Clif a C-like Interpreter Framework for Scientific Computing

T. Hrúz and Ľ. Koreň[1]

Abstract

This paper describes an open C-like interpreter Clif. The main goal of the development is to prepare a syntactic and semantic framework which can be easily enriched with application directed syntactic and semantic subsystems. The interpreter contains an implementation of basic atomary types. A further data abstraction mechanism allows a use of n-dimensional arrays of atomary type elements. The functional abstraction allows a use of recursive function calls. Parameters in function calls are passed by reference and the interpreter architecture is optimized for manipulation with large objects. The interpreter contains also well-defined slots for client-server paradigm as well as user interfaces on various levels.

1. Introduction

There are many situations when it is needed to prepare large programs for computer intensive information processing and for control of technological processes. Usually, to do this task efficiently the most appropriate way is to develop a language. The interpreter Clif is a convenient starting specification and a tool for these purposes.

The general concept which we follow (and is meant with the word *framework*) is to prepare in an open way syntactic and semantic structures with expressiveness similar as in the common programming languages like C, FORTRAN etc. Then user can add syntactic and semantic structures specific to a given problem. A powerful application directed language is easily obtained this way. From this point of view Clif could serve as an empty or skeletal language in the same sense as skeletal expert systems are build and subsequently filled with an application directed information.

Our intention is also driven by similar reasons as had motivated a development of MATLAB and similar environments i.e. to have a convenient and open tool for manipulation with objects in large and optimized scientific computing libraries as BIGNUM [12], LINPACK [6], etc.

In our development we stress user interfaces on all levels of the interpreter system. A user can write procedures within the Clif specification, he can build modules using different languages and link those modules into the Clif but even more he can add new language structures to Clif.

The next sections are organized as follows: in section 2 we describe main syntactic and semantic features of the Clif. In section 3 we follow some aspects of architecture and implementation of the Clif system. In section 4 we conclude with comments about further development of the interpreter. A formal definition of the Clif syntax is in appendix A as well as the syntax of graphics subsystem which is in appendix B. In appendix C the instruction set of the Clif virtual machine is described.

2. Main syntactic and semantic features of the Clif system

The syntax of Clif can be classified to two parts. The first part which represents most of Clif syntactic structures is a proper subset of C language. This subset contains all key constructions of C language except higher type abstractions. Clif currently does not contain structures, unions and pointers. Some other differences are caused by restrictions to the key language subsystems. For example, currently the specification of Clif contains the program flow control statements "for", "while" and "if" but does not contain the "switch" and "do while" specifications because they do not represent anything new in the language concept. The latter syntax can be easily added in the future. The C-compatible part of Clif contains one semantic difference. Parameters are passed to functions by reference because according to the main goal of Clif we suppose manipulations with relatively large objects (matrices, splines, etc.). Also the type checking is stronger in Clif than in C. We have no default types.

Another part of Clif is centered around the directions of future development of the language. This part is represented with "remote" class of commands. This is a slot to remote procedure calls

[1]Slovak Technical University, Faculty of Mechanical Engineering, Department of Automatic Control and Measurement, Námestie Slobody 17, 812 31 Bratislava, Slovak Republic, Phone: +42-7-497193, 493041/ext.497 E-mail: hruz@cvt.stuba.sk, korenl@cvt.stuba.sk

and client server paradigm. We have also in mind specific atomary types as are arbitrary precision integers, matrices and splines.

Clif is a multiplatform compiler based on various UNIX platforms.

The interpreter consists from two main parts: the compiler and the virtual stack machine. A program for Clif is a sequence of zero level syntactic structures. When a complete zero level syntactic structure is read it is compiled to the virtual machine code which is then executed. The debugging and the user control over a program execution is realized with the synchronous interrupt command "csuspend" and with an asynchronous interrupt handling. In both cases the interpreter reacts with opening of a new fully functional interpreter level. A user can finish the computation on the current level with a command "resume" which resumes the computation on the lower level.

Clif has three levels (interfaces) through which user can add new modules and enrich the environment. Firstly, user can write functions in Clif language which is the most natural and simple way. Secondly, more experienced users can write modules in FORTRAN or C language and link them to Clif environment. This sort of openness is realized via "remote" class syntax. The third and the most powerful way how to enrich Clif environment is to change the language itself. This is enabled by systematic use of compiler-compiler technology [7, 11] and careful structural development of Clif [9, 10]. For example, there is a well defined procedure how to add a new atomary type to the language.

The graphics output for applications in optimal control and identification is realized with external subsystem of graphics window channels. There is a special language for the channel specification, which is defined in appendix B.

3. Architecture and implementation of the Clif system

An important decision in the design of C-like *interpreter* is to establish syntactic and semantic blocks on which it is reasonable to suspend the parsing and switch to execution. These considerations have led us to the solution where the interpreter is clearly split to a virtual machine and to a language parsing and code generation engine.

The virtual machine is an abstract model of computer with a low level instruction set. The interpreter works in an alternating two-step mode, where the parser translates a level zero syntactic block and generates a code for the virtual machine. Subsequently the virtual machine executes the code. The code is always finished with the instruction STOP, which gives the control back to the parser.

In the following sections we describe in more detail some aspects of the virtual machine and parser design.

3.1. Virtual machine

During the virtual machine design process we have followed three main objectives:

1. To obtain a machine which naturally supports the C language.

2. To have a simple instruction set consisting of instructions with a medium complexity.

3. To minimize a copying and migration of objects in memory.

The first design objective has led us naturally to a certain sort of stack machine. The virtual machine has no general purpose registers. All operations occur on the top of the arithmetical stack. The fact that a stack machine supports C language was also described in the series of articles [3, 4, 5] where the same objective has led authors during design of a microprocessor chip. The result was a chip which considerably emphasizes stack operations for C language support.

The second design objective fits nicely into the framework of RISC microprocessor instruction sets; therefore we have inspired ourselves with such sets.

The third design objective stems from the intended applications of the interpreter. We expect to work with relatively large objects (with granularity of kilobytes) where any inefficiency caused by object copying can significantly reduce a performance of the interpreter.

To match the above design objectives we use the following structure of the virtual machine. There is a main memory with global variables and function bodies located from the top and the stack growing from the bottom (see figure 1); there is an arithmetic stack and temporary stack. To avoid copying of objects, the arithmetic stack contains addresses of objects located in the main memory, stack or temporary stack. Also the parameter passing mechanism for recursive function calls works strictly by reference. This architecture allows us to avoid at all copying of objects, except the copy operation explicitly demanded with an assignment operator.

In appendix C we briefly describe the RISC-like instruction set of the virtual machine.

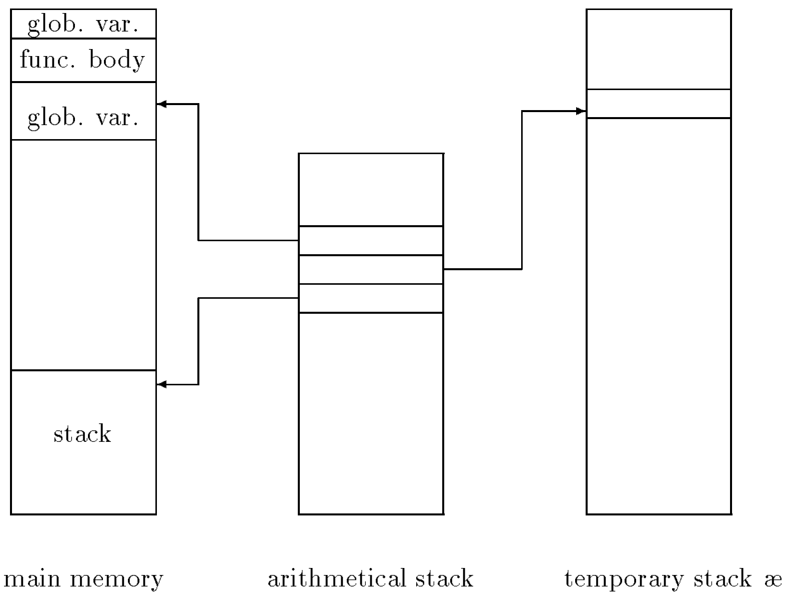

Figure 1:

The top level structure of the Clif virtual machine

3.2. Parser and code generation

The parser subsystem of the Clif interpreter is designed as a regular expression lexical analyzer realized with LEX [11] and LR(1) parser designed with YACC [2]. The BNF form of grammar (parser specification) is in appendix A. The parser is constructed as one-pass parser and code generator. For a parsing of various context dependent semantics of Clif as well as for an object movement optimization we use various sorts of backward and forward fixing techniques.

As we have mentioned at the beginning of this section there must be a decision about the syntactic resp. semantic blocks which trigger a run of the virtual machine and the parser. This design decision is realized with the level_0 and the level_1 grammatical construction (see appendix A). Shortly, we can say that any left bracket "{" opens a list of level_1 statements. They are compiled and the code is generated in the first free area in the main memory. The occurrence of the last right bracket "}" which brings the sentence to the level_0 stops the parsing and triggers a run of the virtual machine.

4. Conclusions

In the concluding section we comment some features of Clif and we discuss a possible future directions of the development.

- Because Clif is a multiplatform compiler we have a well-defined procedure of migration to different platforms. The interpreter is currently ported to the following operating systems and architectures, respectively: UNIX (CD4680 - EP/IX, DEC5000/240 - Ultrix, Sun SPARC Station - Solaris, Linux), MS DOS. We are working on the MS Windows port.
- The remote procedure call slot that we have built currently into the interpreter supports a remote procedure call paradigm, where client (or caller) waits until the finish of the remote procedure. We would like to improve this mechanism with a possibility of concurrent computation and synchronization via internal mechanism which would allow the interpreter to proceed with a computation even if the data from the remote call are not at the disposal. The interpreter runs until there is no reference to the results of the remote call and then blocks just before the first such reference.
- We suppose to implement the following special atomary types: integers with arbitrary precision, matrices with a dynamical allocation of memory and splines.
- To allow a more flexibility in language *syntax* we propose a procedure where for each stage of the compiler development there are two compilers. One is the active compiler with a full semantic and the other one is the same parser but with a special semantic which is directed to a source-source translation towards a new syntax.

- We also think about a possibility to conduct a research towards a compiler with a unified parser working in a heterogeneous environment with different code generation parts for particular environments. We have in mind to apply the methods used in object programming to solve "inheritance anomaly" [8] for this problem. When we look at the parser constructed with YACC it can be seen that parsing and code generation parts are interspersed in such a way that it is not possible to separate and encapsulate those parts from each other. This is an instance of inheritance anomaly which can be approached with recently discovered methods.

References

[1] A. V. Aho, S. C. Johnson: LR parsing, Computing Surveys, Vol. 6, No. 2, June 1974, pp. 99-124

[2] A. V. Aho, J. D. Ullman: The Theory of Parsing, Translation, and Compiling, Prentice Hall, Englewood Cliffs, N.J., Vol. 1, 1972, Vol. 2, 1973

[3] S. Bandyopadhyay, V.S. Begwani, R.B. Murray: Compiling for the CRISP processor, Proceedings of the IEEE Spring, COMPCON, 1987, pp. 96-100

[4] A.D. Berenbaum, D.R. Ditzel, H.R. McLellan: Introduction to the CRISP Instruction Set Architecture, Proceedings of the IEEE Spring, COMPCON, 1987, pp. 86-90

[5] A.D. Berenbaum, D.R. Ditzel, H.R. McLellan: Architectural Innovations in the CRISP Microprocessor, Proceedings of the IEEE Spring, COMPCON, 1987, pp. 91-95

[6] J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, LINPACK User's Guide, SIAM, Philadelphia, 1979

[7] S. C. Johnson: YACC: Yet Another Compiler Compiler, Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, NJ 07974, 1975, pp. 163-196

[8] D.G. Kafura, R.G. Lavender: Concurrent Object-Oriented Language and the Inheritance Anomaly, Proceedings ISIPCALA'93, Prague, 1993, pp. 183-215

[9] Ľ. Koreň: The Interpreter Clif, Programmer's Guide, Research Report, Slovak Technical University, Faculty of Mech. Eng., Bratislava, 1994

[10] Ľ. Koreň: The Interpreter Clif, Technical Guide, Research Report, Slovak Technical University, Faculty of Mech. Eng., Bratislava, 1994

[11] M. E. Lesk, E. Schmidt: LEX - A lexical analyzer generator, Research report, Bell Laboratories, Murray Hill, NJ, 1975, pp. 197-210

[12] B. Serpette, J. Vuillemin, J.C. Hervé: BigNum: A Portable and Efficient Package for Arbitrary-Precision Arithmetic, Research Report, preprint, INRIA, 1989

A. Syntax of Clif

```
<list_stat_0 >::= <list_stat_0 ><stat_0 >
            |empty


<list_stat >::= <list_stat ><stat_1 >
            |empty


<stat_0 >::= INT <list_dekl >;
            |DOUBLE <list_dekl >;
            |FLOAT <list_dekl >;
            |CHAR <list_dekl >;
            |<expr >;
            |IF ( <expr >) <then >
            |WHILE ( <expr >) { <list_stat >}
            |FOR for { <list_stat >}
            |READ ( <l_value >) ;
            |WRITE ( <l_value >) ;
            |PRINT ;
            |EXIT ;
            |INT PROC <proc >
            |DOUBLE PROC <proc >
            |FLOAT PROC <proc >
            |CHAR PROC <proc >
            |VOID PROC <proc >
            |EXPORT_T REMOTE_C INT PROC
<ident >( ) ;
            |EXPORT_T REMOTE_C DOUBLE PROC
<ident >( ) ;
            |EXPORT_T REMOTE_C FLOAT PROC
<ident >( ) ;
            |EXPORT_T REMOTE_C CHAR PROC
<ident >( ) ;
            |EXPORT_T REMOTE_C VOID PROC
<ident >( ) ;
            |REMOTE_C INT PROC <ident >( ) ;
            |REMOTE_C DOUBLE PROC <ident >( ) ;
            |REMOTE_C FLOAT PROC <ident >( ) ;
            |REMOTE_C CHAR PROC <ident >( ) ;
            |REMOTE_C VOID PROC <ident >( ) ;
            |CSUSPEND ;
            |RESUME ;
            |LOAD ( <file_name >);
            |;


<stat_1 >::= <expr >;
            |IF ( <expr >) <then >
            |WHILE ( <expr >) { <list_stat >}
            |FOR <for >{ <list_stat >}
            |BREAK ;
            |CONTINUE ;
            |READ ( <list_expr >) ;
            |WRITE ( <list_expr >) ;
            |PRINT ;
            |EXIT ;
            |RETURN ( <expr >);
            |CSUSPEND ;
            |LOAD ( file_name );
            |;


<list_dekl >::= <ident >
            |<ident >, <list_dekl >
            |<ident ><list_dim >, <list_dekl >


<list_dim >::= <list_dim >[ <numberi >]
            |[ <numberi >] <list_dim >
```

<then >::= { <list_stat ><or >

<or >::= } ELSE { <list_stat >}
|}

<for >::= (<expr >; <expr >; <expr >)
|(; ;)

<proc >::= <ident ><proc1 >

<proc1 >::= (<list_form_param >) <proc2 >
|( ) <proc2 >
|( ) ;

<proc2 >::= { <list_loc_dekl >; <proc3 >
|{ <proc3 >

<proc3 >::= <list_stat >}

<list_form_param >::= INT <ident >
|INT <ident >, <list_form_param >
|DOUBLE <ident >
|DOUBLE <ident >, <list_form_param >
|FLOAT <ident >
|FLOAT <ident >, <list_form_param >
|CHAR <ident >
|CHAR <ident >, <list_form_param >
|INT <ident ><list_loc_dim >
|INT <ident ><list_loc_dim >, <list_form_param >
|DOUBLE <ident ><list_loc_dim >
|DOUBLE <ident ><list_loc_dim >, <list_form_param >
|FLOAT <ident ><list_loc_dim >
|FLOAT <ident ><list_loc_dim >, <list_form_param >
|CHAR <ident ><list_loc_dim >
|CHAR <ident ><list_loc_dim >, <list_form_param >

<list_loc_dim >::= [ ]
|[<numberi >]
|<list_loc_dim >[<numberi >]

<list_loc_dekl >::= IDENT
|IDENT , <list_loc_dekl >

<dekl >::= INT <list_loc_dekl_1 >;
|DOUBLE <list_loc_dekl_1 >;
|FLOAT <list_loc_dekl_1 >;
|CHAR <list_loc_dekl_1 >;

<list_loc_dekl_1 >::= <ident >
|<ident >, <list_loc_dekl_1 >
|<ident ><list_dim >
|<ident ><list_dim >, <list_loc_dekl_1 >

<call >::= <ident >(<call1 >

<call1 >::= <list_param >)
|)

<list_param >::= <expr >)
|<expr >, <list_param >

<expr >::= <expr ><operator ><expr >
|<l_value >= <expr >
|− <expr >
|+ <expr >
|++ <l_value >
|−− <l_value >
|(INT) <expr >
|(DOUBLE) <expr >
|(FLOAT) <expr >
|(CHAR) <expr >
|<prim_expr >

<prim_expr >::= <identifier >
|<numberi >
|<numberd >
|<stringc >
|<numberc >
|(<expr >)
|<call >

<l_value >::= <identifier >

<identifier >::= <ident >
|<identifier >[<expr >]

<operator >::= any character from the set:
`| + - / \% < > & && == <= >= != *` << >> ^~ ! ||

<numberi >::= <number >
|numberi ><number >

<numberc >::= any single character

<numberd >::= <numberi >. <numberi >
|. <numberi >
|<numberi >.

<number >::= digit from the set: 0,1,2,3,4,5,6,7,8,9

<stringc >::= Sequence one or more characters, first character is a letter followed by letters or digits

<ident >::= Sequence one or more characters, first character is a letter followed by letters or digits

The statement LOAD(file_name); is only processed by lexical analyzer - yylex which opens file *file_name* and redirects input to the input from that file.

B. Syntax of the graphical subsystem language

<list_stat_0 >::= <list_stat_0 ><stat_0 >
|empty

```
<stat_0 >::= FIELDS = <numberi >
            |TYPE = <string >
            |PRINT_FORMAT = <string >
            |ON_LEAVE_WINDOW = <string >
            |DIRECTION = <string >
            |START_TIME = <s_time >
            |DURATION_TIME = <d_time >
            |W_RESOLUTION = <numberi >
<numberi >
            |LOWER ( <numberi >) = <numberd >
            |UPPER ( <numberi >) = <numberd >
            |STYLE ( <numberi >) = <numberi >

<d_time >::= <numberd >
            |AUTOMATIC

<s_time >::= <numberd >
            |AUTOMATIC

<numberi >::= <number >
            |numberi ><number >

<numberd >::= <numberi >. <numberi >
            |. <numberi >
            |<numberi >.

<number >::= digit from the set: 0,1,2,3,4,5,6,7,8,9
```

C. Instruction set of the Clif virtual machine

Notation:
ADR- address of the memory cell
ADR_STACK- the address stack register
AST- the arithmetic stack register
BP- the base pointer, it is used in relative address mode
TMP- the temporary stack register, it is used in addressing of temporary variables
$TMPH$- the temporary stack register, it is used in resetting of the temporary stack
NUM- offset in address or value
$STRING$- a string
$STACK$- the stack register
$FRAME$- the stack register used in parameter passing to the intrinsic functions
$[x]$- a value to which x points to
$<integer>$- an integer number
$<double>$- a double precision floating point number
$<float>$- a single precision floating point number
$<char>$- a byte

Instructions have a variable length. The structure of the instructions is the following: major, minor, immediately. Immediately can be either address or value. In the following table is the summary of instruction types.

type	parameters	size
OP_0_ma	major	1
OP_0_mi	major minor	2
OP_1_ma	major address	2
OP_1_mi	major minor address	3
OP_1_i	major minor value	3

C.1. Address instructions and instructions on the arithmetic stack

Instruction **MOV**.
Description: move data from the specified address to another specified address.
Options:
$[ADR] \longleftarrow [[AST]]$ type OP_1_mi
$[BP+NUM] \longleftarrow [[AST]]$ type OP_1_i
$[[BP+NUM]] \longleftarrow [[AST]]$ type OP_1_i
$[ADR+[[AST-1]]] \longleftarrow [[AST]]$ type OP_1_mi
$[BP+NUM+[[AST-1]]] \longleftarrow [[AST]]$ type OP_1_i
$[[BP+NUM+[[AST-1]]]] \longleftarrow [[AST]]$ type OP_1_i
The instructions are specific for each data type. We mean that the each instruction option represents a class of instructions. The instructions in each class differ by minor. For example the very first option is specific for type of operand double, float, integer, char.
$BP \longleftarrow STACK$ type OP_0_mi
$STACK \longleftarrow BP$ type OP_0_mi
$TMPH \longleftarrow TMP$ type OP_0_mi
$FRAME \longleftarrow STACK$ type OP_0_mi

Instruction **PUSHA**
Description: PUSH in to the arithmetic stack.
Options:
$[AST] \longleftarrow [ADR]$ type OP_1_mi
$[AST] \longleftarrow [BP+NUM]$ type OP_1_i
$[AST] \longleftarrow [[BP+NUM]]$ type OP_1_i
$[AST] \longleftarrow [[ADR+[[AST-1]]]]$ type OP_1_i
$[AST] \longleftarrow [BP+NUM+[[AST-1]]]$ type OP_1_i
$[AST] \longleftarrow [[BP+NUM+[[AST-1]]]]$ type OP_1_i
The instructions are specific for each data type.

Instruction **PUSHAI**.
Description: push onto the arithmetic stack immediately
Options:
$[[AST]] \longleftarrow NUM$ type OP_1_i
The instruction is specific for each data type.
$[[AST]] \longleftarrow STRING$ type OP_1_mi

Instruction **POPA**.
Description: POP from the arithmetic stack.
Options:
The arithmetic stack is cleared. type OP_1_mi

C.1.1 Arithmetic-logical instructions:

Address of the result is placed on the top of the arithmetic stack. Evaluation is placed into the temporary stack. Arithmetic-logical instructions are specific for each data type, if it is not stated otherwise in description of an instruction.
Instruction **ADD**.
Description: perform arithmetic addition.
Options:
$[[AST-1]] \longleftarrow [[AST]]+[[AST-1]]$ type OP_0_mi

Instruction **SUB**.
Description: perform arithmetic subtraction on the top of arithmetic stack or from stack pointer. On the stack can be only processed an instruction mentioned below (from stack pointer can be only subtracted an integer number).
Options:
$[[AST-1]] \longleftarrow [[AST-1]]-[[AST]]$ type OP_0_mi
$STACK \longleftarrow (STACK-NUM)$ type OP_1_i

Instruction **MULT**.
Description: perform arithmetic multiplication.
Options:
$[[AST-1]] \longleftarrow [[AST]]*[[AST-1]]$ type OP_0_mi

Instruction **MOD**.
Description: perform arithmetic modulo operation on integers.
Options:

$[[AST-1]] \longleftarrow [[AST-1]]\%[[AST]]$ type OP_0_ma

Instruction **DIV**.
Description: perform arithmetic division.
Options:
$[[AST-1]] \longleftarrow [[AST-1]]/[[AST]]$ type OP_0_mi

Instruction **OR**.
Description: perform logical inclusive OR.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] \; || \; [[AST]]$ type OP_0_mi

Instruction **AND**.
Description: perform logical AND.
Options:
$[[AST-1]] \longleftarrow [[AST-1]]\&\&[[AST]]$ type OP_0_mi

Instruction **ORB**.
Description: perform bitwise OR of integers.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] \; | \; [[AST]]$ type OP_0_ma

Instruction **ANDB**.
Description: perform bitwise AND of integers.
Options:
$[[AST-1]] \longleftarrow [[AST-1]]\&[[AST]]$ type OP_0_ma

Instruction **EQ**.
Description: perform logical test for equality.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] == [[AST]]$ type OP_0_mi

Instruction **GR**.
Description: perform logical test for greater than.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] > [[AST]]$ type OP_0_mi

Instruction **LO**.
Description: perform logical test for lower than.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] < [[AST]]$ type OP_0_mi

Instruction **LE**.
Description: perform logical test for lower or equal.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] <= [[AST]]$ type OP_0_mi

Instruction **GE**.
Description: perform logical test for greater or equal.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] >= [[AST]]$ type OP_0_mi

Instruction **NE**.
Description: perform logical test for non equal.
Options:
$[[AST-1]] \longleftarrow [[AST-1]]! = [[AST]]$ type OP_0_mi

Instruction **NEG**.
Description: perform logical negation.
Options:
$[[AST]] \longleftarrow ! \; [[AST]]$ type OP_0_mi

Instruction **NOT**.
Description: perform one's complement operation of integers.
Options:
$[[AST]] \longleftarrow \tilde{} \; [[AST]]$ OP_0_ma

Instruction **SAL**.
Description: perform arithmetic left shift of integers.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] << [[AST]]$ OP_0_ma

Instruction **SAR**.
Description: perform arithmetic right shift of integers.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] >> [[AST]]$ OP_0_ma

Instruction **XOR**.
Description: perform logical exclusive OR of two integers.
Options:
$[[AST-1]] \longleftarrow [[AST-1]] \; \hat{} \; [[AST]]$ OP_0_ma

C.1.2 Integer and floating point instructions:

Instruction **CVT**.
Description: Convert a signed quantity to a different signed data type.
Options:
$<integer> \longrightarrow <double>$
$<double> \longrightarrow <integer>$
$<integer> \longrightarrow <float>$
$<float> \longrightarrow <integer>$
$<float> \longrightarrow <double>$
$<double> \longrightarrow <float>$
$<char> \longrightarrow <integer>$
$<integer> \longrightarrow <char>$
$<double> \longrightarrow <char>$
$<char> \longrightarrow <double>$
$<char> \longrightarrow <float>$
$<float> \longrightarrow <char>$
$<void> \longrightarrow <integer>$
$<void> \longrightarrow <double>$
$<void> \longrightarrow <float>$
$<void> \longrightarrow <char>$
The conversion to the wider type (more bits) can be executed either on the top of the arithmetical stack or one operand under the top of the arithmetical stack. Above instructions are of the type OP_0_mi.

C.2. Stack instructions

Instruction **PUSH**.
Description: Push value onto the stack.
Options:
$[STACK] \longleftarrow BP$ type OP_0_mi

$[STACK] \longleftarrow TMPH$ type OP_0_mi

Instruction **POP**.
Description: Pop value from the top of the stack.
Options:
$BP \longleftarrow [STACK]$ type OP_0_mi

$TMPH \longleftarrow [STACK]$ type OP_0_mi

C.3. Address stack instruction

Instruction **PUSHAD**.
Description: PUSH address into the address stack.
Options:
$[ADR\_STACK] \longleftarrow [AST]$ type OP_0_ma

Instruction **POPAD**.
Description: POP address from the address stack.
Options:
$[STACK] \longleftarrow [ADR\_STACK]$ type OP_0_ma

C.4. Temporary stack instructions

Instruction **CLRT**.
Description: Clear temporary stack.
Options:
$TMP \longleftarrow TMPH$ type OP_0_ma

C.5. Input and output instructions

Instruction **IN**.
Description: input of the value into address; the address is specified absolutely or relatively.
Options:
$IN[ADR]$ type OP_1_mi
$IN[BP+NUM]$ type OP_1_i
$IN[[BP+NUM]]$ type OP_1_i
$IN[ADR+[[AST-1]]]$ type OP_1_mi
$IN[BP+NUM+[[AST-1]]]$ type OP_1_i
$IN[[BP+NUM+[[AST-1]]]]$ type OP_1_i

Instruction **OUT**.
Description: output of the content of the address; the address is specified absolutely or relatively.
Options:
$OUT[ADR]$ type OP_1_mi
$OUT[BP+NUM]$ type OP_1_i
$OUT[[BP+NUM]]$ type OP_1_i
$OUT[ADR+[[AST-1]]]$ type OP_1_mi
$OUT[BP+NUM+[[AST-1]]]$ type OP_1_i
$OUT[[BP+NUM+[[AST-1]]]]$ type OP_1_i

Instruction **MESS**.
Description: put string message to the standard output.
Options: type OP_1_ma

C.6. Control instructions

Instruction **STOP**.
Description: signalization of end of the virtual machine run.
Options: type OP_0_ma

Instruction **INTER**.
Description: indicating of synchronous interrupt.
Options: type OP_0_ma

Instruction **IRET**.
Description: return from synchronous or asynchronous interrupt.
Options: type OP_0_ma

Instruction **JMP**.
Description: jump to the address.
Options: type OP_1_ma

Instruction **JZ**.
Description: If last operation is equal zero, jump to the address.
Options: type OP_1_ma

Instruction **JNZ**.
Description: If last operation is not equal zero, jump to the address.
Options: type OP_1_ma

Instruction **HALT**.
Description: system halt.
Options: type OP_0_ma

Instruction **CALL**.
Description: call of a function. The function can be either user supplied one or intrinsic one.
Options: type OP_1_ma

Instruction **RET**.
Description: return from a function.
Options: type OP_0_ma