tanulmányok

50/1976

MTA Számítástechnikai és Automatizálási Kutató Intézet Budapest

COMPUTER AND AUTOMATION INSTITUTE
HUNGARIAN ACADEMY OF SCIENCES

ROBERT TREER

A SYNTAX MACRO DEFINITION LANGUAGE

Tanulmányok 50/1976

A kiadásért felelős:
Dr Vámos Tibor

ISBN 963 311 022 X

768116 MTA KESZ Sokszorosító. F. v.: Szabó Gyula

C O N T E N T

PREFACE

The present paper is a description of the language MADE (Syntax MAcro DEfinition language). It presents a version of the language general enough for the considered field of applications and, in the same time, consolidated enough to call for criticism and to be a base for further refinements.

After an introduction which deals mainly with the possible applications, the syntax of the language is given in a meta-language. Parallel with the syntax definition we often use MADE program fragments to illustrate the semantics of syntax -units as well as to give a few hints to the application of the language. We hope that the reader does not mind being introduced to MADE by working through himself on examples.

Until now there is only one implemented version of MADE which includes about 90 percent of the language given in this paper. The complete description of this implementation (for Hewlett-Packard 2100 computers under the operating system TSB 2000 F/C in assembler code) is available at the author under the title "HP-BASIC MADE System description and user's manual". (I.E.I. publication B74-40). A similar implementation might require about 3 months man-labour. We give only a short abstract of the implementation in this paper.

The author wishes to thank:

Prcf.G. Capriz, director of the Istituto di Elaborazione della Informazione del C.N.P., who made possible for him to work on this theme.

0. INTRODUCTION

0.1 General Introduction

MADE is a language independent general purpose syntax-macro definition language implemented by a free-format template matching macroprocessor. It is a translator writing system and therefore, it has two components: a descriptive language (a modification of the well-known Backus-Naur metalanguage) and a procedural language in which semantic routines can be written.

The idea of associating a semantic routine with each rule of a grammar is not new. However, the general trend is to separate the syntax from semantics as much as possible: when the syntax-analyser recognises a syntactic construct, it calls a semantic routine which checks for semantic correctness and does semantic processing.

In MADE we used a different approach: the sintactic and semantic definition of a language unit are inseparable, they form a macro definition or a structure definition MADE syntax entity. Writing a definition in MADE, the programmer has to decide what he want to be syntax-directed and what he leaves for the semantic processing. In this way he may build up syntax-directed translators, but may leave the whole parsing to the procedures of the particular language elements (operators, identifiers, reserved words, etc.). Whilst the former gives a clear picture of the language under definition, the latter often produces a more efficient compiler (See Ch. 1.3). In fact, one of the basic problems of a MADE programmer is to "optimize", that is, to choose among these two extrem and a lot of intermediate possibilities.

The result of the execution of a MADE program will be a "program text" (it can be in intermediate codes, in assembler or in a high-level language). This program text

is the "variable part" of the generated translator of the "new language" defined by the MADE program. The "constant part" of the translator is a part of the MADE system. There exist special data-types and operations to facilitate text-generation and the programmer may use a necessary number of segments to build up fragments of the generated text.

The design aim of MADE (and especially that of BASIC MADE, the implemented version) was to define a system, which can be implemented easily on small computers and in the same time, as a general system-building tool can be used on a wide field of applications. With a 4K implementation (2.5K assembler program, 1.5 K for tables, buffers) behind us we feel, that we may suggest it for implementation also on mini-computers.

MADE programs can be applied to realize software portability, to define special purpose languages, to extend existing programming languages, to translate from one language to another, to generate operating systems, to write in abbreviated form everything which has a pattern. They can be used for text-edition and correction, for data-format conversion and for program parameterisation, too.

0.2 System description, hints on implementation.

A MADE program contains the definition of a language (we often call it a "new language" but it is not necessarily new) and the specification of its translator. (Standard parts of a compiler, such as scanner, symbol-table routines, etc. are included in the MADE language as standard procedures to help the programmer.)

The translation of a MADE program by the MADE compiler occurs at meta-compile time. The resulting translator can then be executed - this occurs at the usual compile-time. Its execution is controlled by the syntax description given in the MADE program (usually by other parts of this program, too) and by the source "program" it is translating. The source "program" can be any string (which is accepted by the input devices of the given computer: it can be a data-structure, simple text, the "flowdiagram" of a plant, a programming language, etc. .

A suggested way to implement a MADE "compiler" is to agree on a set of suitable intermediate codes (about 60 may well do) and making use of the main restriction in the language that the declaration of a variable must precede all use of that variable, to write a simple one-pass compiler which will produce the variable part of the translator of the defined language in intermediate codes. (Instead of a formal parsing algorithm we used a bottom-up recognizer programmed in an intuitive manner). The heart of this system will be a simple structured interpreter which at interpretation-time will be completed by the "intermadiate-code translator" and by the filled - in part of the symbol table.

An implementation using interpreter-technique generally requires less man-labour; besides in this way it is easier to produce good run-time debugging facilities.

The structure of the MADE language is fairly simple. There is only a very simple block-structure, no nested conditionals, no case statements and so forth. MADE is basically line-oriented and the type of a statement in a line can almost always be determined from just the first symbol or two.

The semantic-language part of MADE has almost all of the conventional constructs one has in other procedural languages:

1. data types: integer, boolean, string,
2. simple variables and one-dimensional arrays,
3. assigment and conditional statements,
4. I/O facilities.

In addition, MADE has other elements to help in translator--writing:

1. pointer type (atom and procedure)
   atomclass type and types defined by the programmer
2. stacks,
3. special string generation and manipulation statements,
4. primitives for entry into and searching in tables, universal scanner (lexical analyser), generalized I/O routines.

Both the MADE compiler and the generated translator will use symbol tables. For the former it is convenient to use a simple linear symbol table- the information obtained from local declarations may be stored and may be forgotten parallel with the opening and closing of "blocks".

On the contrary, the generated translator will be supplied with a standard "chained-hash" type symbol table, which makes possible a much more efficient and general table-handling but in the case of using it for the definition of block-structured languages, the implementation might require a garbage-collector, too.

The storage administration of the generated translator can be very simple. A large table of continuous locations may be used as a stack, which is initially empty. The locations should be assigned to declared variables, working variables and string parameters of the MADE program in metacompile--time. When a procedure (which may be also the procedure of a macro or a "structure") is invoked, it should take enough storage for its constant data area from the current top of the stack. However, in case of variable number of parameters in macros or in structures parallel with the scanning of "syntactically correct" parameters, additional locations of the stack could be used, too. In HP-MADE we chained the locations corresponding to different occurrences of a syntactical entity being present in the parameter part of a macro or structure definition. This method makes possible to write definitions with an undetermined number of parameters.

When the procedure returns to the point of the call, it "pops" the stack, freeing all the locations it used.

0.3. APPLICATIONS

0.3.1. The definition of special-purpose problem-oriented languages and their compiler.

There exists a need for a wide variety of programming languages, thus we have scientific languages, data processing languages, languages for list processing, simulation languages, etc. . It is true, that a good general purpose language can almost always be used for any application but this is often at the expense of considerable inefficiency and obscurity.

The programmer's need is to have a language which is appropriate and natural for his particular application. Yet new computer applications appear regularly (and move frequently in countries without a full-scale computerization) and the approach of designing a new language for a new application and writing its compiler (by hand) is rarely an economic solution, nor is it a solution which can be easily realized.

Moreover, the computers may be found at almost any level of the modern socienty: in factories, in offices, in universities. People of different professions use them and their usual way to draw up a problem is often far away from the "style" of the programming language at hand. There is a need for special, "non-programming-languages" which are close to the conventional professional usage - less coding error, less computer - and man-labour, less obstacle on the way of introducing computers onto new fields of applications may be the outcome.

The MADE language is suitable for this type of applications. In chapter 6. we give a MADE program for the definition of a special purpose language which is basically the "flow diagram-language" ("blue print") of chemical plants for simulation, control or optimization of the plant.

0.3.2. Abbreviation of patterns, text editing and correction, data-format conversion.

The MADE programmer may define macros which enable their user to write in abbreviated form anything which involves considerable repetition of certain patterns, even though the repetition is with variations too elaborate to permit simple use of "ditto".

These is no restriction as far as the nature of the "pattern-language" is concerned, but it must be able to serve as input to a computer, of course. We give a very simple example (Chapter 4.3) how to generate from an abbreviated form a pattern-language: a kind of nursery rhyme. (This example is not intended to solve any problem of computer-poetry).

0.3.3. Extension of existing programming languages.

MADE programs may be applied to provide the users of a particular computer language with simple means of adding extra statements and other syntactic forms to the language making it more appropriate for a special field of applications. Such an extended language may become the common language of a team working on the same project.

The adding of convenient abbreviations, program parameterisation (e.g. a parameter may determine whether debugging statements have to be included into the program) are belonging to this class of applications.
In chapter 3.2.3 we show how to extend FORTRAN-IV by a "when statement".

0.3.4. Portability by MADE

Software is an expensive commodity and being practically one hundred per cent intensive labour it is likely to remain so.

It is a wide-spread wiew in the computing community that we are in the midst of a software crisis. The main difficulty is the immense magnitude of the task of providing operating systems, compilers and application programs for new and existing computers. The example of the IBM is well known: whilst in 1953 they supplied about 10.000 lines of code to their model 650, in 1962 for the IBM 7090 they provided over 100.000 lines. In 1964 when the 360 serie was released one million lines of programming system were provided for it and by 1968 they reached the 8 million lines.

A large proportion of this software effort is exhausted in rewriting existing software for different computers.

One solution to this problem is to write programs in machine independent form: to produce mobile or portable software.

Of course, the portability of application programs may be easily improved by writing them in FORTRAN, ALGOL or COBOL.

Unfortunately, for systems software the picture is less good. Most systems software is written in assembly language and can only be transferred by complete recoding.

A technique for writing a mobile program is to code it using only macrocalls. To transfer such a system to another machine is merely necessary to supply a MADE program with a set of macrodefinitions fot the mapping of programs into the symbolic assembler language of the target-machine.

The idea of a descriptive language for writing software is that we first decide what statements and data types we need to write the software and then define an appropriate language embedding these features. (A similar idea is the so called abstract machine concept).

By writing a MADE program we can map each statement of the descriptive language into assembler statements for any machine on which it is desired to implement the software.

Having implemented in this way a descriptive language we can write the software in a language which contains all of the necessary facilities for writing it with ease, but no more. ("It is better to tailor the software writing language to the software than vice-versa" (Brown).)

Such a scheme is efficient and easily portable. It can be transferred to another machine by writing a set of macros (about 30 in a typical case) to map the software into the assembler language of the target computer.

0.3.5. Translation from one programming language to another.

Reprogramming is one of the great unsolved problems of computing today. During the life of a computer a lot of money may have been spent on developing programs for it, and when it comes to replace the machine, the saving of this investment is of utmost importance.

A solution to the problem of moving assembly language programs from one machine to another may be the use of a macroprocessor like MADE. When the two computers have a high degree of similarity in the number and length of central processor registers and instruction formats, an efficient translation may be produced.

MADE can be used to translate from one high-level language to another or to generate a compiler of a high--level but we do not consider this type of application as typical for reasons of efficiency. However, the question is often not to have an efficient compiler for a language a particular software is written in but to have this software or not. In such a case MADE could give a quick solution.

In chapter 5.3. we give a mapping of general arithmetic expressions to Hewlett-Packard assembly code.

0.3.6. System macro generation.

Virtually all modern computers run under the aegis of an operating system. The heart of the operating system is the supervisor: a master control program which runs concurrently with the user program and performs for it such activities as I/O, dump, restart, protecting the programs & the supervisor itself, program loading, etc.. A program running in such an environment will necessarily make numerous calls of the supervisor. Usually a number of system macros are provided for such calls, because otherwise a supervisor call should involve two or more lines of code containing special flags.

A MADE program can be written for the definition of such system macros, that is, to generate a new language for the communication with the supervisor. The program may make use of an existing operating system and modify only this communication language (if it was not satisfying from some point of view) but by defining the "meaning" of the system macros in machine code it may generate a completely new operating system, too. (However, for this application it might be necessary to

make some modifications on the proposed mode of implementation as far as efficiency is concerned).

0.4. Further work.

We plan more implementations and more applications (more practical ones). We are sure that the use of MADE will raise further problems and their solution may cause minor changes on the present form(s) of the language.

We are particularly interested in an improvement of MADE so that it could be "portable". We think that after some minor modifications MADE could be implemented on other computers with relative ease by a bootstrapping technique. (Chapters 1.3,2.3,5.3 may be considered as a first step in this direction).

An other point which needs further clarification is the role of recursivity in building up such system. In the proposed version a macro or structure call may not appear in its own definition. It is not an inevitable restriction, because MADE has a dynamic data area for every execution of a procedure, but it seems that the interdependence of syntax and semantics which is one of the main peculiarityes of the language, if we do not make this restriction, may give rise to difficulties.

0.5. Introduction to the syntax definition of MADE.

In the following chapters we define the syntax of MADE in a meta-language which seems to be somewhere middleway between the well-known BNF meta-language and the notation one uses in the macro and structure definitions of MADE programs. (We want to facilitate by the language of the syntax description the apprehension of a part of MADE itself).

In the syntax description the use of brackets [and] serves to denote syntax-allowed optional repetition of the terms included. If the right bracket is followed by an integer number, this gives the maximum number of repetitions allowed, which is otherwise an implementation constant. The minimum number of possible repetitions is 0. The terms inside a bracket pair must be repeated all together (or none of them), except if internal bracket-pairs make optional some of them.

Bracket-pairs define the range of the "immediate inside"! signs. (! replaces the BNF / sign).

It is necessary to make a sharp distinction between the symbols of the metalanguage and those of the language being described.

The method we use in a macro definition or in a structure definition is to write between quotation marks the symbols of the latter language. This method avoids ambiguity but perhaps seems a bit clumsy.

This is why in the description of MADE syntax we use heavy type for the < > ! [ ] symbols of the language MADE in places where not making a distinction could imply ambiguity.

To avoid the proliferation of dialects and with consideration on implementations on minicomputers, in MADE we use a minimum character set. (<character> ::=...)

This is the following:

space ! " $ % ( ) * + , - . / : ; < = > ? [\ ] #
"car return" "line feed", digits,

the letters of the English alphabet.

There is no distinction made between capital and small letters. In the examples we denote a line feed car return in a generated text by %.

In MADE:

<identifier> ::= <letter>[<letter> ! <digit>]
<integer constant>::= <digit> [<digit>]
<symbol>::= <identifier> ! <integer constant> !

space ! ! ! " ! $! % ! (!) !*!+!-,! , !-!.!
/=!=! /\ ! \/ !:=!:!;! <=! < !>>!
>!?![!]

The identifiers may be defined as newtype, attribute, newprocedure names, as atom, atomclass and newtype constants. They may be used as labels, macroparameters and structure names. However, certain identifiers are reserved and are not to be used in this way by the programmer. These are: madbegin madend begin end head tail segments newtype attributes integer boolean procedure atom atclass marker text astack ostack ivector avector newprocedure return next same new goto if then gener close delete call comment length no none true false default error scan look sclook scback copy dummy maca acla state empty symb string this last input standard.

The given set of reserved words makes possible an implementation in wich only the first 4 characters of an identifier are taken into consideration. However, such an implementation migh allow the use of longer identifiers to provide that redundancy which human beings like and computers prefer to be without.

The reserved identifiers may be used as macro names making possible to define by MADE macrodefinitions a language which is formally equivalent with MADE itself.

In the description of the MADE language we separated the syntactical and semantical parts. However, we made a concession to the "MADE approach" in which their interdependence is emphasized: an <identifier> having taken part in a declaration of some type becomes (by semantics) <declared ... variable> (of the same type as the declaration was). The identifier after the reserved word newprocedure will be referenced as <declared newprocedure> and the row of characters in a <newtype declaration> will become <declared newtype constant> (of the same newtype). The <atom constant> and <atomclass constant> categories of a <made program> born the same way -- from the evaluation of the <atomdefinition part>.

"Atom" has the same meaning as symbol. We use this term to make distinction between the different language-levels: atoms are the symbols of the language we define by a MADE program.

We tried to write meaningful examples to demonstrate the syntax. As a consequence they usually involve much more syntactical element than the chapter they close. They may need some cross-word decoding work but we hope that more frequently they prove to be self-explaining.

1. THE STRUCTURE OF MADE PROGRAMS

1.1 Syntax

```
<MADE program>          ::=<program start>
                           <instruction part>
                           <program finish>

<program start>         ::=MADBEGIN [ <comment> ] ;
<program finish>        ::=MADEND [ <comment> ] ;
<instruction part>      ::=[ <instruction>   ]

<instruction>           ::=<declaration> ! <statement> !
                           <definition>

<declaration>           ::=<segment declaration> !
                           <newtype declaration> !
                           <attribute declaration> !
                           <variable declaration> !
                           <newprocedure declaration>

<variable declaration>  ::=<simple variable declaration> !
                           <text declaration> !
                           <stack declaration>
                           <vector declaration>

<statement>             ::=[ <label> :]
                           <unlabelled statement>

<unlabelled statement>  ::=<assignment statement> !
                           <conditional statement> !
                           <goto statement> !
                           <text generation> !
                           <stack operation> !
                           < call statement> !
                           <input specification> !
                           <comment statement>
```

```
<definition>              ::=<atom definition part> !
                             <macro definiton> !
                             <structure definition>

<atom definition part>    ::=<character class definition>
                             <atom class definition part>

<macro definition>        ::=$ <macro name> [ <parameter part> ]==
                             <macro body>

<macro body>              ::=[ HEAD ]!
                             [ <local instruction> ]
                             [ TAIL ]!
                             [ <local instruction> ]
                             % [ <comment> ] ;

<macro name>              ::=<atomclass constant> ! <keyword atom>
<keyword atom>            ::=  " <row of non-" characters>"
<local instruction>       ::=<variable declaration> ! <statement>

<structure definition>    ::=$ <structure name>::=
                             <structure syntax>
                             ==<structure body>

<structure name>          ::=< <identifier> >
<structure syntax>        ::=<parameter part>
<structure body>          ::=<macro body>
```

1.2 Semantics

The place of a particular instruction in the program flow is restricted by the following semantical rules:

1. The declaration or definition of a language element must precede all of its application occurences.
2. Every "open block" symbol (MADBEGIN , NEWPROCEDURE , BEGIN , $, HEAD) must have one and only one corresponding "close block" symbol pair (MADEND , RETURN , END , % , TAIL, respectively).
3. There may be only one <atom definition part>,<segment declaration> and <attribute declaration> in the program. The <attribute declaration> must precede all use of attributes.

We will often talk about a variable being global to several program segments. This means that this variable is declared in the main block (the one between MADBEGIN and MADEND).

1.3 Example

In this example we describe the supervisor part of the implemented HP-MADE compiler-interpreter -- in MADE . The example reflects the simplicity of the "bottom-up recognizer" used. We hope that it also gives an idea about the procedural part of the MADE language.

The example presents one of the cases when a formal definition (that of the structure of MADE programs) based on macro definitions should be less efficient.

```
MADBEGIN basic made;
NEWTYPE (typt: intv, boolv, atomc, atomv, procv, procc,
               atclv, atclc, textv, textc, astav, ostav, labv,
               labc);
```

```
ATTRIBUTES PACKED ( typt type, BOOLEAN dim, INTEGER pointer,
                    INTEGER  pri(3), BOOLEAN uary, BOOLEAN
                    bary);
COMMENT the standard attribute MACA is of procedure type and
        denotes the procedure of the macro having the refer-
        enced atom as its keyword. The standard attribute ACLA
        denotes the atomclass value of the referenced atom.
NEWPROCEDURE smerror;
COMMENT error message and actual atom printing;
RETURN;
ERROR:= smerror;
COMMENT ERROR is a standard procedure variable of MADE.
        Newprocedure supvsave saves the actual state of the
        system , newprocedure supvrestore restores it ;

NEWPROCEDURE supverror ;
CALL ERROR ;
CALL supvrestore ;
RETURN ;

GOTO definitions;

ATOM a, chat;
INTEGER gbi,icind,iw;
COMMENT icind is the counter of the generated intermediate
        codes, gbi is the index of the next free element of
        array  gbs , which is used as a label-table;
IVECTOR gbs(4ØØ);

start : CALL SCAN;
        IF <THIS> = "MADBEGIN"
        THEN GOTO calab;
        CALL supverror;
        GOTO start;
scan1 : CALL SCLOOK ;
```

```
calab : a:=<THIS> ;
        IF a.MACA = NONE THEN GOTO elsl;
        CALL supvsave;
        CALL a.MACA;
        GOTO scanl;
elsl  : IF a.ACLA /= identifier THEN
        GOTO assignl;
        IF a.type = NONE \/ a.type = labvar THEN
        BEGIN
        CALL SCAN;
        IF <THIS>/= " : " THEN CALL supverror;
        iw:=gbi;
        IF a.type=labv THEN BEGIN iw:=a.pointer;
                                  GOTO 11; END;
        a.pointer:=gbi;
        gbi:=gbi+1;
        a.type:=labc;
11    : gbs.iw:=icind:
        GOTO scanl;
        END ;
assignl:until:= " := " ;
       CALL savun;
        until:= " : " ;
        CALL savun;
COMMENT Savun scans atoms until finds an atom which is equal
        (until) and "saves" them for further processing.
        (They must represent a <variable> and a <value> ). ;

        CALL store;
COMMENT Store generates codes corresponding to <assigment
        statement> -s. It may be the procedure of a macro
        named " := " ;

        GOTO scanl;
```

```
DEFINITIONS:

$ "GOTO" <iden!.type=NONE \/.type=labc> ==
  INTEGER gbv;
  gbv:=gbi;
  IF  .type=NONE THEN
                BEGIN  .type:=labv: .poin:=gbi  ;
                       gbi:=gbi + 1; GOTO 11;
                END ;
  gbv:= .pointer;
11 : GENER 1," JMP L", SYMB gbv;
% Intermediate code generation corresponding to the goto
statement resulting in a JMP code with the label-table index
of the label ;
```

2. DECLARATIONS

2.1 Syntax

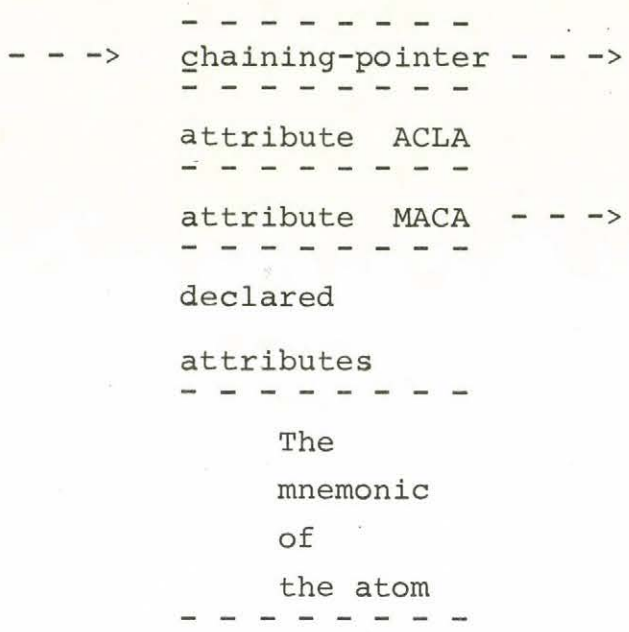
```
<segment declaration>      ::=SEGMENTS <integer constant>
                              [,<integer constant> ] ;

<newtype declaration>      ::= NEWTYPE
                              (<identifier>:<cosntant list>)
                              [,(<identifier>:<constant list>)];
<constant list>            ::=<newtype constant>
                              [,<newtype constant>]

<newtype constant>         ::=<atomconstant>
<attribute declaration>    ::=<simple attribute declaration>!
                              <packed attribute declaration>

<simple attr.declaration>  ::=ATTRIBUTES
                              (<simple v. declarator><identifier>
                              [ , <simple v.declarator>
                              <identifier>]);

<packed attr. declaration>::= ATTRIBUTES PACKED
                              (<simple v. declarator>
                              <identifier>[(<bitnumber>)]1
                              [,<simple v. declarator>
                              <identifier>[(<bitnumber>)]1 ]);

<bitnumber>                ::=<integer constant>

<simple v. declarator>     ::=<standard simple v. declarator>!
                              <newtype declarator>
<standard simple v. declarator>
                           ::=INTEGER ! BOOLEAN ! PROCEDURE !
                              ATOM ! ATCLASS ! MARKER
<newtype declarator>       ::=<declared newtype declarator>
```

```
<simple variable declaration>
                              ::=<simple v. declarator>
                                 <identifier>
                                 [ , <simple v. declarator>
                                 <identifier> ] ;

<text variable declaration>::= TEXT <name and length list>;

<atomstack v. declaration>::=ASTACK <name- and length list>;
<outstack v. declaration> ::=OSTACK <name- and length list>;

<integer vector variable declaration>
                              ::=IVECTOR <name- and length list>;
<atom vector variable declaration>
                              ::=AVECTOR <name- and length list>;

<name- and length list>    ::=<identifier>[(<length>)]1
                                 [,<identifier>[(<length>)]1]

<length>                   ::=<integer constant>

<newprocedure declaration>::=NEWPROCEDURE<identifier>;
                                 [ <local instruction> ]
                                 RETURN [ <comment> ] ;
```

2.2. Semantics.

An implementation will necessarily limit the maximum number of segments and their length. The most natural implementation of segments is a realization by files.

An atomtable entry of the generated language will have the following structure:



The standard attribute acla (atom class attribute) contains the internal representation of the atomclass value of the particular atom. If the atom is a macroname the standard attribute maca (macro attribute) will be a pointer to the generated procedure of the macro, otherwise it is none.

The use of a packed attribute declaration will result in compressed entries and thus in a better memory utilization.

Every element of an output stack is a text. We found convenient to denote the actual length of a text by changing the content of the last cell it occupies to negative. (It needs only a bit of space).

A common practice in programming is to make everything that may be input to a subroutine a formal parameter of that routine. While this method produces general independent procedures, in compiler writing it is often better to make use of global parameters to get more efficient code by an easier implementation. This is why in the present language the procedures do not have parameters. A procedure in its declaration is called "new procedure". The declaration of a <new procedure> will result a <procedure value> which will be denoted by its name, the identifier after the reserved work PROCEDURE in the newprocedure declaration.

2.3. Example

In this example we continue the definition of the BASIC-MADE and its compiler in MADE. In the examples we demonstrate a "real" compiler (not an interpreter as in our implementation), which generates HP assembler code.

```
INTEGER wri;
wri:=Ø ;
NEWPROCEDURE genworkvar:
GENER 2, "WR", SYMB wri , "OCT Ø %" :
wri:= wri + 1 ;
RETURN generate working variable ;
COMMENT wri is a global counter . It is advised to clear
it now and then ;

$ "INTEGER" <ide ! .type = NONE>
            [ "," <ide ! . type = NONE>. NEXT ] ==
     wi:=1 ;
L :  chat:=<ide.wi>;
     chat.type:=intv;
     GENER 2, SYMB chat , "OCT Ø % " ;
     wi:=wi+1 ;
     IF wi <= ide.NO THEN GOTO 1;
% Segment 2 will contain the memory locations reserved for
  assembler variables. The identifiers of the HP-MADE are
  valid identifiers in the HP-assembler , this is why we
really do not use the attribute pointer ;

$ "IVECTOR" <ide ! .type=NONE> ["("<icons>·1")"]1
              [ "," <ide ! .type=NONE>.NEXT
            [ "(" <icons>.SAME ")" ]1 ] ==
wi:=1;
L :chat:=<ide.wi>;
   A:=<icons.wi>.
   IF A=NONE THEN a:=1Ø;
```

```
   chat.type:= intv ;
    chat.dim:=TRUE;
   GENER 2,SYMB chat , "BSS" ,SYMB a , "%" ;
% The default length of vectors is 1Ø ;
```

3. DEFINITIONS

3.1 The definition of atoms and atomclasses

3.1.2 Syntax

```
<atom definition part>::=
                <special atom definition part>
                <character class definition part>
                <atomclass definition part>
                        <end-of-line sign>

<special atom definition part>::=
                [ C <standard special atom>
                    <assigned special atom> ,]

<standard special atom>
                <end-of-line sign> !
                <metasymbol begin sign> !
                <metasymbol end sign> !
                <union sign> !
                <complement sign> !
                <option begin sign> !
                <option end sign>

<end-of-line sign>::=          #
<metasymbol begin sign > ::= <
<metasymbol end sygn>    ::= >
<union sign>             ::= !
<complement sign>        ::= -
<option begin sign>      ::= [
<option end sign>        ::= ]

<appointed special atom> ::=<character>

<character class definition part>::=
                <character class definition>
                [ <character class definition> ]
                <end-of-line sign>
```

```
<character class definition>::=
                <metasymbol>::=<metasymbol>
                [[<union sign>!<complement sign>]<metasymbol>
                ]<end -of-line sign> !
                <character>[<character>]<end-of-line sign>

<metasymbol> ::=<metasymbol begin sign><identifier>
                  <metasymbol end sign>

<atomclass definition part>::=
                  <atomclass definition level>
                  [<atomclass definition level>]
                  <end-of-line sign>

<atomclass definition level>::=
                  <atomclass definition>
                  [ <atomclass definition> ]
                  <end-of-line sign>

<atomclass definition>      ::=
                  [* ] 1 <metasymbol>
                  [ <option begin sign><integer constant>
                    <option end sign> ]1 ::=
                  <metasymbol sequence>
                  [ <union sign><metasymbol sequence> ]
                  <end-of-line sign>

<metasymbol sequence>       ::=
                  <metasymbol> [<metasymbol>]
                  [<option begin sign><metasymbol>
                   <option end sign>]
```

3.1.2. Semantic

The function of the special atom definition part is to give the programmer an opportunity to change standard special atom-s (that is, the atoms needed to describe the syntax of character and atom classes) to any other character of the character set. The appointed special atoms will inherit the function of the standard ones.

The <character class definition part> is to define sets of characters belonging to character classes. There are two operations defined on character classes: the union operation and the complement operation. The latter may be used only once in a character class definition and it must be before the last metasymbol.

Among the characters in a character class definition the characters <end-of-line sign> and <metasymbol begin> sign may not be present.

There may be any level of <atomclass definition> in an <atom class definition part>.

On the left-hand side of an <atomclass definition> there must be a <metasymbol> (different from any other <metasymbol> on the same level) which may be preceded by an asterisk. The presence of an asterisk signifies that the metasymbol in question is a final atom class constant but may present on the right-hand side in the definitions of the succeeding levels. There is an option to specify the maximum length of the metasymbol if it is different from the standard length.

On the right hand side of an <atom class definition> there may be any number of <metasymbol>-s defined as atomclass names on preceeding levels.

The meaning of option signs is the same as in the syntax descriptions of this manual. Two sequence of <metasymbol>-s must be different before "arriving to" an optional meta-symbol.

For the description of an implementation of the <atom definition part> see M. Martelli, Analizzatore lessicale per linguaggi autoestensibili, Tesi di Laurea, Universita degli Studi di Pisa, 1974.

3.1.3. Example

In the following example we give the definition of <identifier>-s, integer constants (<ic>) and <real constant>-s of FORTRAN.

```
<digit>::=Ø123456789#
<letter>::=abcdefghijklmnoprstuvwxyz#
<ec>::=E#
<signo>::=+-#
<pointo>::=.#
<alfa>::=<letter> ! <digit>##

<e>::=<ec>#
<identifier>[6]::=<letter>[<alfa>]#
 <ic>::=<digit>[<digit>]#
<point>::=<pointo>#
<sign> ::=<signo>##

* <rc1>::=<point><ic>!<ic><point>!<ic><point><ic>#
<rc2>::=<e><sign><ic>!<e><ic>##

<real constant>::=<rc1> ! <rc1><rc2>###
```

We give an other example of <atom definition part> in Chapter 6.

3.2 The definition of macros

3.2.1. Syntax

```
<macro definition>::= $ <macro name>[<parameter part>]
                      == <macro body>

<parameter part>  ::=<left sub-list><right sub-list>
                     [ <left sub-list><right sub-list> ]

<left sub-list>   ::=<left list element>
                     [<left list element>]
<right sub-list>  ::=<right list element>
                     [ <right list element> ]

<left list element>::=
                     [[]  <compulsory parameter list>
                     [[<compulsory parameter list>] [!]

<right list element>::=
                     [][]  <compulsory parameter list>
                     [][<compulsory parameter list>] [!]]1

<compulsory parameter list>::=
                      [ <compulsory parameter> ]

<compulsory parameter>::=
                      " <row of non-" characters> " !
                      < STRING TO <closing word> >
                      [  . <parameter index> ] !
                      <  <identifier> [
                         [ !<condition part> ] >
                      [  . <parameter index> ] !
                      < <structure name> >
                         [ . <parameter index> ]
<condition part>      ::=<boolean value>
```

```
<parameter index>     ::=<integer constant> ! NEXT ! SAME

<closing word>        ::=<atomclass constant> !
                          " <row of non-" characters> "
```

3.2.2. Semantics

For the use of [,] and ! marks the same semantical rules are valid as for the same marks in the metalanguage used in the syntax descriptions of this paper(Chapter 1.5.).

The meaning of the 1st-, 3rd- and 4th- type of <compulsory parameter>-s is very close to the meaning of the metasymbols used in the syntax descriptions. There are two main differences:

1. The compulsory parameters of these types may be indexed and in this way referenced in a <macro body>. The index NEXT provides an automatic index-updating facility: the value of the index used as latest with the same identifier or structure name will be updated by one. The index SAME will be the same as the index value of the closest proceeding <compulsory parameter> supplied by an index. The lack of index in compulsory parameters is equivalent by using 1 as index.

2. A 3rd- type <compulsory parameter> may have a condition part. This option gives an opportunity to describe parts of the "semantic parsing" in the "syntax description", that is, in the parameter part. (In MADE we might write into 3rd- type <compulsory parameter> in the place of <identifier> the more exact <identifier! . type = aclav \/. type = none>. (This example is a part of the semantics of the 3rd--type <compulsory parameters> !). Besides, the <conditional part> option provides a tool for the easy definition of "working-meta-symbols". We call "working-meta-symbols" such meta-symbols of a meta--language which are not "essential", are used only in a limited part of the syntax to express connections between "essential" meta-symbols more clearly or more shortly. For example, <parameter index> and <closing word> may be considered as working-meta-symbols.

In MADE the <structure definition> of such meta-symbols may be spared, we may write in place of <parameter index> (in the syntax description of <compulsory parameter>) <any!. ACLA=icons \/=NEXT \/= = SAME>.

The 2nd-type <compulsory parameter> is to be used in cases when a part of the input stream (text of some language) is not of interest as a sequence of atoms. A string parameter in the definition of a macro will result at execution time in the simple scanning and storing of the mnemonics of the constituent atoms as one and only one text value (it may by referenced in the macrobody as <STRING.parameter ref. index>). The <closing word> will not be a part of this text, it remains the next atom to be scanned. (See Chapter 6.).

3.2.3. Example.

In this example we define an extension of the FORTRAN-IV by a <when statement>. The syntax of this when statement is the following:

<when statement>::= WHEN <logexp> THEN <uncond statement>
ELSE <uncond statement> %

Let us suppose that the <logexp> and <uncond statement> ("uncstat") are the same in both of the languages -- basically we want only to copy them but there is a little manipulation, too. (This is why we use INCLUDE as default procedure.)

The macro may be the following:

```
$ "WHEN" <logexp> "THEN" <uncstat> "ELSE" <uncstat> "%"==
HEAD;
DEFAULT:=INCLUDE;
TAIL;
MARKER mv;
INTEGER othlab;
mv:=1.STATE;
othlab:=newlab+1;
COMMENT integer newlab is global counter. Its value is
the next "free" FORTRAN label;
GENER 1, empty," GO TO " ,SYMB newlab ,"%  " INCLSTACK.1,
        "%     ","GO TO " ,SYMB othlab , "%",SYMB newlab,
        " ",INCLSTACK.2,"%",SYMB othlab," CONTINUE%" ;

DELETE INCLSTACK;
DELETE INCLSTACK;
GENER 1.mv,"      IF ",INCLSTACK.1;
newlab:=newlab+2;
% when finish ;
```

The application of this macro definition will result in the translation of a

```
    WHEN I .EQ. J THEN X=A+2 ELSE X=B+3
```

statement into the

```
    IF I .EQ. J GO TO 10000
    X=B+3
GO TO 10001
10000 X=A+2
10001 CONTINUE
```

FORTRAN program fragment.

4. STATEMENTS

4.1. Syntax

```
<assignment statement>
              ::=<integer variable>:=<integer value>; !
                <boolean variable>:=<boolean value>; !
                <procedure variable>=<procedure value>; !
                <atom variable>:=<atom value>; !
                <atomclass variable>:=<atomclass value>: !
                <marker variable>:=<marker value>; !
                <newtype variable>:=<newtype value>; !
                <text variable>:=<text value>; !
                <atomstack variable>.NEW:=<atom value>; !
                <outstack variable>.NEW:=<text value>; !
                <integer vector variable>·<integer primitiv>
                :=<integer value>; !
                <atom vector variable>.<integer primitiv>
                :=<integer value>; !
                <atom vector variable>·<integer primitiv>
                ::=<atom value>;

<conditional statement>
                ::=IF <boolean value> THEN
                [ BEGIN ]1
                [ <statement> ]
                [ END; ]1

<go to statement>::=<GOTO <label> ;

<label>          ::=<identifier>

<text generation>::=GENER <output field> ,
                 <text value> [ , <text value> ] ;
<output field>    ::=<segment number> ! <text variable> !
                     <segment number>·<marker value>
                     ! <outstack variable>
```

```
<segment number> ::=<integer primitiv>

<stack operation>::=<close text statement> !
                    <delete stacktop statement>

<close text statement>
                    ::=CLOSE <cutput field> ;
<delete stacktop statement>
                    ::=DELETE <stackvariable>[.1]1;
<stack variable>  ::=<atomstackvariable> !
                    <outstack variable>   ! INCLSTACK

<stackindex>      ::= 1 ! 2

<call statement>  ::=CALL < procedure value> ;

<input specification>
                    ::=INPUT <segment number> [ ,
                       <segment number> ] : ! INPUT STANDARD;
<comment statement>
                    ::=COMMENT <comment> ;
<comment>           ::=<row of non-" characters>
```

4.2. Semantics

Only the first and the second element of a stack variable may be referenced directly. A <stack variable> without a <stack index> means <stack variable> .1, that is, the stack top element of the stack variable.

The <delete stack top statement> deletes <stack variable>.1 and updates the stack pointer. After "delete" the "old" <stack variable>.2 becomes <stack variable>.1 and so forth.

A new element may be added to the stack as a new stack top by assigning it to the <stack variable>•NEW. The element must be an <atom value> or <text value>•

There are two modes of text-transfer in MADE: the open-mode and the closed-mode.

Open-mode transfer may be realized only by <text generation>, closed-mode transfer only by <assignment statement>•

The closed-mode transfer includes an automatic close of the text equivalent with the given <close text statement>, every assignment to the same <outstack variable>• NEW will result in a new stack top element, the <delete stack top statement> may be applied at any place among or after the assignments. A <text value> is always a closed-text.

The open-mode transfer do not include automatic close of the text: the <text value>-s newly generated into an <output field> will be unified by the text already existing in the output field (if there exists such a text). Only the use of a <close text statement> will close this string and in case of an <outstack variable> or <text variable> will procedure a <text value>•

In its internal representation, a segment will surely have a pointer to the memory location of the next entering character. This memory location may be saved by a <marker variable>::= <segment number>• STATE; statement and a <text generation> may be accomplished into this part of the segment by using <segment number>•<marker value> as <output field>• (See Chapter 3.2.3.)

The <input specification> statement is to specify a list of segments as the input device or to "switch back" to the standard input device of the implementation. This statement makes possible the generation of several-pass translators.

<Comment>-s may be deleted by the scanner or at supervisor-level. INCLSTACK is a standard atom stack variable. Its description may be found in Chapter 5.2.3.

4.3. Example

The following macro generates nursery rhymes of a certain type on the basis of the pattern of rhymes of this type -- given in the macro definition and in a "condensed rhyme"" -- which contains only the variables" of the pattern.

```
$ "*" [ <ide>•NEXT ] "+" ==
INTEGER i;

TEXT t;
OSTACK os;
i:=1;
t:="Hey do diddledy ho%":
1:GENER os, "The",SYMB<ide.i>,"wants a " ,SYMB<ide.i+1>,"%";
CLOSE  os;
GENER Ø,os , os, t ,os "%" ;
i:=i+1;
IF ide.NO>>i THEN GOTO 1;
DELETE os;
GENER os, "We all pat the",  SYMB <ide.i> ;
CLOSE os;
GENER Ø,os,os,t,os,"%%";
% nursery rhyme type 1;
```

Let us suppose that the condensed rhyme is the following:

```
* farmer wife child dog +
```

The result then will be the nursery rhyme:

```
The farmer wants a wife
The farmer wants a wife
Hey do diddledy ho
The farmer wants a wife
```

The wife wants a child
The wife wants a child
Hey do diddlesy ho
The wife wants a child

The child wants a dog
The cild wants a dog
Hey do diddledy ho
The cild wants a dog

We all pat the dog
We all pat the dog
Hey do diddledy ho
We all pat the dog

5. VARIABLES AND VALUES

5.1. Syntax

5.1.1. Integers

```
<integer variable>::<declared integer variable> !
                    [<atomvalue>]1.<
                    <declared integer attribute>!LENGTH

<integer primitiv>::<integer constant> !
                    <integer variable> !
                    <integer vector variable>.
                    <integer primitiv> !
                    <identifier>.NO !
                    - <integer primitiv> !
                    ( <integer value> )

<integer value>   ::=<integer expression>

<integer expression>
                  ::=<integer primitiv>
                     [ <binary arithmetic operator>
                       <integer primitiv> ]
<binary arithmetic operator>
                  ::= + ! - ! * ! /
```

5.1.2. Booleans

```
<boolean variable>::=<declared boolean variable> !
                     [ <atomvalue> ]1
                     . <declared boolean attribute>

<boolean primitiv>::=NONE ! TRUE ! FALSE !
                     <boolean variable> !
                     -, <boolean variable> !
                     ? <atomclass value> !
                     =<simple atomvalue> !
                     (<boolean value>)

<boolean value>   ::=<boolean expression>

<boolean expression>
                  ::=<boolean primitiv>
                     [ <binary boolean operator>
                       <boolean primitiv> ] !
                     <integer value><relational operator>
                     <integer value> !
                     [<atomvalue>]1 =  <atom value>!
                     <atomclass value> =
                     = <atomclass value> !
                     <newtype value> =   <newtype value>
<binary boolean operator>
                  ::= /\ !  \/
<relational operator>
                  ::= <= ! >> ! = ! /=
```

5.1.3. Procedures

```
<procedure variable> ::=<declared procedure variable> !
                        DEFAULT ! ERROR !
                        [ <atom value> ]1
                        . <declared procedure attribute>

<procedure value>    ::=NONE ! DEFAULT ! ERROR !
                        <procedure variable> !
                        <declared newprocedure> !
                        <basic procedure>

<basic procedure>    ::=SCAN ! LOOK ! SCLOOK !
                        SCBACK ! COPY ! DUMMY ! INCLUDE
<procedure attribute>::=MACA !
                        <declared procedure attribute>
```

5.1.4. Atomclasses

```
<atomclass variable> ::=<declared atomclass variable> !
                        <atom value> . ACLA

<atomclass value>    ::=NONE ! <atomclass variable> !
                        <atomclass constant>
```

5.1.5. Markers

```
<marker variable>    ::=<declared marker variable>

<marker value>       ::= NONE ! <marker variable> !
                        <segment number> . STATE
```

5.1.6. Newtype

<newtype variable> ::=<declared newtype variable> !
[<atom value>]1
. <declared newtype attribute>

<newtype value> ::=NONE ! <newtype variable> !
<declared newtype constant>

5.1.7. Text

<text constant> ::= EMPTY ! " <row of non-"
characters>"
::=<THTX> ! <declared text variable>

<text value> ::=<text constant> ! <text variable>!
SYMB <atom value> ! SYMB
<integer value> !
<outstack variable>
[. <stackindex>]1 !
<STRING [·<parameter ref. index>]1>

<parameter ref. index>
::=<integer constant> !
<simple.int.var.>

5.1.8. Atoms

<atom variable> ::=<declared atom variable> !
[<atomvalue>]1
. <declared atom attribute>

<atom constant> ::=<defined atom>

<atomconstant reference>
::= "<row of non-" characters>"

```
<atom value>      ::=<atom variable> !
                     <atomconstant reference> !
                     <atomstack variable>.<stackindex> !
                     <atom vector variable> •
                     <integer primitiv> !
                     < THIS > ! < LAST > !
                     <<parameter reference>>
<parameter reference>
                  ::=<parameter name> [ •<parameter index>]1
```

5.2. Semantics

The lack of an - - in the syntax description as optional defined - - <atom value> is understood as if the atom <this> were there.

5.2.1. Integers

The standard integer variable LENGTH contains the length of the standard text variable <THTX> in characters. It is updated by the standard procedure SCAN.

The use of <identifier>•NO has semantical meaning only if the same identifier had taken part in a <macro definition> or in a <structure definition> as a 3rd-type <compulsory parameter>-- in this cases it means the value of the index used as latest with the same identifier. (If we use the automatic index-updating facility it will be the number of the actual occurances of constructs which correspond the conditions in the compulsory parameter denoted by the identifier in question).

5.2.2. Booleans

Among the <boolean primitiv>-s the ? <atom class value> is the abbreviation of <THIS>•ACLA = <atom class value>• The=<simple atom value> is a shorter form of <THIS>= =<simple atom value>• We use these forms and the - a bit clumsy - <=(less or equal),>>(greater),/= (non equal) symbols to facilitate the implementation of the BASIC-MADE subset.

Simple variables whose value-set is "limited" (boolean, atom, newtype, atom class, procedure, marker types) are initiated by the NONE value, hereby the user is supplied with a mean to check whether a variable has ever been used in an assignment on the left-hand side before.

5.2.3. Procedures

The MADE system supplies the user with a number of standard procedures to help him in "writing" of his translator:

SCAN is a lexical analyzer working on the basis of the informations given in the atom definition part. The call of SCAN will result in the input of an <atom constant>. (The actual input device is specified by the latest input specification. If no <input specification> is given, the input device is the standard input device of the implementation). The mnemonic of the atom constant will be stored in the global <THTX> text variable (THIS TEXT). The atom class value of the atom constant will be saved, too.

LOOK is the standard atom table handling procedure. Its task is to search for the atom with mnemonic equal with the content of <THTX> in the atom table. If it does not find such an atom, it will add to the atom table a new entry with the mnemonic in <THTX> filling out the attribute ACLA with the actual atom class value. The actual atom value (in the internal representation it is the (relativ) address of the entry) will be stored in the global atom variable <THIS>. The "old" content of <THIS> will be saved in the global atom variable <LAST>.

The standard procedure SCLOOK contains a call of SCAN followed by a call of LOOK.

The function of the standard procedure SCBACK is to make possible the repeated scanning of the same atom. It will be automatically called at the end of the input of a source "program" part corresponding to a 2nd-type <compulsory parameter> (that is, to a "string parameter").

See Chapter 6 for further applications of SCBACK.

In the course of the parsing process, the recognition of an element which corresponds to the "actual" compulsory parameter in the macro or structure definition will cause a call of the procedure the standard variable DEFAULT is pointing to. There are three standard procedures (DUMMY, COPY,INCLUDE) which may be assigned to DEFAULT, but the MADE programmer may write his own default procedure, too.

The standard procedure DUMMY is a no-operation.
The standard procedure COPY will copy the mnemonic of the "actual parameter" onto segment 1.

The standard procedure INCLUDE will include the actual atom into the standard atom stack named INCLSTACK as if INCLSTACK . NEW:= <THIS> ; were written.

If the generated translator fails to recognize a part of the source "program" or this part does not match the description given in the parameters part of the macro or structure definition, the general procedure of this definition will call an error procedure: the one the standard procedure variable ERROR is pointing to.

The programmer may write several new procedures for "printing" error messages (onto segment 0, for example) and for the definition what to do when a particular error-situation occurs. (See Chapters 1.3 and 5 for example).

5.2.4. Atom class

See Chapter 2.2.

5.2.5. Markers

See Chapter 2.2.

5.2.6. New types

No comment.

5.2.7. Text

The text constant EMPTY is a row of "emty" characters occupying standard text-length number of memory locations. In the internal representation an empty character may be a number which does not correspond to any character on the input devices. At the final printing or punching out of the segments these characters will be totally ignored. A <text variable> is initialized by EMTPY.

The function of the SYMB "operator" is to transform from an <atom value> or <integer value> into text-form. The text-form of an <atom value> is its mnemonic.

5.2.8. Atoms

An <identifier> having taken part as the first <identifier> in a 3rd-type <compulsory parameter> becomes a <parameter name>. In the same time, if it was an <atom class constant>, it remains <atom class constant>, too.

5.3. Example

In this example we continue the description of the translator of a MADE-like language by adding the definition of integer items and general arithmetic expressions. (The first is a restricted, the second is a generalized version of the corresponding MADE syntax element).

In the example the task of semantic processing is assigned to semantic procedures: at the beginning of the evalution of a structure one of these procedures will be choosed (the choise is environment-directed!) and will be specified as "default-procedure". This default-procedure will then do the semantic processing working parallel with the (syntactic) parsing.

This method is very close to the technique we used in a earlier version of MADE. In that version every atom had a procedure (DUMMY or a newprocedure) and the succesfull parsing of an atom implied the call of this procedure. That method made possible the definition of more efficient translators by eliminating a large part of "case statements" one writes programming in the presented version of MADE, but the elements belonging to the evaluation of a macro or structure were cut into small pieces and were scattered in the MADE program.

We remark that in the example particular care has been taken to differenciate between unary +,-and binary+,--signs. This is responsible for some of the complications.

```
INTEGER pno;
TEXT stl (3);
BOOLEAN index;
OSTACK opnd (40);
DEFAULT := INCLUDE;

";".pri:=0; ":=".pri:=0;
"(".pri:=0; ")".pri:=0;
"+".pri:=1; "-".pri:=1;
"*".pri:=2; "↑".pri:=3;
"abs".pri:=4;
"+".uary:TRUE; "-".uary:=TRUE; "abs".uary:=TRUE;
"+".bary:=TRUE; "-".bary:=TRUE; "*".bary:=TRUE;
"/".bary:=TRUE; "↑".bary:TRUE;

NEWPROCEDURE inclinit;
COMMENT to include an atom when in an integer item;
ATOM a;
a:=SYMB <THIS>;
IF ? iden THEN GENER opnd , a;

IF = "." THEN BEGIN IF -, <LAST>.dim THEN CALL ERROR;
                    index:=TRUE; END;

IF ? intconst /\  index THEN BEGIN GENER opnd ,"+",a ;
                                   GOTO ret; END ;

IF ? intconst THEN GENER opnd, a, "=D" ;
ret: RETURN ;

<init>::=<intconst>!<iden!.type=intv>
         [ "." [<intconst>!<iden!.type=intv>]1]
      ==head;
        PROCEDURE saved;
        saved:=DEFAULT;
        DEFAULT:=inclirit;
        index:=FALSE ;
```

```
          TAIL ;
          CLOSE opnd;
          DEFAULT:=saved;
% end of structure "integer item" ;

NEWPROCEDURE inclaex ;
COMMENT to include an atom when in arithmetic expression;
IF .uary /\  =aop.1 THENGOTO uapm ;
IF ="(" THEN pno :=pno+1; IF =")" THEN pno :=pno-1;
beg:
IF .bary  \/ =")" \/ = ";" THEN GOTO yes1;

CALL INCLUDE;
GOTO ret;
jes1:
st1:=INCLSTACK. 1;
IF .pri >> st1.pri THEN GOTO no1;
IF st1="(" THEN BEGIN DELETE INCLSTACK; GOTO ret; END;
IF st1=":=" THEN GOTO lets;
IF st1 .bary THEN GOTO baop;
IF st1.uary THEN GOTO uaop;
IF st1="$"/\ pno /= ∅ THEN CALL ERROR;
GOTO ret;
uaop:
IF st1 /= "abs" THEN CALL ERROR;
GENER 1," LDA " , opnd.1,"% SSA% CMA,INA%";
GOTO common;
lets:
GENER 1," LDA ",, opnd.1,"% STA ", opnd.2,"%";
st1:=opnd.1;DELETE opnd;DELETE opnd;opnd.NEW:=st1;CLOSE opnd;
GOTO com2;
BAOP:
GENER 1," LDA ",opnd.1,"%";
IF st1= "-" THEN GENER 1, " CMA,INA"%";
IF st1="+" THEN GENER 1, " ADA ",opnd.2, "%";
IF st1="*" THEN GENER 1, " CLB% MPY "<opnd.2,"%";
```

```
DELETE opnd;
COMMENT we leave code optimization and other operators for
the reader;
common:
CALL genworkvar;
GENER 1," STA WR" , SYMB wri;
DELETE opnd;
GENER opnd,"WR",SYMB wri;
CLOSE opnd;
com2:
DELETE INCLSTACK;
GOTO yes1;
uapm:
COMMENT to eliminate unary + and to transformate unary -
into binary - ;
IF ="+" THEN BEGIN DELETE INCLSTACK;GOTO ret;END;
IF ="-" THE BEGIN ; GENER opnd , SYMB ∅ ; CLOSE opnd;
    GOTO beg;END;
ret:
RETURN end of inclaex;

$ <aexandassign>::=[<init!index \/.acla/=intconst> ":="]
[ "(" ] [<aop!.uary>]1 ["("] <init> [")"]
[ <aop!.bary>•2 ["("[<aop!.uary>]1  ] <init> [")"] ]
[ ";" ]    ==
HEAD;
PROCEDURE save ;
pno:=∅ ;
save :=DEFAULT;
DEFAULT:=inclaex;
INCLSTACK.NEW:="$";
wri:=∅;
TAIL;
DEFAULT:=save ;
% end of arithmetic expression and/or assignment;
```

6. A SPECIAL PURPOSE PROBLEM ORIENTED LANGUAGE

In this chapter we give a complete MADE program for the definition of a new simulation language and for the generation of its translator. The new simulation language is a version of the "computational flow-chart" of chemical plant. It will be transformed by the generated translator into a SIMULA 67 program. This program is prefixed by the "chemical plant" SIMULA class which consists of classes and procedures for the dinamic simulation of devices, connections, etc. of a chemical plant.

Though the application of this SIMULA - language simulation system (J.H. Kardasz, G. Molnar: A Simula-based structure oriented language for the dynamic simulation of chemical plants, the Computer Journal, Vol. 17, No.1, 1974) is easy enough, it requires a by-hand-transformation of the structure of the particular plant, its devices, its streams and the actual data into the form of a valid SIMULA program: this activity involves the "generation" of so-called "connections" to describe the situation of the singular devices with respect to their "neighbours" (this "connections" are lacking on the flow-chart of the plant); the informations concerning a particular device have to be written to about 5 different places in the programs, etc.

Though these are basically automatic activities, they require some knowledge of the language SIMULA and the produced program has to be tested on a big computer.

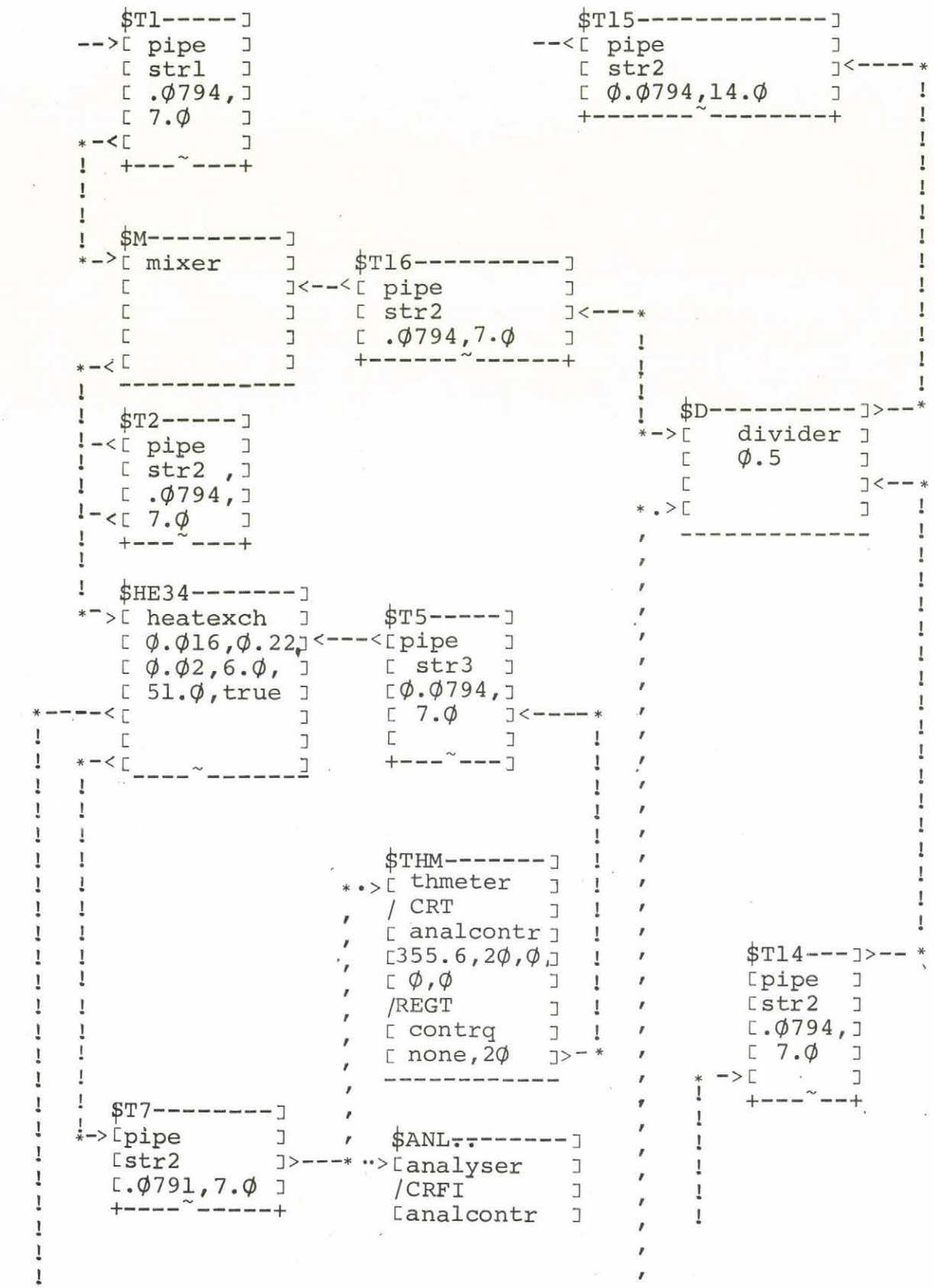
In the followings we give one version of "computational flow--chart" for a chemical plant on the base of the figure 1 and figure 2 of the article mentioned.

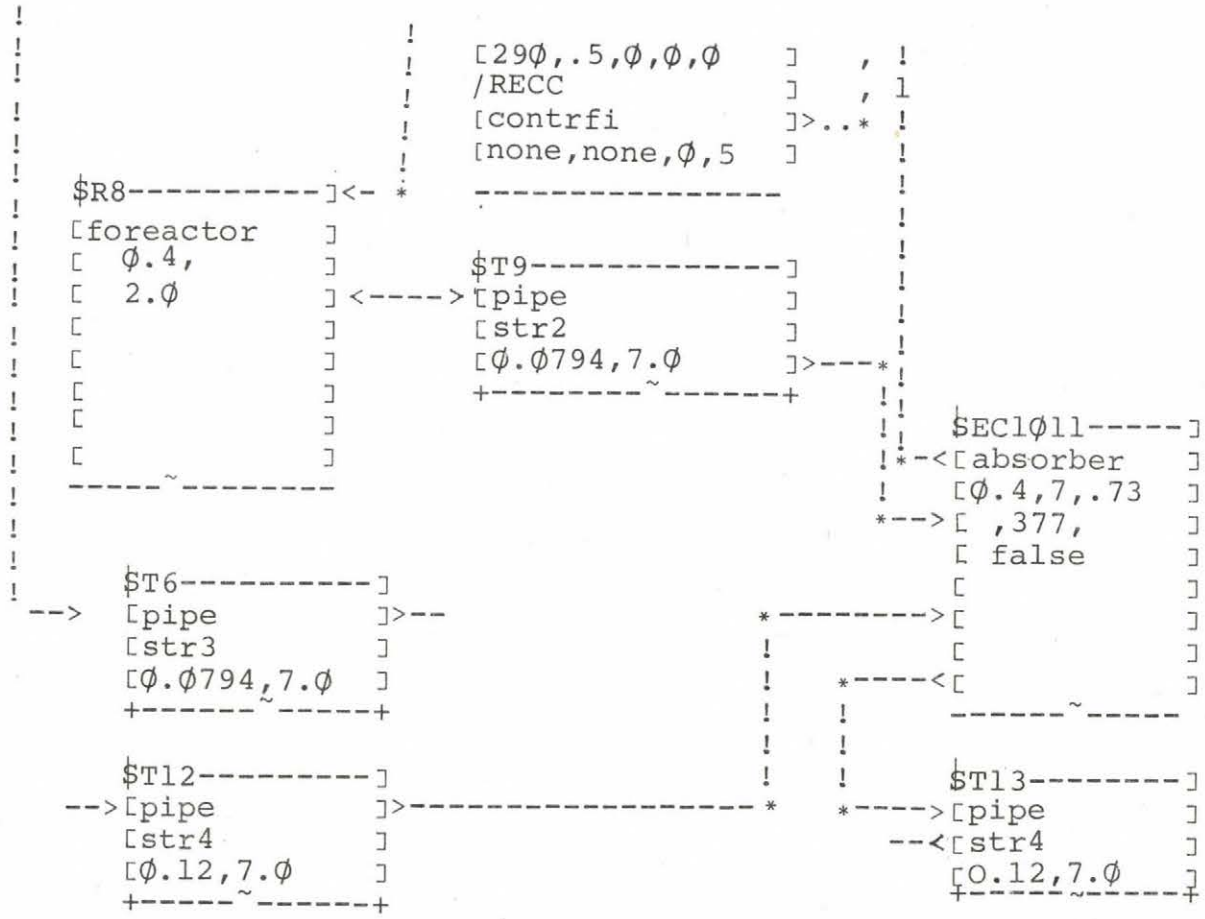
Each "box" is beginning with its identifier then comes the type of the device. In pipes, the types of the stream is the next parameter. As to the meaning of the other parameters we

refer to the original article.

The plant has a recycle and is controlled by two controller for the temperature and composition before the reactor.

```
simtime:=Ø.Ø5; dt := Ø.ØØØ5;
```





=

We copy from the referenced article the part of the SIMULA program which corresponds to this part of the plant. We made some minor changes on the names.

```
chemical plant 'begin' 'comment' definition of system values;
simtime:=Ø.ØØ5; dt:=Ø.ØØØ5;
'comment' structural elements declarations;
'ref' (connect) E1,E2,E3,C1,C2,C3,C4,C5,C6,C7,C8,C9,C1Ø,C11,
C12,C13,C14,C15,C16,C17,C18,C19,C21,C22;
'ref' (mixer) M;
'ref' (divider) D;
'ref' (pipe) T1,T2,T5,T6,T7,T9,T12,T13,T14,T15,T16;
'ref' (reactor) R8;
'ref' (heatexch) HE34;
'ref' (absorber) EC1Ø11;
```

```
'ref' (analyser) ANL;
'ref' (thmeter) THM;
'ref' (contrq) REGT;
'ref' (contrfi) REGC;
'ref' (analcontr) CRFI,CRI;
'comment' declaration of pseudodvices;
'ref' (print) stel,ste2,ste3,stcl,stc2,stc3,stc4,stc5,stc6,
stc7,stc8,stc9,stclØ,stcl2,stcl3,stcl4,stcl5,stcl7,stcl8,stcl9,
stc2l,stc22;
'comment' generation of the system;
'comment' generation of connections;
El:-'new' connect (4,strl);
E2:- 'new' connect (4,str3);;
E3:- 'new' connect (4,str4);
Cl:- 'new' connect (4,str2);
C2:-'new' connect (4,str2);
C3:-'new' connect (4,str2);
C4:- 'new' connect (4,str2);
C5:-'new' connect (4,str2);
C6:- 'new' connect (4,str2);
C7:- 'new' connect (4,str2);
C8:- 'new' connect (4,str2);
C9:- 'new' connect (4,str2);
ClØ:- 'new' connect (4,str2);;
Cl2:- 'new' connect (4,strl);
Cl3:- 'new' connect (4,str3);
Cl4:- 'new' connect (4,str3);
Cl5:- 'new' connect (4,str3);
Cl7:- 'new' connect (4,str4);
Cl8:- 'new' connect (4,str4);
Cl9:- 'new' connect (4,str4);
C2l:- 'new' connect (4,str2);
C22:- 'new' connect (4,str2);
'comment' generation of devices other than controllers and
input pseudodevices;
M:- 'new' mixer (Cl2,ClØ,Cl);
```

```
T1:- 'new' pipe (Ø.Ø794,7.Ø,E1,C12 );
T2:- 'new' pipe (Ø.Ø794,7.Ø,C1,C2 );
T5:- 'new' pipe (.Ø794,7.Ø,E1,C15 );
T6:- 'new' pipe ( Ø.Ø794,7.Ø,C13,C14);
T7:= 'new' pipe (Ø.Ø794,7.Ø, C3,C4 );
T9:- 'new' pipe (Ø.Ø794,7.Ø, C5,C6 );
T12:- 'new' pipe (Ø.12,7.Ø,E3,C19 );
T13:- 'new' pipe (Ø.12,7.Ø,C17,C18 );
T14:- 'new' pipe (Ø.Ø794,14.Ø,C7,C8 );
T15:-'new' pipe (Ø.Ø794,14.Ø,C21,C22 );
T16:- 'new' pipe (Ø.Ø794,7.Ø,C21,C22 );
R8:- 'new' foreactor (Ø.4,2.Ø,C4,C5 );
HE34:- 'new' heatexch (Ø.Ø16,Ø.22,Ø.Ø2,6.Ø,51.Ø,'ture',C2,C15,
                       C3,C13 );
EC 1Ø11:- 'new' absorber (Ø.4,7.Ø.Ø.73,377.Ø,'false',C6,C19,C7,
                          C17 );
D:- 'new' divider ( Ø.5,C8,C9,C21 );
'comment generation of effector's and sensors;
ANL:- 'new' analyser (C4);
REGC:- 'new' contrfi ('none','none',Ø.5,D);
THM:= 'new' thmeter ( C4 );
REGT :- 'new' contra ( 'none' , 2Ø.Ø,E2 );
'comment' generation of controllers;
CRFI:-'new' analcontr (355,6,2Ø.Ø,Ø.Ø,Ø.Ø,Ø.Ø,THM,REGT );
CRFI:- 'new' analcontr ( 29Ø,Ø.5,Ø.Ø,Ø.Ø,Ø.Ø,ANL,REGC );
'comment' definition of output requirements;
stel:-'new' print ( E1 );
ste2:-'new' print (E2);
        ......one print for every connection.....
T1.profile ('pipe1',1 ); T2.profile ("pipe2",1 );
        ......and so and for every pipe ..............
R8.profile ("reactor8",1 );
HE34.profile ("heat exchanger34",1 );
EC1Ø11.profile ("extraction column 1Ø11",1 );
'end';
```

The made program written for the definition of our flowchart language is as follows:

```
madbegin plant flowdiagram=>simula program;
c ?,c<(,c>9,
comment character classes;
(digit)::=123456789Ø?
(le)::=-.? (arrow)::=<>? (cross)::=+? (eq)::==?
(finis)::=*!S[/ %?
(rbra)::=]? (else)::= %,;: ?
(alfa)::=(letter) ! (digit) ??
comment atom classes;
(le)::-(le)? (cross)::=(cross)? (arrow)::=(arrow)?
(fins)::=(fins)? (iden)::=(letter) [ (alfa) ]?
*(rbra)::=(rbra)? (else)::=(else)? (eq)::=(eq)?
 (strf)::=(rbra)(arrow)??
(rwall)::= (rbra) ! (strf) ???

newtype (tt: boxt,cont);
attributes (tt type, atom tofb, atom tofs, integer xØ,
        integer xl, integer yØ, integer yl, atom insl, atom
        ins2, atom outl, atom out2, integer segi);

integer x,y,xor,xsav,bno,osno,cr,wv,pti,str,bi,bsli,
        pri ;
boolean bwv,bw;
atom box,stream, bseg,awv;
text twv,tw;
avector bsl(6Ø) ;
ostack os (7Ø) ,wos  (4Ø) ;
procedure bqr,p;
y:=Ø;
bno:=Ø;
osno:=Ø;
bi:=Ø;
stri:=Ø;
```

```
Pro:=Ø;
bsli:=Ø;
newprocedure coord;
x:=x+length;
return updates an x coordinate;

default:=coord;
newprocedure bbox;
bw:=(awv.type=boxt) /\ (awv.yl=Ø) /\ ((awv.xØ=x)    /\
(awv.xl=x));
return Is actual point on the left- or right wall of box awv?;

newprocedure bstr;
bw:=(awv.type=cont) /\ (x=awv.xØ) /\ (y=awv.yØ+1);
return Is the actual point the continuation of stream awv?;

newprocedure idbos;
cr:=1;
bli:
awv:=bsl.l;
call bqr;
if bw then goto fini;
if cr=bsli then begin awv:=none; goto fini; end;
cr:=cr+1;
goto bli;
fini:
return Search in bsl for actual box or stream;
newprocedure idbox;
bqr:=bbox;
call idbos;
box:=awv;
return to identify the box of the actual point;

newprocedure idstr;
bqr:=bstr; call idbos; stream:=awv;
return to identify the stream of the actual point;
```

```
newprocedure skipsp;
bacs:
call scan;
if <this>=" " then goto sps;
goto fins;
sps:
x:=x+1;
goto bacs;
fins:
x:= x-length;
call scback;
return procedure skipspace;

newprocedure errlp;
gener ∅, "error line  ",symb y," atom ",symb <this>,"% box ",
          symb box, "%";
back:
call scan;
if <this>="%" then goto foly;
goto back;
foly:return To write error message to segment ∅ ;

error:=errlp;

newprocedure newbox;
bsl.bsli:=box; bi:=bi+1; bsli:=bsli+1;
box.type:=boxt;
box.x∅:=xor; box.x1:x; box.y∅:=y;
box.ins1:=none;box.ins2:=none; box.out1:=none; box.out2:=none;
bno:=bno+1;
if bno=1 then bseg:=box;
if bno =2 then osno :=1;
return new box accounting and initialzation;
```

```
newprocedure cseg;
boolean b1,b2,b3;
box.segi:=3;
if awv="analcontr" then box.segi:=5;
b1:=awv="analyser"; b2:=awv="thmeter";
b3:=awv="contrfi"; bw:=awv:="contrq";
if (b1 \/ b2) \/  (b3 \/ bw) then box.segi:=4;
return computers the segment number of the actual box;

newprocedure bcon; bw:=awv.type=cont;return;
newprocedure bb; bw:=awv.type=boxt; return;

newprocedure fine;
gener 1,"pr1;%'ref' (connect)";
15:
if str1=ø then goto 16;
bpr:=bcon; call idbos; gener 1, symb awv,","; stri:=stri-1;
goto 15;
16:
gener 1,"c1;%";
11ø:
if bi=ø then goto 17;
bqr:=bb; call idbos;
gener 1," 'ref' (",symb awv.tofb,") ",symb awv,";%" ;
goto 11ø;
17:
return To generate all connection and box declarations;

newprocedure c1pr;

integer prin,prfn;
xsav:=x; y:=y-1; x:=xor; call idbox; x:=xsav; y:=y+1;
box.y1:=y; bno:=bno-1;
if ="+" then prin:=prin+1;
prfn:=ø;
if <last>="~"  then prfn:=prfn+1;
```

```
l2:
if prin=Ø then goto tovl;
gener wos,"pr",symb pri,"," ; close wos; gener l, wos;
gener 6, wos," :- 'new' print (",symb box.insl," );%" ;
prin:=prin-l; delete wos; goto l2;
tovl:
if prfn=Ø then tov2;
gener 6,symb box,".profile(",symb box.tofb," ",symb box,",l);%";
tov2:
tw:=symb box.insl;
gener 2,tw,":- 'new' connect (4," , symb box.tofs," );" ;
gener wos , "," , tw ;
awv:=box.ins2; wv:=box.segi;
if awv = none then goto tov3;
gener 2,symb awv,":- 'new' connect(4,",symb box.tofs," );" ;
gener wos , "," ,symb awv;
tov3:
gener wos,",", symb box.outsl;
awv:=box.outs2;
if awv=none then goto tov4;
gener wos,",",symb awv;
tov4:
gener wos ," );%" ; close wos;
if box = bseg then goto sel;
gener os,wos; close os; osno:=osno+l; goto finl;
sel:
gener wv,wos;
bll:
if osno=Ø then goto finl;
gener wv,os; delete os; osno:=osno-l; goto bll;
finl:
return to finish the generations belonging to a box when
closing the box;
```

```
$ "%"[ " " ] == x:=Ø; y:=y+1;xor:=x; % % is a "crlf% atom;
$ "$" <iden> <string to "]" > == box:=<iden>; call newbox ;
%open a new box;

$ "/" <iden> <string to "]" > ==
  box:=<iden>; call newbox; y:=y-2;
x:=xor; awv:=box; call idbox; box . outl:=awv;
call clpr;
box:=awv; y:=y+2; x:=xsav;
% open a new box in a complex box;
$ "["[ " "] [<iden>] <string to rwall> ==
xsav :=x; x:=xor; call idbox; x:=xsav; wv:=y-box.yØ;
if wv=1 then goto 11;
if wv=2 then goto 14,
1Ø:
wos.new:=<string>;
goto 13;
11,
awv:=<iden>; box.tofb:=awv; call cseg;
gener wos,"%",symb box,":- 'new' ",symb awv,"(" ;close wos;
goto 13;
13:
wv:=box.segi;
if box=bsegm then goto sel;
gener os,wos; goto finl;
sel:
gener wv,wos; goto finl;
14:
if box.tofb="pipe" then goto 15; goto 1Ø;
15:
box.tofs:=<iden> ;
finl:
% inside of a box ;

$ "-" [ "-" ] [ "~" ] 1 [ "-" ] [ "+" ]1 --
 prin:=Ø; call clpr;
 %  last line of a box;
```

```
$ "+" [ "-" ] [ "~" ] 1 [ "-" ] [ "+" ]1 ==
prin :=1; call clpr;
%  last line of a box;
$ "]" <string to fins> == xor:=x; % space after box;
$ "!" [ " " ] == xsav:=x; x:=xor+1; call idstream;
stream.yØ:=y; x:xsav;
% element of a stream - line;

$ "," [ " " ] == xsav:=x; x:=xor+1; call idstream;
stream.yØ:=y;x:=xsav;
% element of a stream line;

newprocedure newstr;
stri:=stri+1;
gener twv, "c" ,symb stri;
<thtx>:=twv; call look;
<this>.type:=cont; bsl.bsli:=<this>; bsli+1;
return New stream accounting;

newprocedure iostream;
if = "<" then goto 17;
if box.ins1=none then goto 11Ø;
box.ins2:=stream; goto 112;
11Ø:
box.ins1:=stream; goto 112;
17:
if box.out1=none then goto 111;
box.out2:=stream; goto 112;
111:
box.out1:=stream;
112:
return mark input or output stream in box;

newprocedure star;
call idstream;
if stream = none then call newstream;
```

```
stream.xØ:=x; stream.yØ:=y;
return mark turning point the stream;

$ "*" <string to arrow> <arrow> ==
xsav:=x; x:=xor; call star; x:=xsav + 1; call idbox;
x:=x-1; call idstream;
% to follow a stream from a turning point;

$ <strf> [  <le> ] <any> ==
call idstream;
if = "*" then goto strr ;
if ?arrow then goto finl;
strr:
call star; call scan; if ?le then goto foly;call skipsp;
goto end;
foly:
<thtx>:="*" ; call scback; goto end;
finl:
call idstream;
end:
% to follow a stream from the source;
$ "=" == goto ufi; % flowchart finish;

gener 1,"%
          chemical plant 'begin'% " ;
ll:
call scan; if /=":" then begin gener 1, symb <this>;goto ll;
end;
gener 1,"% 'ref'(print) ";
lm:
call scan; call look;
p:<this>•maca;
call p;
goto lm;
ufi:
gener 6, "%" 'end' ; " ;
madend;
```

The execution of this MADE program would result in a translator. The translator should be able to read in flowcharts (like the one presented) and translate them into SIMULA programs. We do not give here a formal description of the "flowchart language" nor its limitations -- they are described in the MADE program.

In the followings we give the translated flowchart---a different displacement of the boxes should give a different SIMULA program but with the same meaning. It may be easily seen that the generated program is equivalent with the program copied from the original article.

```
chemical plant 'begin'
simtime:=Ø.ØØ5; dt:=Ø.ØØØ5;
'ref' (print) prØ,pr1,pr2,pr3,pr4,pr5,pr6,pr7,pr8,pr9,pr1Ø,
pr11,pr12,pr13,pr14,pr15,pr16,pr17,pr18,pr19,pr2Ø,pr21,pr1;
'ref' (connect) cØ,c1,c2,c3,c4,c5,c6,c7,c8,c9,c1Ø,c11,c12,c13,
c14,c15,c16,c17,c18,c19,c2Ø,c21,c1;
'ref' (pipe) T1; 'ref' pipe T15; 'ref' (mixer) M;
'ref' (pipe) T16; 'ref' (divider) D; 'ref' (pipe) T2;
'ref' (pipe) T5; 'ref' (heatexch) HE34; 'ref' (thmeter) THM;
'ref' (analcontr) CRI; 'ref' (contrq) REGT; 'ref' (pipe) T14;
'ref' (pipe) T7; 'ref' (analyser) ANL; 'ref' (analcontr) CRFI;
'ref' (contrfi) RECC; 'ref' (foreactor) R8; 'ref' (pipe) T9;
'ref' (absorber) EC1Ø11; 'ref' (pipe) T6; 'ref' (T12);
'ref' (pipe) T13;

cØ:-'new' connect (4,str1);
c3:-'new' connect (4,str1);
      remark: for the input and output stream of every box
T1:-'new' pipe (,Ø794.7,cØ,c3);
T15:-'new' pipe (Ø.Ø794,14Ø,c2,c1);
M:- 'new' mixer (c3,c4,c6);
T16:- 'new' pipe (.Ø794,7.Ø,c5,c4);
D:= 'new' dividier( Ø.5,c5,c7,c2);
T2:-'new' pipe (Ø.Ø794,7.Ø,c6,c8);
```

```
HE34:- 'new' heatexch (Ø.Ø16,Ø.22,Ø.Ø2,6.Ø,51.Ø,true,c8,c9,
                       c11,c12);
T5 :- 'new' pipe (Ø.Ø794,7.Ø,c1Ø,c9);
T14:- 'new' pipe (.Ø794,7.ØØc13,c7);
T7:- 'new' pipe  (.Ø794,7.Ø,c12,c14);
R8:- 'new' foreactor (Ø.4,2.Ø,c14,c15);
T9:- 'new' pipe (Ø.Ø794,7.Ø,c15,c16);

EC1Ø11:- 'new' absorber (Ø.4,7,.73,377,false,c16,c18,c13,c19);
T6:- 'new' pipe(Ø,Ø794,7.Ø,c11,c17);
T12:- 'new' pipe (Ø.12,7.Ø,c2Ø,c18);
T13:- 'new' pipe (Ø.12,7.Ø ,c19.c21):

THM :- 'new' thmeter (c14);
REGT:- 'new' contrq (none,2Ø,c1Ø);
ANL:- 'new' analyser  (c14);
REGC :- 'new' contrfi (none,none,Ø.5,D );

CRT:- 'new' analcontr ( 355.6,2Ø,Ø,Ø,Ø,THM,REGT);
CRFI:- 'new' analcontr (29Ø,.5,Ø,Ø,Ø,ANL,REGC);

prØ:- 'new' print (cØ);
pr1:- 'new' print (c3);
T1.profile ("pipe T1",19;
pr2:-'new' print (c2);
pr3:-'new'print (c1);
T15.profile('pipe T15',1);
       and so and for all of the boxes marked by + or ~

pr2Ø:- 'new' print (c19);
pr21:- 'new' print (c21);
T13.profile('pipe T13',1);

'end' ;
```

7. THE IMPLEMENTATION OF HP-BASIC MADE

The present chapter is an abstract of the implementation of the language MADE on the Hewlett Packard 2100 computer under the operating system TSB 2000/F/C, at the installation at I.E.I. (C.N.R.), Pisa, Italy. We give only some typical details about this implementation here.

The work of the HP-MADE system may be divided into two phases.

At compilation-time a MADE program is translated to intermediate codes. The intermediate codes give the variable part of the translator of the "new language" described in the MADE program.

Then at interpretation time the interpreter part at HP-MADE will execute the generated translator which will be able to input any program written in the "new language". By the end of the interpretation this input "program" will be translated to an other language text, the way it is described in the original BASIC-MADE program.

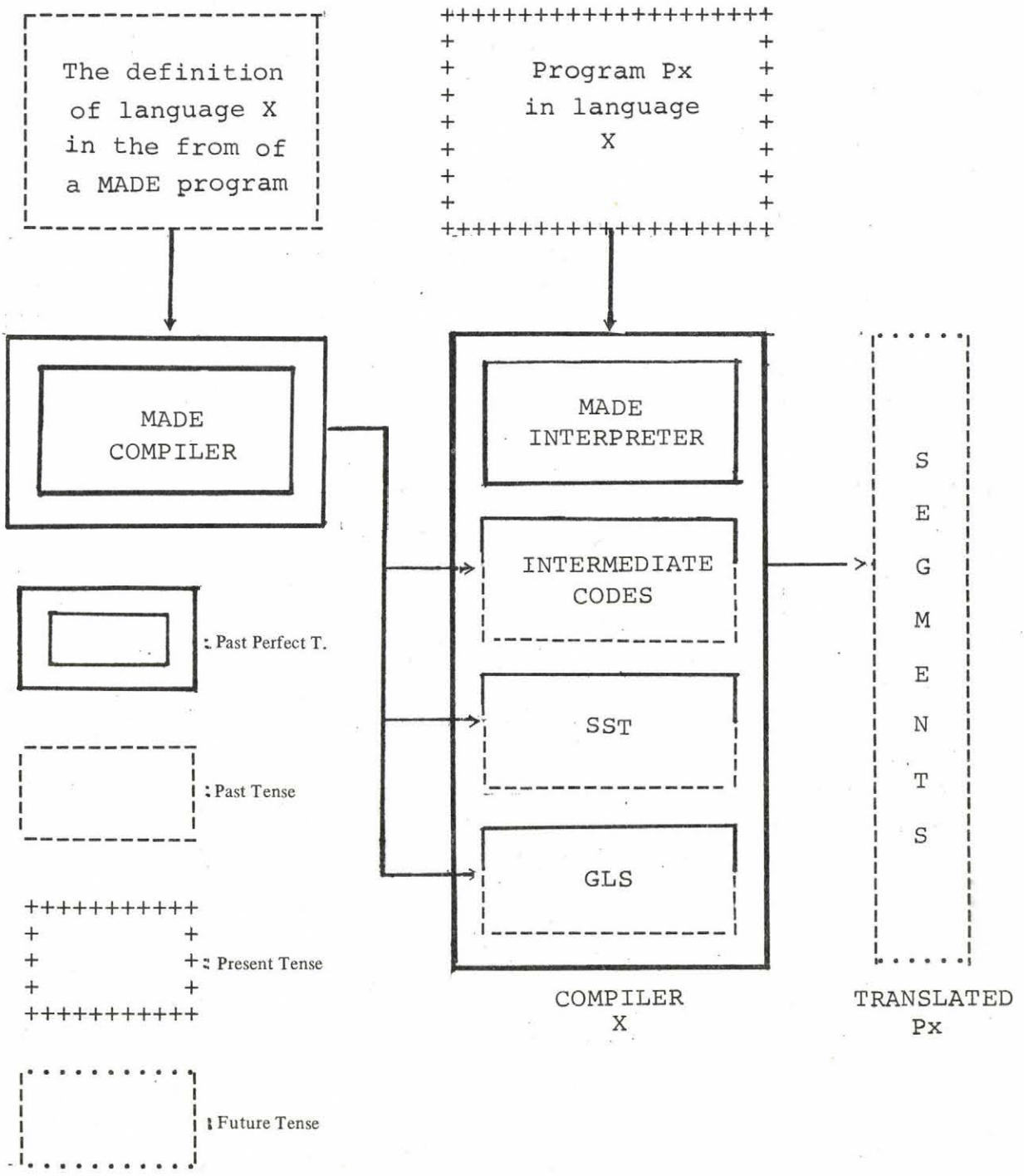


FIGURE 1.
THE HP-BASIC-MADE SYSTEM

7.1. General introduction

The primary aim of the HP-MADE implementation was to test in pratice certain ideas and concepts described in the preceeding chapters. However, under an other operating system which gives more memory to the user, it may present a system building tool for system engineers. The present implementation (using 4K memory) may be used for demonstrative purposes or for experience-gathering. Its conversional mode facilitates this kind of application.

The implementation consists of three parts which work "independently" from each other using the full 4K memory.

The first part is a general scanner definition system. It accepts an atomclass definition part (Ch. 3.1.) and builds up the transition matrix on the basis of the given grammar of the atoms. Its general scanner will accept any string as input and will transform it to a serie of atoms accompanied by their atomclasses. These atomclasses will be representated by their internal representations (which are two-digit octal numbers). A list of the atomclass names (their first four letter) together with their internal representations will be produced, too. This will serve for the HP-MADE compiler to extend its symbol table by the atomclass constants.

The second part of the implementation is the HP-BASIC-MADE compiler. It accepts a BASIC-MADE program instruction by instruction on the typewriter-terminal. An erroneous instruction will cause an error-message on the typwriter and will annulate the instruction. After the error-message the typing of the program may be continued. "Fatal-errors" are exceptions.

The main parts of the HP-MADE compiler are the followings:

1. asimple scanner for the lexical analysis of BASIC-MADE programs. It inputs one character and starts comparing it with the words of SL (See Table 1), but if it finds an entry in SL with a negative class value, gives the control to the procedure of this entry. Either one of these procedures or one of the comparisons must result in recognizing the symbol.

2. a linear symbol table (ST) having the reserved words of the BASIC-MADE as constant entries (see Table 3, Figure 2), and procedures to include a new symbol or to look for a symbol in the table. An ST entry is four words long.

3. a "supervisor" which has a very simple botton-up parsing algorithm programmed in an intuitive manner. In BASIC-MADE an instruction is beginning with a label (which can be easily handled), a reserved word (which implies a call of the respective procedure), or it is an assignment statement.

4. a chained-hash type atom table (SST) for the defined macro names and for text constants (Figure 3). Procedure LOOK is to include atoms into SST and to search for them. The text constants with be placed into SST, too. They can be distinquished by their "mnemonic" which is the negative value of their length in characters. The text itself will be stored in SST. The second word of its ST entry will point to its SST entry (Figure 4.). Procedure GETEX and PUTEX handle such entries.

5. procedures to analyse BASIC-MADE instructions and to translate them into intermediate codes.

6. procedures to analyse BASIC-MADE variables and values, and to generate the necessary intermediate codes for their reductions to GLS, IC or SST adresses. Because of the possibly of failure in recognizing the group of symbols under examination, they will be stored in a "save-buffer" (SBUF) by procedure SAVUN and the place of operator-symbols will be marked in the OPLIS list. Procedure SCANS will replace the scanner after that.

7. buffers. Besides the already mentioned SBUF and OPLIP, HP-MADE has a buffer TBUF for storing "text symbols". The whole input of the HP-MADE compiler is bufferized.

8. a "global-local" stack (GLS) with appropriate stack building facilities.

The third part of the HP-MADE implementation is the BASIC-MADE interpreter. We discuss it in Ch. 6.4.

SYMBOL	CLASS OF SYMBOL	procedure call
ATAB:		SPLFP--TO SKIP SPACE AND LINEFEED
	IDE	IDENP--TO RECOGNIZE IDENTIFIERS
	INO	INUMP--TO RECOGNIZE INTEGER
	TEX	NUMBERS
%	ELS	TEXTP--TO RECOGNIZE TEXT
,	P55	
(	ELS	
)	ELS	
[	OPT	
]	OPT	
.	ELS	
;	ELS	
!	OPT	
+	AOP	
*	AOP	
?	ELS	
	-1ØØ	GOMCH--TO GET MORE CHARACTER
:=	ELS	
/\	LOP	
\/	LOP	
<=	ROP	
-,	ELS	
>>	ROP	
/=	ROP	
==	ELS	
	-1ØØ	GOLCH--TO GIVE BACK ONE CHARACTER
=	ROP	
-	AOP	
:	ELS	
<	ELS	
/	ELS	
>	P55	
$	ELS	

TABLE 1

LIST OF THE SYMBOLS OF THE HP-MADE (SL)
Remark: AOP, LOP, ROP and P55 are operator classes (OP)

variable			constant		
mnemonic	internal	repr.	mnemonic	internal	repr.
INT	1		INTC	-1	
BOOL	2		BOOLC	-2	
LAB	3		LABC	-3	
PROC	4		PROCC	-4	
ATOM	6		ATOMC	-6	
ATCL	7		ATCLC	-7	
TEXT	1Ø		TEXTC	-1Ø	
ASTA	1ØØ		ASTAC	-1ØØ	
OSTA	1Ø1		OSTAC	-1Ø1	

TABLE 2

HP-MADE TYPES

Remark: Some of the reserved words of HP-MADE (standard variables, constants) are of this types. A group of reserved words belongs to a type DECLARATOR (-5) and an other to type PWPROC (5).

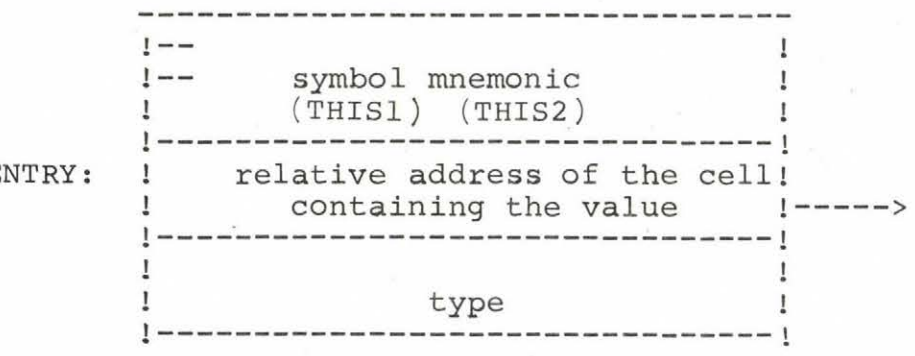


FIGURE 2. A typical ST entry

	SYMBOL MNEMONIC	((ENTRY))	COMMENT	TYPE
STABE:	SCAN	2		PROCC
	LOOK	4	INTERMEDIATE	PROCC
	SCLO	6	CODE	PROCC
	SCBA	1Ø	RELATIVE	PROCC
	DUMY	12	ADDRESS	PROCC
	DEFA	12	GLS	PROC
	LENG	1Ø	RELATIVE	INT
	ERRO	13	ADDRESS	PROC
	ATOM	6	INTERNAL	DECLARATOR
	ATCL	7	REPRESENTATION	DECLARATOR
	BOOL	2	OF	DECLARATOR
	INTE	1	THE	DECLARATOR
	PROC	4		DECLARAOTR
	TEXT	1Ø	CORRESPONDING	DECLARATOR
	ASTA	1ØØ	TYPE	DECLARATOR
	OSTA	1Ø2		DECLARATOR
	DELE	DELET		RWPROC
	MADE	MADEN		RWPROC
	CLOS	CLOSE	PROCEDURE	RWPROC
	GENE	GENER	OF	RWPROC
	IF	IF	THE	RWPROC
	ATTR	ATTRI	RESERVED	RWPROC
	MADB	MADBE	WORD	RWPROC
	NEWP	NEWP		RWPROC
	RETU	RETP		RWPROC
	CALL	CALL		RWPROC
	GOTO	GOTO		RWPROC
	%	RETP		RWPROC
STFP1:	$	MDLHS		RWPROC

TABLE 3

HP-MADE SYMBOL TABLE
(ORIGINAL STATE)

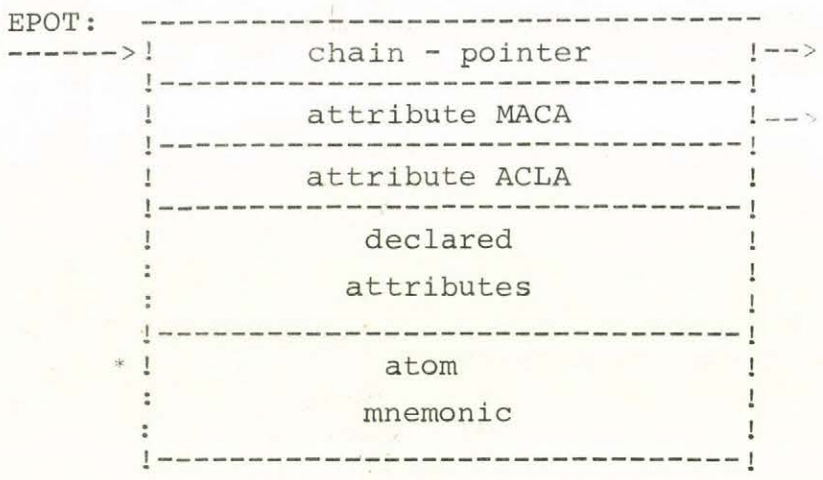


Figure 3. A typical SST entry

Remark: The last word occupied by the atom mnemonic will be negative.

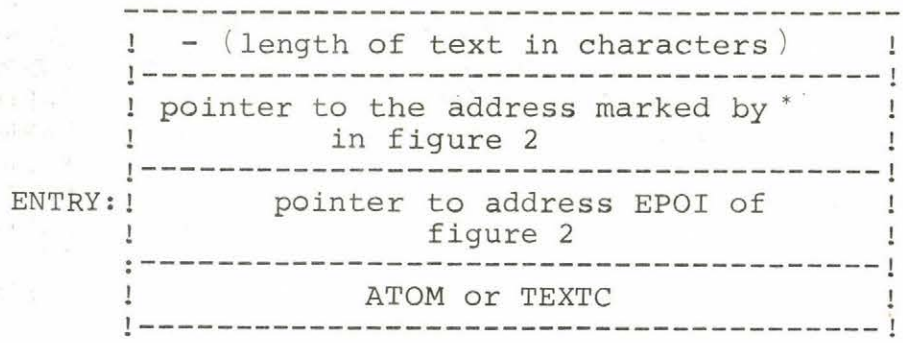


Figure 4. A macro name or text constant entry in ST

NAME	POINTER	INITIAL VALUE	REMARK
SL		ATAB	List of HP-MADE symbols for the scanner
ST	STBEG	STABE	Lower limit: STSTO
	STLAS	STFP1	Max.no. of new entries:44
ALIST		ALBEG	List of attributes Max.no. of attributes: 1Ø
HT		HTB	Hash table Length (H1): 64 words
SST	FREEP	SST	Atom table: 29Ø words
IC	ICIND	2Ø	Intermediate codes. (ICIND) is the actual no. of words occupied.
	ICSTP(abs.)	INCOD=(ICBERG)	(ICSTP) is the address of the intermediate code (ICSTP)=(ICBERG)+(ICIND)mod4ØØ
GLS	GBI(rel.)	4	(GBI) is the actual no. of labels+1, that is, the relative address of the beginning of the "real" GLS.
	SPOIN(rel.)	13	(SPOIN) is the no. of words reserved for global identifiers.(rel.to(GBI))
BUFER	BUFP	BUFER	"line-buffer":max.64 words
TBUF	TBUFP	(TBUBE)=TBUF	"text-buffer":max.64 words
SVUF	SBUFP	(SBUBE)=SBUF	"save-buffer":max.2Ø symbol
OPLIS	OPLIP	(OPLIB)=OPLIP	list of operators in SBUF

TABLE 4

Tables and buffers in the HP-MADE compiler

The HP-MADE distinguishes between two error-types: fatal errors and non-fatal errors.

A fatal error signifies, that one of the system tables is full -- you have to moderate your memory requirements and start again typing the program from the last saved state. (We mean saving by TSB on disk.) The consequence of a fatal error will be the printing out of the address of the checking instruction as an error-message, and a jump to FINISH COMPILER, STOP.

A non-fatal error will cause the print of the address of the checking instruction and will restore the state of the system at the end of the last right instruction, then will wait for a new instruction.

Intermediate codes

We consider the use of intermediate codes a convenient method for writing compilers.
The internal form of a MADE intermediate code consists of two parts: a head part and a parameter part. The head part (one word) includes the operation code (see Table 5), the complemeter of the number of parameters and the mode of each parameter (C,G or L). The parameter part (one word for each parameter) gives the values of the parameters (if the value of a non-C parameter is negative its absolute value will be considered as a pointer to the memory location containing the actual value of the parameter).

7.3. The global-local stack GLS

The GLS consists of 3 parts and only the third part is really a stack, but for the sake of simplicity we call stack this continuous memory-field (Figure 5).

In the first part of this memory field the relative ("ICIND") adresses of all the labels will be stored in compilation time. This rather inefficient method makes possible to compile in one pass and in a situation when only a relatively small part of the translated program can be reached. The "label part" of GLS will be punched out and thus "saved" for interpretation-time.

The second part of GLS is the storage assigned to global variables and working registers (GS).

The third part of GLS is the "local stack". It begins after the last word occupied by global variables (the exact number of them will be known by the end of the compilation and will be stored into GØ. The "elements" of the local stack are memory fields - each one corresponding to a block of the the MADE progaram. The actual number of the local stack elements is the number of the "opened blocks", that is, procedures or macros called but not yet "returned". When a block is opened (that is a BLS intermediate code is interpreted), it will take a "necessary number" of words beginning from the actual end of the preceeding local stack element or GS.

In case of procedure calls the "necessary number" is well known at compilation time - the length of the stack element is constant. (one of the BLS - functions is to store it into LØ).

In case of macros the number of parameters will generally be known only at interpretation time, this is why the "chaining" of a new actual parameter (CHAI) to the actual--parameter-chain in LS must update the length of the actual LS element. See figure 6 for details.

At interpretation time GØ always "points" to beginning of the actual LS element or just to the first free memory location. When a block "returns", (LØ) number of memory locations will be liberated.

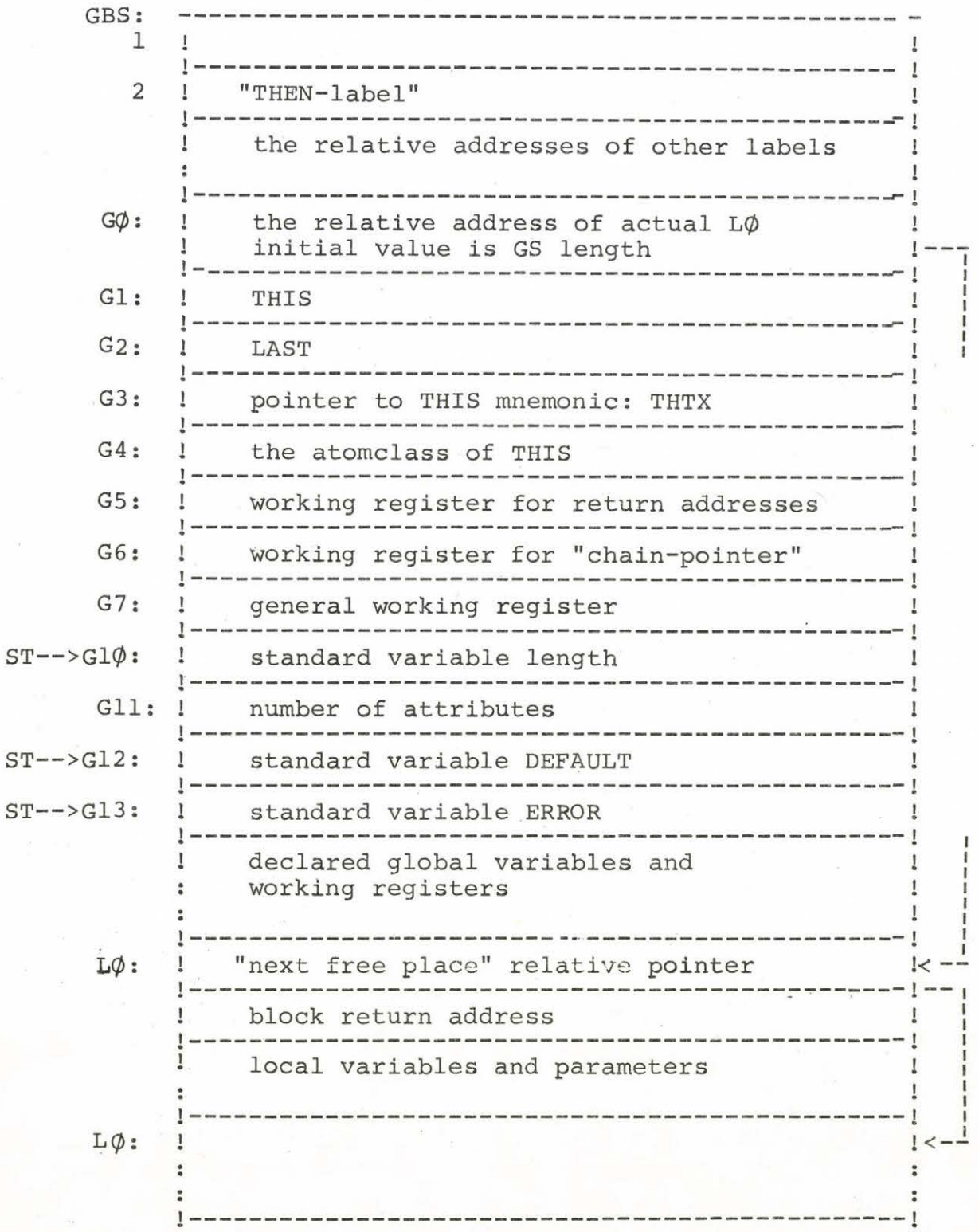


Figure 5. The structure of GLS

7.4. The HP-MADE interpreter

The task of the interpreter is to execute the intermediate code translator generated by the HP-MADE compiler.

In Tbale 5. we give the list of the MADE intermediate codes. They have maximum 3 parameters, which may be basically of 3 different types: type 1 means a normal GLS address (or a fixed-point constant), type 2 is an ICIND address, type 3 is an SST address (for text values).

The meaning of intermediate codes OR, MULT, PLUS, MINUS, AND, LEQS (less or equal), EQS (equal), and GRS (greater) is very simple: execute the corresponding operation on the first and second parameters and store the result into the address specified by the third parameter. STORE means store of the first parameter into the address given by the second (TSTOR if the parameter's type is text), INCR and DECR are abbreviated forms of PLUS and MINUS, respectively, if the second parameter would be the constant 1. JMP means a simple jump to an other intermediate code,
in EQJ and NEQJ (for text TEQJ and TNEQJ) the jump is effectuated if the condition (equality or non-equality) is NOT fulfilled.
ITOT and ATOT transform integer or atom values to text, GENS and GENO store text values to segments or to other output fields, respectively.

DELO (delete from output stack), DELA (delete from atom stack), AST (compute atomstack address), OST (compute outstack address) manipulate with stacks.

There are intermediate codes to compute the GLS address of a macro - actual parameter being presently inputted (CHAI) and actual parameters already in GLS (CHAO). (Figure 6.).

CODE NUMBER	MNEMONIC	THE TYPE OF PARAMETERS 1TH	2ND	3RD
34	OR	1	1	1
35	BLS	1		
36	CHAI	1		
37	CHAO	1	1	
4ϕ	AST	INO	1	1
41	OST	INO	1	1
42	DELA	1		
43	DELO	1		
44	CLOSE	1		
45	GENS	3	INO	
46	GENO	3	1	
47	INCR	1	1	
5ϕ	MDEND			
51	DECR	1	1	
52	MULT	1	1	1
53	PLUS	1	1	1
54	RET			
55	MINUS	1	1	1
56	----			
57	AND	1	1	1
6ϕ	STORE	1	1	1
61	TSTOR	3	3	
62	TEQJ	3	3	2
63	TNEQJ	3	3	2
64	ITOT	1	3	
65	ATOT	1	3	
66	JSB	2		
67	JSBM	2		
7ϕ	JMP	2		
71	CALL	2	1	2
72	EQJ	1	1	2
73	NEQJ	1	1	2
74	LEQS	1	1	1
75	EQS	1	1	1
76	GRS	1	1	1

TABLE 5

HP-MADE INTERMEDIATE CODES

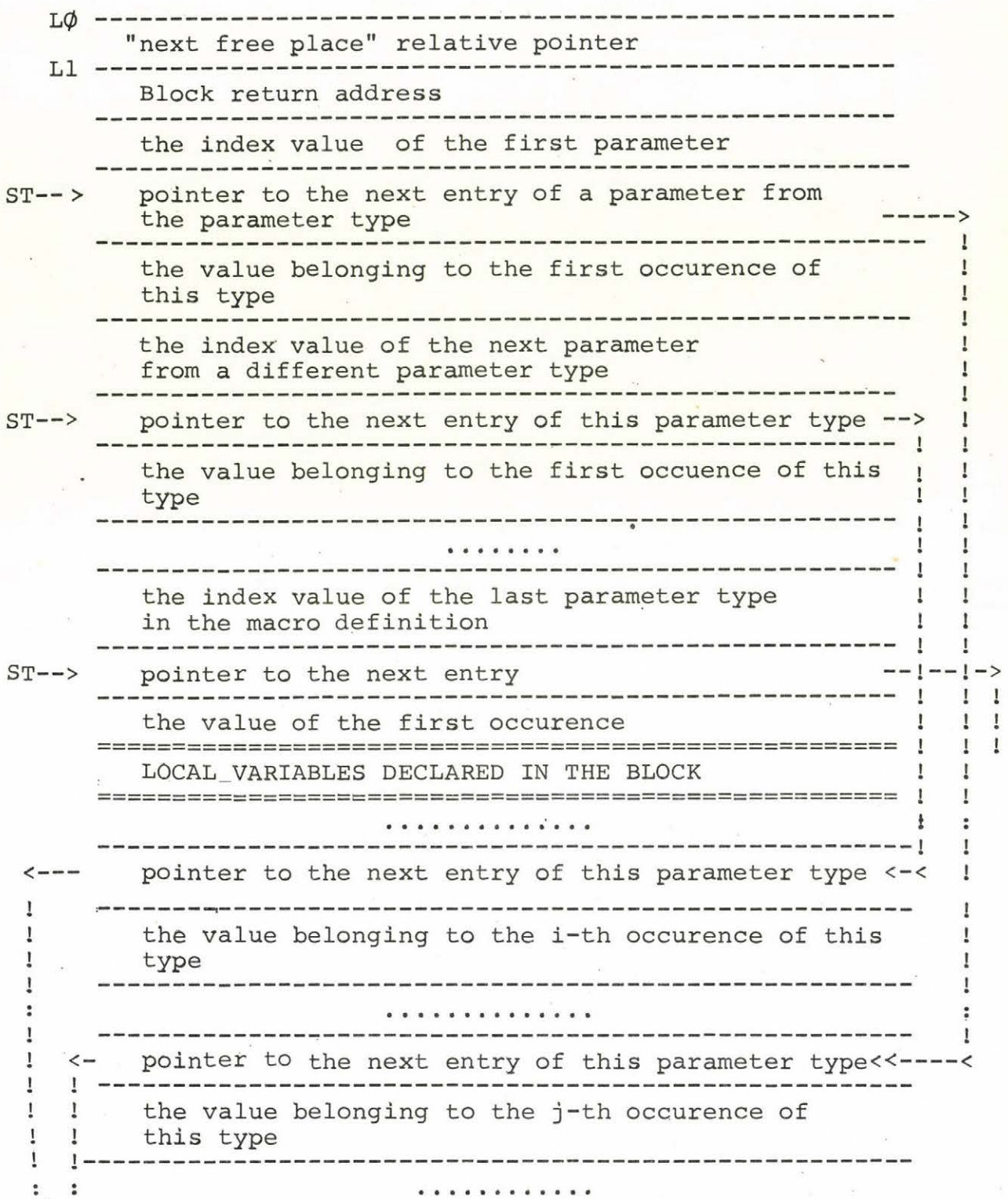


Figure 6. Macros in LS

The interpreter is something like a "case statement" (implemented by a jump table), the 35 cases are corresponding to the 35 intermediate codes.

There are only 3 important subrutines in the interpreter corresponding to the 3 type of parameters: OPND1, OPND2 and OPND3. They compute the absolute address of the parameter and store it into the accumulator.
In Table 6. we give the meaning of the three operand types as a function of their modes. We call icaddress a location containing the absolute address of the actual intermediate code. The content of icaddress is the index (in some cases it is really an index). GLSB is the beginning address of GLS, GØB is that of the global stack. ICB signifies the address of the first intermediate code.

OPERAND TYPE	OPERAND MODE	SIGN	ADDRESS EXPRESSION	REMARK
1	C		(icaddress)	
	G	+	index+GØB	
	L	+	index+ GØB	
	G	-	(-index+GØB)	
	L	-	(-index+(GØB))	
2	C		index+ICB	
	G	+	(index+GLSB)+ICB	
	L			non exists
	G	-	(-index+GØB)	
	L	-	(-index+ GØB))	
3	C		((ICADDRESS))	
	G	+	index+GØB+1	(index+GØB)->CHI
	L	+		non exists
	G	-	(-index+GØB)	(MAXI)->CHI
	L	-	(-index+ GØB))	(MAXI)->CHI

Table 6.
Operand types and modes

The CHI (check index) will be checked in store-type statements. MAXI contains the actual upper-limit address characteristic to the type of the operation we are involved in.

A TANULMÁNYOK sorozatban eddig megjelentek:

1/1973 Pásztor Katalin: Módszerek Boole-függvények minimális vagy nem redundáns, {∧,∨,¬} vagy {NOR} vagy {NAND} bázisbeli, zárójeles vagy zárójel nélküli formuláinak előállitására

2/1973 Вашкеви Иштван: Расчленение многосвязных промышленных процессов с помощью вычислительных машин

3/1973 Ádám György: A számitógépipar helyzete 1972 második felében

4/1973 Bányász Csilla: Identification in the Presence of Drift

5/1973* Gyürki J.-Laufer J.-Girnt M.-Somló J.: Optimalizáló adaptiv szerszámgépirányitási rendszerek

6/1973 Szelke E.-Tóth K.: Felhasználói Kézikönyv /USER MANUAL/ a Folytonos Rendszerek Szimulációjára készült ANDISIM programnyelvhez

7/1973 Legendi Tamás: A CHANGE nyelv/multiprocesszor

8/1973 Klafszky Emil: Geomatriai programozás és néhány alkalmazása

9/1973 R. Narasimhan: Picture Processing Using Pax

10/1973 Dibuz Á.-Gáspár J.-Várszegi S.: MANU-WRAP hátlaphuzalozó, MSI-TESTER integrált áramköröket mérő, TESTOMAT-C logikai hálózatokat vizsgáló berendezések ismertetése

11/1973 Matolcsi Tamás: Az optimum-számitás egy uj módszeréről

12/1973 Makroprocesszorok, programozási nyelvek. Cikkgyüjtemény az NJSzT és SzTAKI közös kiadásában.
Szerkesztette: Legendi Tamás

13/1973 Jedlovszky Pál: Uj módszer bonyolult rektifikáló oszlopok vegyészmérnöki számitására

14/1973 Bakó András: MTA kutatóintézeteinek bérszámfejtése számitógéppel

15/1973 Ádám György: Kelet-nyugati kapcsolatok a számitógépiparban

16/1973 Fidrich I.-Uzsoky M.: LIDI-72 listakezelő rendszer a Digitális Osztályon, 1972.évi változat

17/1974 Gyürki József: Adaptiv termelésprogramozó rendszer /APS/ termelőmühelyek irányitására

18/1974 Pikler Gyula: MINI-számitógépes interaktiv alkatrészprogramiró rendszer NC szerszámgépek automatikus programozásához

19/1974 Gertler,J.-Sedlak,J.: Software for process control

20/1974 Vámos,T.-Vassy,Z.: Industrial Pattern Recognition Experiment - A Syntax Aided Approach

21/1974 A KGST I.-15-1.: "Diszkrét rendszerek automatikus vezérlése" c. témában 1973. februárban rendezett szeminárium előadásai

22/1974 Arató,M.-Benczur,A.-Krámli,A.-Pergel,J.: Stochastic Processes, Part I.

23/1974 Benkó S.-Renner G.: Erősen telitett mágneses körök számitógépes tervezési módszere

24/1974 Kovács György-Franta Lászlóné: Programcsomag elektronikus berendezések hátlaphuzalozásának tervezésére

25/1974 Járdán R. Kálmán: Háromfázisu tirisztoros inverterek állandósult tranziens jelenségei és belső impedanciája

26/1974 Gergely József: Numerikus módszerek sparse mátrixokra

27/1974 Somló János: Analitikus optimalizálás

28/1974 Vámos Tibor: Tárgyfelismerési kisérlet nyelvi módszerekkel

29/1974 Móricz Péter: Vegyészmérnöki számitási módszerek fázisegyensulyok és kémiai egyensulyok vizsgálatára

30/1974 Vassy,Z.-Vámos,T.: The Budapest Robot - Pragmatic Intelligence

31/1975 Nagy István: Frekvenciaosztásos középfrekvenciás inverterek elmélete

32/1975 Singer D., Borossay Gy., Koltai T.: Gázhálózatok optimális irányitása különös tekintettel a Fővárosi Gázmüvek hálózataira

33/1975 Vámos,T.-Vassy,Z.: Limited and Pragmatic Robot Intelligence

Mérő,L.-Vassy,Z.: A Simplified and Fastened Version of the Hueckel Operator for Finding Optimal Edges in Pictures

Галло В.: Программа для распознавания геометрических образов, основанная на лингвистическом методе описания и анализа геометрических структур

34/1975 László Nemes: Pattern Indentification Method for Industrial Robots by Extracting the Main Features of Objects

35/1975 Garádi-Krámli-Ratkó-Ruda: Statisztikai és számitástechnikai módszerek alkalmazása kórházi morbiditás vizsgálatokban

36/1975 Renner Gábor: Elektromágneses tér számitása nagyhőmérsékletü anyagban

37/1975 Edgardo Felipe: Specification problems of a process control display

38/1975 Hajnal Andrásné: Nemlineáris egyenletrendszerek megoldási módszerei

39/1975* A.Abd El-Sattar: Control of induction motor by three phase thyristor connections in the secondary circuit

40/1975 Gerhardt Géza: QDP Grafikus interaktiv szubrutinok a CDC 3300-GD'71 grafikus konfigurációra

41/1975 Arató M.-Benczur A.-Krámli A.-Pergel J.: Stochastic Processes, Part II.

42/1975 Arató M.: Fejezetek a matematikai statiszkikából számitógépes alkalmazásokkal

43/1975 Matavovszky Tibor- dr. Pásztorné Varga Katalin: Programrendszer Boole-függvény együttes egyszerüsitésére vagy minimalizálására

44/1975 Bacsó Nándorné: Pneumatikus áramköri hazardok

45/1975 Varga András: Ellenpárhuzamos félvezetőpárokkal vezérelt aszinkronmotoros hajtások számitási módszerei

46/1976 Galántai Aurél: Egylépéses módszerek lokális hibabecslései

47/1976 Abaffy József: A feltétel nélküli függvényminimalizálás kvadratikus befejezésü módszerei

48/1976 Strehó Mária: Stiff tipusu közönséges differenciálegyenletek megoldásáról

49/1976 Gerencsér László: Nemlineáris programozási feladatok megoldása szekvenciális módszerekkel

Készült Pisában 1974-ben, ahol a szerző C.N.R. ösztöndijasként dolgozott.

A * -gal jelölt kivételével a sorozat kötetei megrendelhetők az Intézet könyvtáránál /Budapest, XIII. Victor Hugo u. 18-22/