Eidgenössische
Technische
Hochschule
Zürich

*Berichte des
Instituts
für
Informatik*

K. V. Nori, U. Ammann
K. Jensen, H. H. Nägeli

The PASCAL ‹P› Compiler: Implementation Notes

Dezember 1974 *10*

Eidgenössische
Technische
Hochschule
Zürich

Berichte des
Instituts
für
Informatik

K. V. Nori, U. Ammann
K. Jensen, H. H. Nägeli

The PASCAL ‹P› Compiler: Implementation Notes

The PASCAL 'P' Compiler: Implementation Notes

K.V. Nori*, U. Ammann, K. Jensen, H.H. Nägeli

Abstract

The PASCAL 'P' compiler is a portable compiler for a subset of 'Standard PASCAL'. This compiler is written using exactly the subset it processes and it generates object code for a hypothetical stack computer. This report is a documentation of the stack computer and of the compiler. The latter part of the documentation has proved to be very useful to one of the authors (K.V. Nori) in informally verifying the compiler.

* United Nations TAO Fellow,

on leave from: Computer Group, TIFR, Homi Bhabha Road, Bombay-400005, India.

performance of various kinds of optimization as well. As an experiment, these steps were repeated for the CDC 6000 series and for a hypothetical stack computer. For the CDC 6000 series, the refinements were embellished to include all of 'Standard PASCAL' as well as optimization of object code. The main purpose of the experiment was to indicate the feasibility of the idea that object code for different machines could be generated by versions of a compiler which had a great deal of substructure in common (and, no doubt, to obtain an efficient PASCAL compiler for the CDC 6000 series too).

A very useful by-product of the above experiment was a basis for a portable compiler. The stack computer, SC, was not specifically designed for portability, as was the OCODE machine for the portable BCPL compiler (Richards 71). Rather, it was evolved to conveniently couch code generation and address assignment in the parser delivered by step 3. Exploiting this by-product for the purposes of portability is the genesis of the PASCAL 'P' compiler.

The Implementation Kit

An implementation kit was prepared, by two of the authors of this report (U. Ammann and K. Jensen) in early 1973, by means of which the PASCAL 'P' compiler could be ported. This kit consisted of:

a) The PASCAL 'P' compiler in source form (Ammann 73b);
b) The PASCAL 'P' compiler in the assembly language of SC;
c) An assembler/interpreter for programs of SC (Jensen and Wirth 73);
d) PASCAL definition reports (Jensen and Wirth 74; Hoare and Wirth 72);
e) Documentation of SC.

One way of quickly acquiring the PASCAL 'P' compiler was to write an interpreter for SC; running part (b) of the kit on this interpreter would then make PASCAL available on the system. This method could be used to bootstrap the PASCAL 'P' compiler: one of the authors (U. Ammann) has used such a bootstrap as a means for validating the compiler by exploiting the fact that the initial object code file and the object code file resulting from the bootstrap must be identical. Another method of implementing this compiler is to translate part (b) of the kit to a machine language program of the implementation machine (Laws and Wichman 73; Friesland et al 73).

Several factors have led to the current version of this compiler. The most important ones are: a) Feedback from implementation efforts such as those above; b) The need to restrict the compiler into using a subset of the standard and extending the compiler so that it processes exactly this subset; and c) Parameterizing SC so that it may be more easily implemented on an actual implementation machine. One of the authors (K.V. Nori) undertook to examine these issues, consider the trade-offs involved and effect the necessary changes. It should be noted however that, essentially, the implementation kit consists of the same parts for the present version of the PASCAL 'P' compiler.

The Language Processed by the PASCAL 'P' Compiler

The language processed by the PASCAL 'P' compiler is 'Standard PASCAL' with some omissions and one change. These are of no consequence to the bootstrap process; the omission can be filled in quite easily as there are indications in the compiler where the required extensions should be filled in.

The features which are not processed are:
a) procedures/functions as parameters.
b) goto's leading out of procedure/function bodies.
c) all kind of files except predefined character files (of type 'text').
d) all features associated with 'packing'.
e) characters not in the restricted ISO set represented by the CDC Display Code.

Change:
f) the standard procedure 'dispose' is replaced by 'mark' and 'release'

For further details see Appendix IV.

Implementing Strategies

Assuming that an implementor of the 'P' compiler is already familiar with PASCAL, the first thing he has to familiarize himself with is the SC. The syntax of assembly programs for the SC is given in Appendix I. The semantics of SC programs can be easily deciphered from part (c) of the implementation kit. This assembler/interpreter is expressed in PASCAL. The implementor should take note that this program is only a guideline to understanding the SC and not necessarily the best way to implement it on any computer. The reason for such a strong statement should by no means be taken to reflect upon the quality of this program! We wish to emphatically bring to the reader's notice that the data manipulated by the SC is parameterized according to the storage space it requires: the assembler/interpreter is written for the case where 1 storage unit is required to preserve any kind of data. Additional explanatory notes are given in Appendix II.

Apart from several pragmatic issues, a compiler has to perform two very important tasks. Firstly, it has to check whether the source program is well formed. And secondly, if it is well formed, then the compiler has to generate an object program which is semantically equivalent to the source program. The design of PASCAL is such that the first task is greatly simplified. The second task concerns a relationship between the source and object languages. This relationship is concisely presented in Appendix III.

Having digested the information in these appendices, an implementor has sufficient information to devise an implementation strategy. Three such strategies are presented here as examples; they are definitely not an exhaustive coverage of the possibilities.

If the expected use of PASCAL is for teaching purposes and only short programs are to be compiled and executed, then the simplest method of implementing the 'P' compiler is by writing an efficient assembler/interpreter for SC. The only logical hurdle that may confront an implementor using such a strategy is the fetching and storing of data from and to the memory of the stack computer. This problem needs to be efficiently solved especially when different kinds of data, e.g. integers, characters, sets, pointers, etc., require different units of store for their representation. Once the assembler/interpreter is implemented, part (b) of the implementation kit could be used interpretively to compile short programs. The output of the compiler can be then processed by the assembler/interpreter to obtain an interpretive execution of these programs. It should be noted that despite interpretation, the overall throughput will compare favourably with commercial compilers for large languages. The resources required for such an implementation will approximately be: (a) 54 K bytes for storing the object code of the 'P' compiler in machine language form of the SC; (b) about 20-30 K for the data segment of the 'P' compiler to compile programs usually given as student exercises; and (c) the store required for the assembler/interpreter. Bootstrapping the PASCAL 'P' compiler by this method can be very expensive and is not recommended.

Another strategy, more suited to the bootstrapping of the compiler, is to convert part (b) of the implementation kit to an assembly language program, by using a conventional macro-processor or some such scheme. This will no doubt increase the storage requirements initially. But as the implementor is forearmed with the code generation patterns of the 'P' compiler (Appendix III) and also the exact scheme of converting part (b) of the kit to assembly code, an effective 'peephole optimization' (McKeeman 65) could be planned. Once this has been accomplished, part (a) of the kit, 'P' compiler in PASCAL, can be directly modified to generate code for the implementation machine.

In case storage is the main constraint, a judicious mixture between interpretation and machine execution can be used. A possible technique is that of 'threaded code' (Bell 73). A considerable gain in speed with marginal increase in storage requirements, in comparison with the first strategy above, could result. This technique therefore seems viable for attempting a bootstrap. The plan of action is very much like that of the second strategy above. An intermediate language suited to the implementation machine architecture can be designed (under the constraint of the ease of mapping the code generation pattern of the 'P' compiler to code in this language) along with a run time package to support this language. Then part (b) of the kit could be transformed to this code in this language. The result is that PASCAL is now available on the implementation machine. The bootstrap can now be effected by modifying part (a) of the kit and using it.

In our experience neither portability nor bootstrapping can be obtained at very little expense (as one is sometimes led to believe from papers on this topic). In the absence of 'heavy artillery' like sohpisticated macro-processors (Waite 70, 73) or very high-level machine languages (Poole 74) or a host machine (Wirth 71c; Welsh and Quinn 72), the work required to move an especially machine and system dependent software - like a compiler - is not negligible. However, if the work required is approximately an order of magnitude less than that for writing the whole software from 'scratch', we would consider the method as viable for purposes of portability. Within such a constraint, we feel that the solution we offer is a feasible one.

Portability and adaptability considerations demand a generality in the solution of problems which could prove to be redundant in most of the specific cases of their use. More often than not, this aspect is looked upon as a deterrent to general solutions because severe constraints arise due to the limited resources (or their poor management) of the implementation machine. Something akin to 'conditional assembly' is required to prevent loss of efficiency. For PASCAL, a separate project is currently in progress at ETH, which tries to get around this issue (Wirth 74).

The PASCAL 'P' Compiler Options

The PASCAL 'P' compiler uses 3 external files: they are labelled INPUT, OUTPUT and PRR respectively and are all of type 'text'. The file INPUT contains the source program; the file OUTPUT contains the source program listing (optionally), the symbol tables (optionally) and compiler messages; and the file PRR optionally contains the object program, for SC, in its assembly language form. In the preceding sentence, by 'optionally' we mean that these components may or may not be included in the respective files depending on the user's discretion. These user-options can be set at any point in the user program by means of a pseudo-comment. A pseudo-comment is one in which the first symbol within the comment is a '$' symbol. Following the '$' symbol are option settings separated by ',' symbols. An option setting starts with the letter 'L', to indicate 'listing', or 'T', to indicate 'tables', or 'C', to indicate 'code'; this letter is followed by either a '+' symbol or a '-' symbol, the former indicating that the option is to be turned on and the latter indicating that the option should be turned off. Default setting for these options are: L is on; T is off; and C is off.

So much for user options; we now turn to implementor-options and restrictions.

A compiler by its very nature cannot be machine-independent: after all the compiler has to be expressed in <u>some</u> language, generate code for <u>some</u> machine and run in some operating system. Be that as it may, in the interest of portability the commitments arising from the above aspects should be minimal. The design of PASCAL is such that it needs a very simple run-time support and hence is largely independent of the environment provided by an operating system. The only commitment we make is that of the character set. It is possible to be independent of the character set too, but the expense is unwarranted compared to the gain in conciseness of the program (mainly, the lexical analyzer). The character set used is a subset of the ISO code as implemented by the CDC Display code. It is imperative, therefore, that the implementor does code conversion in the early phases of his bootstrap process. Since the PASCAL 'P' compiler is expressed in PASCAL, no further linguistic or representational commitments are necessary. The rest of the considerations arise from the nature of the SC.

It is quite possible to design a very clean machine, one in which no commitment is made at all with regard to representation of basic units and structures of both data and operations; in fact, the design can be predicated purely upon logical considerations stemming from PASCAL, e.g., JANUS (Coleman et al 74). The hitch is that the interpreter for such a language is quite complex and is bound to be slow. Even if interpretation is never intended as the mode of usage of code of such a machine, the description or definition of this machine is complex enough to impair perfect communication between its designer and implementor. Another solution is to make a complete commitment with regard to data and operation representations, e.g. OCODE for BCPL (Richards 71). The rigidity of this solution with respect to efficient adaptation to different systems along with the inherent problem with BCPL, that programs in it may not be truly machine-independent because of the word-length problem, makes us discard this approach too. Our solution lies between these two: all basic operations of the SC arise out of logical requirements due to PASCAL, though some extra operations come in because programs are 'linearized' and data are accessed within a 'tree'; all basic data types of PASCAL have attributes for both representation as well as size - furthermore, sets appear as basic units in this machine. This specific choice is dictated by reasons of simplicity and efficiency of the compiler as well as the interpreter.

Having explained our motivation, we give below the specific constants which can be set by an implementor, in order to obtain a part (b) of the implementation kit which is most suited to his implementation machine:

a) MAXINT: The value of this constant is implementation dependent;

it indicates the largest integer which the compiler will process.

b) CHARSIZE, PTRSIZE, INTSIZE, BOOLSIZE, REALSIZE, SETSIZE:
These constants indicate the number of storage units required to preserve values of type character, pointer, integer, boolean, real and set. All storage allocation is done in terms of these constants. Also, storage allocation of data is according to the simple rule that consecutively declared entities are allocated the requisite number of consecutive storage units. Note that sets have to be able to hold at least 59 elements: this restriction arises from the fact that the compiler itself uses large sets. Implementors wishing to perform the full bootstrap should pay particular attention to the address alignment problem when all these constants are not equal to one another.

c) DIGMAX: This constant represents the maximum length of a string of characters which may be used to represent unsigned numeric constants. At present its value is 15. This restriction is not a machine-dependent feature but a pragmatic one. Note that real numbers are not converted into an internal form but preserved as character strings. Consequently, the compiler itself uses no real numbers.

d) STRGLGTH: At present, this constant has the value 16. It represents the maximum length of strings processed by this compiler. That STRGLGTH is greater than DIGMAX is not coincidental: An implementor desiring to change one of them should also change the other in accordance with the above relationship.

We now turn to the commitments and restrictions imposed by the address assignment (for data) and code generation patterns of the 'P' compiler, and the requirement of simplicity of the assembler/interpreter for the SC.

a) Organization of programs as generated by the compiler:

absolute address	instruction	
0	MST	0
1	CUP	label for main program
2	STP	
3	ENT	name indicating a constant which is the size of data segment of procedure whose body is encountered first.

The reader is referred to the assembler/interpreter for SC and to Appendix II for further explanation about the instructions.

b) Organization of data segments of procedures (they all start at relative address 0) according to the address assignment done by the compiler:

type of data	remarks
mark stack information	space for value returned by function + 3 pointers (see Appendix II);
parameters	call by reference parameters: address call by value parameters: if size of value parameter is one storage unit, then the value, else the address; for all value parameters whose size is not equal to 1 storage unit, requisite local storage is allocated so that the value may be copied at procedure entry at run-time (see Appendix III on code generation for procedure bodies);
local variables	allocation according to the simple rule: consecutively declared entities are allocated consecutive storage units.

Note that in the outermost block, after the mark stack information, space is pre-allocated for files INPUT, OUTPUT, PRD (second input file) and PRR (second output file) respectively.

c) Due to the nature of the mark-stack information, function return has an additional parameter, viz. the type of value returned by the function. This information allows a proper adjustment of the top of the stack when the return is effected.

With this, we hope to have covered what every implementor should definitely make a note of. The various details are reserved for the appendices which follow.

Acknowledgments

It is a pleasure to acknowledge the help, advice and criticism of our colleagues H. Sandmayr and Dr. E. Miyamoto. We are grateful to V.K. Le for updating the assembler/interpreter. Prof. Wirth's guidance, taste for programming and acumen with regard to balancing theory and practice in his work has influenced us greatly: we hope it has shown up in our effort. One of the authors (K.V. Nori) wishes to acknowledge the TAO Fellowship so generously granted by ECE, UNO: his participation in this effort would not have been possible otherwise.

References

Ammann U. 73a The method of structured programming applied to the development of a compiler: Proceedings of ACM International Computing Symposium, Davos, Switzerland.

— 73b Step 5: A PASCAL compiler developed by stepwise refinement; program listing.

Bell 73 Threaded Code: CACM, Vol. 16, No. 6, pp. 370-372.

Colemann S.S., Pode P.C. and Waite W.M.
74 The mobile programming system: JANUS: Software - Practice and Experience, Vol. 4, No. 1, pp. 5-23.

Dijkstra E.W. 70 Notes on Structured Programming: Technical report, T.H., Eindhoven; see also 'Structured Programming' by Dahl, Dijkstra and Hoare, Academic Press, 1972.

Friesland G., Gross-Lindemann C.O., Lorenz P.W., Nagel H.H. and Stirl P.J.
73 A PASCAL compiler bootstrapped on a DEC System 10: (Hamburg University) private communication.

Hoare C.A.R. and Wirth N.
72 An axiomatic definition of the programming language PASCAL: Technical Report 6, ETH Zürich; see also Acta Informatica, Vol. 3, pp. 335-355, 1973.

Jensen K. and Wirth N.
73 An assembler/interpreter for a hypothetical stack computer; program listing.

— 74 PASCAL - User Manual and Report, Springer-Verlag, 1974.

Laws J. and Wichman B.A.

73 Notes on the PASCAL P-code system: (NPL, UK) private communication.

McKeeman W. 65 Peephole optimization: CACM, Vol. 8, pp. 443-444.

Poole P.C. 74 Portable and adaptable compilers: Lecture notes of 'Advanced Course on Compiler Construction', TU, München; to be published by Springer-Verlag.

Richards M. 71 The portability of the BCPL compiler: Software - Practice and Experience, Vol. 1, pp. 135-146.

Waite W.M. 70 The mobile programming system: STAGE2: CACM, Vol. 13, pp. 415-421.

—— 73 Implementing software for non-numeric applications: Prentice Hall.

Welsh J. and Quinn C.

72 A PASCAL Compiler for the ICL 1900 Series Computers: Software - Practice and Experience, Vol. 2, pp. 73-77.

Wirth N. 71a Program development by step-wise refinement: CACM, Vol. 14, No. 4, pp. 221-227.

—— 71b The programming language PASCAL: Acta Informatica, Vol. 1, pp. 35-63.

—— 71c The design of a PASCAL compiler: Software - Practice and Experience, Vol. 1, pp. 309

—— 72 On PASCAL, Code Generation and the CDC 6500: Technical Report, Stanford University.

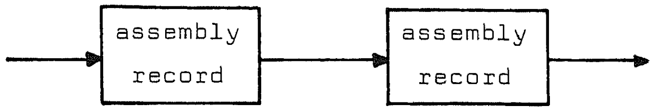
73 Systematic Programming: an introduction, Prentice-Hall.

74 PASCAL and New PASCAL: PASCAL Newsletter no. 2.

APPENDIX I: The Syntax of Stack Computer Assembly Language

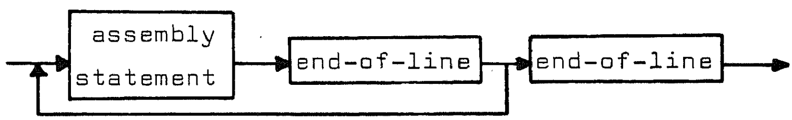
The syntax of Stack Computer Assembly Language is given in the form of Syntax Diagrams.

Assembly program

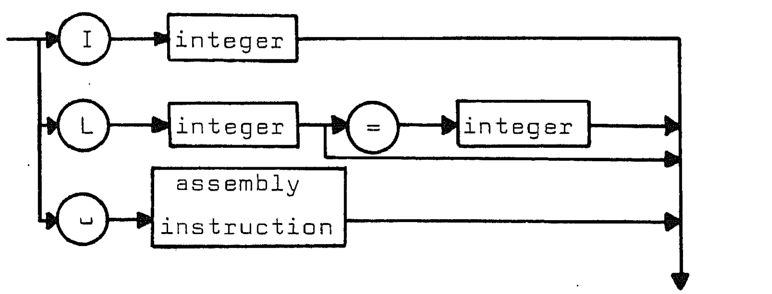


Note: The first assembly record consists of the whole program and should be loaded from absolute location 3 of the SC. The second record, to be loaded from absolute address 0 of the SC, consists of a call to the outermost block (please see the subsection "Organisation of programs as generated by the compiler" in the section "The PASCAL 'P' Compiler Options").

Assembly record



Assembly statement



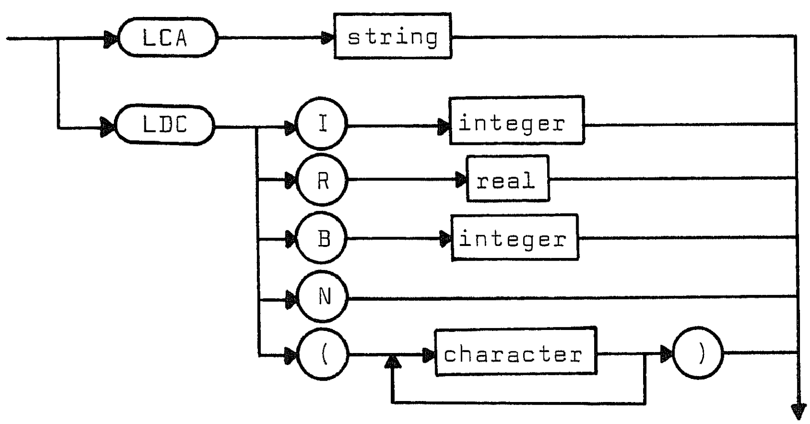
Notes: 1) The top part of the above diagram indicates the value of the location counter at which the next instruction should be assembled; its purpose is only to allow the reader of this code to relate to the source listing produced by the compiler and is generated for every tenth instruction.

2) The middle part allows the definition of symbolic names;

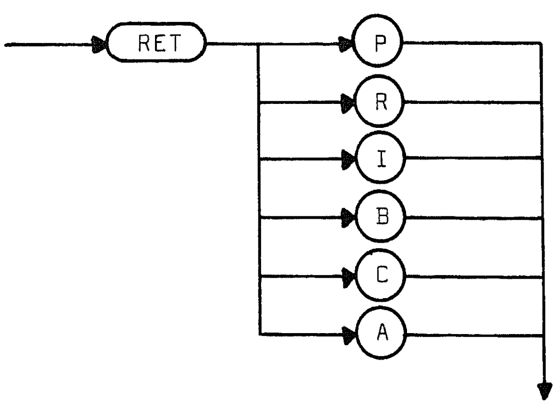
these names are very simple - an L followed by an integer which uniquely identifies the name; names are used either as labels or as constants; the former allow all control transfers to be symbolic and hence allow the code to be automatically manipulated - such names are defined by the point of their occurence, i.e., the value of the location counter at the time when such a statement is encountered; the use of names as constants is in defining the data segment length for a procedure or function or the outermost block - in such a case, the value of this constant follows a '=' symbol.

3) An assembly instruction is always preceded by a blank.

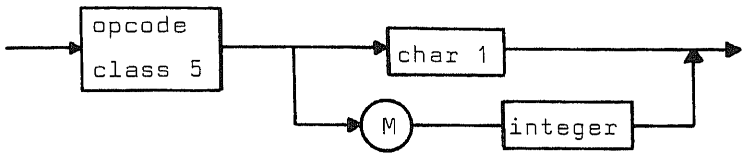
Instruction for loading constants



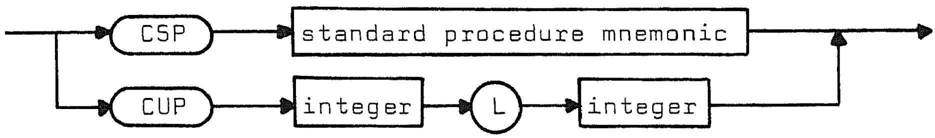
Return instruction



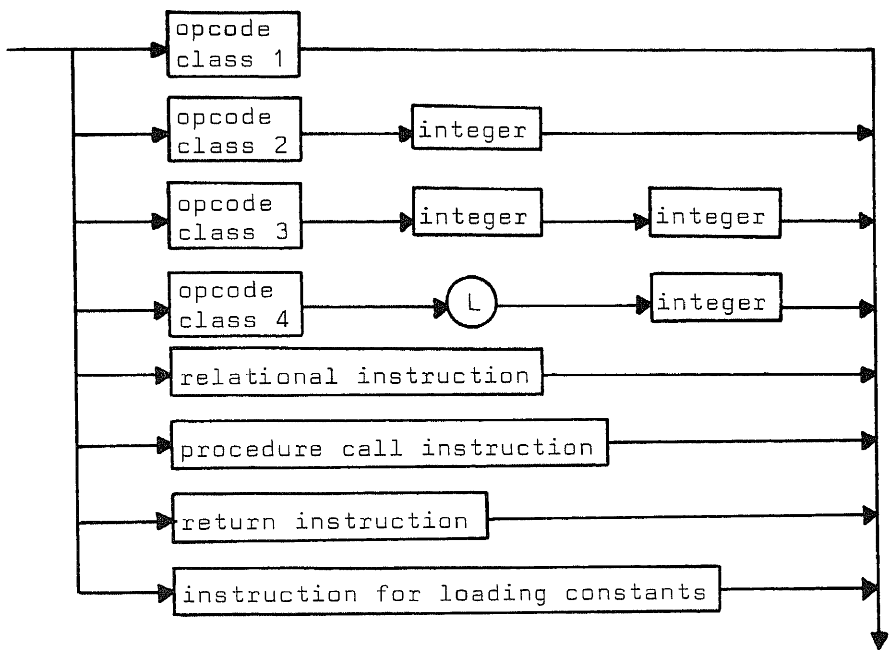
Relational instruction



Procedure call instruction



Assembly instruction



opcode class 1 = {ABI, ABR, ADI, ADR, AND, DIF, DVI, DVR, EOF, FLT,
FLO, INN, IOR, INT, MOD, MPI, MPR, NGI, NGR, NOT,
ODD, SBI, SBR, SGS, SQI, SQR, STO, STP, TRC, UNI}

opcode class 2 = {CHK, DEC, INC, IND, IXA, LAO, LDO, MOV, MST, SRO}

```
opcode class 3 = {LDA, LOD, STR}

opcode class 4 = {ENT, UJP, XJP, FJP}

opcode class 5 = {EQU, NEQ, GEQ, LEQ, GRT, LES}

char 1         = {A, I, R, B, S}

standard procedure mnemonic
               = {GET, PUT, ELN, NEW, WRS, WLN, WRI, WRR, WRC, RDI,
                  RDR, RDC, RST, SAV, SIN, COS, EXP, LOG, SQT, ATN, RLN}
```

integer, string, real as defined in PASCAL syntax.

APPENDIX II: Explanatory Notes on the Stack Computer

This appendix consists of two parts: the first part gives an informal description of the Stack Computer, SC; and the second part gives the list of opcodes and standard procedure mnemonics with brief comments to indicate their meanings. For a more precise definition of the Stack Computer, the reader is referred to the assembler/interpreter of the SC which is included in the implementation kit.

PART A: Description of the Stack Computer

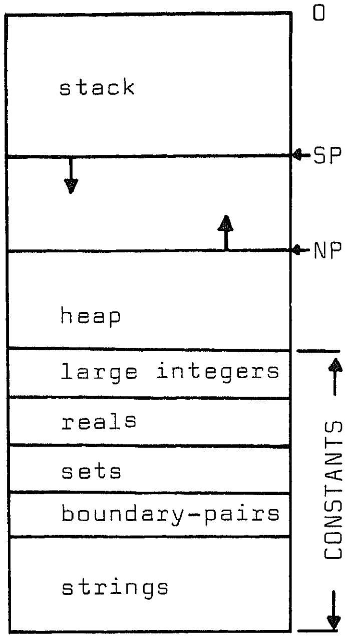
The stack computer, SC, consists of 4 registers and a memory. The registers are:

1) PC the program counter;
2) SP the 'stack' pointer;
3) MP the 'mark' pointer;
4) NP the 'new' pointer.

The meaning of SP, MP and NP will become apparent when we consider the organisation of the memory. The memory can be thought of as linear arrays of storage units (words): one of the parts of the memory is referred to as the code store, labelled CODE and the other part is referred to as the data store, labelled STORE. Their functions are obvious. Note that PC is always an index into CODE and SP, MP and NP are indices into STORE.

Each element of CODE has three fields: the OP field, the P field and the Q field. The actual length of these fields is implementation dependent with the restriction that the OP field should be at least 6 bits long and the P field at least 4 bits long respectively.

For STORE, each element has two fields: the tag field and the value field. The interpretation of the value field is dependent on the contents of the tag field. Furthermore, STORE is subdiveded into two parts: one part contains constants of various kinds whereas the other caters to the varying demands of data store, as required by the execution of PASCAL programs. This is depicted below.



The stack grows from 0 upwards and consists of all directly addressable data according to the data declarations.

Storage overflow occurs if SP and NP meet. The heap grows downwards from the point where the constants begin: its growth is dictated by use of the standard procedure NEW.

At present, the areas for each of the types of constants is fixed. Hence provision for checking table overflow is made. The use of boundary-pairs is not yet implemented in the current version of the P-compiler.

The following points are worth noting regarding the dynamic use of elements of STORE: the compiler's use of the heap resembles a second stack and so a very simple heap mechanism suffices. However, a user of the implementation kit desiring more flexibility should implement a more complex free-storage handling mechanism. Though it should be clear from the above picture, please note that SP points to the top of the stack and NP points to the top of the heap.

The stack has further internal structure; this structure allows a correspondence between the dynamic evaluation of a PASCAL program and its static text in that necessary links are maintained, dynamically, so that the accessable objects are those dictated by static program text (except for parameters - of course). To amplify, the stack consists of a sequence of 'data-segments', each of them 'belonging' to an activation of some procedure or function (except the first data segment, which starts at location 0, and which belongs to the outermost block, viz., the program block).

A data-segment consists of the following sequence of information: a 'mark-stack' part; a 'parameter' section if there are any parameters to the procedure or function to which the data segment belongs; a 'local-data' section if there are any local variables declared within the procedure or function to which the data-segment belongs; and finally, any temporary elements which may be required in the program evaluation process.*

In turn, the 'mark-stack' part consists of 4 consecutive fields: the first field is space for preserving the value returned by a function - it is not used by procedures, but included in their 'mark-stacks' too for the sake of uniformity and ease of implementation; the second field consists a pointer called the static link; the third field consists also of a pointer called the dynamic link; and finally, the fourth field consists of a pointer called the return address. Note that the static and dynamic link are indices into STORE whereas the 'return-address' is an index into CODE.

The parameter section consists of two parts, both of which may be empty. The first part consists of elements which are either: (a) pointers (indices into STORE) in case the corresponding parameters are of type 'call-by-reference' or of type 'call-by-value' but the size of the parameter is larger than the PTRSIZE; or (b) the parameter is 'call-by-value' and the value itself is passed as it requires only PTRSIZE or less. The second part pertains only to call-by-value parameters whose size is larger than PTRSIZE. In such a case, for each of such parameters, space is allocated as required by their respective sizes.

In order to effect a procedure/function call, a mark-stack instruction (MST) is executed with a parameter which allows the links to be filled.

* In this connection, the register MP points to the mark stack part of the most recently allocated data segment in the stack.

Then follows a series of expression evaluations to fill in the first part of the parameter section. After this a call-user-procedure instruction (CUP) or a call-standard-procedure instruction (CSP) is executed with appropriate parameters. The reserving of space for the second part of the parameter section as well as the local data is done by the ENT instruction, the first to be executed in the procedure body. The copying of large call-by-value parameters into the second part of the parameter section is done by instructions immediately after the ENT instruction (see Wirth 71c, 72 for information on the maintenance of the mark stack information).

It was mentioned earlier that the mark-stack information has space for preserving the value of functions, i.e. of type integer, real, character, boolean, any scalar or subrange, and pointer. Because in general values of the above types could require different amounts of storage units for their representations, the space reserved equals the largest number of storage units required (usually for type real). This is done purely for standardizing the mark-stack information and simplifying its maintenance. It follows that the return instruction for functions should take heed of the type of function being computed and thereby adjust the top of the stack, i.e. SP, accordingly.

The relational instructions, the load constant instruction and the return instruction are parameterized by type information as specified in the syntax in Appendix I. We trust that the reader will have no trouble in deciphering the correspondence between the characters and the types they signify.

PART B:Symbolic Instructions of PASCAL-CODE

Each instruction is packed into a 30-bit field. The op-code occupies a 6-bit field, parameter P a 4-bit field, and parameter Q a 20-bit (address) field. Sometimes, the Q field may be symbolic. This is indicated by an asterisk (*) in the description below.

Alphabetic List of Instructions:

code	mnemonic	parameters	description
40	ABI		absolute value of integer
41	ABR		absolute value of real number
28	ADI		integer addition
29	ADR		real addition
43	AND		Boolean "and"
26	CHK	Q	check against upper and lower bounds
15	CSP	Q	call standard procedure
12	CUP	P Q	call user procedure *
57	DEC	Q	decrement address
45	DIF		set difference
53	DVI		integer division
54	DVR		real division
13	ENT	Q	enter block *
27	EOF		test on end of file
17	EQU	P (Q)	compare on equal
24	FJP	Q	false jump*
34	FLO		float next to the top
33	FLT		float top of the stack
19	GEQ	P (Q)	greater or equal
20	GRT	P (Q)	greater than
10	INC	Q	increment address
9	IND	Q	indexed fetch
48	INN		test set membership (in)
46	INT		set intersection
44	IOR		Boolean "inclusive or"
16	IXA	Q	compute indexed address

code	mnemonic	parameters	description
5	LAO	Q	load base-level address
56	LCA	Q	load address of constant
4	LDA	P Q	load address
7	LDC	P Q	load constant
1	LDO	Q	load contents of base-level address
21	LEQ	P (Q)	less than or equal
22	LES	P (Q)	less than
0	LOD	P Q	load contents of address
49	MOD		modulus
55	MOV	Q	move
51	MPI		integer multiplication
52	MPR		real multiplication
11	MST	P	mark stack
18	NEQ	P (Q)	not equal
36	NGI		integer sign inversion
37	NGR		real sign inversion
42	NOT		Boolean "not"
50	ODD		test on odd
14	RET	P	return from block
30	SBI		integer subtraction
31	SBR		real subtraction
32	SGS		generate singleton set
38	SQI		square integer
39	SQR		square real
3	SRO	Q	store
6	STO		store at base-level address
58	STP		stop
2	STR	P Q	store at address
35	TRC		truncation
23	UJP	Q	unconditional jump*
47	UNI		set union
25	XJP	Q	indexed jump*
8		P Q	load constant indirect, an assembler-generated instruction

Type of the operands on the top of the stack for the instructions

(the first element corresponds to the top of the stack)

Before	after	the instruction
int	int	ABI, NGI, SQI
real	real	ABR, NGR, SQR
bool	bool	NOT
adr	adr	DEC, INC
int	real	FLT
int	bool	ODD
int	set	SGS
real	int	TRC
adr	bool	EOF
int int	↓ int	ADI, DVI, MOD, MPI, SBI
real real	↓ real	ADR, DVR, MPR, SBR
bool bool	↓ bool	AND, IOR
int adr	↓ adr	IXA
set set	↓ set	DIF, INT, UNI
↑	any	LOD
↑	adr	LAO, LCA, LDA
↑	int,bool,adr (depending on P parameter)	LDC, LDO
any	↓	SRO, STR
bool	↓	FJP
int,real,bool,set,adr int,real,bool,set,adr (depending on P parameter)	↓ bool	EQU, GEQ, GRT, LEQ, LES, NEQ
set int	↓ bool	INN
any int	any real	FLO
adr adr	↓ ↓	MOV

Before	after	the instruction
any adr	↓ ↓	STO
no action on top of stack		CHK, WP, STP, UJP
special		CSP, ENT, MST, RET
adr	any	IND

PART C: Mnemonics of Standard Procedures/Functions of the Hypothetical Stack Computer

The only argument of the CSP (Call Standard Procedure) instruction is of mnemonic representing a Standard Procedure/Function. The integer representation or code of this mnemonic is put into the address part of an instruction by the assembler.

Alphabetic List of Standard Procedures/Functions:

Mnemonic	Code	Description
ATN	19	Computes the arctan function for the value on the top of the stack and leaves the value of the result on the top of the stack.
COS	15	Computes the cosine function for the value on the top of the stack and leaves the value of the result on the top of the stack.
ELN	2	Checks the EOLN condition for the file specified on the top of the stack; the result of this check is left on the top of the stack.
EXP	16	Computes the function e^y where y is the value on the top of the stack; the result is left on the top of the stack.
GET	0	Performs get on the file specified by the top of the stack and appropriately fills the buffer associated with it.
LOG	17	The natural logarithm is computed for the value on the top of the stack; the computed value is left on the top of the stack.
NEW	4	The top of the stack specifies the size of element to be allocated from the free storage; the address of the element is to be stored in the pointer variable whose address is to be found below the top of the stack.

Mnemonic	Code	Description
PUT	1	Performs the put operation on the file specified by the top of the stack; the buffer is now initialized to 'undefined'.
RDC	13	Reads a character from the file specified on the top of the stack and assigns it to the variable whose address is below the top of the stack; note the automatic updating of the buffer associated with the file.
RDI	11	Reads an integer from the file specified on the top of the stack and assigns it to a variable whose address is below the top of the stack; note the automatic updating of the buffer associated with the file.
RDR	12	Reads a real number from the file specified on the top of the stack and assigns it to a variable whose address is below the top of the stack; note the automatic updating of the buffer associated with the file.
RLN	20	The top of the stack specifies a file on which a READLN is performed; note the automatic updating of the buffer associated with the file.
RST	2	Sets the 'new' pointer (heap pointer) to the pointer value on the top of the stack.
SAV	20	Saves the current value of the 'new' pointer (heap pointer) at the address specified on the top of the stack.
SIN	14	Computes the <u>sine</u> function for the value on the top of the stack; the result is left on the top of the stack.
SQT	18	The <u>square root</u> of the value on the top of the stack is computed and the result is left on the top of the stack.

Mnemonic	Code	Description
WLN	7	Performs WRITELN on the file specified on the top of the stack.
WRC	10	Writes on the file specified on the top of the stack a character whose ordinal value is found immediately below the element below the top of the stack. Just below the top is the number of characters to be written out.*
WRI	8	Writes on the file specified by the top of the stack an integer whose value is given immediately below the element below the top of the stack. Just below the top is the number of characters to be written out.*
WRR	9	Writes on the file specified by the top of the stack a real number whose value is given immediately below the element below the top of the stack. Just below the top is the number of characters to be written out.*
WRS	6	Writes on the file specified by the top of the stack a string of characters; immediately below the top of the stack is the value of the actual length of the string; below this is specified the actual length to be written out - if the actual length is less than the number of characters to be written out, then sufficient number of initial blank characters are written out; below the number of characters to be written out is an address where the actual string can be found.

As is standard with all stack machines, the use of operands by the above standard procedures/functions causes them to be removed from the top of the stack. In the use of functions, the result is pushed on to the stack.

* if necessary leading blanks are filled in.

Type of operands on the top of the stack for the standard procedures and functions

(the first element corresponds to the top of the stack)

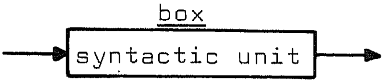
real	real	ATN, COS, EXP, LOG, SIN, SQT
adr	bool	ELN
adr	↓	GET, PUT, RLN, SAV, RST, WLN
adr	↓	RDC, RDI, RDR
adr	↓	
int	↓	NEW
adr	↓	
adr	↓	WRC, WRI
int	↓	
int	↓	
adr	↓	
int	↓	WRR
real	↓	
adr	↓	
int	↓	WRS
int	↓	
adr	↓	

APPENDIX III: Code Generation Pattern of the P-compiler

A compilers job is to produce a syntactic transformation under the constrain of semantics equivalence of source and object programs. This syntactic transformation is certainly a context sensitive one. In this sense, the ensuring description of code generation pattern of the P-compiler for the hypothetical stack computer is inadequate. However, as the reader is assumed to be familiar with PASCAL, it is hoped that he will fill in the context conditions under which the generation pattern is valid.

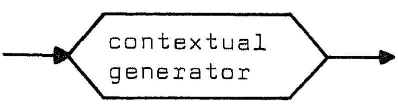
To simplify the description, syntax diagrams, like those used in the definition of PASCAL syntax, will be used here. The conventions followed with regard to the 'boxes' in the diagrams are:

<u>box</u>

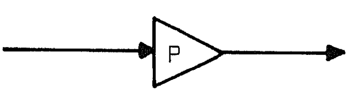


<u>meaning</u>

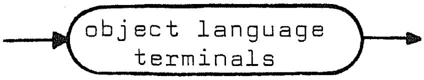
indicates the generation of code for the syntactic unit enclosed in the box at the point where such a box is encountered.



indicates the use of a specified contextual code generator by the compiler; e.g. LOAD, LOADADDRESS and STORE: the context is usually represented by compiler variables LATTR and GATTR which preserve local and global attributes of expressions, variables, etc.



indicates that the path will be followed only if the predicate p is true: these predicates, represented by letters in the diagrams, are informally explained in the footnotes that follow every diagram.

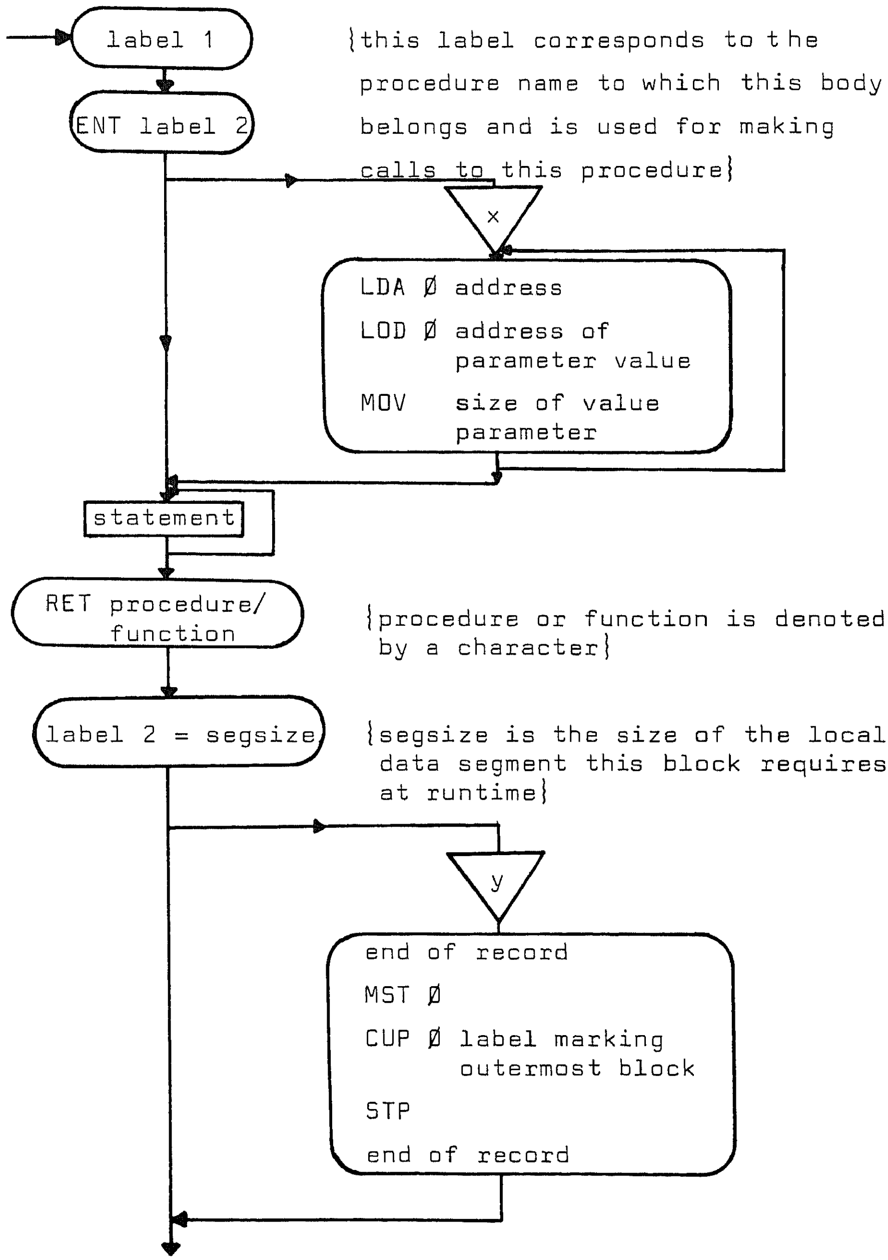


the enclosed information is immediately put out on the object code file 'LGO' and is considered to be a terminal symbol of the object language.

1) program

2) block

3) body

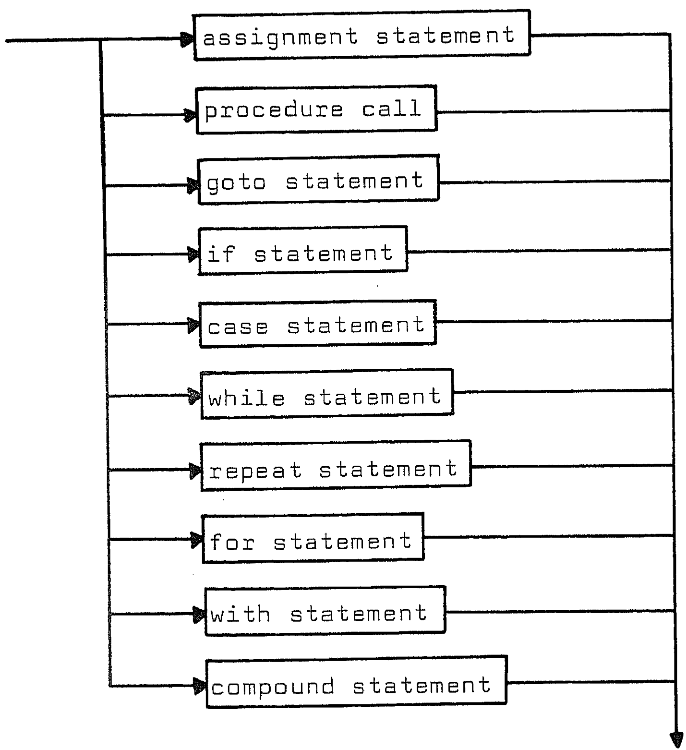


Predicates

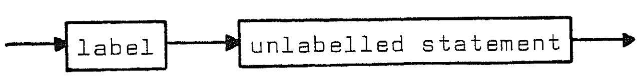
x: when the body being considered is that of a procedure/function and this procedure/function has value parameters which are of size greater than the PTRSIZE; this could happen for arrays/records passed as value parameters and also for some scalars (say reals and integers) and sets.

y: when the body being compiled is that of the main program (i.e., the outermost block); the code generated makes a call to this block and stops on return.

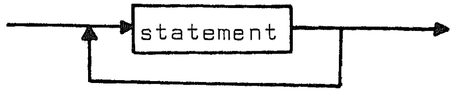
4) unlabelled statement



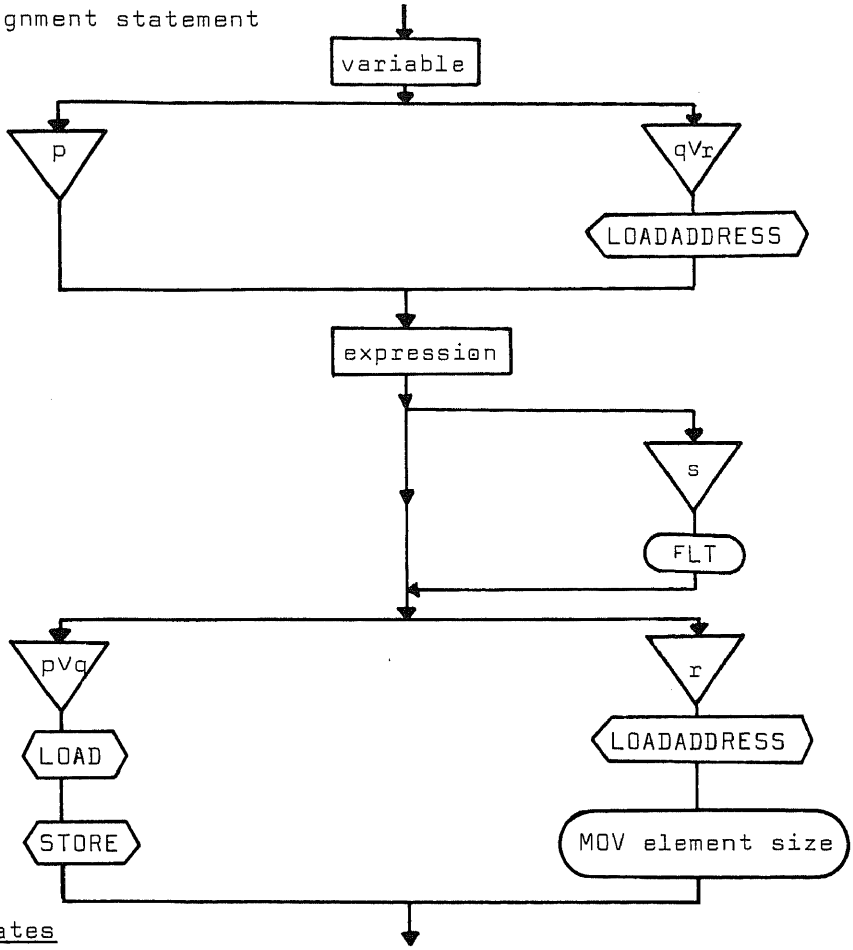
5) statement



6) compound statement



7) assignment statement



<u>Predicates</u>

p: where <variable> is of type scalar, subrange, pointer or set and is directly accessable.

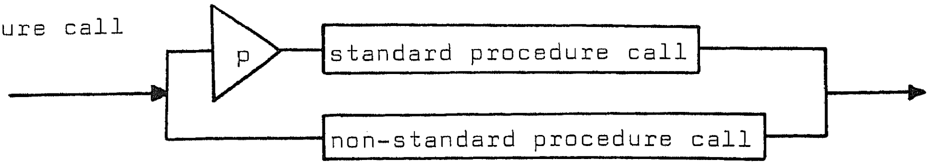
q: where <variable> is of type scalar, subrange, pointer or set and computation is required to access the demand, e.g., subscript evaluation, or indirection due to use of pointers.

r: where <variable> is of type record or array.

s: where<expression>is of type integer and <variable> is of type real.

Note: In case all paths are blocked because none of the correpsonding entry predicates is true, a definite contextual semantic error is detected.

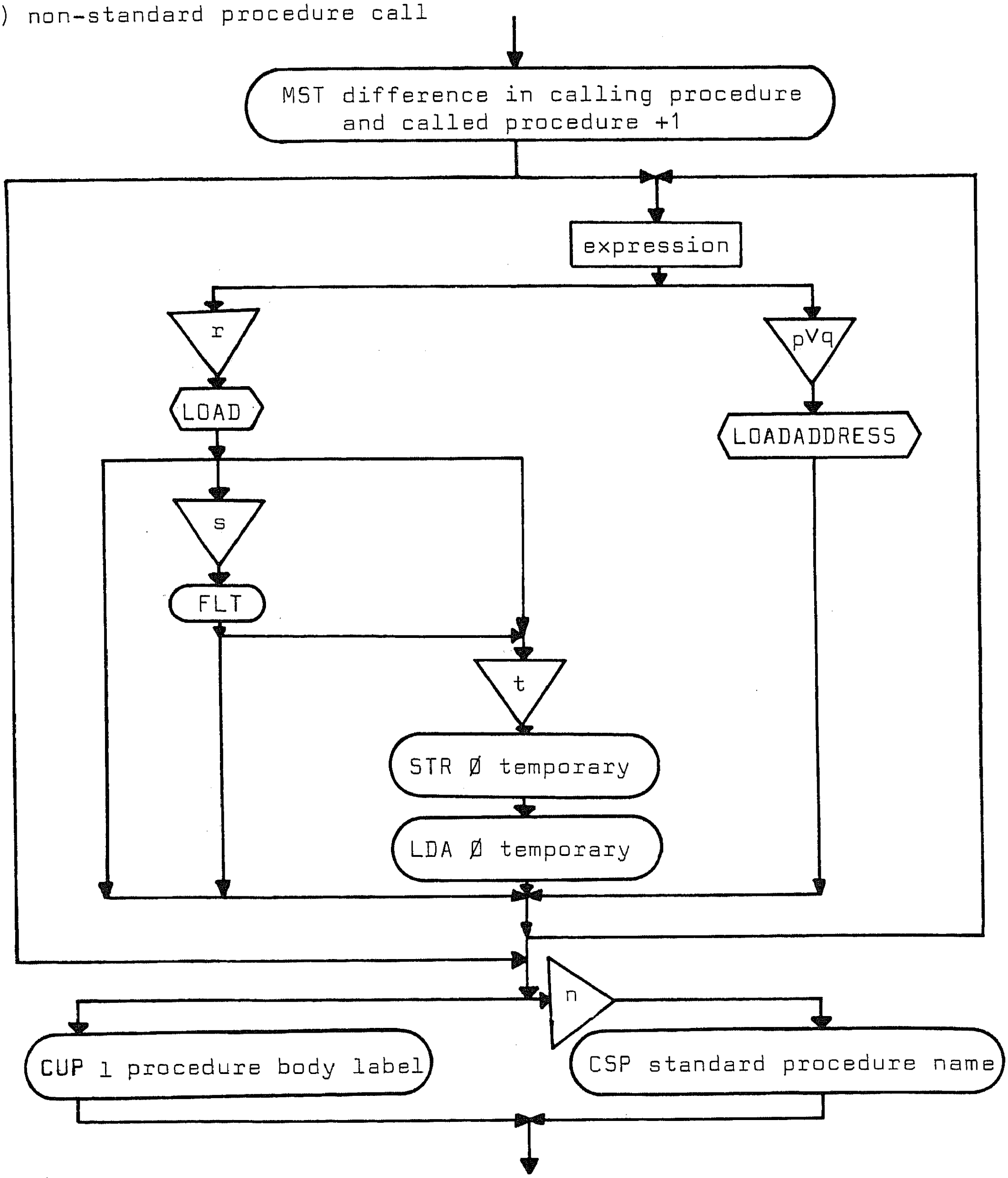
8) procedure call



Predicate

p: where the 'procedure identifier' is one of the standard names.

9) non-standard procedure call



{l is the number of locations units required by the parameters (machine dependent)}

Predicates

p: when the corresponding formal parameter is a var parameter and <expression> is a <variable>.

q: when the corresponding formal parameter is a value parameter and <expression> is a <variable> and size of variable >PTRSIZE and type of <variable> and the formal parameter are compatible.

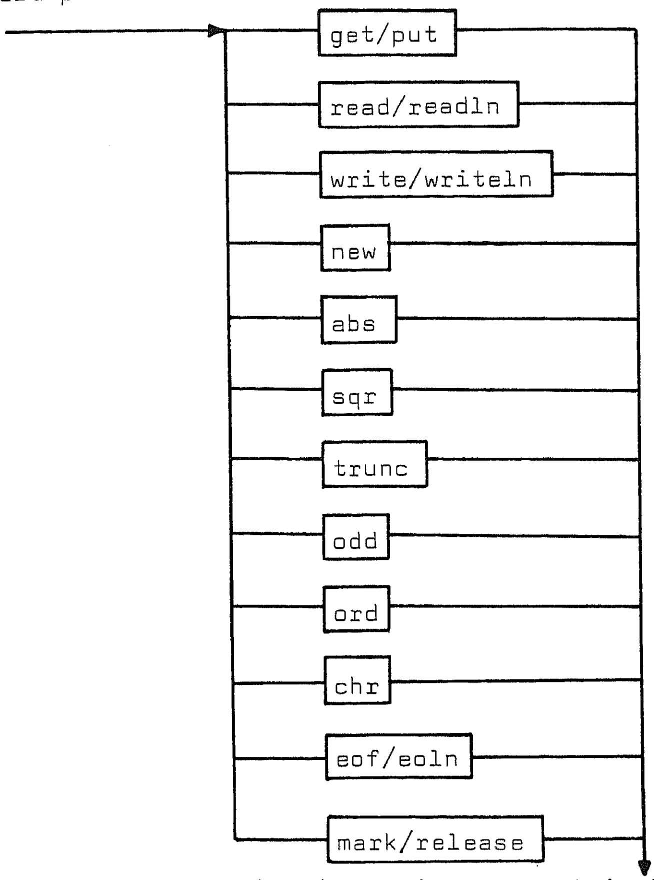
r: when the corresponding formal parameter is a value parameter and <expression> results in a computed value or a constant or if it is a <variable> then either size $\leq$PTRSIZE and its type is integer whereas the formal parameter is of type real or its size is $\leq$ PTRSIZE .

s: when <expression> results in an integer value and the corresponding formal parameter is of type real.

t: when size of formal parameter is >PTRSIZE and its value is available on the stack and should be passed; the value is stored in a temporary location in the current data segment (of the calling procedure) and the address of this temporary location is passed on; {note implicit declaration of pointer type variable}

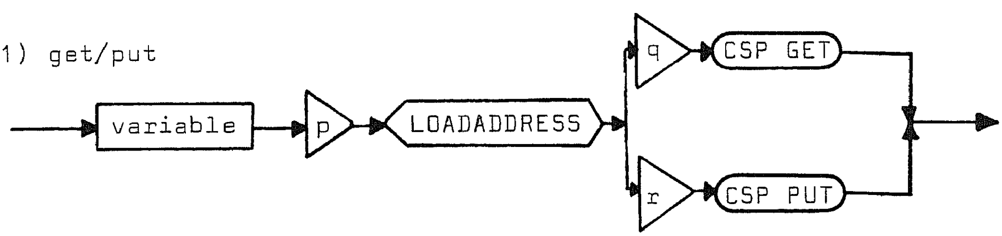
n: when the procedure is external and is one of SIN, COS, ATAN, LN, EXP and SQRT.

10) standard procedure call



{Those not mentioned in the above list are not implemented by the P-compiler}

11) get/put



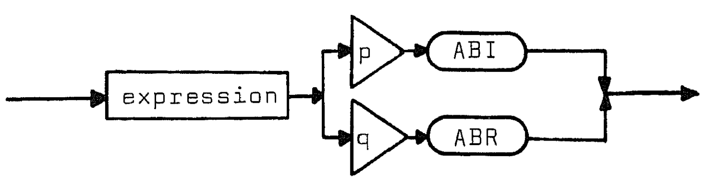
Predicates

p: where <variable> has the type attribute 'file'

q: The standard name used is GET.

r: The standard name used is PUT.

12) abs

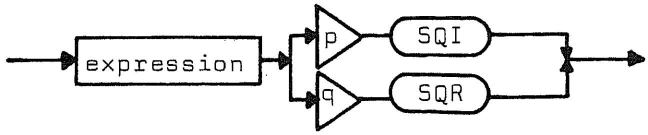


Predicates

p: <expression> is of type integer.

q: <expression> is of type real.

13) sqr

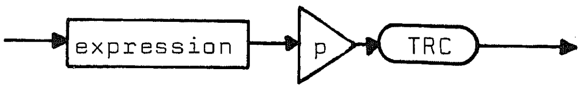


Predicates

p: <expression> is of type integer

q: <expression> is of type real

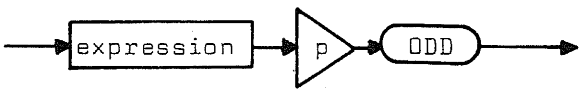
14) trunc



Predicate

p: <expression> is of type real.

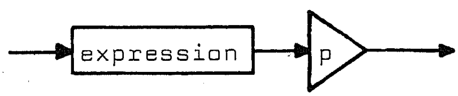
15) odd



Predicate

p: <expression> is of type integer

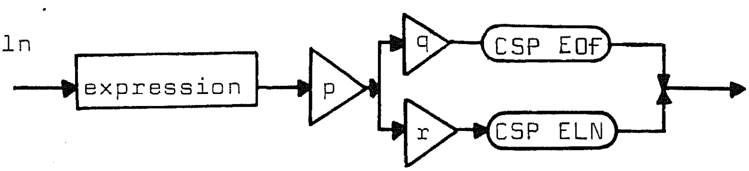
16) ord
chr



Predicate

p: <expression> is of type scalar or subrange for ord and is of type integer for chr {note: type pointer is non standard}

17) eof/eoln



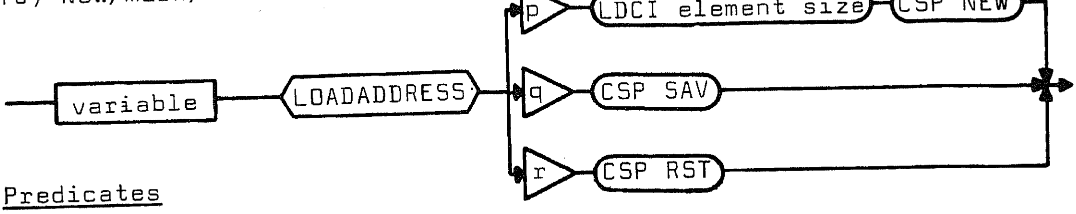
Predicates

p: <expression> is of type file

q: the standard name used is EOF

r: the standard name used is EOLN

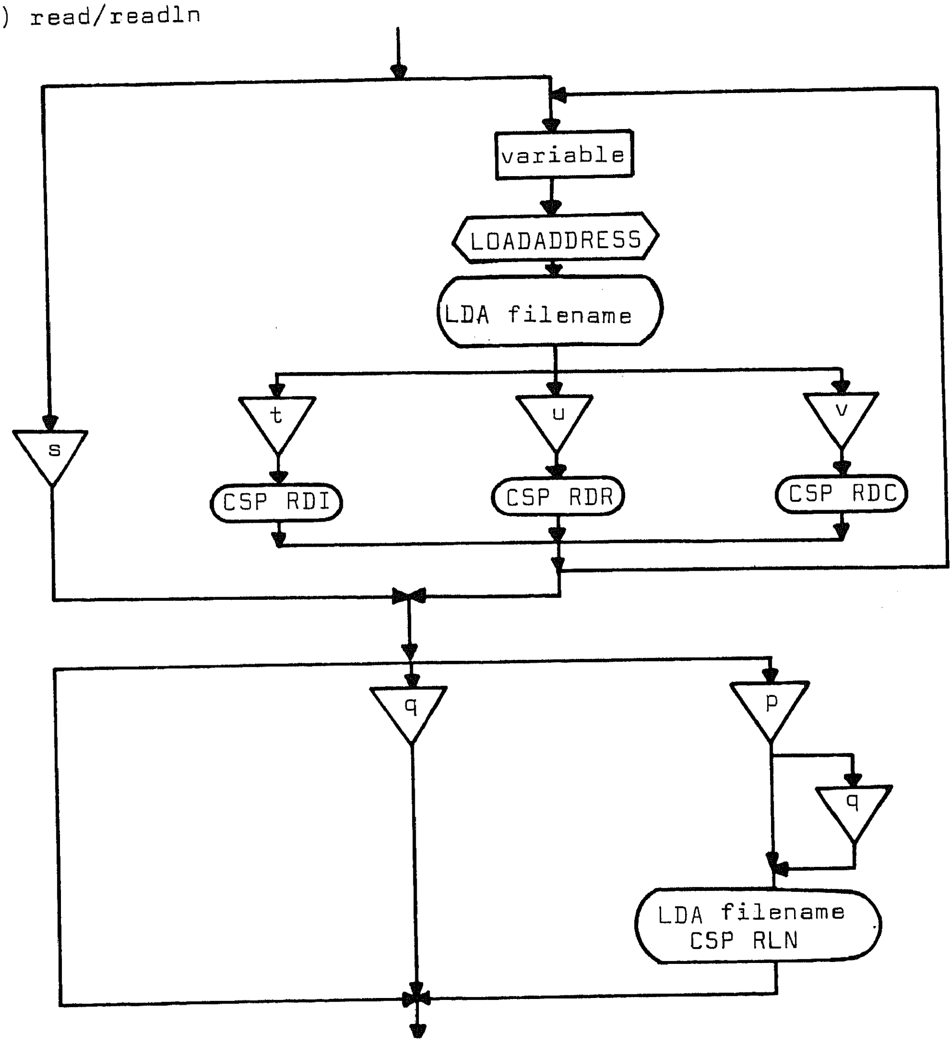
18) New/mark/release



Predicates

p,q,r the standard name used is NEW/MARK/RELEASE

19) read/readln



Predicates

p: the standard name used was READLN

q: the first parameter is a file name; {the other path makes the assumption that the file is INPUT}

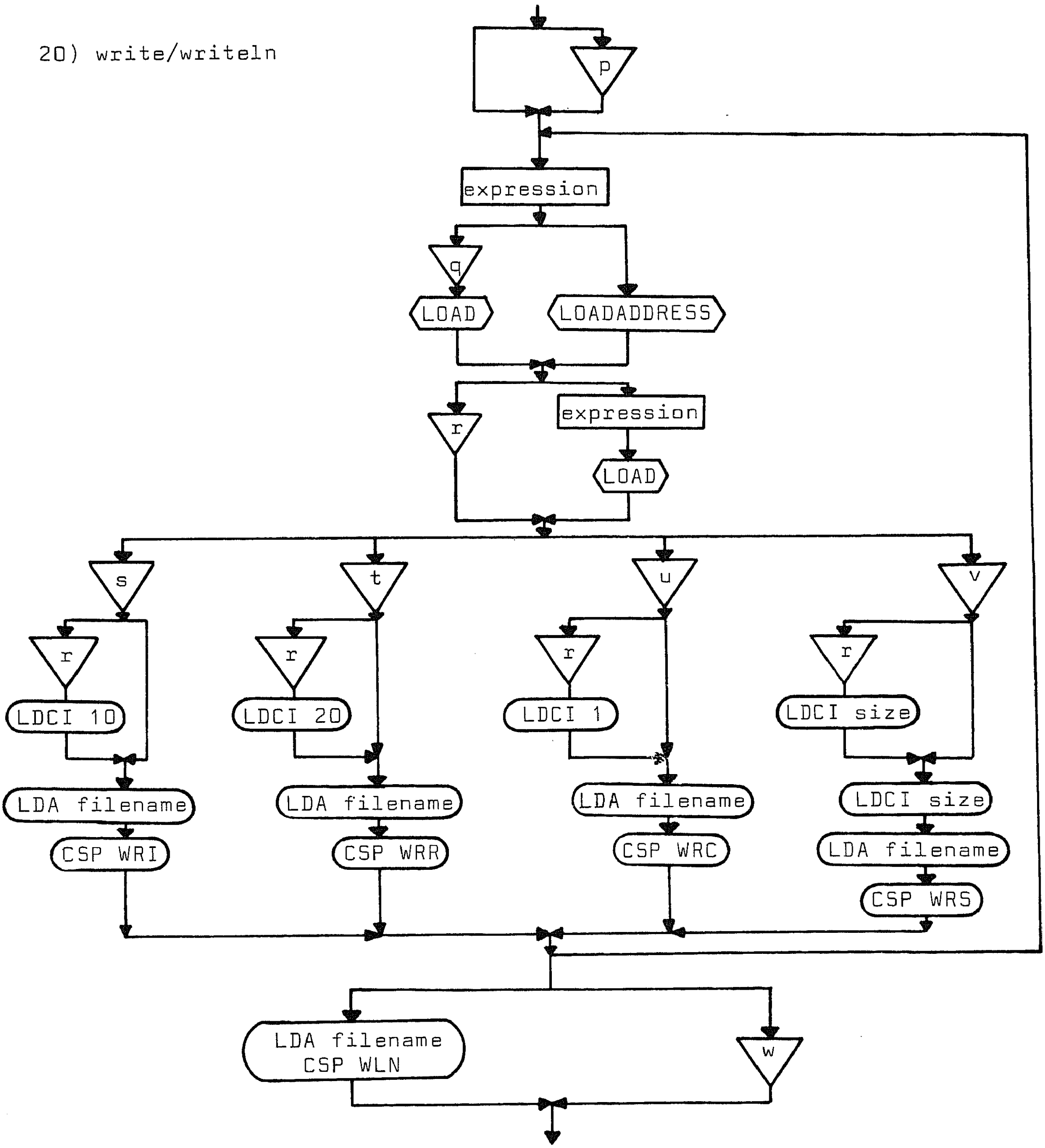
s: no further arguments?

t: <variable> is of type integer

u: <variable> is of type real

v: <variable> is of type character

20) write/writeln



Predicates

p: the first parameter is a file name; {the other path makes the assumption that the file is OUTPUT}

q: <expression> is of type integer, real or character

r: there is no field length specification

s: the parameter to be printed out is of type integer

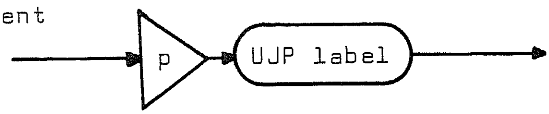
t: the parameter to be printed out is of type real

u: the parameter to be printed out is of type character

v: the parameter to be printed out is of type string

w: the standard name used was WRITE.

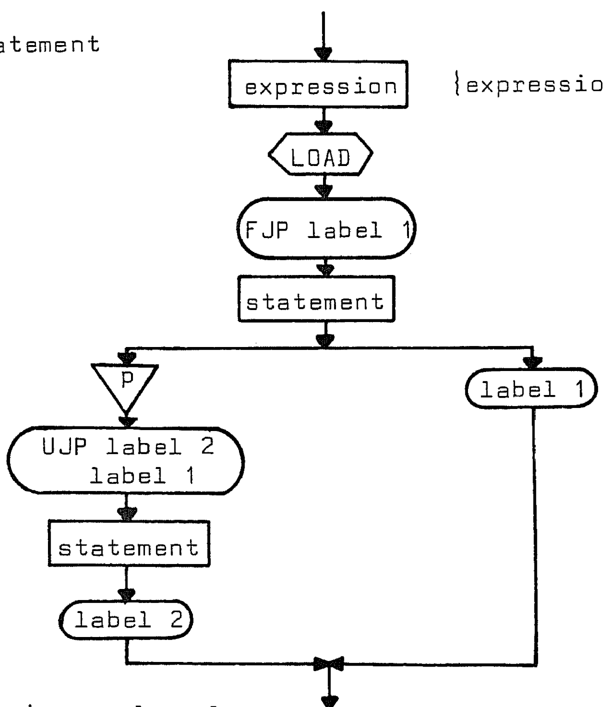
21) goto statement



Predicate

p: the integer following the reserved word goto has been declared in the label declarations of the block in which the goto occurs.

22) if statement

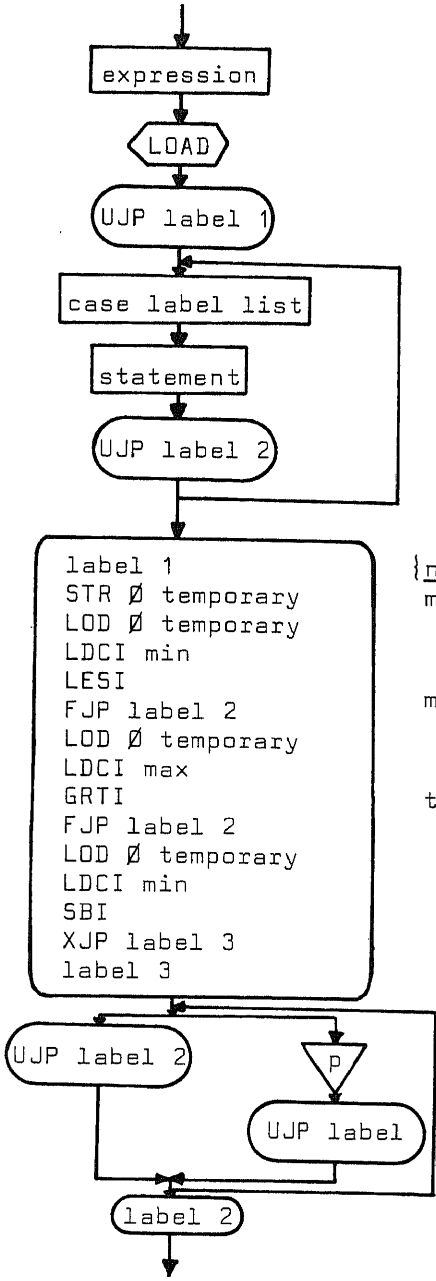


{expression is of type boolean}

Predicate

p: there is an else clause.

23) case statement



{notation

min: minimum value of case label found in the processing above.

max: maximum value of case label found in the processing above.

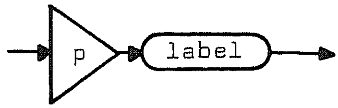
temporary: a location within the current data segment which preserves the value of the <expression>; this location is assumed to be implicitly declared and is of type integer}

{note that <u>label</u> is a context sensitive feature and it corresponds to each of the case labels found in the corresponding case label list}

<u>Predicate</u>

p: has the label {which is between min and max} occured in any of the <u>case</u> <u>label</u> <u>lists</u> processed in the case body?

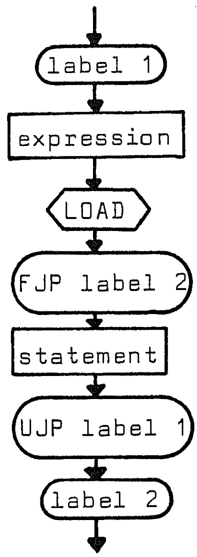
24) case label list



Predicate

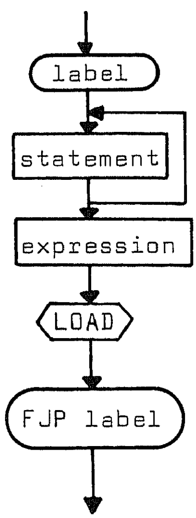
p: do types of all the case labels found in the list agree with the type of <expression> ?

25) while statement



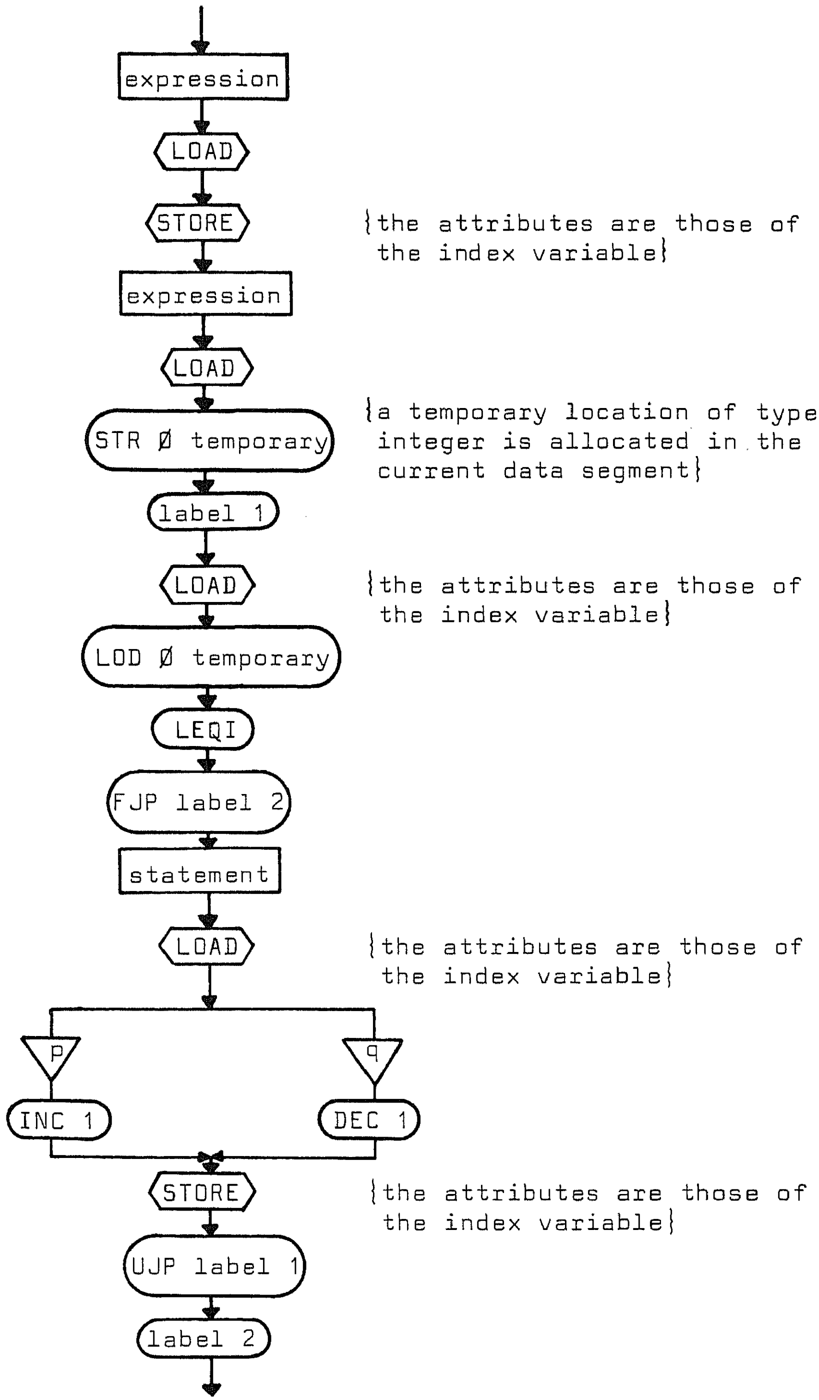
{<expression must be of type boolean}

26) repeat statement



{<expression> must be of type boolean}

27) for statement

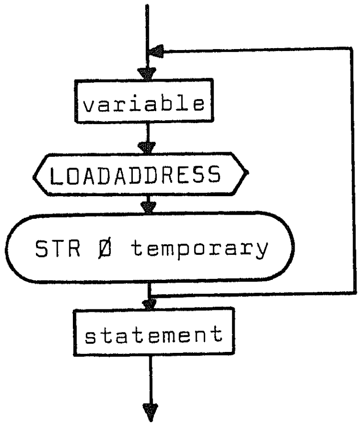


Predicates

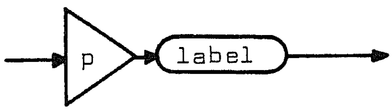
p: TO symbol was used in this for statement

q: DOWNTO symbol was used in this for statement.

28) with statement



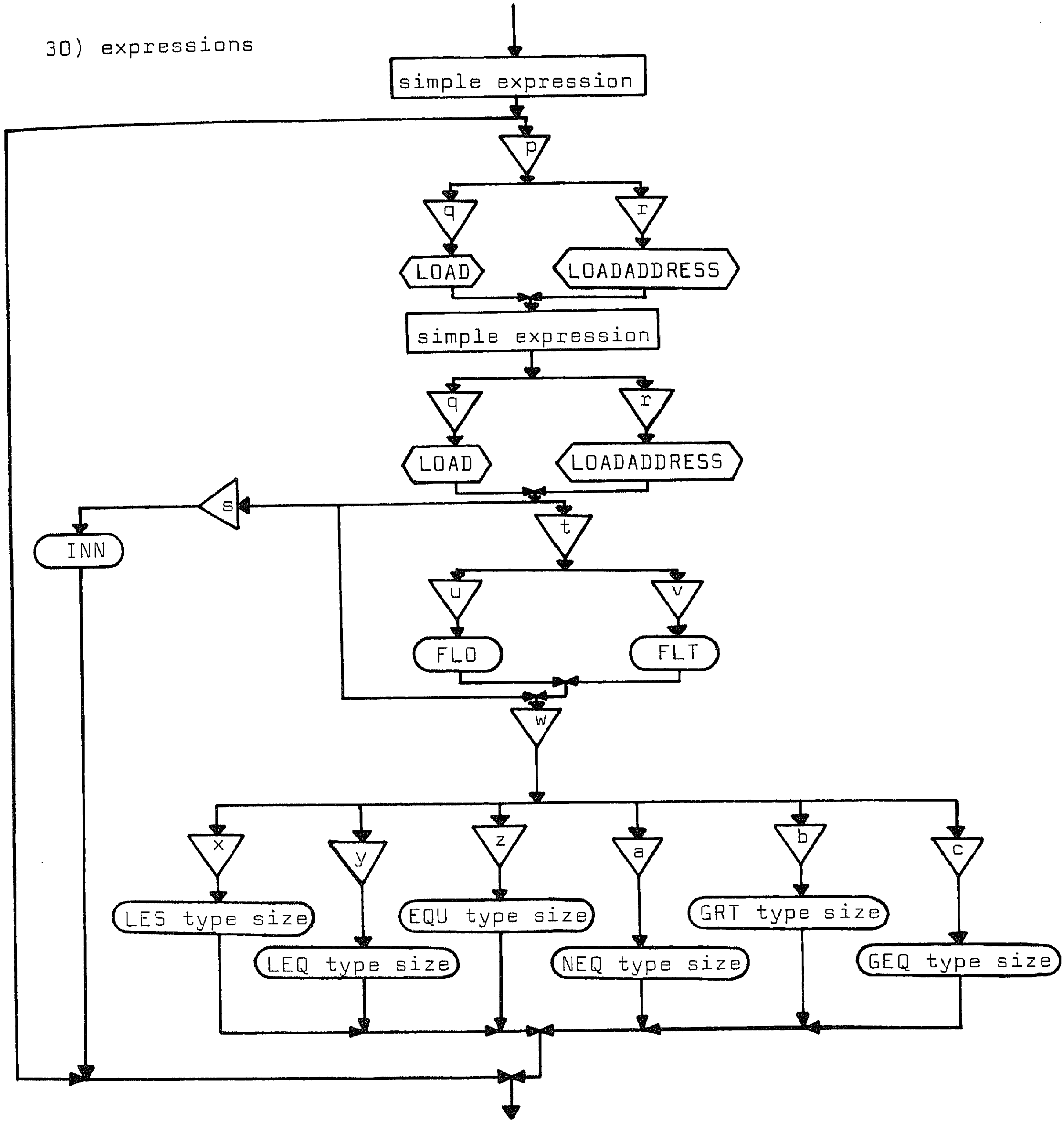
29) label



Predicate

p: the integer used as a label has been declared in the label declarations of the block in which this integer label occurs.

30) expressions



{<u>type</u> is one of I, R, A, B, M indicating the relational operation sought is for values of type integer, real, pointers (addresses), boolean and multiple (for records and arrays) respectively}

{<u>size</u> is meaningfull only in the case when type is M}

Predicates

p: a relational operation is used

q: the type of <simple expression> is scalar, pointer or set

r: the type of <simple expression> is arrays or records

s: the relational operator used was IN and the first <simple expression> yields a possible element of the type of set yielded by the second <simple expression>

t: the types of the two <simple expressions> are not the same

u: the first <simple expression> is of type integer whereas the second <simple expression> is of type real

v: the first <simple expression> is of type real whereas the second <simple expression> is of type integer

w: the types of the two <simple expressions> are compatible

x: the relational operator was '<'

y: the relational operator was '<='

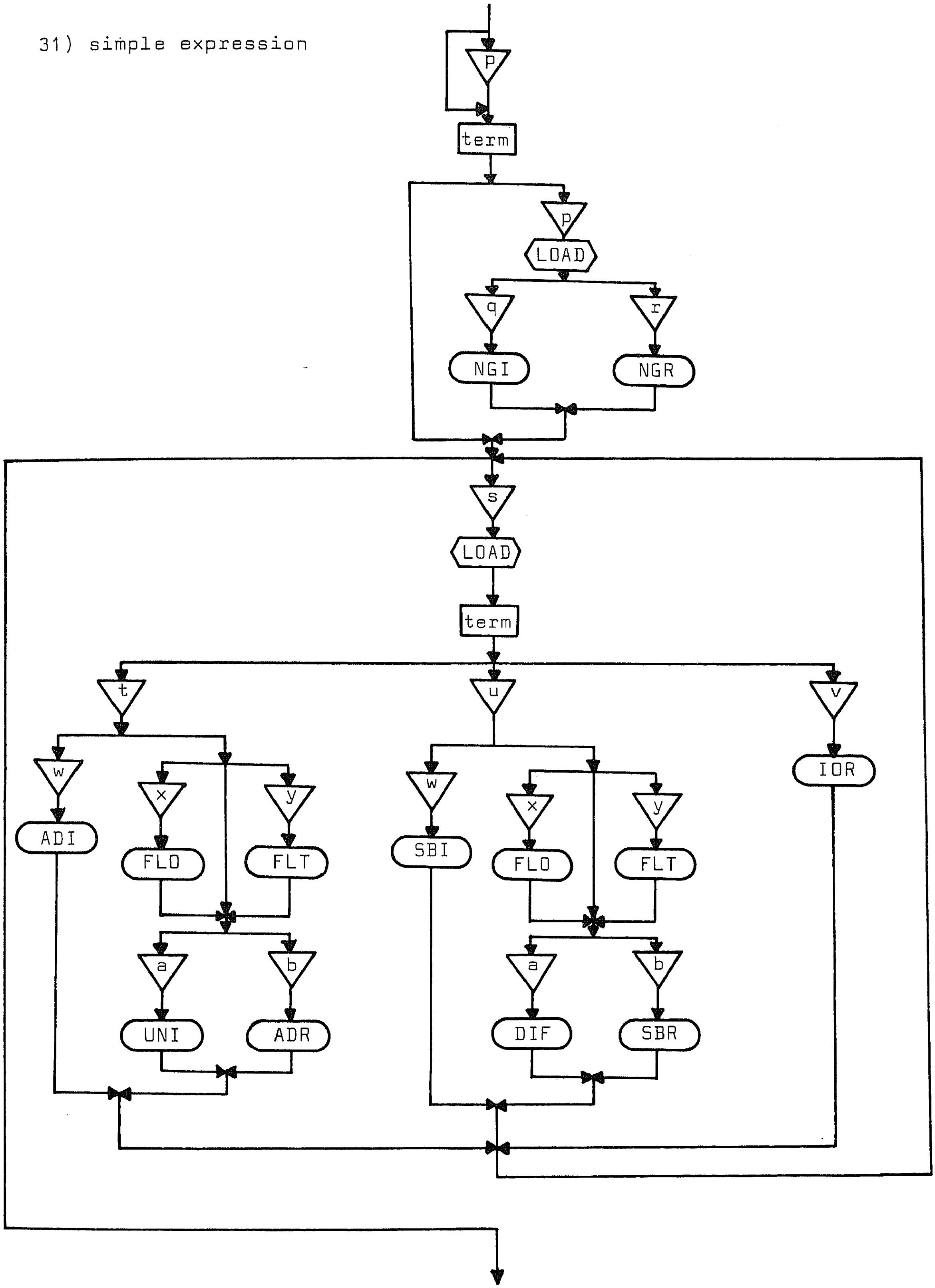
z: the relational operator was '='

a: the relational operator was '<>'

b: the relational operator was '>'

c: the relational operator was '>='

31) simple expression



Predicates

p: there is an unary minus

q: <term> is of type integer

r: <term> is of type real

s: there is an adding operator; i.e., one of '+', '-', 'V'

t: the operator is '+'

u: the operator is '-'

v: the operator is 'V' and both <term>'s are boolean

w: both <term> s are of type integer

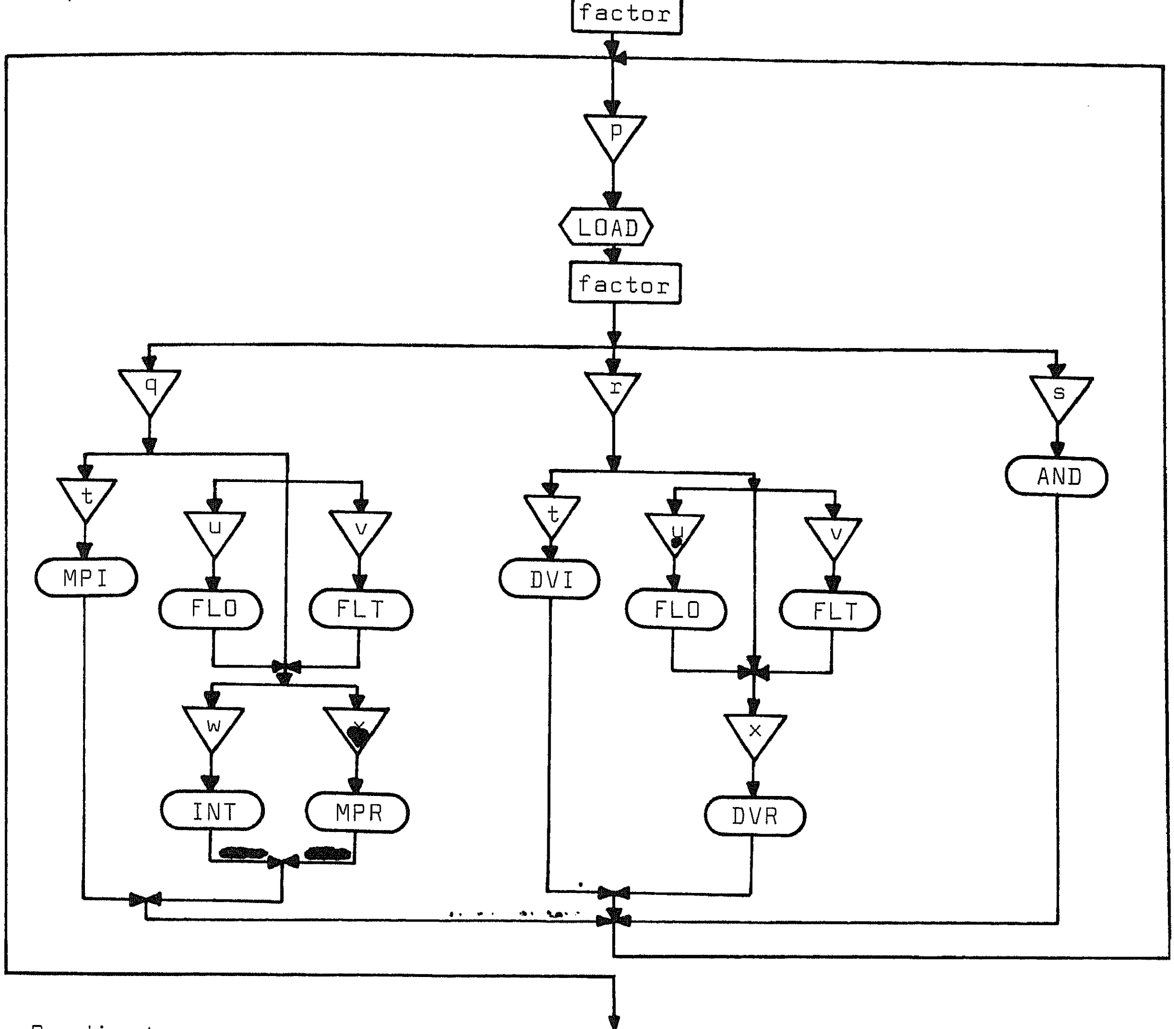
x: the first term is of type integer

y: the second term is of type integer

a: both <term> s are of type set

b: both <term> s are of type real

32) term



Predicates

p: there is a multiplying operator (i.e., one of '*', '/', '∧')

q: the operator is a '*'

r: the operator is a '/'

s: the operator is a '∧' and both the <factor>'s are boolean

t: both <factor> s are of type integer

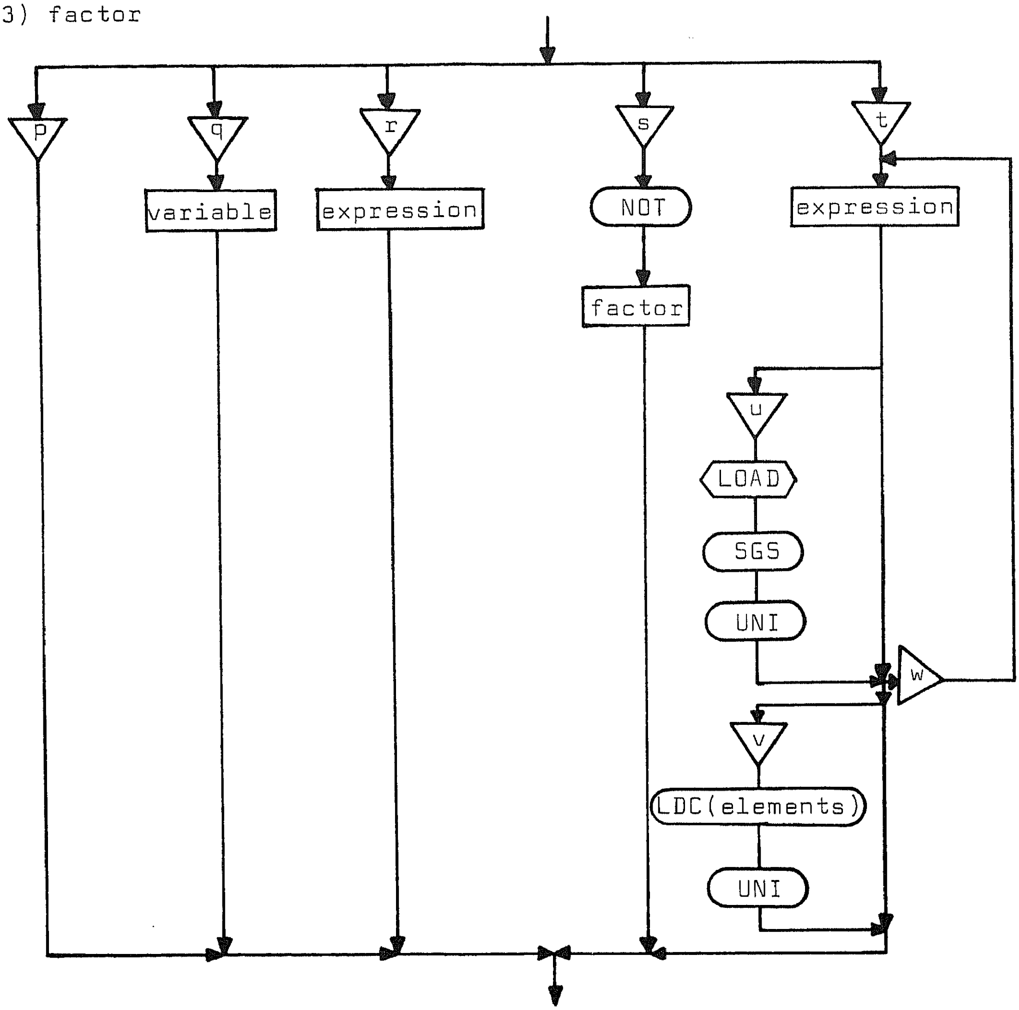
u: the first <factor> is of type integer

v: the second <factor> is of type integer

w: both <factor> s are of type set

x: both <factor> s are of type real.

33) factor



Predicates

p: the <factor> happens to be a constant (either an integer constant or a real constant or a string constant or an identifier which represents a constant

q: the <factor> is a <variable>

r: the <factor> is an <expression> enclosed in paranthesis

s: the <factor> is the logical inverse of the <factor> that follows (which must be of type boolean)

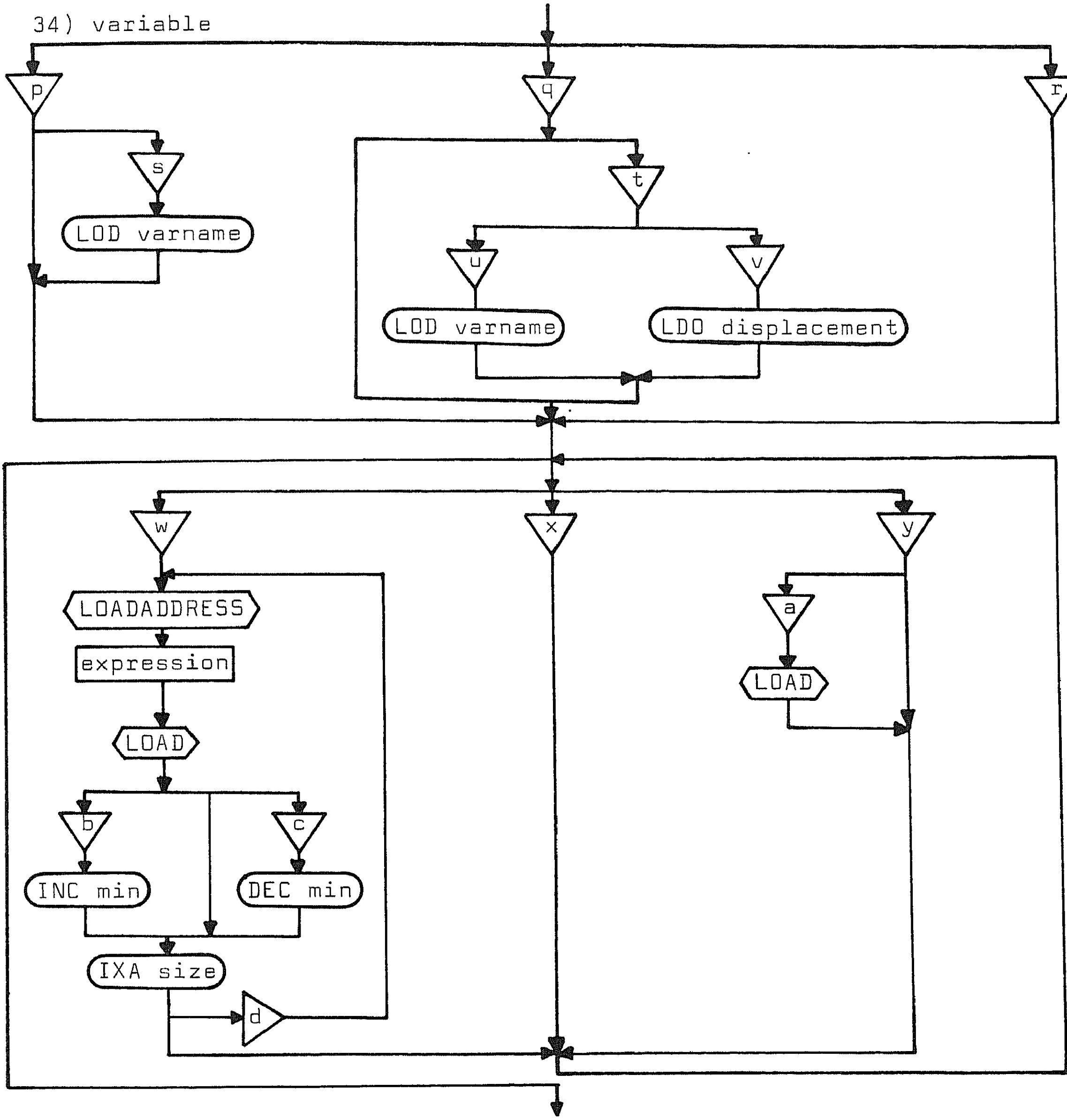
t: the <factor> is a set expression

u: the set expression consists of set variables or set expressions

v: a constant set has been found in the set expression

w: there are more elements of a set to be computed.

34) variable



{varname refers to the addressing of a variable by <level difference, displacement>;
min is the minimum value of the corresponding index;
size is the size of an element of the array}

Predicates

p: the variable starts with a variable name

q: the variable starts with field name

r: the variable is a function name (the function is a declared one and is not a formal function)

s: the variable is not a local variable

t: the field name is a field of an indirectly accessed variable (e.g., via pointers or subscripts or formal parameter etc.)

u: the indirectly accessed element has its base in the outermost block

v: the indirectly accessed element has its base in a block other than the outermost one

w: the variable is of type array and subscripts follow.

x: the variable is of type record and a field follows

y: the variable is of type pointer or file

a: the variable is of pointer type

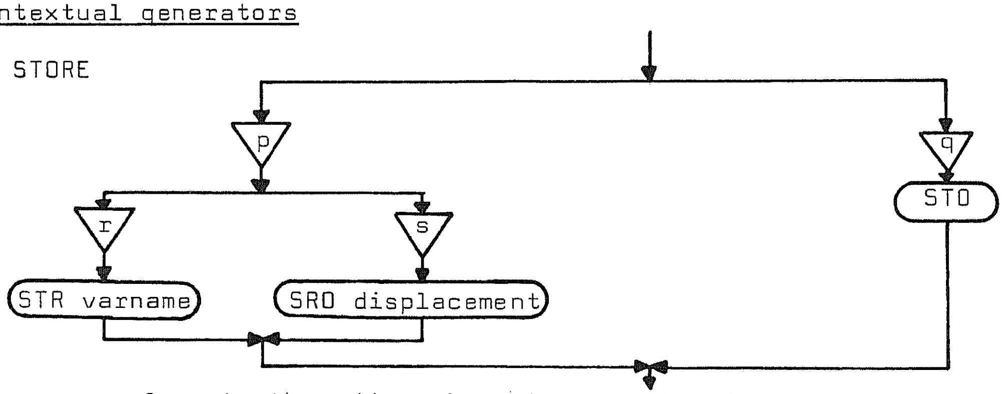
b: the minimum value of the corresponding index is greater than Ø

c: the minimum value of the corresponding index is less than Ø

d: more subscripts remain to be evaluated.

<u>Contextual generators</u>

1) STORE



{<u>varname</u> refers to the addressing of a variable by <level difference,displacement>}

<u>Predicates</u>

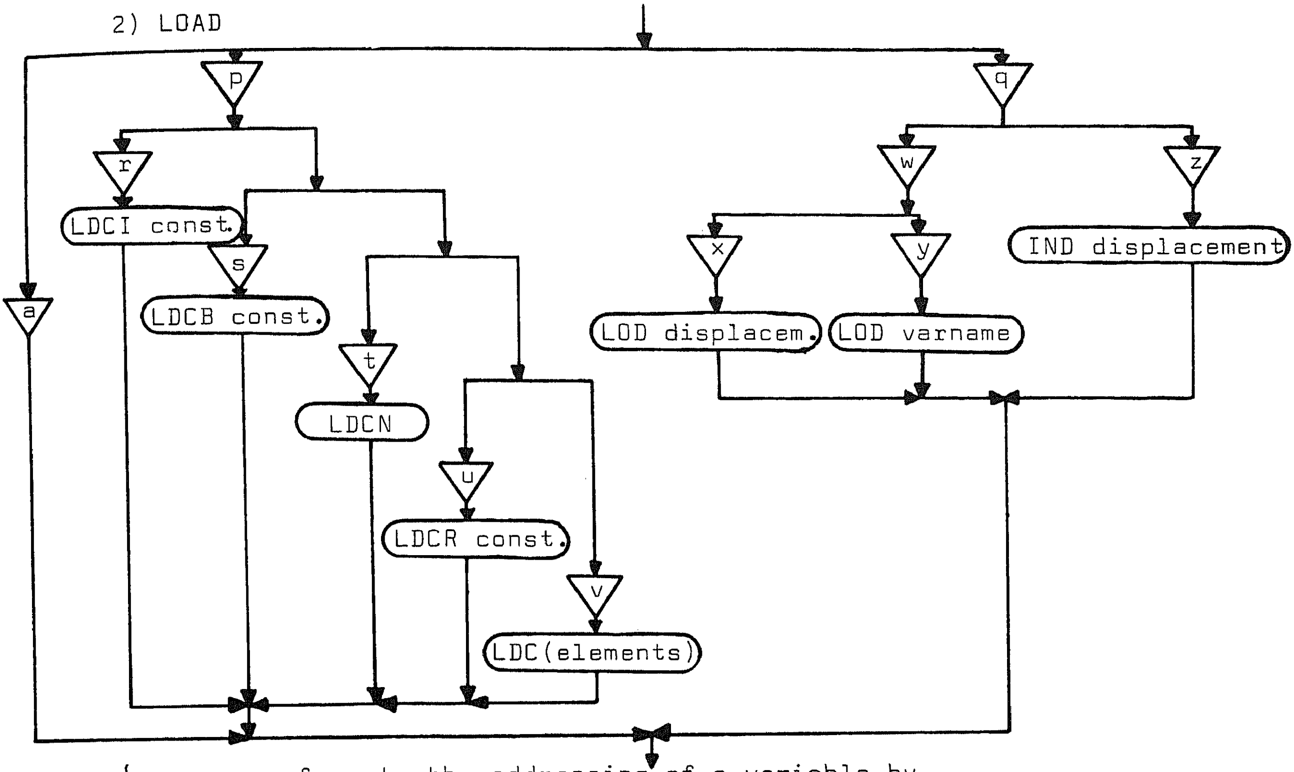
p: the variable is directly accessable

q: the variable is accessable only after having computed its address (so the address is already on the stack)

r: the variable is not in the outermost block

s: the variable is in the outermost block.

2) LOAD



{varname refers to the addressing of a variable by <level difference,displacement>}

Predicates

p: the element to be loaded is a constant

q: the element to be loaded is a variable

r: the constant to be loaded is an integer

s: the constant to be loaded is a boolean value

t: the constant to be loaded is NIL

u: the constant to be loaded is a real number

v: the constant to be loaded is a set

w: the variable to be loaded is directly accessable

x: the variable to be loaded is in the outermost block

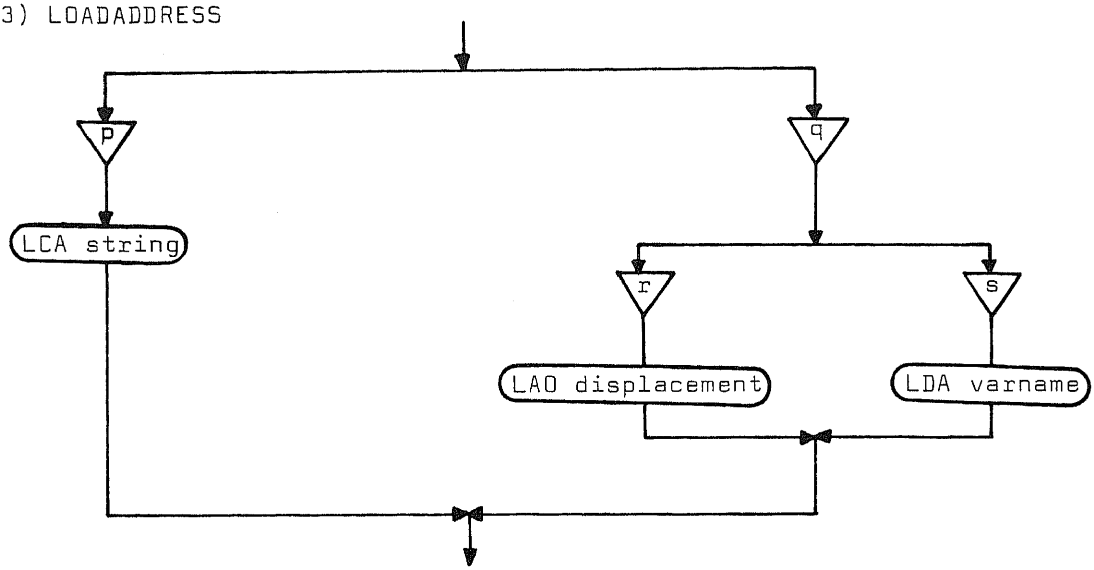
y: the variable to be loaded is not in the outermost block

z: the variable is indirectly accessable

a: the element is already on the stack

{note: loaded means loaded on the top of the stack}

3) LOADADDRESS



{varname refers to the addressing of a variable by <level difference,displacement>}

Predicates

p: the address to be loaded is that of a string constant

q: the address to be loaded is that of a variable

r: the variable is in the outermost block

s: the variable is not in the outermost block

APPENDIX IV: Comments on the Difference between the Language processed by the PASCAL 'P' Compiler and the 'Standard PASCAL'

1) Predefined files: the predefined standard text files (i.e. of type <u>file</u> <u>of</u> char) are:

   INPUT,PRD (input files)
   OUTPUT,PRR (output files)

   Every standard procedure/function involving files must specify the file concerned: the usual default interpretations

   $$\left.\begin{array}{l}\text{READLN} \equiv \text{READLN(INPUT)} \\ \text{EOLN} \equiv \text{EOLN(INPUT)}\end{array}\right\} \text{ do not hold.}$$

   etc.

2) The standard procedure DISPOSE is not part of the language processed by the 'P' compiler. It is replaced by MARK and RELEASE.

   MARK(P) where P is of any pointer type: marks the heap in the current state.

   The variable P should not be altered until the corresponding RELEASE.

   RELEASE(P) releases all items created by a NEW instruction since the corresponding MARK(P) .

   The use of MARK and RELEASE is much more suited to the bootstrap process than DISPOSE.

APPENDIX V: Description of the Tapes containing the PASCAL 'P' Code System

1) Format of the tape

No. of tracks : 7

Density : 800 bpi

Parity : odd

Physical record length : 5120 frames
the last physical record of a file may be shorter than 5120 frames.

Code:

first octal digit \ second octal digit	0	1	2	3	4	5	6	7
0		A	B	C	D	E	F	G
1	H	I	J	K	L	M	N	O
2	P	Q	R	S	T	U	V	W
3	X	Y	Z	Ø	1	2	3	4
4	5	6	7	8	9	+	-	*
5	/	(	)	$	=	␣	,	.
6	'	[	]	:				
7	↑		<	>				;

The <u>end of line</u> is represented as a series of <u>two</u> to <u>eleven</u> 00_8 frames.

The last eight frames of a files have no meaning (the last 8 frames of the trailing short record of a file).

Interrecord gap : 3/4"

End of file gap : 6"

End of information = 2 end of file gaps

2) Content of the tapes

1st file	:	interpreter (source)
2nd file	:	compiler (source)
3rd file	:	compiler (P code)
4th file	:	= 1st file
5th file	:	= 2nd file
6th file	:	= 3rd file

- interpreter: 9 physical records

  It starts with:

```
.
.(*ASSEMBLER AND INTERPRETER ...
.PROGRAM PCODE(...
```

  and ends with:

```
.
.1:
.END.
```

- compiler (source): 34 physical records

  It starts with:

```
.
.(*$T-,L-,C+*)
.(*****...
```

  and ends with:

```
.   PROGRAMME(BLOCKBEGSYS+...
.
.END.
```

- compiler (P code): 60 physical records

It starts with:

```
·|L    3
·| ENT          L    4
·| LDO               520
```

and ends with:

```
·|I    0
·| MST                    0
·| CUP     0    L    1519
·| STP
·|
```

Berichte des Instituts für Informatik

Nr. 1	Niklaus Wirth:	The Programming Language Pascal (out of print).
Nr. 2	Niklaus Wirth:	Program development by step-wise refinement (out of print).
Nr. 3	Peter Läuchli:	Reduktion elektrischer Netzwerke und Gauss'sche Elimination.
Nr. 4	Walter Gander, Andrea Mazzario:	Numerische Prozeduren I.
Nr. 5	Niklaus Wirth:	The Programming Language Pascal (Revised Report).
Nr. 6	C.A.R. Hoare, Niklaus Wirth:	An Axiomatic Definition of the Language Pascal (out of print).
Nr. 7	Andrea Mazzario, Luciano Molinari:	Numerische Prozeduren II.
Nr. 8	E. Engeler, E. Wiedmer, E. Zachos:	Ein Einblick in die Theorie der Berechnungen.
Nr. 9	Hans-Peter Frei:	Computer Aided Instruction: The Author Language and the System THALES.
Nr. 10	K.V. Nori, U. Ammann, K. Jensen, H.H. Nägeli:	The PASCAL 'P' Compiler: Implementation Notes.