type of the element indexed by that INDEX-list (or INDEX). The third position is the type of the whole array construct including the INX-LIST (or INDEX)*)

(27) FIELD-LIST[C↓,F↓,K↓,F_1↑,K_1↑] =
(27.1) ID-LIST[F↓,field(T)↓,F_1↑] : FIELD-TYPE[C↓,K↓,K_1↑,T↑]

(28) FIELD-TYPE[C↓,K↓,K↑,(passive,S)↑] =
(28.1) TYPE[C↓,(passive,S)↑]

(28) FIELD-TYPE[C↓,passive↓,reference↑,(reference,S)↑] =
(28.2) TYPE[C↓,(reference,S)↑]

(28) FIELD-TYPE[C↓,reference↓,reference↑,(reference,S)↑] =
(28.3) TYPE[C↓,(reference,S)↑]

(28) FIELD-TYPE[C↓,active↓,active↑,(reference,S)↑] =
(28.4) TYPE[C↓,(reference,S)↑]

(28) FIELD-TYPE[C↓,K↓,active↑,(active,S)↑] =
(28.5) TYPE[C↓,(active,S)↑]

(*The third (resp. fourth) position of FIELD-LIST is the kind of the record-fields, up to but excluding (resp. including) the construct. The second and third position of FIELD-TYPE are similar. The fourth position of FIELD-TYPE is the represented type*)

(29) GLOBAL-VAR-DCL-PART[L↓,L_1↑] =
(29.1) **var** { GLOBAL-VAR-DCL[L↓^{1}L_1↓,L_1↑] ; }+

(30) GLOBAL-VAR-DCL[L↓,L**U**L_1↑] =
(30.1) ID-LIST[*e*↓,var(T)↓,L_1↑] : TYPE[L↓,T↑]
INITLIST[L↓,L_1↓,T↓]

(31) LOCAL-VAR-DCL-PART[C↓,L↓,L_1↑] =
(31.1) **var** { LOCAL-VAR-DCL[C↓,L↓^{1}L_1↓,L_1↑] ; }+

(32) LOCAL-VAR-DCL[C↓,L↓,L**U**L_1↑] =
(32.1) ID-LIST[*e*↓,var(passive,S)↓,L_1↑] : TYPE[C\L↓,(passive,S)↑]
INITLIST[C\L↓,L_1↓,(passive,S)↓]
(32.2) | ID-LIST[*e*↓,var(reference,S)↓,L_1↑] : TYPE[C\L↓,(reference,S)↑]

(33) INITLIST[C↓,L↓,T↓] =
(33.1) EMPTY

(34) INITLIST[C↓,L↓,(passive,S)↓] =
(34.1) := *tuple*[initlist,TO↓] *table*[L↓,TO↑]
INITELEMENT[C↓] { , INITELEMENT[C↓] }* *tuple*[endinitlist↓]

(35) INITELEMENT[C↓] =
(35.1) { EXP[C↓,TO↑,const↑,word↑] | EXP[C↓,TO↑,const↑,byte↑] }

```
            tuple[repetition↓,c(1)↓,TO↓]
(35.2)  | EXP[C↓,TO↑,const↑,word↑] *
            { EXP[C↓,TO1↑,const↑,word↑] | EXP[C↓,TO1↑,const↑,byte↑] }
            tuple[repetition↓,TO↓,TO1↓]
        (*The second position of INITLIST is the list of identifiers
          to be initialized *)

(36)    COMPOUND-STATEMENT[C↓,L↓,ND↓] =
(36.1)      begin STATEMENT[C↓,L↓,ND↓]
              { ; STATEMENT[C↓,L↓,ND↓] }*
            {;}? end
        (*The third position of COMPOUND-STATEMENT and STATEMENT is
          the current nesting depth of if and while statements etc. *)

(37)    STATEMENT[C↓,L↓,ND↓] =
(37.1)      COMPOUND-STATEMENT[C↓,L↓,ND↓]
(37.2)  | ASSIGNMENT-STATEMENT[C\L↓,S↑,TO↑]
(37.3)  | repeat tuple[repeat↓,t(ND)↓,TO↓]
              STATEMENT[C↓,L↓,ND+1↓] {; STATEMENT[C↓,L↓,ND+1↓] }* {;}?
            until EXP[C\L↓,TO↑,EK↑,word↑]
            tuple[endrepeat↓,t(ND)↓,TO↓]
(37.4)  | while tuple[while↓,t(ND)↓,TO↓] EXP[C\L↓,TO↑,EK↑,word↑]
            do tuple[whiledo↓,t(ND)↓,TO↓]
              STATEMENT[C↓,L↓,ND+1↓]
            tuple[endwhile↓,t(ND)↓,TO↓]
(37.5)  | if tuple[if↓,t(ND)↓,TO↓] EXP[C\L↓,TO↑,EK↑,word↑]
            then tuple[ifthen↓,t(ND)↓,TO↓] STATEMENT[C↓,L↓,ND+1↓]
            { else tuple[ifelse↓,t(ND)↓,TO↓] STATEMENT[C↓,L↓,ND+1↓] }?
            tuple[endif↓,t(ND)↓,TO↓]
(37.6)  | exit tuple[exit↓]
(37.7)  | halt tuple[halt↓]
(37.8)  | lock tuple[lock↓,t(ND)↓,TO↓] VAR[C\L↓,TO↑,(reference,reference)↑]
            to tuple[lockto↓,t(ND)↓,TO↓] id[N↑]
            : NAMED-TYPE[C\L↓,(passive,S)↑]
            in tuple[lockin↓,t(ND)↓,TO↓,TO1↓]
              table[{N→var(passive,S)}↓,TO1↑]
              STATEMENT[C↓,LU{N→var(passive,S)}↓,ND+1↓]
            tuple[endlock↓,t(ND)↓,TO↓]
(37.9)  | with tuple[with↓,t(ND)↓,TO↓] VAR[C\L↓,TO↑,(K,record(F))↑]
            do tuple[withdo↓,t(ND)↓,TO↓]
              STATEMENT[C↓,L\(FXTO)↓,ND+1↓]
            tuple[endwith↓,t(ND)↓,TO↓]
        (*The fieldlist of the record variable in production 37.9 is extended
          with the value of TO for all defined fields. This is done in order
          to obtain access to the record variable when the fields are referred
```

```
          in the statement. See production 51.2. This is in fact the only
          situation where EXTENDED-DENOTATION is used.*)
(37.10) | case tuple[case↓,t(ND)↓,TO↓] CASE-EXP[C\L↓,TO↑,EK↑,S↑]
            of tuple[caseof↓,t(ND)↓,TO↓]
              CASE-ELM[C↓,L↓,ND+1↓,S↓] {; CASE-ELM[C↓,L↓,ND+1↓,S↓] }*
          {;}? end tuple[endcase↓,t(ND)↓,TO↓]
(37.11) | PROCEDURE-ID[C\L↓,N↑,I(PL)↑]
              tuple[procedurecall↓,v(N)↓,TO2↓,TO↑]
              ACTUAL-PARAMETERS[C\L↓,PL↓,TO↓,e↑,TO1↑]
              tuple[endprocedurecall↓,TO1↓,TO2↑]
(37.12) | PREDEFINED-PROCEDURE-CALL[C\L↓]

(38)   ASSIGNMENT-STATEMENT[C↓,S↑,TO3↑] =
(38.1)     LEFTSIDE[C↓,TO1↑,var(K,S)↑] :=
             { EXP[C↓,TO2↑,EK↑,S↑]
             | ASSIGNMENT-STATEMENT[C↓,S↑,TO2↑] }
           tuple[:=↓,TO1↓,TO2↓,TO3↑]

(38)   ASSIGNMENT-STATEMENT[C↓,word↑,TO3↑] =
(38.2)     LEFTSIDE[C↓,TO1↑,localfunction(P)↑] :=
             { EXP[C↓,TO2↑,EK↑,word↑]
             | ASSIGNMENT-STATEMENT[C↓,word↑,TO2↑] }
           tuple[functionresult↓,TO1↓,TO2↓,TO3↑]
         (*The second position of ASSIGNMENT-STATEMENT is the structure
           of the variables (and/or functions) being assigned. Notice the
           use of localfunction (ref. prod. 21-22).
           The third position is the number of the last assign tuple.*)

(39)   LEFTSIDE[C↓,TO↑,var(passive,S)↑] =
(39.1)     VAR[C↓,TO↑,(passive,S)↑]
(39.2)   | VAR[C↓,TO↑,(reference,S)↑]

(39)   LEFTSIDE[C↓,v(N)↑,localfunction(P)↑] =
(39.3)     DECL-ID[C↓,N↑,localfunction(P)↑]
         (*The second position of LEFTSIDE is the operand to be assigned.
           The third position is the denotation of the operand (variable
           or function). *)

(40)   CASE-EXP[C↓,TO↑,EK↑,word↑] =
(40.1)     EXP[C↓,TO↑,EK↑,word↑]

(40)   CASE-EXP[C↓,TO↑,EK↑,byte↑] =
(40.2)     EXP[C↓,TO↑,EK↑,byte↑]

(41)   CASE-ELM[C↓,L↓,ND↓,S↓] =
(41.1)     CASE-LABEL[C\L↓,ND↓,S↓] {, CASE-LABEL[C\L↓,ND↓,S↓] }*
           : STATEMENT[C↓,L↓,ND↓] tuple[caseelement↓,t(ND)↓]
         (*The last position of CASE-ELM and CASE-LABEL is the
```

structure of the case-expression (word or byte) *)

(42) CASE-LABEL[C↓,ND↓,S↓] =
(42.1) EXP[C↓,TO↑,const↑,S↑] *tuple*[caselabel↓,t(ND)↓,TO↓]
(42.2) | **default** *tuple*[caselabel↓,t(ND)↓,c('default')↓]
(*notice that check for identical case-labels has not been performed*)

(43) PROCEDURE-ID[C↓,N↑,I(PL)↑] =
(43.1) DECL-ID[C↓,N↑,procedure(I(PL))↑]
(43.2) | DECL-ID[C↓,N↑,formalprocedure(I(PL))↑]
(43.3) | DECL-ID[C↓,N↑,externalprocedure(I(PL))↑]
(43.4) | DECL-ID[C↓,N↑,forwardprocedure(I(PL))↑]
(43.5) | DECL-ID[C↓,N↑,formalprocedure(recursive(N_1))↑]
RECURSIVE-PARAM[C↓,N_1↓,paramlist(I(PL))↑]

(44) ACTUAL-PARAMETERS[C↓,PL↓,TO↓,PL↑,TO↑] =
(44.1) EMPTY

(44) ACTUAL-PARAMETERS[C↓,PL↓,TO↓,PL_2↑,TO_4↑] =
(44.2) (PARAM[C↓,PL↓,TO_1↑,PL_1↑] *tuple*[parameter↓,TO_1↓,TO↓,TO_3↑]
{, PARAM[C↓,PL_1↓^{1}PL_2↓,TO_2↑,PL_2↑PL_1↑^{0}]
tuple[parameter↓,TO_2↓,TO_3↓^{1}TO_4↓,TO_4↑TO_3↑^{0}] }*)
(*The second (fourth) position of ACTUAL-PARAMETERS and PARAM is the rest of the parameter list which has to be substituted by actual parameters, excluding (including) the construct. The third position of ACTUAL-PARAMETERS is the number of the tuple with the operation procedurecall. The fifth position is the number of the last parameter tuple. *)

(45) PARAM[C↓,{N→formalconst(K,S)}**U**PL↓,TO↑,PL↑] =
(45.1) EXP[C↓,TO↑,EK↑,S↑]

(45) PARAM[C↓,{N→formalvar(T)}**U**PL↓,TO↑,PL↑] =
(45.2) VAR[C↓,TO↑,T↑]

(45) PARAM[C↓,{N→formalprocedure(I(PL))}**U**PL_1↓,v(N_1)↑,PL_1↑] =
(45.2) PROCEDURE-ID[C↓,N_1↑,PL↑]

(45) PARAM[C↓,{N→formalfunction(I(PL))}**U**PL_1↓,v(N_1)↑,PL_1↑] =
(45.3) FUNCTION-ID[C↓,N_1↑,PL↑]

(45) PARAM[C↓,{N→formalprocedure(recursive(N_1))}**U**PL↓,v(N_2)↑,PL↑] =
(45.4) PROCEDURE-ID[C↓,N_2↑,PL_1↑]
RECURSIVE-PARAM[C↓,N_1↓,paramlist(I(PL_1))↑]

(45) PARAM[C↓,{N→formalfunction(recursive(N_1))}**U**PL↓,v(N_2)↑,PL↑] =
(45.5) FUNCTION-ID[C↓,N_2↑,PL_1↑]
RECURSIVE-PARAM[C↓,N_1↓,paramlist(I(PL_1))↑]

(46) RECURSIVE-PARAM[C↓,N↓,c(N)↑] =

(46.1) EMPTY

(47) EXP[C↓,TO↑,EK↑,S↑] =
(47.1) EXP[C↓,TO_1↑,EK↑,S_1↑] DYADIC[S_1↓,S_2↓,S↑,OP↑]
TERM[C↓,TO_2↑,EK↑,S_2↑] *tuple*[OP↓,TO_1↓,TO_2↓,TO↑]

(47) EXP[C↓,TO↑,var↑,S↑] =
(47.2) EXP[C↓,TO_1↑,var↑,S_1↑] DYADIC[S_1↓,S_2↓,S↑,OP↑]
TERM[C↓,TO_2↑,const↑,S_2↑] *tuple*[OP↓,TO_1↓,TO_2↓,TO↑]
(47.3) | EXP[C↓,TO_1↓,const↑,S_1↑] DYADIC[S_1↓,S_2↓,S↑,OP↑]
TERM[C↓,TO_2↑,var↑,S_2↑] *tuple*[OP↓,TO_1↓,TO_2↓,TO↑]

(47) EXP[C↓,TO↑,EK↑,S↑] =
(47.4) TERM[C↓,TO↑,EK↑,S↑]
(*The second position of EXP and TERM is the number of the tuple being the root of the expression tree. The third position indicates whether it is a constant expression (const) or not (var). The fourth position is the STRUCTURE of the expression. *)

(48) DYADIC[word↓,word↓,word↑,OP↑] =
(48.1) **operator**[OP↑]

(48) DYADIC[word↓,word↓,byte↑,extract↑] =
(48.2) **extract**

(48) DYADIC[byte↓,word↓,word↑,extend↑] =
(48.3) **extend**

(49) TERM[C↓,TO↑,var↑,S↑] =
(49.1) VAR[C↓,TO↑,(K,S)↑]

(49) TERM[C↓,TO↑,var↑,word↑] =
(49.2) FUNCTION-CALL[C↓,TO↑]

(49) TERM[C↓,TO↑,const↑,S↑] =
(49.3) DECL-ID[C↓,N↑,const(TO,(K,S))↑]

(49) TERM[C↓,c(V)↑,const↑,word↑] =
(49.3) **const**[V↑]

(49) TERM[C↓,c(CH)↑,const↑,byte↑] =
(49.4) **char**[CH↑]

(49) TERM[C↓,c('true')↑,const↑,word↑] =
(49.5) **true**

(49) TERM[C↓,c('false')↑,const↑,word↑] =
(49.6) **false**

(49) TERM[C↓,TO↑,const↑,word↑] =

```
(49.7)    CONST[V1↑] ↑ CONST[V2↑]
            tuple[radix↓,c(V1)↓,c(V2)↓,TO↑]

(49)   TERM[C↓,TO↑,EK↑,S↑] =
(49.8)    ( EXP[C↓,TO↑,EK↑,S↑] )

(49)   TERM[C↓,TO↑,EK↑,word↑] =
(49.9)    MONADIC[OP↑] TERM[C↓,TO1↑,EK↑,word↑] tuple[OP↓,TO1↓,TO↑]

(50)   MONADIC[neg↑] =
(50.1)    neg

(50)   MONADIC[not↑] =
(50.2)    not

(51)   VAR[C↓,v(N)↑,T↑] =
(51.1)    DECL-ID[C↓,N↑,var(T)↑]
        | DECL-ID[C↓,N↑,formalvar(T)↑]

(51)   VAR[C↓,TO↑,T↑] =
(51.2)    DECL-ID[C↓,N↑,(field(T),TO1)↑] tuple[field↓,TO1↓,v(N)↓,TO↑]
(51.3)  | VAR[C↓,TO1↑,(K,record(F))↑] . DECL-ID[F↓,N↑,field(T)↑]
            tuple[field↓,TO1↓,v(N)↓,TO↑]
(51.4)  | ARRAY-VAR[C↓,TO↑,T↑] )

(52)   ARRAY-VAR[C↓,TO↑,T↑] =
(52.1)    { VAR[C↓,TO1↑,(K,array(TO3,TO4,T))↑] (
          | ARRAY-VAR[C↓,TO1↑,(K,array(TO3,TO4,T))↑] , }
          EXP[C↓,TO2↑,EK↑,word↑] tuple[index↓,TO1↓,TO2↓,TO3↓,TO4↓,TO↑]

(53)   FUNCTION-CALL[C↓,TO↑] =
(53.1)    FUNCTION-ID[C↓,N↑,I(PL)↑]
            tuple[functioncall↓,v(N)↓,TO2↓,TO↑]
            ACTUAL-PARAMETERS[C↓,PL↓,TO↓,e,TO1↑]
            tuple[endfunctioncall↓,TO1↓,TO2↑]

(54)   FUNCTION-ID[C↓,N↑,I(PL)↑] =
(54.1)    DECL-ID[C↓,N↑,function(I(PL))↑]
(54.2)  | DECL-ID[C↓,N↑,formalfunction(I(PL))↑]
(54.3)  | DECL-ID[C↓,N↑,externalfunction(I(PL))↑]
(54.4)  | DECL-ID[C↓,N↑,forwardfunction(I(PL))↑]
(54.5)  | DECL-ID[C↓,N↑,formalfunction(recursive(N1))↑]
            RECURSIVE-PARAM[C↓,N1↓,paramlist(I(PL))↑]

(55)   PREDEFINED-PROCEDURE-CALL[C↓] =
(55.1)    alloc tuple[alloc↓,TO4↓,TO↑]
            VAR[C↓,TO1↑,(reference,reference)↑]
          from tuple[parameter↓,TO1,TO↓,TO2↑]
            VAR[C↓,TO3↑,(active,pool(PI))↑]
```

tuple[endalloc↓,TO_3↓,TO_2↓,TO_4↑]

(55.2) | **alloc** *tuple*[alloc↓,TO_6↓,TO↑]
VAR[C↓,TO_1↑,(reference,reference)↑]
from *tuple*[parameter↓,TO_1↓,TO↓,TO_2↑]
VAR[C↓,TO_3↑,(active,pool(PI))↑]
with *tuple*[parameter↓,TO_3↓,TO_2↓,TO_4↑]
VAR[C↓,TO_5↑,(active,semaphore)↑]
tuple[endalloc↓,TO_5↓,TO_4↓,TO_6↑]

(55.3) | **return** *tuple*[return↓,TO_2↓,TO↑]
VAR[C↓,TO_1↑,(reference,reference)↑]
tuple[endreturn↓,TO_1↓,TO↓,TO_2↑]

(55.4) | **signal** *tuple*[signal↓,TO_4↓,TO↑]
VAR[C↓,TO_1↑,(reference,reference)↑]
to *tuple*[parameter↓,TO_1↓,TO↓,TO_2↑}
VAR[C↓,TO_3↑,(active,semaphore)↑]
tuple[endsignal↓,TO_3↓,TO_2↓,TO_4↑]

(55.5) | **wait** *tuple*[wait↓,TO_4↓,TO↑]
VAR[C↓,TO_1↑,(reference,reference)↑]
from *tuple*[parameter↓,TO_1↓,TO↓,TO_2↑]
VAR[C↓,TO_3↑,(active,semaphore)↑]
tuple[endwait↓,TO_3↓,TO_2↓,TO_4↑]

(55.6) | **setcode** *tuple*[setcode↓,TO_3↓,TO↑]
EXP[C↓,TO_1↑,EK↑,word↑]
in *tuple*[parameter↓,TO_1↓,TO↓,TO_2↑]
VAR[C↓,TO_3↑,(reference,reference)↑]
tuple[endsetcode↓,TO_3↓,TO_2↓,TO_4↑]

(55.7) | **readcode** *tuple*[readcode↓,TO_4↓,TO↑]
VAR[C↓,TO_1↑,(passive,word)↑]
in *tuple*[parameter↓,TO_1↓,TO↓,TO_2↑]
VAR[C↓,TO_3↑,(reference,reference)↑]
tuple[endreadcode↓,TO_3↓,TO_2↓,TO_4↑]

(55.8) | **load** *tuple*[load↓,TO_4↓,TO↑]
DCL-ID[C↓,N↓,process(P)↑]
from *tuple*[parameter↓,v(N)↓,TO↓,TO_1↑]
EXP[C↓,TO_2↑,EK↑,word↑]
report *tuple*[parameter↓,TO_2↓,TO_1↓,TO_3↑]
PROCEDURE-ID[C↓,N_1↑,P_1↑]
tuple[endload↓,v(N_1)↓,TO_3↓,TO_4↑]

(55.9) | **create** *tuple*[create↓,TO_7↓,TO↑]
VAR[C↓,TO_1↑,(active,shadow)↑]
like *tuple*[parameter↓,TO_1↓,TO↓,TO_2↑]
PROCESS-CALL[C↓,TO_3↑]
with *tuple*[parameter↓,TO_3↓,TO_2↓,TO_4↑]
EXP[C↓,TO_5↑,EK↑,word↑]

report *tuple*[parameter↓,TO_5↓,TO_4↓,TO_6↑]
PROCEDURE-ID[C↓,N↑,P↑]
tuple[endcreate↓,v(N)↓,TO_6↓,TO_7↑]

(55.10) | **start** *tuple*[start↓,TO_3↓,TO↑]
VAR[C↓,TO_1↑,(active,shadow)↑]
report *tuple*[parameter↓,TO_1↓,TO↓,TO_2↑]
PROCEDURE-ID[C↓,N↑,P↑]
tuple[endstart↓,v(N)↓,TO_2↓,TO_3↑]

(55.11) | **stop** *tuple*[stop↓,TO_3↓,TO↑]
VAR[C↓,TO_1↑,(active,shadow)↑]
report *tuple*[parameter↓,TO_1↓,TO↓,TO_2↑]
PROCEDURE-ID[C↓,N↑,P↑]
tuple[endstop↓,v(N)↓,TO_2↓,TO_3↑]

(55.12) | **remove** *tuple*[remove↓,TO_3↓,TO↑]
VAR[C↓,TO_1↑,(active,shadow)↑]
report *tuple*[parameter↓,TO_1↓,TO↓,TO_2↑]
PROCEDURE-ID[C↓,N↑,P↑]
tuple[endremove↓,v(N)↓,TO_2↓,TO_3↑]

(55.13) | **unload** *tuple*[unload↓,TO_2↓,TO↑]
DECL-ID[C↓,N↑,process(P)↑]
report *tuple*[parameter↓,v(N)↓,TO↓,TO_1↑]
PROCEDURE-ID[C↓,N_1↑,P_1↑]
tuple[endunload↓,v(N_1)↓,TO_1↓,TO_2↑]

(56) PROCESS-CALL[C↓,TO↑] =
(56.1) DECL-ID[C↓,N↑,process(I(PL))↑]
tuple[processcall↓,v(N)↓,TO_2↓,TO↑]
ACTUAL-PARAMETERS[C↓,PL↓,TO↓,*e*↑,TO_1↑]
tuple[endprocesscall↓,TO_1↓,TO_2↑]