The Interpreter Clif
Programmer's Guide

Ľ. Koreň and T. Hrúz

For version 0.93
Last updated January 5, 1999

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below,

refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

The document explains, comments and shows some decisions, programming techniques and solutions in Clif. Its purpose is to highlight some important parts of the implementation and to help with possible extensions for new users.

The document also helps authors to have the latest form of the implementation details in comprehensible form, as a commented manual.

The document consists of the Clif grammar in BNF form, description of the main interpreter organization, possibilities for including new library functions and conditions for extending the language of Clif.

Interrupt services as an option for debugging are also described. Optional graphical interface with graphic primitives is a part of the document as well.

The substantial part of the document is devoted to the internal representation of types in Clif. The chapter will be interesting for users, who want to extend the Clif with new non atomary data types.

Contributors to the Clif

In addition to Ľudovít Koreň several people contributed to the Clif.

- Tomáš Hrúz principal planning decisions, overall design and many ideas for implementation
- Jozef Repiský first basic (and very restricted) implementation

Chapter 1

Compiling options

There are several additional options, that are supported during a run of configure script. They are described in the following sections.

1.1 Option –enable-CONTROL

If a command line option –enable-CONTROL was specified, the synchronous and asynchronous interrupts are enabled (see 7).

1.2 Option –enable-CODE

If a command line option –enable-CODE was specified, the output of the virtual machine code is enabled. The output file name is code.

1.3 Option –enable-DEBUG

If a command line option –enable-DEBUG was specified, the debug output is enabled on the stderr.

1.4 Option –enable-CHKSTACK

If a command line option –enable-CHKSTACK was specified, the run-time check of stack and code area is done. I.e. each instruction which grows the stack, assures that the stack does not interfere with generated code.

Chapter 2

Invoking Clif

2.1 Options summary

-bc Size of the memory in 512-Byte pages.

-c Compile only. (Not fully supported yet.)

-copying Show copying.

Options controlling Clif behavior. -fcall-by-reference, -fno-call-by-reference, -fcall-by-value, -fno-call-by-value, -fhandle-main

Debugging options. -g, -dy

-help Show short help.

-v Show version.

-version Show version.

-verbose Verbose.

-warranty Show warranty.

Warning options. -w, -Wcomment, -Wformat, -Wimplicit, -Wreturn-type, -Wtrigraphs, -Wuninitialized, -Wall All of the above warnings.

-W, -Waggregate-return, -Wunused

2.2 Memory size options

`-bc=<number>`

option specifies number of 512-Byte pages for the Clif environment main memory. The arithmetical and temporary stack is multiple of this option as well.

2.3 Options controlling Clif behavior

-fcall-by-reference call by reference parameter passing mechanism.

-fno-call-by-reference do not pass parameters by reference.

-fcall-by-value call by value parameter passing mechanism.

-fno-call-by-value do not pass parameters by value.

From the above mentioned options, only one should be specified in positive form and one in negative form.

-fhandle-main simulate compiler-like behavior. The files on the command line and included files are compiled. The 'main' function must be defined. After parsing pass, the generated code is executed. The main function is the beginning of execution.

2.4 Debugging options

-g produce debugging information. The source lines are output during virtual machine code execution.

-dy dump debugging information during parsing to standard error.

2.5 Warning options

-w Inhibit all warning messages.

-Wcomment Warn when a comment-start sequence '/*' appears in a comment.

-Wformat Check calls to 'printf' and 'scanf', etc., to make sure that the arguments supplied have types appropriate to the specified format string.

-Wimplicit Warn if a function or parameter is implicitly declared.

-Wreturn-type Warn if the return statement is without return value in non-void function, or with a value in 'void' function.

-Wtrigraphs Warn about trigraphs usage.

-Wuninitialized An automatic variable is used without first being initialized.

-Wall All of the above warnings.

-W Print extra warning messages.

-Waggregate-return Warn if any functions that return structures or unions are defined or called.

-Wunused Warn whenever a variable is unused aside from its declaration.

Chapter 3

Errors

3.1 Clif error messages

In this chapter is a list of Clif error messages.

3.1.1 Syntax error messages

```
    case 1000:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"variable ‘%s’ isn’t declared\n",
text);
      break;
    case 1001:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"variable ‘%s’ was already declared\n",
text);
      break;
    case 1002:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"remote procedure %s is not declared\n",
proc_name_text[proc]);
      break;
    case 1003:
      ERROR_INFO;
      fprintfx (stderr,
"local variable ‘%s’ was already declared\n",
text);
      break;
    case 1004:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"at the %d-th char, near the ‘%s’\n",
char_counter, yytext);
      break;
    case 1005:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid type of the operand, %d-th character\n",
char_counter);
      break;
    case 1006:
      ERROR_INFO;
      fprintfx (stderr,
"remote function ‘%s’ already declared\n",
text);
      break;
    case 1007:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"remote function isn’t declared\n");
      break;
    case 1008:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"remote functions are not in the load table\n");
      break;
    case 1009:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"‘void’ type in expression\n");
      break;
    case 1010:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"‘void’ type assigned to l_value\n");
      break;
    case 1011:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"load can’t open file ‘%s’\n",
yytext);
      break;
    case 1012:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"variable or field ‘%s’ declared void\n",
text);
      break;
    case 1013:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"switch quantity not an integer\n");
      break;
    case 1014:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"case label does not reduce to an integer constant\n");
      break;
    case 1015:
      ERROR_FULL_INFO(tmp_c->line_number);
      fprintfx (stderr,
"duplicate case value\n");
      ERROR_FULL_INFO(tmp_m->line_number);
      fprintfx (stderr,
"this is the first entry for that value\n");
      break;
    case 1016:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"case label not within a switch statement\n");
      break;
    case 1017:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"struct tag ‘%s’ was already declared\n",
text);
      break;
    case 1018:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"union tag ‘%s’ was already declared\n",
text);
      break;
    case 1019:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"enum tag ‘%s’ was already declared\n",
text);
      break;
    case 1020:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"conversion to non-scalar type requested\n");
```

```
      break;
    case 1021:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid type argument of ‘->’\n");
      break;
    case 1022:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid lvalue in unary ‘&’\n");
      break;
    case 1023:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"storage size of ‘%s’ isn’t known\n",
text);
      break;
    case 1024:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"parameter ‘%s’ has incomplete type\n",
text);
      break;
```

3.1.2 Clif compilation error messages

```
    case 2000:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid number of subscripts\n");
      break;
    case 2001:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"‘%s’ is not an array variable\n",
text);
      break;
    case 2002:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"invalid type of array subscript\n");
      break;
    case 2003:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"type of formal parameter does not match previous declaration\n");
      break;
    case 2004:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"number of formal parameters does not match previous declaration\n");
      break;
    case 2005:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"name of formal paramter does not match previous declaration\n");
      break;
    case 2006:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"size of array subscript of formal parameter does not match previous declaration\n");
      break;
    case 2007:
      print_source_line ();
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"number of array subscripts of formal parameter does not match previous declaration\n");
      break;
    case 2008 :
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"structure has no member named ‘%s’\n",
text);
      break;
    case 2009:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"request for member ‘%s’ in something not a structure or union\n",
text);
      break;
```

3.1.3 Clif control statement error messages

```
    case 3000:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"‘break’ outside loop or switch\n");
      break;
    case 3001:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"bad used ‘continue’\n");
      break;
    case 3002:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"default label not within a switch statement\n");
      break;
    case 3003:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"multiple default labels in one switch\n");
      ERROR_FULL_INFO(fixp->switch1.def_use.line_number);
      fprintfx (stderr,
"this is the first default label\n");
      break;
    case 3004:
      ERROR_FULL_INFO(line_counter);
      fprintfx (stderr,
"duplicate label ‘%s’\n",
text);
      break;
    case 3005:
      ERROR_FULL_INFO(error_line_number);
      fprintfx (stderr,
"label ‘%s’ used but not defined\n",
text);
      break;
    case 3006:
      ERROR_FULL_INFO(error_line_number);
      fprintfx (stderr,
"invalid lvalue in assignment\n");
      break;
```

3.1.4 Clif run-time error messages

```
    case 4000:
      ERR_NO_INFO;
      fprintfx (stderr,
"interpreter: full memory\n");
      break;
    case 4001:
      ERR_NO_INFO;
      fprintfx (stderr,
"interpreter: stack overflow\n");
      break;
    case 4002:
      ERR_NO_INFO;
      fprintfx (stderr,
"operating system out of memory\n");
      break;
```

3.1.5 Clif fatal error messages

```
    case 5000:
      fprintfx (stderr,
"Compile Fatal ");
      ERR_NO_INFO;
      fprintfx (stderr,
"Interpreter Internal Error (unknown operand type) in line %d e-mail: %s\n",
line_counter, EMAIL);
      print_source_line ();
      break;
    case 5001:
      fprintfx (stderr,
"Run-time Fatal ");
      ERR_NO_INFO;
      fprintfx (stderr,
" Internal Interpreter Error (unknown instruction) e-mail: %s\n",
EMAIL);
      print_source_line ();
      break;
    case 5002:
      fprintfx (stderr,
"Compile Fatal ");
      ERR_NO_INFO;
      fprintfx (stderr,
" Interpreter Internal Error (error in book-keeping) in line %d e-mail: %s\n",
line_counter, EMAIL);
      print_source_line ();
      break;
    case 5003:
      fprintfx (stderr,
"Compile Fatal ");
      ERR_NO_INFO;
      fprintfx (stderr,
" Internal Interpreter Error (error in operand type) in line %d e-mail: %s\n",
line_counter, EMAIL);
      print_source_line ();
      break;
```

3.1.6 Clif warning messages

```
    case 6000:
      if (warning_yes)
{
  ERROR_FULL_INFO(line_counter);
  fprintfx (stderr,
    "remote function %s already declared\n",
    text);
}
      return;
    case 6001:
      if (warning_yes)
{
  ERROR_FULL_INFO(line_counter);
  fprintfx (stderr,
    "‘return’ with no value, in function returning non-void\n");
}
```

```
   return;
  case 6002:
   if (warning_yes)
{
 ERROR_FULL_INFO(line_counter);
 fprintfx (stderr,
  "‘return’ with a value, in function returning void\n");
}
   return;
  case 6003:
   if (warning_yes)
{
 if (proc)
  {
   print_file_name ();
   fprintfx (stderr,
" In function ‘%s’:",
proc_name_text[proc]);
   ERROR_INFO;
   fprintfx (stderr,
"unused variable ‘%s’\n",
text);
  }
 else
  {
   print_file_name ();
   fprintfx (stderr,
" In block finishing at line %d:\n",
line_counter);
   ERROR_INFO;
   fprintfx (stderr,
"unused variable ‘%s’\n",
text);
  }
}
   return;
  case 6004:
   if (warning_yes)
{
 ERROR_FULL_INFO(error_line_number);
 fprintfx (stderr,
  "label ‘%s’ defined but not used\n",
  text);
}
   return;
  case 6005:
   if (warning_yes)
{
 ERROR_INFO;
 fprintfx (stderr,
  " ‘/*’ within comment\n");
}
   return;
  case 6006:
   if (warning_yes)
{
 if (proc)
  {
   print_file_name ();
   fprintfx (stderr,
" In function ‘%s’:",
proc_name_text[proc]);
   ERROR_INFO;
   fprintfx (stderr,
"‘%s’ might be used uninitialized in this function\n",
text);
  }
 else
  {
   print_file_name ();
   fprintfx (stderr,
" In block finishing at line %d:",
line_counter);
   ERROR_INFO;
```

```
      fprintfx (stderr,
"‘%s’ might be used uninitialized in the block\n",
text);
    }
}
      return;
    case 6007:
      if (warning_yes)
{
  print_file_name ();
  fprintfx (stderr,
    " In function ‘%s’:",
    proc_name_text[proc]);
  ERROR_INFO;
  fprintfx (stderr,
    "number of locals is greater than the ANSI allows\n");
}
      return;
    case 6008:
      if (warning_yes)
{
  print_file_name ();
  fprintfx (stderr,
    " In function ‘%s’:\n",
    proc_name_text[proc]);
  ERROR_INFO;
  fprintfx (stderr,
    "number of params is greater than the ANSI allows\n");
}
      return;
    case 6009:
      if (warning_yes)
{
  char *tmp_line, *beg, *end, *com;
  int n;
  n = strlen(line_buf);
  tmp_line = malloc(n+1);
  if (NULL == tmp_line)
    {
      perror ("");
      abort ();
    }

  strcpy (tmp_line, line_buf);
  beg = strrchr (tmp_line, ’(’);
  if (NULL == beg)
    {
      perror ("");
      abort ();
    }

  beg++;
  com = strrchr (beg, ’,’);
  if (NULL != com)
    beg = com + 1;
  for (; *beg == ’ ’ || *beg == ’\t’; beg++);

  end = strrchr (beg, ’ ’);
  com = strchr (beg, ’ ’);
  if (end != com)
    *end = ’\0’;
  else
    tmp_line[n - 1] = ’\0’;

  ERROR_INFO;
  fprintfx (stderr,
    "‘%s’ declared inside parameter list its scope is only this definition or declaration, which is pr
    beg);
  free (tmp_line);
}
      return;
```

3.1.7 Clif initialization error messages

```
    case 7000:
      print_error_number (err_no);
      fprintfx (stderr,
"in run-string and/or in ‘clif.ini’ file\n");
      break;
    case 7001:
      print_error_number (err_no);
      fprintfx (stderr,
"interpreter: can’t open file %s\n",
argvv[argc_counter]);
      break;
    default:
      fprintfx (stderr, "Fatal error invalid error number (%d) e-mail: %s\n", err_no, EMAIL);
      break;
```

Chapter 4

Syntax of the language

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

<list_stat >::= <list_stat ><stat_1 >
             |
;

<stat_0 >::= <declarations >
             |<statement >
             |RESUME ';'
             |';'
             |<error >';'
;

<stat_1 >::= <statement >
             |GOTO IDENT ';'
             |';'
             |<error >'}'
             |<error >';'
;
```

```
<jump_statement >
::= BREAK ';'
        |CONTINUE ';'
        |RETURN <expression >';'
        |RETURN ';'
;
```

```
<declaration_specifiers >
::= <storage_class_specifier >
          |<storage_class_specifier ><declaration_specifiers >
          |<type_specifier >
          |<type_specifier ><declaration_specifiers >
          |<type_qualifier >
          |<type_qualifier ><declaration_specifiers >
;
M ::=
;

<declarations >::=
M <declaration_specifiers ><first_dekl >
          |REMOTE '{' INTRINSIC ',' STRINGC '}' IDENT ';'
          |REMOTE '{' RPC ',' STRINGC '}' IDENT ';'
          |UNLOAD IDENT ';'
;
```

```
<statement >::= <labeled_statement >
              |<compound_statement >
              |
<expression >';'
              |<selection_statement >
              |<iteration_statement >
              |<jump_statement >
              |EXIT ';'
              |CSUSPEND ';'
;
```

```
<selection_statement >
::= IF '(' <expression >')'
<then >
            |SWITCH '(' <expression >')'
<switch_body >
;
```

```
<iteration_statement >
::= WHILE
'(' <expression >')'
<while_stat >
          |DO
<do_while_stat >
WHILE '(' <expression >')' ';'
          |FOR <for ><for_stat >
;

<while_stat >::= <stat_1 >
;

<do_while_stat >
::= <stat_1 >
;

<for_stat >::= <stat_1 >
;

<for >::= '(' <expression >';'
<for_expr1 >
          |'(' ';'
<for_expr1 >
;

<for_expr1 >::= <expression >';'
<for_expr2 >
          |';'
<for_expr2 >
;

<for_expr2 >::= <expression >')'
          |')'
;
```

```
<type_specifier >::= INT
             |DOUBLE
             |FLOAT
             |CHAR
             |VOID
             |LONG
             |SHORT
             |SIGNED
             |UNSIGNED
             |<struct_or_union_specifier >
             |<enum_specifier >
             |TYPENAME
;

<type_qualifier >
::= CONST
             |VOLATILE
;

<pointer >
::= '*'
             |'*' <type_qualifier_list >
             |'*' <pointer >
             |'*' <type_qualifier_list ><pointer >
;

<type_qualifier_list >
::= <type_qualifier >
             |<type_qualifier_list ><type_qualifier >
;

<struct_or_union_specifier >
::= <struct_or_union >IDENT
'{' <struct_declaration_list >'}'
             |<struct_or_union >
'{' <struct_declaration_list >'}'
             |<struct_or_union >IDENT
;

<struct_or_union >
::= STRUCT
             |UNION
;

<struct_declaration_list >
::= <struct_declaration >
             |<struct_declaration_list ><struct_declaration >
;

<struct_declaration >
::= M <specifier_qualifier_list ><struct_declarator_list >';'
;

<specifier_qualifier_list >
::= <type_specifier >
             |<type_specifier ><specifier_qualifier_list >
             |<type_qualifier >
             |<type_qualifier ><specifier_qualifier_list >
;
```

```
<struct_declarator_list >
::= <struct_declarator >
          |<struct_declarator_list >',' <struct_declarator >
;

<struct_declarator >
::= <declarator >
          |':' <constant_expression >
          |<declarator >':' <constant_expression >
;

<enum_specifier >
::= ENUM '{'
<enumerator_list >'}'
          |ENUM IDENT '{'
<enumerator_list >'}'
          |ENUM IDENT
;

<enumerator_list >
::= <enumerator >
          |<enumerator_list >',' <enumerator >
;

<enumerator >
::= IDENT
          |IDENT '=' <constant_expression >
;

<declarator >
::= <pointer ><direct_declarator >
          |<direct_declarator >
;

<direct_declarator >
::= IDENT
          |IDENT <list_dim >
;

<initializer >
::= <assignment_expression >
          |'{'
<initializer_list_complete >
;

<initializer_list_complete >
::= <initializer_list >'}'
          |<initializer_list >',' '}'
;

<initializer_list >
::= <initializer >
          |<initializer_list >','
<initializer >
;

<type_name >
::= M <specifier_qualifier_list >
          |M <specifier_qualifier_list ><abstract_declarator >
;
```

```
<abstract_declarator >
::= <pointer >
            |<direct_abstract_declarator >
            |<pointer ><direct_abstract_declarator >
;

<direct_abstract_declarator >
::= '(' <abstract_declarator >')'
            |'[' ']'
            |<direct_abstract_declarator >'[' ']'
            |'[' <constant_expression >']'
            |<direct_abstract_declarator >'[' <constant_expression >']'
            |'(' ')'
            |<direct_abstract_declarator >'(' ')'
            |'(' <list_type_spec >')'
            |<direct_abstract_declarator >'(' <list_type_spec >')'
;

<storage_class_specifier >
::= TYPEDEF
            |EXTERN
            |EXPORT_T
            |STATIC
            |AUTO
            |REGISTER
;
```

```
<list_type_spec >::= M <declaration_specifiers >
              |M <declaration_specifiers >',' 
<list_type_spec >
              |M <declaration_specifiers ><list_dim_or_pointer >
              |M <declaration_specifiers ><list_dim_or_pointer >',' 
<list_type_spec >
;

<list_dim_or_pointer >::= <list_dim >
              |<pointer >
              |'(' <pointer >')' '(' ')'
              |'(' <pointer >')' '(' <list_type_spec >')'
              |'(' <pointer >')' <list_dim >
;
```

```
<first_dekl >::= IDENT
<initializer_optional >
               |IDENT <list_dim >
<initializer_optional >
               |<pointer >IDENT
<initializer_optional >
               |<pointer >IDENT <list_dim >
<initializer_optional >
               |IDENT '('
<func_first >
               |<pointer >IDENT '('
<func_first >
               |';'
;

<func_first >::= ')'
<initializer_optional >
               |<list_type_spec >')'
<initializer_optional >
               |<list_form_param >')'
<initializer_optional >
               |<list_form_param >')'
<compound_statement >
               |')'
<compound_statement >
;

<func_rest >::= ')'
<initializer_optional >
               |<list_type_spec >')'
<initializer_optional >
               |<list_form_param >')'
<initializer_optional >
               |<error >'{'
;
```

```
<list_dekl >::= IDENT
<initializer_optional >
              |IDENT <list_dim >
<initializer_optional >
              |IDENT '('
<func_rest >
              |<pointer >IDENT
<initializer_optional >
              |<pointer >IDENT <list_dim >
<initializer_optional >
              |<pointer >IDENT '('
<func_rest >
;

<initializer_optional >
::= ';' R
              |<initialization >';'
              |',' R <list_dekl >
              |<initialization >
',' <list_dekl >
;
R ::=
;

<initialization >
::= '=' <initializer >
;
```

```
<then >::= <stat_1 >
ELSE <stat_1 >
                |<stat_1 >
;

<switch_body >::= <stat_1 >
;
```

```
<labeled_statement >
::= IDENT ':'
<stat_1 >
            |CASE
<constant_expression >
':' <stat_1 >
            |DEFAULT ':'
<stat_1 >
;
```

```
<compound_statement >
::= '{' N <list_stat >'}'
        |'{' N <list_loc_dekl >
<list_stat >'}'
;
N
::=
;

<list_form_param >
::= M <declaration_specifiers >IDENT
        |M <declaration_specifiers >IDENT ','
<list_form_param >
        |M <declaration_specifiers >IDENT <list_dim >
        |M <declaration_specifiers >IDENT <list_dim >','
<list_form_param >
        |M <declaration_specifiers ><pointer >IDENT
        |M <declaration_specifiers ><pointer >IDENT ','
<list_form_param >
        |M <declaration_specifiers ><pointer >IDENT <list_dim >
        |M <declaration_specifiers ><pointer >IDENT <list_dim >','
<list_form_param >
;

<list_dim >::= '[' ']'
        |'[' NUMBERI ']'
        |<list_dim >'[' NUMBERI ']'
;

<list_loc_dekl >
::= M <declaration_specifiers ><list_loc_dekl_1 >
        |M <declaration_specifiers >';'
        |M <declaration_specifiers >';'
<list_loc_dekl >
;

<list_loc_dekl_1 >
::= IDENT
<initializer_optional_loc >
        |IDENT <list_dim >
<initializer_optional_loc >
        |IDENT '(' ')'
<initializer_optional_loc >
        |IDENT '(' <list_type_spec >')'
<initializer_optional_loc >
        |IDENT '(' <list_form_param >')'
<initializer_optional_loc >
        |<pointer >IDENT
<initializer_optional_loc >
        |<pointer >IDENT <list_dim >
<initializer_optional_loc >
        |<pointer >IDENT '(' ')'
<initializer_optional_loc >
        |<pointer >IDENT '(' <list_type_spec >')'
<initializer_optional_loc >
        |<pointer >IDENT '(' <list_form_param >')'
<initializer_optional_loc >
;

<initializer_optional_loc >
```

```
::= ';' P
            |';' P
<list_loc_dekl >
            |<local_initialization >';'
            |<local_initialization >';'
<list_loc_dekl >
            |',' P <list_loc_dekl_1 >
            |<local_initialization >',' <list_loc_dekl_1 >
;
P ::=
;

<local_initialization >
::= '=' <initializer >
;

<call >::= <list_param >
            |')'
;

<list_param >::= <assignment_expression >')'
            |<assignment_expression >','
<list_param >
;
```

```
<primary_expression >
::= <ident >
            |NUMBERI
            |NUMBERUI
            |NUMBERLI
            |NUMBERLUI
            |NUMBERD
            |NUMBERLD
            |NUMBERF
            |STRINGC
            |WSTRINGC
            |NUMBERC
            |'(' <expression >')'
;

<ident >::= IDENT
;

<postfix_expression >
::= <primary_expression >
            |<postfix_expression >
'[' <expression >']'
            |<postfix_expression >'.' IDENT
            |<postfix_expression >PTR IDENT
            |<postfix_expression >PP
            |<postfix_expression >MM
            |<postfix_expression >'('
<call >
;

<unary_expression >
::= <postfix_expression >
            |'&' <unary_expression >
            |'∗' <unary_expression >
            |NEG_T <unary_expression >
            |NEG_B <unary_expression >
            |'+' <unary_expression >
            |'-'
<unary_expression >
            |PP <unary_expression ><ae_empty >
            |MM <unary_expression ><ae_empty >
            |SIZEOF <unary_expression >
            |SIZEOF '(' <type_name >')'
;

<cast_expression >
::= <unary_expression >
            |'(' <type_name >')' <cast_expression >
;

<multiplicative_expression >
::= <cast_expression >
            |<multiplicative_expression >'∗' <cast_expression >
            |<multiplicative_expression >'/' <cast_expression >
            |<multiplicative_expression >'´<cast_expression >
;

<additive_expression >
::= <multiplicative_expression >
            |<additive_expression >'+' <multiplicative_expression >
```

```
              |<additive_expression >'-' <multiplicative_expression >
;

<shift_expression >
::= <additive_expression >
              |<shift_expression >SHIL <additive_expression >
              |<shift_expression >SHIR <additive_expression >
;

<relational_expression >
::= <shift_expression >
              |<relational_expression >'<' <shift_expression >
              |<relational_expression >'>' <shift_expression >
              |<relational_expression >LQ <shift_expression >
              |<relational_expression >GQ <shift_expression >
;

<equality_expression >
::= <relational_expression >
              |<equality_expression >EQ_A <relational_expression >
              |<equality_expression >NE_A <relational_expression >
;

<bit_AND_expression >
::= <equality_expression >
              |<bit_AND_expression >'&' <equality_expression >
;

<exclusive_OR_expression >
::= <bit_AND_expression >
              |<exclusive_OR_expression >'^' <bit_AND_expression >
;

<inclusive_OR_expression >
::= <exclusive_OR_expression >
              |<inclusive_OR_expression >'|' <exclusive_OR_expression >
;

<logical_AND_expression >
::= <inclusive_OR_expression >
              |<logical_AND_expression >AND_A
<inclusive_OR_expression >
;

<logical_OR_expression >
::= <logical_AND_expression >
              |<logical_OR_expression >OR_A
<logical_AND_expression >
;

<conditional_expression >
::= <logical_OR_expression >
              |<logical_OR_expression >'?'
<expression >':'
<conditional_expression >
;
```

```
<assignment_expression >
::= <conditional_expression >
            |<unary_expression >
'=' <assignment_expression >
            |<unary_expression ><assignment_operator >
;

<ae_empty >::=
;

<assignment_operator >
::= MUL_ASSIGN <ae_empty ><assignment_expression >
            |DIV_ASSIGN <ae_empty ><assignment_expression >
            |MOD_ASSIGN <ae_empty ><assignment_expression >
            |ADD_ASSIGN <ae_empty ><assignment_expression >
            |SUB_ASSIGN <ae_empty ><assignment_expression >
            |LEFT_ASSIGN <ae_empty ><assignment_expression >
            |RIGHT_ASSIGN <ae_empty ><assignment_expression >
            |AND_ASSIGN <ae_empty ><assignment_expression >
            |XOR_ASSIGN <ae_empty ><assignment_expression >
            |OR_ASSIGN <ae_empty ><assignment_expression >
;
```

<expression >
::= <assignment_expression >
|<expression >','
<assignment_expression >
;

<constant_expression >
::= <conditional_expression >
;

<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.

4.1 Syntax of the graphical subsystem language

<list_stat_0 >::= <list_stat_0 ><stat_0 >
|
;

<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
|<error >
;

<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

Chapter 5

Interpreter organization

5.1 Instruction set of the virtual machine

Notation:
ADR- address of the memory cell
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 literal
$<double>$- a double precision floating point literal
$<float>$- a single precision floating point literal
$<char>$- 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

5.1.1 Address instructions and instructions on the arithmetic stack

Instruction **MOV**.
Description: move data from the specified address to another specified address. The addresses are specified either as 2 consecutive addresses on the arithmetical stack, or one address is on the arithmetical stack and the second is created in the temporary stack.
Options:
$[ADR] \longleftarrow [[AST]]$ type OP_0_mi
The instruction is 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, etc.
$BP \longleftarrow STACK$ 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
$STACK \longleftarrow FRAME$ type OP_0_mi
$[AST+1] \longleftarrow [AST]$ type OP_0_mi
$[AST] \longleftarrow [[AST]]$ type OP_0_mi
$[AST] \longleftarrow [AST]$ type OP_0_mi
$[AST] \longleftarrow [(AST-1)+[AST]]$ type OP_0_mi $[(AST-2)+[AST]] \longleftarrow [(AST-1)+[AST]]$ 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
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
The arithmetic stack is popped to the stack. type OP_1_i
The instruction is implemented for each basic data type.
Instruction **XCHG**
Description: Exchange two addresses on the top of the arithmetic stack.
Options:
$[AST-1] \longleftrightarrow [AST]$ type OP_0_ma

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 on the top of arithmetic stack or to the stack pointer. On the stack can be only processed an instruction mentioned below (to the stack pointer can be only added an integer number).
Options:
$[[AST-1]] \longleftarrow [[AST]] + [[AST-1]]$ type OP_0_mi
$STACK \longleftarrow (STACK+NUM)$ type OP_1_i

Instruction **SUB**.
Description: perform arithmetic subtraction on the top of arithmetic stack or from the stack pointer. On the stack can be only processed an instruction mentioned below (from the 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:
$[[AST-1]] \longleftarrow [[AST-1]] \quad || \quad [[AST]]$ 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:
$[[AST-1]] \longleftarrow [[AST-1]] \quad | \quad [[AST]]$ 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

Integer and floating point instructions

Instruction **CVT**.
Description: Convert a signed quantity to a different signed data type.
Options:
$<integer> \longrightarrow <double>$
$<double> \longrightarrow <integer>$
$<integer> \longrightarrow <float>$
$<float> \longrightarrow <integer>$
$<float> \longrightarrow <double>$
$<double> \longrightarrow <float>$
$<char> \longrightarrow <integer>$
$<integer> \longrightarrow <char>$
$<double> \longrightarrow <char>$
$<char> \longrightarrow <double>$
$<char> \longrightarrow <float>$
$<float> \longrightarrow <char>$
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.

There are not listed all possibilities. Basically, there is an instruction from each type to every type.

5.1.2 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

$[STACK] \longleftarrow FRAME$ 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
$FRAME \longleftarrow [STACK]$ type OP_0_mi

5.1.3 Temporary stack instructions

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

5.1.4 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

5.1.5 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. The intrinsic functions are either call by value or call by reference.
Options: type OP_1_ma

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

5.2 Storage of variables, hash tables

Local and global variable names and addresses are stored in hash tables [10]. Structure record of the hash table for global variables is:

```
struct tab                     /* Hash table structure. */
{
  char *name;
  int def; /* Position in the hash table. */
  int count; /* Is the variable used at all? */
  int use_line_number; /* The first use of the variable. */
  int l_value_flag; /* Is the variable initialized before
   the use? */
  int declaration_line; /* Variable declaration line number. */
  struct tab *next;
};
```

```
static struct tab *hastab; /* Pointer to the hash table. */
```

Another table is for global identifiers. Its structure is as follows:

```
struct ident_tab              /* Table of identifiers. */
{
  struct internal_type *type;
  int body;
  struct ident_list_str *list_formal_param;
  struct FIX *next;
  char *adr;
};
```

```
static struct ident_tab *identtab;   /* Pointer to the table of identifiers. */
```

The structure internal_type is used for type of a variable, e.g. array, function etc. The field body flags if the variable is declared or defined. The DIM structure is for storing of dimensions of arrays and has the following content:

```
struct range
{
  int lower;
  int upper;
};
```

The structure FIX is used for fixing of calls of the function and looks as follows:

```
struct FIX       /* List of addresses where to backpatch
       * undefined function.
       */
{
  char *address;
  struct FIX *next;
};
```

It is a linked list. address point to the virtual machine code where the call address should be fixed.

The table of local variables is allocated when local variables are defined for the first time. Then, for each new block new table is allocated. After return to scope level zero the tables are deallocated. The table looks like:

```
struct ident_tab_header
{
  int scope_level;
  int pi_loc;
  int offset;
  char *file_scope;
  struct ident_tab_header *previous_level;
  struct ident_tab_loc *all;
  struct tab *hastab_loc;
  struct ident_tab_loc *table;
};

struct ident_tab_loc          /* Table of local identifiers. */
{
  struct internal_type *type;
  int body;
  int offset;
  char *adr;
  struct ident_list_str *list_formal_param;
  struct ident_tab_loc *previous;
};
```

The struct ident_tab_header is header for the list of tables for local identifiers. scope_level is nesting level of the block. The zero level is the prime level. The offset is size in bytes that has to be added to the offset of the local variable in nested scope (i.e. in the scope_level + 1, if the current level is scope_level). The previous_level points to the block scope_level - 1. The all is a pointer to the list of all local variables. This list can be traversed and additional information can be found. The hastab_loc and table point to the hash table and identifier table, respectively of the current level.

The struct ident_tab_loc consists of:

type - internal representation of type

offset - offset of the variable in the current level

previous - points to the previous declared variable

Pointer to the table is:

```
static struct ident_tab_header
*identtab_loc;                /* Pointer to the table of local identifiers. */
```

The interpreter is always searching for variables in the table of the local identifiers. If the names do not match with searched name, the interpreter proceeds in the table of the global identifiers. If the interpreter does not find the name of the variable in the hash table, it announces an error.

Structures of the hash tables of intrinsic functions are the following:

```
struct remote_tab             /* Hash table structure for intrinsic
       * functions.
       */
{
  char *name;
  void (*adr) PROTO((char **));
  struct remote_tab *next;
};
```

```
struct remote_has_tab         /* Hash table structure for intrinsic
```

```
       * functions.
       */
{
  char *name;
  int offset;
  struct remote_has_tab *next;
};
```

Intrinsic functions are initially stored in the list remote_tab. The functions are loaded from this list into the hash table of the remote_has_tab structure. Pointers to these structures are:

```
struct remote_has_tab *hastab_remote;
```

```
struct remote_tab
*remote_ptr_C;                /* Pointer to the structure of remote
       * function table.
       */
```

Hash function is as follows:

```
/*
 * Hash function.
 */
static unsigned int
hash_code (s, size)
  char          *s;
  unsigned int size;
{
  int             c = 0;

  while (*s)
    {
      c = c << 1 ^ (*s);
      s++;
    }
  if (0 > c)
    c = (-c);
  return (c % size);
}
```

MAX_HAS is size of the hash table.

In addition, there is a table of defined types:

```
static struct ident_tab_header
```

```
*tagtab; /* Pointer to the table of tags. */
```

In this table are stored new types and structure tags, etc. For types, the lazy allocation is used, i.e. table is allocated only when first type is declared or defined.

5.3 Fixation and fixative structures

We use fixation in the statements if, while, for, continue, break, return and switch. The fixation are used in the loop statements because of nested loops or unknown length of the, loop respectively.

Fixative structures are following:

```
/*
 * control.h
 *
 * Header of fixative structures.
 */

#ifndef _CONTROL_H
#define _CONTROL_H

typedef struct
{
 int major;
 char *jmp;
 char *jz;
   struct cont1 *cnext;
 struct break1 *bnext;
} WHILE1;

typedef struct
{
 int major;
 char *jn; /* Label of the JZ */
 char *jmp2; /* Label of the JMP. The first JMP */
/* instruction. It is between expr2 */
/* and expr3. */
 char *jmp3; /* Address where to jump if all */
/* statements of the loop are */
/* done. (See manual) */
 struct break1 *bnext;
 struct cont1 *cnext;
} FOR1;

struct break1
{
 char *adr;
 struct break1 *next;
};

struct cont1
{
 char *adr;
 struct cont1 *next;
};
```

```
typedef struct
{
 int major;
 char *jz;
 char *jmp;
} IF1;

struct default_usage
        {
  int line_number; /* Line number where the default label
   was used. */
  int def_flag; /* Flag if the default label was used.
   It can be used only once per switch
   statement. (ANSI) */
  char *adr; /* Address where to jump to in the
   virtual machine code. */
};

typedef struct
        {
  int major;
  char *jz;
  char *jmp;
  struct default_usage def_use;
  struct break1 *bnext;
  struct list_const1 *next;
} SWITCH1;

struct list_const1 /* The list of labels in switch
   statement is created. The list is
   at the end of switch statement
   checked if labels in this list are
   not duplicit. If there are duplicit
   error message is issued. */
        {
  int line_number; /* Line number where the label was
   used in the switch statement. */
  int constant; /* Label in switch statement. */
  struct list_const1 *next;
};

union fix
{
 WHILE1 while1;
 FOR1 for1;
 IF1 if1;
 SWITCH1 switch1;
        };
```

```
union fix *fixp;              /* Pointer to the fixative stack. */
```

The fixation of for statement is fully optimized, i.e. there are no jumps on jumps. It is tested if all parts, i.e.

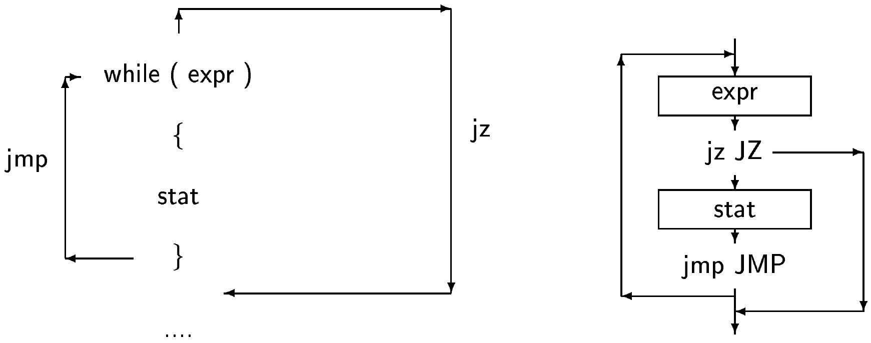


Figure 5.1: Fixation in while

expr2 and expr3 are in the for statement.

The statement break, continue are fixed to the end of the most inner loop.

Another case is the statement return. The fixation is made another way as it is mentioned above. The fixative structure is:

```
struct return1       /* List of addresses where to backpatch
       * returns from a function.
       */
{
  char *adr;
  struct return1 *next;
};
```

```
struct return1 *rp; /* Pointer for backpatch address of */
/* return in a function. */
```

The epilogue of the procedure is fixed in this case (i.e. the epilogue is always executed, before the control flow reaches caller).

The statement switch has complicated fixative structure. You can follow it in the comments attached to the fixative structure. It can be be clearer from the fig. 5.7 as well.

See figures 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7 for the all cases of the fixations.

Unconditional program branching is supported by a goto statement. The goto statement has the following fixative structure:

```
struct goto_adr /* List of addresses with goto
   statements for the current label. */
{
  char *adr; /* Address of the goto statement. */
  int line_number; /* Line number of the goto statement. */
  struct goto_adr *gnext;
};
```

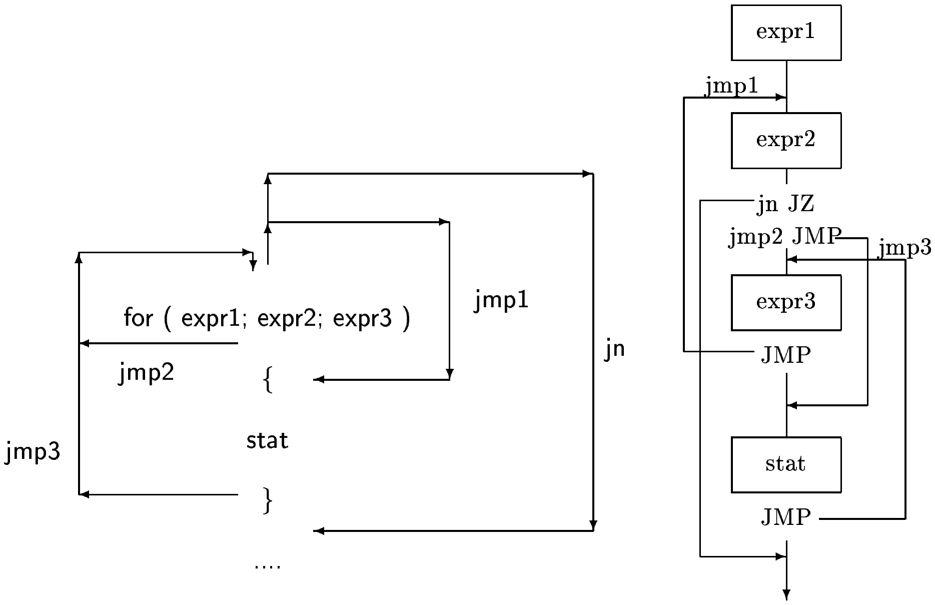


Figure 5.2: Fixation in for

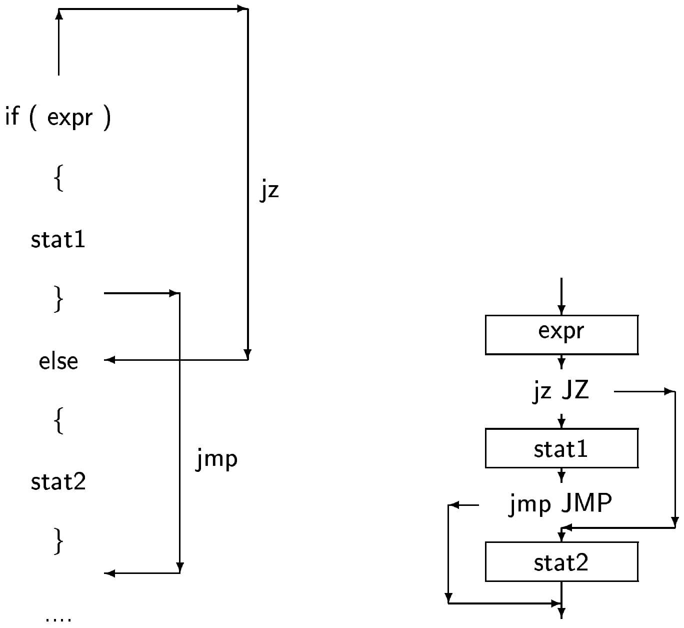


Figure 5.3: Fixation in if

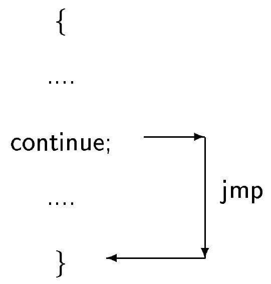


Figure 5.4: Fixation of continue

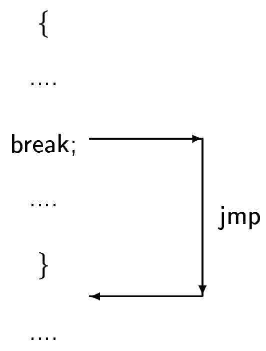


Figure 5.5: Fixation of break

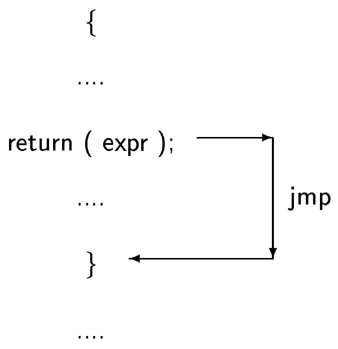


Figure 5.6: Fixation of return

```
switch (expr)
    {
    case const_expr_1 : stat_1 ;
                        break;
    case const_expr_2 : stat_2 ;
    case const_expr_3 : stat_3 ;
                        break ;
    default : stat_4 ;
              break;
    }
```

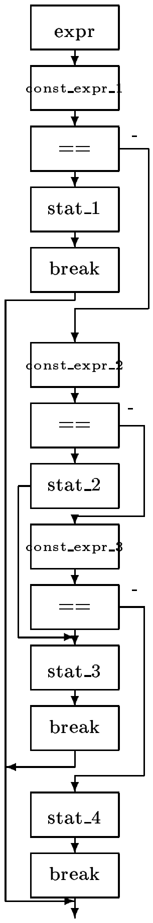


Figure 5.7: Fixation of switch

```
struct goto_tab /* Hash table structure for goto
   labels. */
{
  char *name; /* Name of the label. */
  char *label_adr; /* Address of the label in the
   generated code. */
  int line_number; /* Line number of the label in the
   source file. */
  struct goto_adr *gnext; /* List of goto's to the label. */
  struct goto_tab *next;
};
```

```
static struct goto_tab *hastab_goto; /* Pointer to the hash
   table. */
```

If labels or goto's are encountered during parsing, the addresses of code are put into the fixative structure. After successful compilation, addresses of goto's are fixed to the corresponding labels. Then the code is executed.

The goto statement can appear only in the level one (all statements that are compiled) statements. The goto cannot jump to a label across functions.

5.4 Parameter passing mechanism

Parameters are passed by reference [10]. The parameters are over the pointer BP ($BP+offset$), i.e. toward greater addresses. Local variables are below the BP ($BP-offset$), i.e. toward lesser addresses. The parameters are pushed onto the stack in the reverse order as they appear in the function call. The first parameter has offset less than the last parameter (the first parameter is closer to the BP). The last parameter is the deepest parameter on the stack.

Addresses of the parameters are stored in the stack. During compilation of the list of formal parameters it is leaving out the place for them in the stack. It causes different accesses to the parameters and the local variables, because local variables are stored by value (only one indirection, parameters need two indirections). Different virtual machine instructions must be generated for parameters and local variables.

5.5 Stack

The stack is used for storing of the addresses of parameters and for the local variables as it was mentioned in section 5.4. The location of parameters and local variables is depending on the level in which the function is called. The parameters and the local variables are destroyed after a return from the procedure.

5.6 Compiled statement

The statements mentioned on the level 1 (<list_stat >(see appendix 4)) are compiled. Procedures are also compiled. The procedures must be processed when they are called. The procedures usually consist of prologue, epilogue and body. The following instructions are in the prologue:

```
GEN_PUSHb;
GEN_PUSHt;
GEN_MOVt;
GEN_MOVbs;
GEN_SUBss(count);
```

It is namely: **PUSH** $[STACK] \longleftarrow BP$, **PUSH** $[STACK] \longleftarrow TMPH$, **MOV** $TMPH \longleftarrow TMP$, **MOV** $STACK \longrightarrow BP$ and **SUB** $STACK \longleftarrow (STACK-count)$ (see 5.1). The epilogue of the procedure is generated:

```
GEN_MOVsb;
GEN_POPt;
GEN_POPb;
GEN_RET;
```

These instructions are: $\mathbf{MOV}\ STACK \longleftarrow BP$, $\mathbf{POP}\ TMPH \longleftarrow [STACK]$, $\mathbf{POP}\ BP \longleftarrow [STACK]$ and **RET** return to the caller (i.e. the instruction **CALL**).

The procedure call is also compiled. The reason is that the list of the parameters can contain expressions.

Chapter 6

Different level of user interfaces to the interpreter Clif

6.1 Intrinsic functions

Intrinsic functions are linked to the compiler. The user should update two tables and write an intrinsic function in C or FORTRAN then to link the object code of the compiler with the tables and with the code of the intrinsic function. This way is enriched the compiler with a new intrinsic function. The structures of the tables are as follows:

```
/*
 * intrinsic.h
 *
 * List of intrinsic functions.
 */

#ifndef _INTRINSIC_H
#define _INTRINSIC_H

#include <math.h>
#include "myintrinsic.c"

#ifdef CONTROL
#include "example/apl.c"
#endif

typedef void (*INTRINSIC_FUNCTION) PROTO((char **));

struct INTR
{
char *name;
INTRINSIC_FUNCTION adr;
} intr_name[]=
{{"cclose", (INTRINSIC_FUNCTION)cclose},
         {"cgetc", (INTRINSIC_FUNCTION)cgetc},
 {"chclose", (INTRINSIC_FUNCTION)chclose},
 {"chflush", (INTRINSIC_FUNCTION)chflush},
 {"chopen", (INTRINSIC_FUNCTION)chopen},
 {"chwrite", (INTRINSIC_FUNCTION)chwrite},
         {"copen", (INTRINSIC_FUNCTION)copen},
         {"cputc", (INTRINSIC_FUNCTION)cputc},
         {"exp", (INTRINSIC_FUNCTION)exp_},
 {"fflush", (INTRINSIC_FUNCTION)cfflush},
         {"fprintf", (INTRINSIC_FUNCTION)cfprintf},
         {"fscanf", (INTRINSIC_FUNCTION)cfscanf},
 {"printf", (INTRINSIC_FUNCTION)cprintf},
         {"scanf", (INTRINSIC_FUNCTION)cscanf},
         {"sin", (INTRINSIC_FUNCTION)sin_}
#ifdef CONTROL
 ,
         {"green_func", (INTRINSIC_FUNCTION)green_func},
         {"csigpause", (INTRINSIC_FUNCTION)csigpause},
         {"open_termo", (INTRINSIC_FUNCTION)open_termo},
         {"read_termo", (INTRINSIC_FUNCTION)read_termo},
         {"write_termo", (INTRINSIC_FUNCTION)write_termo},
         {"close_termo", (INTRINSIC_FUNCTION)close_termo},
         {"init_wait", (INTRINSIC_FUNCTION)init_wait}
#endif
};



/*
 * Number of intrinsic functions.
 */
#define SIZE_REM sizeof(intr_name)/sizeof(intr_name[0])



/*
 * Size of an array of intrinsic functions.
 */
void (*(f[SIZE_REM])) PROTO((char **));
```

The names of the intrinsic functions are stored in hash table (mentioned earlier). The compiler knows about the names of the intrinsic function from its initializing part. When the remote keyword is parsed, remote functions are loaded into the hash table of remote functions. Only remote functions previously declared are known to the environment. If there are any declaration after remote statement the names are known but have no corresponding code. The example of the remote statement usage can be found in the 'io.ci'.

The body of remote function or declaration of the header respectively must be in the file 'myintrinsic.c'. Continuing in the above example the file 'myintrinsic.c' should look like:

```
double plus(double *a,double *b)
{
printf("a=%g\n",*a);
printf("b=%g\n",*b);
return(*a+*b);
}




double sin_(a)char **a;{return(sin(*(double *) a[0]));}
```

For the proper function of the compiler user should keep the following steps:

- Update the variable intr_name. The first field is a name of the intrinsic function; the second field is the name of the user supplied function.

- Write the code of a new intrinsic function in the file 'myintrinsic.c'.
- Compile remote call subsystem. Link new version of the compiler.

6.2 Main areas of allocated memory

The allocated memory of the interpreter is split to two parts. The first part is static. The user can not change the size of these memory areas at run-time, only by recompiling the interpreter. The size is as follows:

```
#define PAGE        512       /* Size of a page. */
#define SIZE_HAS   1999       /* Size of hash table. */
#define SIZE_HAS_LOC 401      /* Size of hash table for locals. */
#define SIZE_HAS_GOTO 257     /* Size of hash table for goto
 labels. */
#define MAX_IDENT    1999     /* Max number of variables */
#define MAX_IDENT_LOC  401    /* Max number of local variables. */
#define SIZE_ADR_STACK 256    /* Size of address stack (obsolete). */
#define SIZE_STRUCT_FIX 256   /* Size of the stack of fixative structures. */
#define SIZE_REMOTE 1999      /* Size of hash table for remote function
 names. */
```

Macros **SIZE_HAS, SIZE_HAS_LOC, SIZE_HAS_GOTO, MAX_IDENT, MAX_IDENT_LOC, SIZE_REMOTE** are primes according to [18]. If you want to change these macros you should change them to primes again for better performance of hash functions.

The second part is

```
#define SIZE_SPACE  210       /* Memory size in pages. */
#define SIZE_ARIT_STACK 7     /* Size of the arithmetical stack in pages. */
#define SIZE_TMP_STACK 52     /* Size of stack of temporaries in pages. */
```

Each size is specified by number of pages (PAGE 512 B). The user can change these sizes. The easiest way is to run the interpreter with the parameter /bc=<number>. E.g.

```
clif /bc=5
```

Starting the interpreter with argument 5 caused that the size of the memory, size of the arithmetical stack and size of the temporary stack are lowered to half compared to the default set (default is constant equal 10). The user can extend the size of the three memory parts if the interpreter is ran with the argument greater than 10.

6.3 Hash tables and hash function

The most important is that the size of the hash table must be a prime. We chose the primes 1999 for the size of the global hash table and 401 for the size of the local hash table.

We chose the following hash function:

```
/*
 * Hash function.
 */
static unsigned int
```

```
hash_code (s, size)
  char          *s;
  unsigned int size;
{
  int             c = 0;

  while (*s)
    {
      c = c << 1 ^ (*s);
      s++;
    }
  if (0 > c)
    c = (-c);
  return (c % size);
}
```

The core of the hash function is one bit shifting and then the result is XOR-ed with the ASCII value of the character. The problems may arise when the variable's identifier is longer than the size of the integer on the computer.

6.4 Adding new data type

The user has to follow these steps:

- Update the file 'cast.h' with cast operators for the new data type.
- Update the file 'geninstr.h' with appropriate instructions for the virtual machine. User can be inspired by previous types.
- Update the file 'type.h'. To define the constant of new data type.
- Enrich the file 'instr.h' with a new structure for a new instruction of the virtual processor if it is needed.
- Update the scanner in file 'ls.l' with the pattern matching of the new data type constant.
- Enrich the list in file 'keyword.gperf' with needed tokens name.
- Update the grammar of the language in file 'ys.y'. An user should be searching for previously defined data type (i.e. INT) and he/she has to take inspiration from the surrounding structures. Namely the following functions should be rearranged:
  - The function 'exe' ('virtual_machine.c'). Instructions of the virtual machine are executed in the function.
  - The function 'init' ('comp_maint.c'). Update the array 'pri' with the new data type.
  - The function 'implicit_cast' (comp_maint.c) contents all possible options of implicit cast instructions.
  - The function 'l_value_cast' (comp_maint.c) is casting to the <l_value >

Chapter 7

Interrupt services

As it was mentioned earlier two types of interrupts are implemented synchronous and asynchronous (see 1). Each of them call a specific service function. The interpreter is interrupted only if the virtual machine is running. Interrupt handling functions are system dependent. Therefore these functions should be specific for each platform. Current ports are on CD 4680 (file 'inter_handl_svr3.c'), DEC 5000/240 (file 'inter_handl_bsd.c'). There is a port for Linux (using BSD-like signals), SVR4 port (file 'inter_handl_svr4.c') and generic POSIX port (file 'inter_handl_posix.c')[16]. The configuration of ours CD 4680 is rather confusing, that is why we need specific interrupt handler for it. To further complicate things, the new terminal setting is added to these files as well. Because each platform to which we ported Clif has a specific terminal handling, this part is specially designed for them as well. Most of currently ported platforms can use the generic POSIX implementation as well. However, it arises two problems:

- For the file 'inter_handl_svr3.c', it cannot be compiled afterwards, because it knows the functions but only in SVR4 mode and the default is SVR3 (CD4680). If we switch to the default SVR4 some other functions are not known.
- Some of platforms need to push newline into the input stream and some do not. We use interrupt (DC4) to break the run of the virtual machine as well as to resume the run if it is pressed again. After pressing DC 4 for the second time, the resume statement is internally generated.

  In interrupt handler, on some platforms, there must not be a newline character (or is generated automatically by the terminal line discipline), on others, it must be pushed by interrupt handler function. See sections 7.1, 7.2.

Generally, user may use 'inter_handl_posix.c' whenever appropriate. If the interrupt does not function, the user should consider to add the following line to the interrupt_service just after assignment to handler variable

```
ioctl (handle_fd, TIOCSTI, "\n");
```

The following function registers interrupt handler:

```
#include <signal.h>
#include <termio.h>
#include <fcntl.h>
#include <setjmp.h>

#define OFF(x, y) (x) & (~(y))
#define ON(x, y) (x) | (y)
#define SPACE 0x20
#define NL '\n'

int handler = 0;
int handle_fd;
struct termio term,term_initial;

RETSIGTYPE (*interrupt_handler) (void);
RETSIGTYPE interrupt_service PROTO((void));
```

```
void interrupt_register PROTO((void));
void term_restore PROTO((void));

RETSIGTYPE fatal_handler PROTO((void));
void fatal_handler_register PROTO((void));
extern jmp_buf jmpbuf;
extern int error_count;

/*
 * Registers interrupt handler.
 */
void
interrupt_register ()
{
  interrupt_handler = interrupt_service;
  handle_fd = fileno (stdin);
  ioctl (handle_fd, TCGETA, &term);
  term_initial = term;
  term.c_cc[0] = 0x14; /* DC4 */
  term.c_cc[5] = 0x12; /* DC2 */
  term.c_lflag = OFF(term.c_lflag, LNEW_CTLECH); /* dalsi flag ktory ma pre nas vyznam je : term.c_lflag=ON
  ioctl (handle_fd, TCSETA, &term);
  sigset (SIGINT, interrupt_handler);
}
```

on the CD 4680. Via ioctl system call is reset interrupt signal to DC4. Via sigset system call is set the interrupt handler. On the DEC 5000/240 looks the function as follows:

```
/*
 * Registers interrupt handler.
 */
void
interrupt_register ()
{
  interrupt_handler = interrupt_service;
  handle_fd = fileno (stdin);
  ioctl (handle_fd, TCGETA, &term);
  term_initial = term;
  term.c_cc[0] = 0x14; /* DC4 */
  term.c_cc[5] = 0x12; /* DC2 */
  ioctl (handle_fd, TCSETA, &term);
  vec.sv_handler = interrupt_handler;
  sigvec (SIGINT, &vec, &ovec);
}
```

The interrupt handler is set here through the system call sigvec.

When the interpreter session terminates, the terminal is reset to the initial values:

```
/*
 * Restores setting of the terminal at the termination of Clif session.
 */
```

```
void
term_restore ()
{
  ioctl (handle_fd, TCSETA, &term_initial);
}
```

7.1 Synchronous interrupt service function

The function is as follows:

```
/*
 * Synchronous interrupt service.
 */
void
interrupt_service_sync ()
{
  handler = 1;
}
```

The virtual machine instruction **INTER** is calling the function interrupt_service_sync in which the handler is set to 1. Before the next instruction is executed, the handler is checked. If it is set the virtual machine switches its context. The context of the virtual machine is stored in the following structure:

```
struct CONTEXT
{
  char *bp;
  char *frame;
  char *kodp;
  char *kodp1;
  char *kodp2;
  char *kodp3;
  char *kodp4;
  char *pc;
  char *stack;
  char *tmp;
  char *tmph;
#ifdef FLEX_SCANNER
  void *state;
#else
  int (*input) PROTO((void));
#endif
  struct CONTEXT *previous;
};
```

There are no restriction on the use of the statements. The statement resume resumes after an interrupt.

7.2 Asynchronous interrupt service function

Not only synchronous interrupt is provided by the interpreter, but also asynchronous interrupt. Only the virtual machine can be interrupted. The user interrupts the virtual machine by pressing the DC4 key. As it is mentioned above there are no restrictions on the use of the statements. The user can interrupt more than once. The interpreter interrupt level can rise. To return to resume initially interrupted program, the user should specify equal number of the resume statements to the interrupts. If the interrupt level is again zero initially interrupted program continues.

The context of the virtual machine is stored in the same structure as for the synchronous interrupt. The asynchronous interrupt service function is the following:

```
/*
 * Asynchronous interrupt handler.
 */
void
interrupt_service ()
{
#ifdef DEBUG_INTER
  printfx ("interrupt\n");
#endif
  if ((clif_interrupt_level > 0) || (!virtual_machine_suspended))

/*
 * Test of the virtual machine running and the level of interrupt.
 * Interrupt is only accepted if the virtual machine is running.
 */

    {
#ifdef DEBUG_INTER
      printfx ("virtual machine is running, interrupt accepted\n");
#endif
      handler = 1;
    }
  return;
}
```

The function is valid for CDC. The asynchronous interrupt service function for DEC is as follows:

```
/*
 * Asynchronous interrupt service.
 */
RETSIGTYPE
interrupt_service ()
{
#ifdef DEBUG_INTER
  printfx ("interrupt\n");
#endif
  if ((clif_interrupt_level > 0) || (!virtual_machine_suspended))

/*
 * Test of the virtual machine running and the level of interrupt.
 * An interrupt is only accepted if the virtual machine is running.
 */
```

```
    {
#ifdef DEBUG_INTER
      printfx ("virtual machine is running, interrupt accepted\n");
#endif
      handler = 1;
#ifdef HAVE_TIOCSTI
      ioctl (handle_fd, TIOCSTI, "\n");
#endif
    }
  return;
}
```

Switching of the virtual machine to a new context is connected with switching of the interpreter input. The different function for each platform exists because of features of the stream on those platforms. The CDC stream contents interrupt character as well as newline character. The DEC stream only contents interrupt character. Therefore we must put into the DEC stream any character before the interpreter input is switched.

The interpreter input is switched to the standard input. Returning from the interrupt the interpreter input is switched to the initially set input.

7.3 Interpreter input functions

There are three interpreter input functions. The first function is initialized at the beginning of the interpreter session. Depending on the number of parameters of the run-string the interpreter input is either in run-string specified program or the standard input (function used in this case is input_komp). When the virtual machine process the instruction of synchronous interrupt or if the asynchronous interrupt happened the input is switched to the standard input (function input_std). If the virtual machine is interrupted by the user the input is switched to the input from the buffer (function input_buf). The input from the buffer is the same as it is by synchronous interrupt. After processing it the input is switched to the standard input. The mentioned input functions follow:

```
/*
 * input.c
 *
 * Initialization of the main compiler
 * and redefinition of its input functions.
 * Different input function are used during
 * synchronous and asynchronous interrupt
 * handling.
 */

#include <stdio.h>
#include <fcntl.h>
#include "global.h"
#include "lex_t.h"
#include "input.h"

extern FILEATTR pf, spf[];
#ifndef FLEX_SCANNER
extern int yylineno;
extern int yytchar;
extern char *yysptr, yysbuf[];

extern int getcx PROTO((FILE *));
static int buf_pointer = 0;
#else
extern FILE *yyin;
```

```
#endif

#define U(x) x

int (*input) PROTO((void));

char string_resume[]="resume;"; /* Buffer for input by return
 * from asynchronous interrupt.
 */
extern int no_compile_only; /* Flag in the case of errors or */
/* compile only. */
extern int handle_main; /* If set, compiler like behavior. */
extern int source_line_number;
/* Source line number. Detecting if */
/* the current line was already */
/* printed. Using in error messages. */
extern void exit_file_scope PROTO((void));

#ifndef FLEX_SCANNER
/*
 * Redefinition of the input for the main compiler.
 */
int
input_komp ()
{
#ifdef DEBUG_INTER
  printfx("in front of getc in input\n");
#endif
  yytchar=yysptr>yysbuf?U(*--yysptr):getcx(pf.fp);
#ifdef DEBUG_INTER
  printfx("behind of getc in input, read character %c - %x\n",
  yytchar, yytchar);
#endif
  if(yytchar == '\n') /* LF */
    {
      yylineno++;
      spf[s].line_counter++;
      char_counter = 0;
      line_buf[0] = 0;
    }
  if(yytchar == EOF) /* Is current char EOF? */
    {
      if (s > 0)
{
  spf[s].line_counter = 1; /* Counting lines from beginning. */
  source_line_number = 0; /* Resetting line number in the */
  /* presence of errors. */
  fclose (spf[s].fp);
  exit_file_scope ();
  s = s - 1;
  if (! s && ! no_compile_only)
    return 0;

  /* If we want compiler-like behavior, don't switch to
     stdin. */
  if (! s && handle_main)
    return 0;
  pf = spf[s]; /* Move to the next opened file. */
}
#ifdef DEBUG_INTER
      printfx ("in front of the second getc in input\n");
#endif
      yytchar = yysptr>yysbuf?U(*--yysptr):getcx(pf.fp); /* The first char */
#ifdef DEBUG_INTER
      printfx ("behind of the second getc in input, read character %c - %x\n",
       yytchar, yytchar);
#endif
    }
  return (yytchar);
}

/*
 * Redefinition of the input for the main compiler.
 * It is used during an interrupt.
 */
int
input_std ()
{
#ifdef DEBUG_INTER
  printfx ("in front of the third getc in input\n");
#endif
  yytchar = yysptr>yysbuf?U(*--yysptr):getcx(stdin);
#ifdef DEBUG_INTER
  printfx ("behind of the third getc in input, read character %c - %x\n",
   yytchar, yytchar);
#endif
  if (yytchar == '\n') /* LF */
    {
      yylineno++;
      spf[s].line_counter++;
      char_counter = 0;
      line_buf[0] = 0;
    }

#ifndef NOT_MSWIN_AND_YES_DOS
  if (HANDLER_TEST)
    {
      HANDLER_SET;
      input = input_buf;
      buf_pointer = 0;
      return (input_buf());
    }
#endif
  return (yytchar);
}

/*
 * Redefinition of the input for the main compiler.
 * It is used by return from an interrupt.
 */
int
input_buf ()
{
  yytchar = yysptr>yysbuf?U(*--yysptr):string_resume[buf_pointer++];
  if (yytchar == '\n') /* LF */
```

```
    {
      yylineno++;
      spf[s].line_counter++;
      char_counter = 0;
      line_buf[0] = 0;
    }
  return (yytchar);
}

#else


int
terminate_buffer ()
{
  fclose (spf[s].fp);
  spf[s].fp = NULL;
  spf[s].name = NULL;
  spf[s--].line_counter = 1;
  source_line_number = 0;
  exit_file_scope ();

  /*  Do not switch to stdin, if the user wants compiler-like
      behavior.  */
  if (! s && handle_main)
    return 1;

  return (! s && ! no_compile_only);
}

#endif /* FLEX_SCANNER */

/*
 * Initialization of the main compiler input.
 */
int
init_input (argc1, argv1)
  int argc1;
  char *argv1[];
{
  int b;

#ifndef FLEX_SCANNER
  input = input_komp;
#endif
  s = argc1 - 1;
  spf[0].fp = stdin;
  spf[0].name = "stdin";
  spf[0].line_counter = 1;
  for (b = 1; b < argc1; b++)

/* File opening and storing their pointer. */

    {
      spf[argc1 - b].fp = fopen(argv1[b],"r");
      spf[argc1 - b].name = argv1[b];
      spf[argc1 - b].line_counter = 1;
      if (spf[argc1-b].fp == NULL)
{
  s = argc1 - b;
  error_message (7001);
  return (-1);
}
    }
/*
 * Takes the first file pointer from the stack.
 */
#ifdef FLEX_SCANNER
  yyin = spf[s].fp;
#else
  pf.fp = spf[s].fp;
#endif
  return 0;
}
```

Chapter 8

Graphic interface

As was mentioned earlier there are the four intrinsic functions for programming graphics output channels. There are the following: chopen(), chwrite(), chflush() and chclose(). In addition, these functions are used as a graphical interface from user point of view. These functions are calling graphic primitives functions. The graphics primitives functions are the following: window(), move(), draw() and draw_point().

8.1 Graphic primitives

8.1.1 Function window()

The function is used in creating the window that matches the scope of displayed variable as well as the size of the window in pixels. The function follows:

```
/*
 * Creates a window from user's coordinates.
 * Counts world coordinates.
 * handle - is handle of the window
 * n - specifies line in the window (user can have more lines in the window)
 * rest are coordinates
 */
static void
window (handle, n, x_left, y_down, x_right, y_up)
     int handle, n;
     double x_left, y_down, x_right, y_up;
{
  channel[handle].member[n].ax =
    (channel[handle].w_resolution[0] - 1) / (fabs(x_left - x_right));
  channel[handle].member[n].ay =
    (channel[handle].w_resolution[1] - 1) / (fabs(y_down - y_up));
}
```

The handle is the handle of the channel, the n specifies the displayed variable, the x_left, y_down, x_right and y_up specify scope of the n-th displayed variable.

8.1.2 Function move()

The function is used for moving cursor to the specified position. It looks like follow:

```
/*
```

```
 * Moves cursor to the specified position.
 * handle - is handle of the window
 * n - specifies line in the window (user can have more lines in the window)
 * x, y - coordinates
 */
static void
move (handle, n, x, y)
     int handle, n;
     double x, y;
{
  d_move((int)floor ((x - channel[handle].start_time)
     * channel[handle].member[n].ax),
 channel[handle].w_resolution[1] - 1
 - (int)floor ((y - channel[handle].member[n].lower)
       * channel[handle].member[n].ay));
}
```

where handle is the handle of the channel, n specifies the displayed variable, x and y specify the position in the window to which cursor is moved.

8.1.3 Function draw()

The function draw() is the following:

```
/*
 * Draws a line from the cursor position to the point of coordinates.
 * handle - is handle of the window
 * n - specifies line in the window (user can have more lines in the window)
 * x, y - coordinates
 */
static void
draw (handle, n, x, y)
  int handle, n;
  double x, y;
{
  d_draw (handle, n,
  (int)floor ((x - channel[handle].start_time)
      * channel[handle].member[n].ax),
  channel[handle].w_resolution[1] - 1
  - (int)floor ((y - channel[handle].member[n].lower)
* channel[handle].member[n].ay));
}
```

8.1.4 Function draw_point()

```
/*
 * Draws a point from the coordinates.
 * handle - is handle of the window
 * n - specifies line in the window (user can have more lines in the window)
 * x, y - coordinates
 */
```

```
static void
draw_point (handle, n, x, y)
  int handle, n;
  double x, y;
{
  d_point (handle, n,
   (int)floor ((x - channel[handle].start_time)
       * channel[handle].member[n].ax),
   channel[handle].w_resolution[1] - 1
   - (int)floor ((y - channel[handle].member[n].lower)
 * channel[handle].member[n].ay));
}
```

Chapter 9

Internal representation of types

9.1 Representation of a type

The proper internal representation structure has to uniquely answer the following questions:

- array, simple variable, function with or without exporting parameter types
- function type (distance):
  - global
  - local
  - intrinsic
  - remote
  - ⋮
- arithmetic class:
  - integer
  - double
  - ⋮
- if it is an array - list of dimensions
- if it is a function:
  - list of parameter type specifiers (optional)
  - list of formal parameters (optional)
- if it is a function - it is a definition or declaration (body flag)
- address
- if it is a function definition, it was already formally called (formal call backpatching)

We would like to have an internal structure that is much wider then the actual type possibilities in C. But the semantic actions are only defined for the subset of C-like type definitions and declarations. Therefore we chose the following structure:

```
/* Internal type structure. */
struct internal_type
{
  struct internal_type *input;
  struct internal_type *arity;
  char *field_name;
```

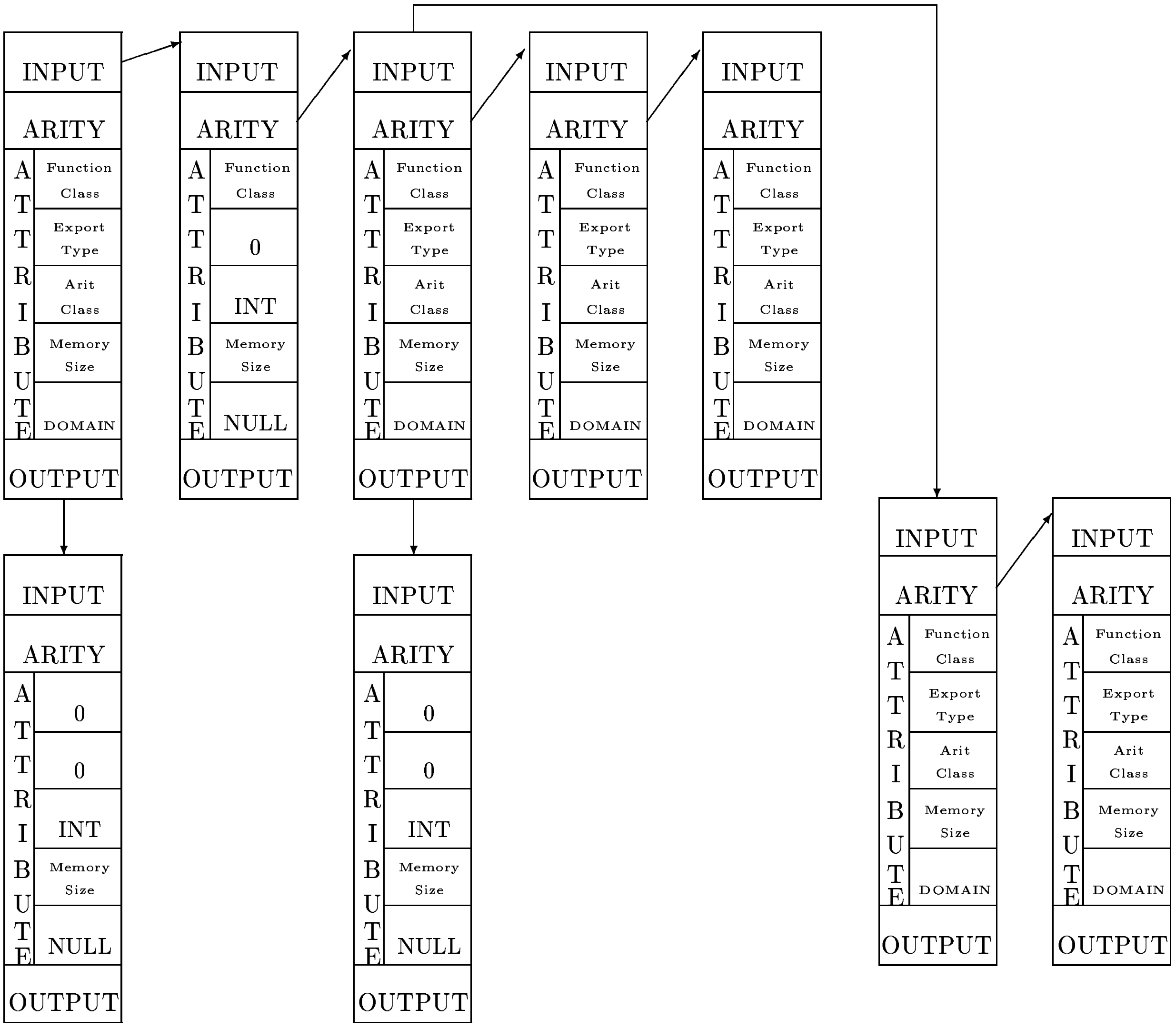


Figure 9.1: Representation of the type declaration int a(int b, int c(.....), ...);

```
  int offset;
  struct attr attribute;
  struct internal_type *output;
};
```

Representation of the type declaration:

```
int a(int b, int c(.....), ....);
```

is in the figure 9.1. The structure member attribute is a substructure. The structure member output is pointer to the structure internal_type.

Structure attr has the following form:

```
/* Type attribute. */
struct attr
{
  enum intern_func_class function_class;
```

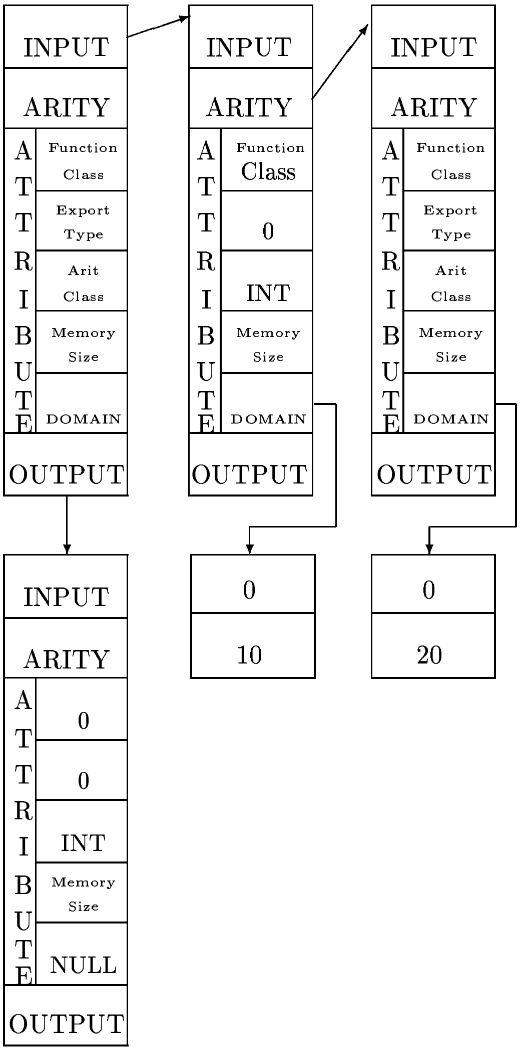


Figure 9.2: Representation of the type declaration int a[10][20];

```
  int export_type;
  enum type_qual type_qualifier;
  enum storage_class_specifier storage_class_specifier;
  enum intern_arit_class arit_class;
  int memory_size;
  char *domain;
};
```

The function_class can be simple variable, matrix, intrinsic function, function, remote function, ... (full listing of possibilities can be found in 'type.h' file). The domain pointer has the type specific form. For example for the type declaration `int a[10][20];`, the internal structure is in the figure 9.2.

In the figure 9.1, there is a general form of representation of a type declaration. Typically, the C-like declarations and definitions have other forms, i.e. there are not declarations of functions in a function declaration. Therefore, the n-ary tree pointed out in the figure 9.1 is simplified to the list of identifiers or binary tree 9.3, respectively.

9.2 C-language subset of type

Some typical patterns are in figures 9.4, 9.5, 9.6.

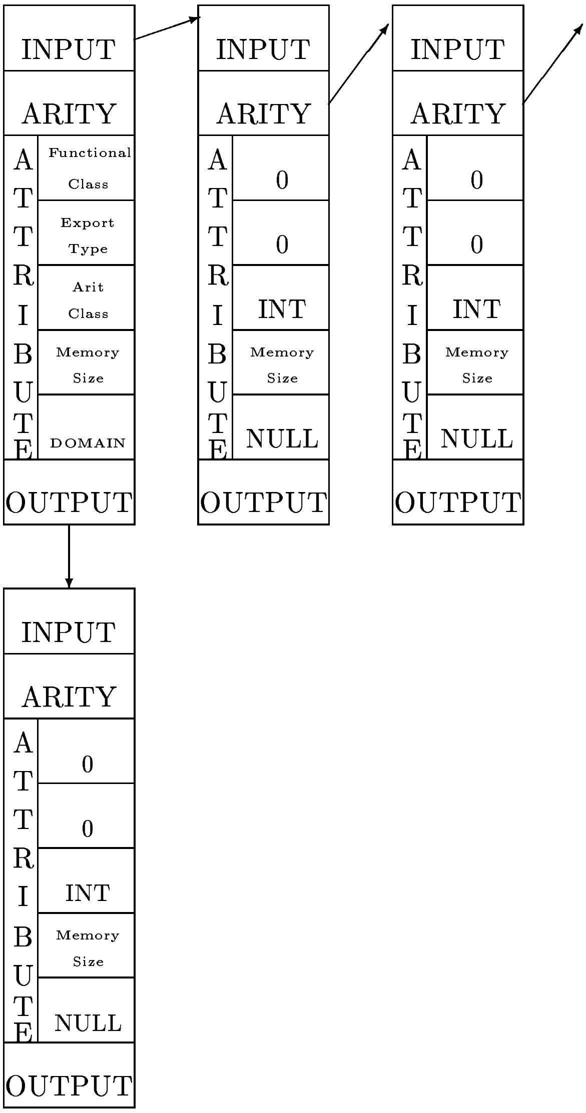


Figure 9.3: Representation of typical C-like function declaration.

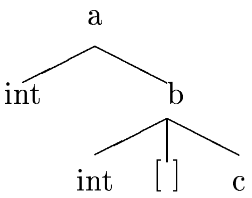


Figure 9.4: Tree pattern of the declaration `int a(int b[ ],int c);`

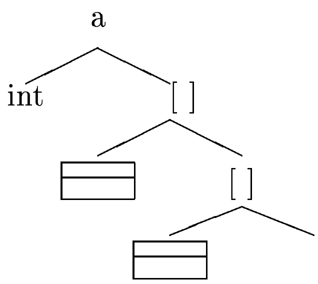


Figure 9.5: Tree pattern of the declaration `int a[ ][ ][ ];`

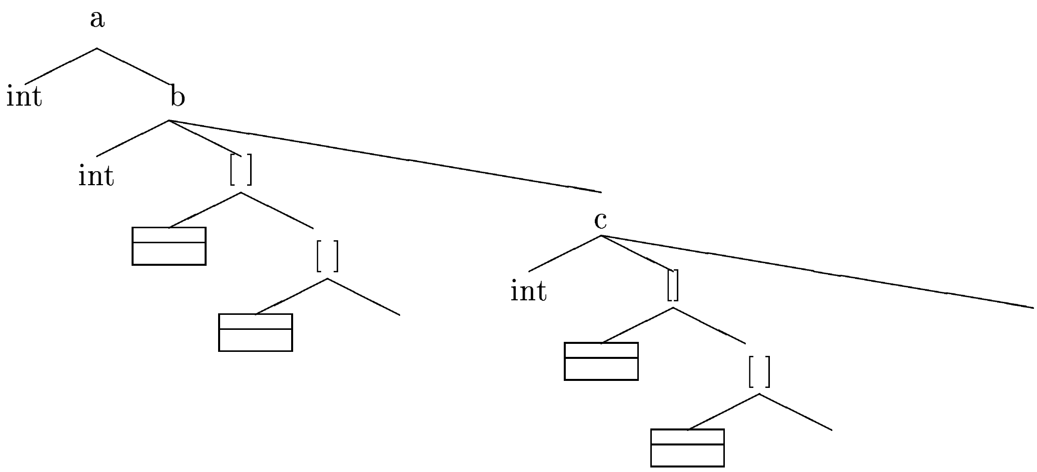


Figure 9.6: Tree pattern of the declaration `int a(int b[ ][ ]..., int c[ ][ ]...,...);`

In the following figures, there are some internal representations of C-language types (fig. 9.7, 9.8, 9.9, 9.10, 9.11). The rest of this chapter describes considerations about different language constructions that are using internal_type structures, but are no necessary fully implemented.

9.2.1 Internal representation of typedef

The typedef type has the function_class *typedef*. Where the typedef is used typically only the part where the first output points is taken and this is the internal type definition of the typedef (see fig. 9.7).

9.2.2 Internal representation of pointer

The pointer type has the function_class *pointer*. The level of indirection is equal to the number of the internal_type structures (see fig. 9.8).

9.2.3 Internal representation of enumeration

The enumeration type has the function_class *enumeration*. The type of enumeration members are integer (see fig. 9.9).

9.2.4 Internal representation of structure

The structure type has the function_class *structure*. The type of structure fields is the same as for simple variables or arrays (see fig. 9.10).

9.2.5 Internal representation of union

The union type has the function_class *union*. Each field of the has its own subtype (see fig. 9.11).

9.3 Coding of C-language subset of type and export type

Note: not implemented yet.
There is a bit for input and a bit for output. The both bits indicate present of input and output structure, respectively. The first 32 bits after current 32 bits is an output member. The structure member domain indicate how many 32-bit words are reserved for the domain information. Currently, domain information are 2 32-bit words for arrays and a 32-bit word for enumeration type. The structure member arit_class defines an atomary type. The structure member export_type indicates if the type should be exported. The structure member function_class has only two options local and remote, now. The structure member arity indicates the distance in number of 32-bit words to the continuing of the structure.

We propose this arrangements of bits:

bit	# of bits	name
0	(1)	input
1	(1)	output
2-4	(3)	domain
5-7	(3)	memory_size
8-13	(6)	arit_class
14	(1)	export_type
15-19	(5)	function_class
20-29	(10)	arity
0-29	(30)	$\sum$

We use 30 bits of each 32 bit word. 2 bits are left unused.

Example of `extern int a(int b, int c[][10], ....)...` is in the figure 9.12 and bit representation in the following table (note that [] is internally represented as -2, i.e. 0xfffffffe):

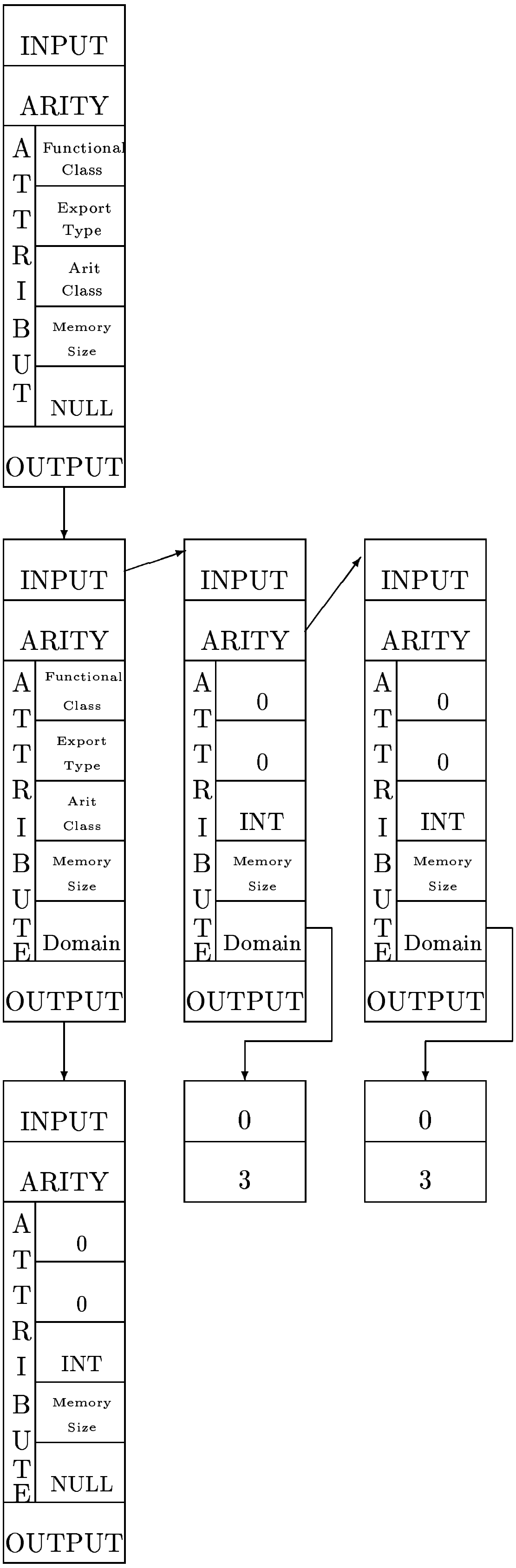


Figure 9.7: Internal representation of `typedef int c[3][3];`

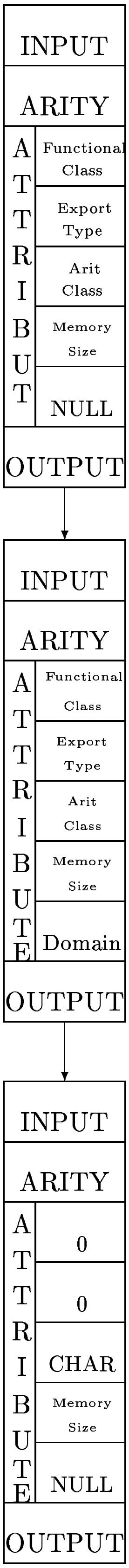


Figure 9.8: Internal representation of `char **a;`

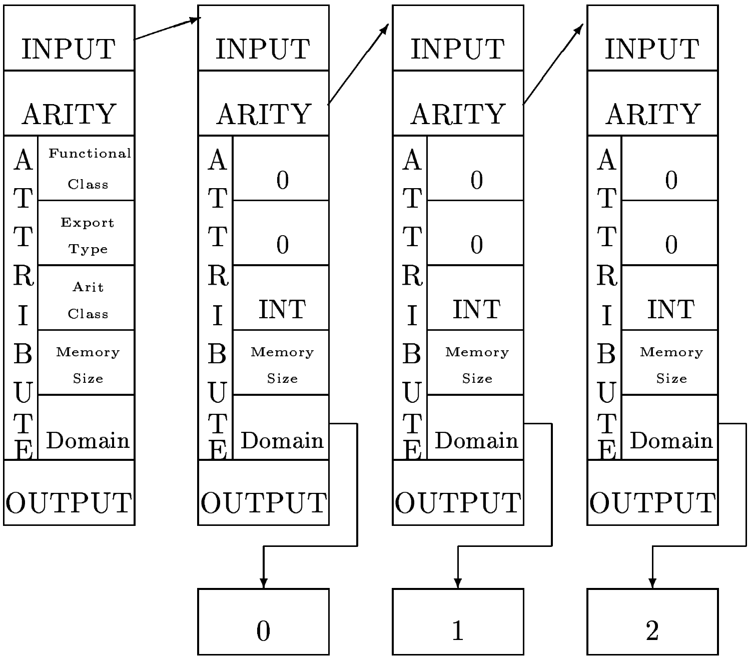


Figure 9.9: Internal representation of enum num { one, two, three };

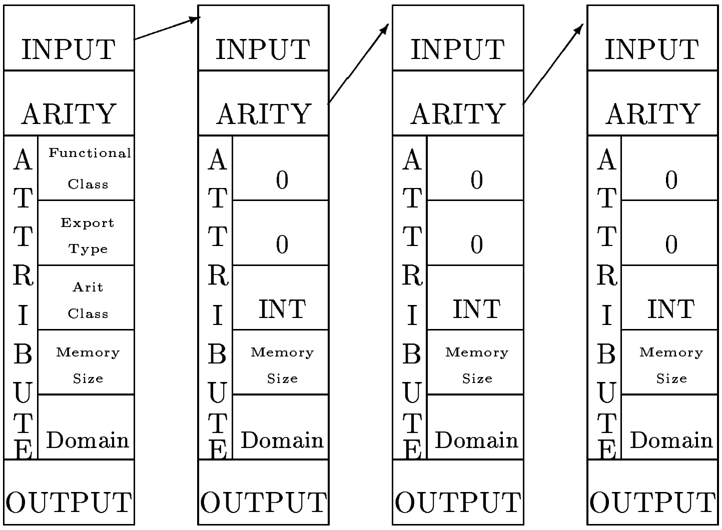


Figure 9.10: Internal representation of struct a { int a, int b, int c};

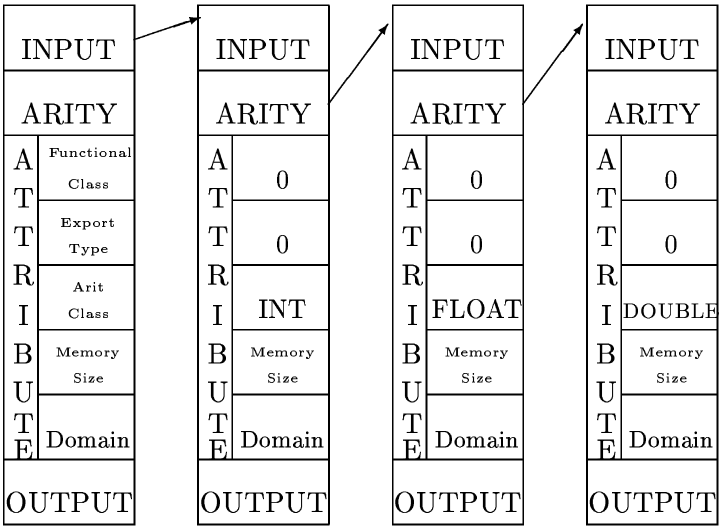


Figure 9.11: Internal representation of union a { int a, float b, double c};

31							24									15								7							0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
															...																

9.4 Notes to the implementation of internal types

Each type is copied when it is defined in structure, union, or enumeration. It is copied only in one level (not recursive). The reason is that the field_name is assigned to the types in aggregate. Each field has to have its own representation of the type.

For deallocation of types, cyclic definition must be checked to avoid endless loops.

Pointers to aggregates should be the same. There is only one instance of each type. If the similar aggregate is declared more than once it is considered to be not compatible. (In the figure 9.13, it is a real example of the following structure definition:

```
struct d {
  struct c *a;
  struct d *b;
};
```

where the c structure is:

```
struct c {
  int a;
  int b;
};
```

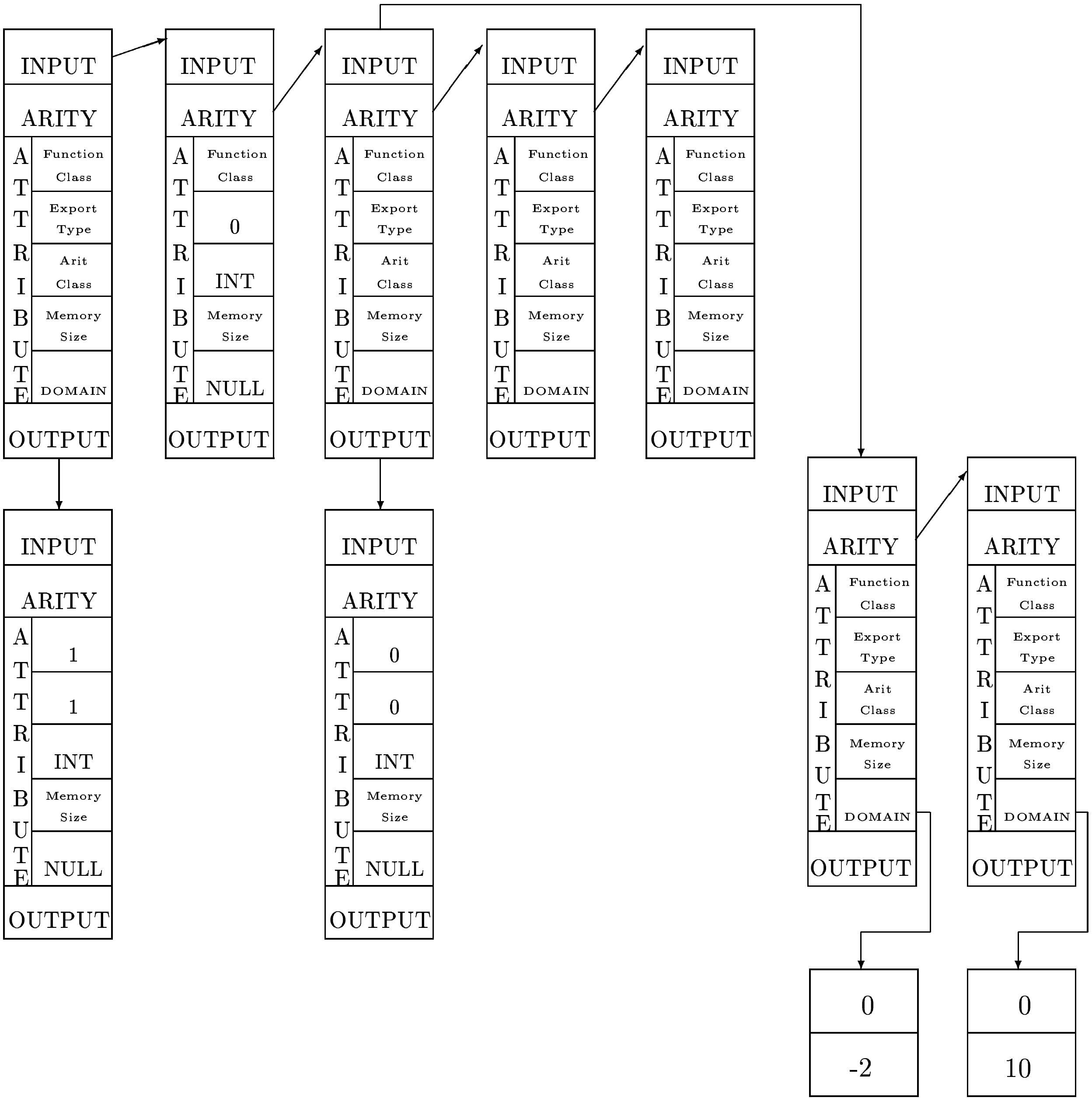
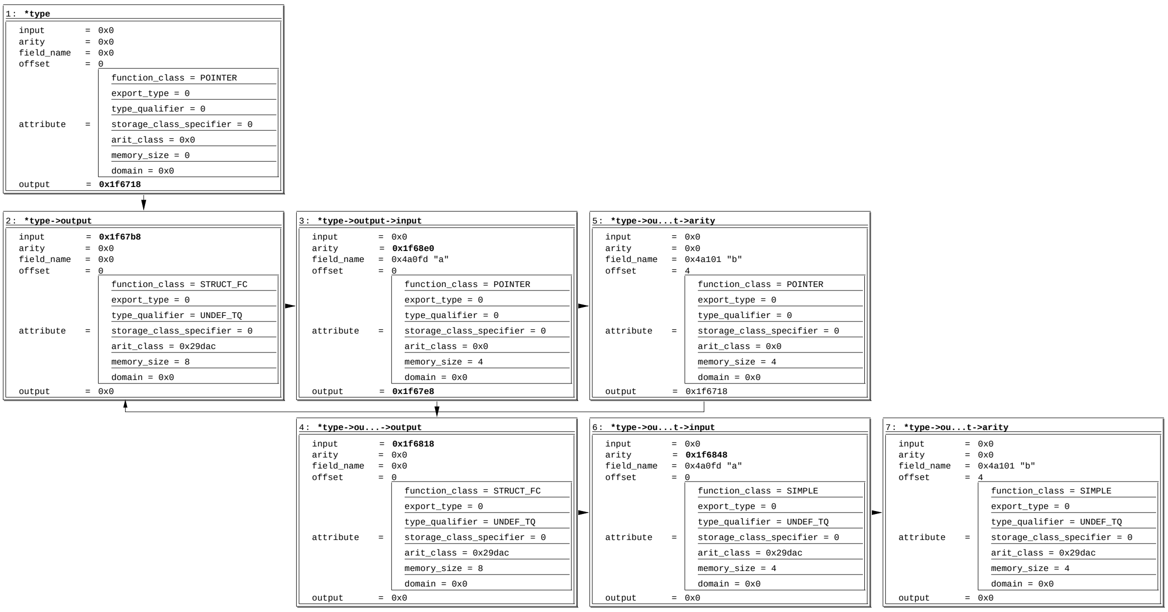


Figure 9.12: Internal type representation of function in form extern int a(int b, int c[][10], ...)...



```
1: *type
  input        = 0x0
  arity        = 0x0
  field_name   = 0x0
  offset       = 0
                   function_class = POINTER
                   export_type = 0
                   type_qualifier = 0
  attribute    =   storage_class_specifier = 0
                   arit_class = 0x0
                   memory_size = 0
                   domain = 0x0
  output       = 0x1f6718

2: *type->output
  input        = 0x1f67b8
  arity        = 0x0
  field_name   = 0x0
  offset       = 0
                   function_class = STRUCT_FC
                   export_type = 0
                   type_qualifier = UNDEF_TQ
  attribute    =   storage_class_specifier = 0
                   arit_class = 0x29dac
                   memory_size = 8
                   domain = 0x0
  output       = 0x0

3: *type->output->input
  input        = 0x0
  arity        = 0x1f68e0
  field_name   = 0x4a0fd "a"
  offset       = 0
                   function_class = POINTER
                   export_type = 0
                   type_qualifier = 0
  attribute    =   storage_class_specifier = 0
                   arit_class = 0x0
                   memory_size = 4
                   domain = 0x0
  output       = 0x1f67e8

4: *type->ou...->output
  input        = 0x1f6818
  arity        = 0x0
  field_name   = 0x0
  offset       = 0
                   function_class = STRUCT_FC
                   export_type = 0
                   type_qualifier = UNDEF_TQ
  attribute    =   storage_class_specifier = 0
                   arit_class = 0x29dac
                   memory_size = 8
                   domain = 0x0
  output       = 0x0

5: *type->ou...t->arity
  input        = 0x0
  arity        = 0x0
  field_name   = 0x4a101 "b"
  offset       = 4
                   function_class = POINTER
                   export_type = 0
                   type_qualifier = 0
  attribute    =   storage_class_specifier = 0
                   arit_class = 0x0
                   memory_size = 4
                   domain = 0x0
  output       = 0x1f6718

6: *type->ou...t->input
  input        = 0x0
  arity        = 0x1f6848
  field_name   = 0x4a0fd "a"
  offset       = 0
                   function_class = SIMPLE
                   export_type = 0
                   type_qualifier = UNDEF_TQ
  attribute    =   storage_class_specifier = 0
                   arit_class = 0x29dac
                   memory_size = 4
                   domain = 0x0
  output       = 0x0

7: *type->ou...t->arity
  input        = 0x0
  arity        = 0x0
  field_name   = 0x4a101 "b"
  offset       = 4
                   function_class = SIMPLE
                   export_type = 0
                   type_qualifier = UNDEF_TQ
  attribute    =   storage_class_specifier = 0
                   arit_class = 0x29dac
                   memory_size = 4
                   domain = 0x0
  output       = 0x0
```

Figure 9.13: Recursive structure declaration

Chapter 10

Files of the Clif

The following sections describe the goal of some important files.

10.1 File 'allocx.c'

Basic allocation functions.

allocate (size, pool) memory of size size, from the pool pool. Return pointer to the allocated memory.

deallocate (pool) memory of pool. It is stored to list and if the memory is needed again it is simply taken from this list.

init_zero (pointer, size) memory pointed to by pointer is initialized to zeroes.

callocx

mallocx

reallocx just wrappers for allocation.

There are only two different pool now: **PERM** and **BLOCK**.

10.2 File 'comp_maint.c'

Catch all file.

info is printed if verbose is specified on the command line.

clif main function. It parses command line options, initializes memory, sets proper input function for scanner and starts parser.

brfix break fixation.

cofix continue fixation.

retfix return fixation.

fix_cont_w fixation of continue in while.

fix_cont_f fixation of continue in for.

fix_break_w fixation of break in while.

fix_break_f fixation of break in for.

fix_ret Backpatching of the return address for return in function.

fix_break_s Backpatching of the address for break in for loop.

init initialization of Clif.

implicit_cast check if an implicit cast in an expression is needed.

l_value_cast check if a cast to <l_value > is needed.

pointer_cast check for pointer casts in <l_value > is needed. Issues appropriate error message.

array_subscript initialization of the dim variable with subscripts according to the array definition.

put_array_subscript store subscripts in internal_type structure.

ERR_NO_INFO

ERROR_INFO

ERROR_FULL_INFO formatting macros for error message.

error_message according to the number send, print the error message.

print_file_name

print_source_line

print_error_number

print_line_number functions formatting error messages.

type2string converts internal type to string representation for error messages.

type_transform transformation of type. It is used when the intrinsic (remote) functions are called.

add_to_spec_list create list of specifiers of function declaration and/or definition to the internal_type structure. This function is used during structure definition as well.

add_to_ident_list create list of identifiers during the function declaration and/or definition.

compare2trees compare two trees for equality. Check if the definition of the function corresponds with its prototype.

copy_type copy internal_type structure. It is used during function declaration and/or definition and structure definition. We need a copy because the type can be slowly added, i.e. on one line can be more identifiers such as simple variables, arrays, function prototypes and all of them have slightly different representation.

add_subs_to_spec_list create internal_type for function declaration if only type specifier and range of array were in the function prototype.

clear_internal_type clear internal type.

search_duplicate_labels after each new label declaration we need to check for duplication.

add_constant_to_list create list of constants of switch.

add_default_to_fixp the same as above but for default label; check for only one default.

func_def_s Functions for setting and resetting of the flag of function definition or declaration context.

enter_scope

exit_scope functions changing scope (hash tables of locals) upon nesting level.

mov2lvalue

move2lvalue generate proper **MOV** instruction.

type_compare compare if pointers are compatible.

cyclic_def find cycles in pointer definitions and declarations.

put2table Combination of all hashing tables. Choose the appropriate one and put the variable there.

get_num_args Number of args for a function is returned. This value is checked against number of actual parameters, if they are equal.

promote_type C-like type promotion.

start_main Arrange all things to start main function. Simulate a compiler like behavior.

check_spec_constr Checks if specified type specifiers are valid. If not issues an error message.

10.3 File 'control.h'

See section 5.3 for full detail.

10.4 File 'define.h'

See section 6.2 for full detail.

10.5 File 'dbg-out.c'

Debugging information generation and printing. The following functions are a simple interface for symbolic debugging.

store_pos Store line number and code pointer in the list. These data are only ones necessary during symbolic information output.

dbg_create Create a list of file names with pointers to the list of line number and code pointer (see above).

get_pos Restore line number from the list of code pointer and line number. The line number is chosen appropriately comparing to the current value of the program counter.

dbg_print Print the currently executed line.

10.6 File 'geninstr.h'

The implementation of generation of virtual machine instructions. The instructions are described in section 5.1.

10.7 File 'input.c'

input_komp the general input for compiling.

input_std this input function is used during the processing of an asynchronous or synchronous interrupt (see 7).

input_buf during the processing of the compiler interrupt after the interrupt key is pressed again this input function sends to the compiler a keyword resume; the compilation resumes and the compiler input is reset to the input_komp function.

init_input this function opens files and stores the pointers to the globally known array. The files could be specified on the command line and/or in the 'clif.ini' file.

terminate_buffer in flex, close file, reset line counter, reinitialize file attribute structure and exit file scope.

Note: The option to switch among several input functions and enable synchronous and asynchronous interrupts must be specified during the compilation of the Clif (see 1).

10.8 File 'instr.h'

This file is declaration of different structures that are used in the virtual machine. Detailed description can be found in section 5.1 (see table 5.1). Each field of structures is integer if it is not stated otherwise.

struct OPERAND_0_ma only major.

struct OPERAND_0_mi major and minor.

struct OPERAND_1_ma major and address as a char pointer.

struct OPERAND_1_mi major, minor and address as a char pointer.

struct OPERAND_1_i major, minor and number (immediately) integer.

struct OPERAND_1_id major, minor and number of type double (immediately double).

struct OPERAND_1_if major, minor and number of type float (immediately float).

struct OPERAND_1_ic major, minor and number of type char (immediately char).

There are definitions of instructions major constants in the file as well.

10.9 File 'keyword.gperf'

The file is a template that has a list of keywords. The C source file is produced by `gperf -t -p -k 2,3 ./keyword.gperf` command.

10.10 File 'parser.h'

param_flag the variable is set during the parsing of parameters to function. Parameters have no such strong constraints as if they are used in expressions. For example, as a parameter can be used array name (pointer).

atom_type_flag variable that specialized if it is a pointer which is pushed as a parameter to a remote function (pointer to the array) or only a dereferenced variable of the type. It is used only for **REMOTE_F** type of function.

variable array of structures. The structure members are address, offset and name. The address (is a pointer to char) of the global variable currently on the stack. The offset (integer) is an address of local variable currently on the stack. It is offset to the current value of the **BP**. The name field is the name of the variable. It is an array because we have to know addresses of all variables currently in the expression. Depending on the context, either address or offset are non NULL, nonzero respectively. If the variable is global, the address is non NULL. If the variable is local, the offset is nonzero. If the local variable has storage class specifier static the address is non NULL and the offset is zero.

it_is_in_case It is always set when the case statement is compiled.

jmp1 Variable for fixing the beginning of code in for loop. The description can be found in section 5.3 and fig. 5.2.

subscript_flag The flag is set after the first subscript of an array was parsed. The first subscript needs special treatment. The variable is an array, i.e. subscript_flag has each variable currently on the stack. It is cleared by macro **TYPE_CLEAR**.

is_address It is set when the address operator was used in the current expression.

struct_union_field The variable is set when structure or union identifier is parsed. If there are more dots or pointer operators in the variable additional code must be generated. Each variable on the stack which currently parsed in expression has a struct_union_field. This variable is reset by **TYPE_CLEAR**.

or_jmp

and_jmp both are used for generation of jumps in expressions with || and && operands. || expression is evaluated until a subexpression is true. && is evaluated until a subexpression is false.

main_defined set when the main function is defined.

initialize_only set when there is a need for execution of initialization code of variables.

full_bracketing Flag for proper code generation for fully bracketed initialization.

aggregate_memory_size During initialization, the size of subtypes are remembered. The right number of initializers is checked. If the initialization was incomplete, zeros are added.

TYPE_CLEAR macro that resets the following variables: subscript_flag, struct_union_field, type, type_com

SET_ADDRESS macro that resets variable and offset variables.

FUNCTION_PROLOGUE it generates code for function prologue.

FUNCTION_EPILOGUE it generates code for function epilogue.

DELETE_SUBSCRIPT runs code for deleting dimensions. The variable is used for generating proper offset of array variable in parsing expression. The dimensions stored in this variable are needed for computing of map function and map instructions of virtual machine for variable of the type array.

OFFSET_LAST_ADD generates last instructions after variable dereferencing structure and/or union was parsed. The special code is needed when the variable has more than one dereferencing operator.

OFFSET_LAST_ADD_ARRAY generates instructions after variable dereferencing structure and/or union was parsed and the last dereference was an array. The special code is needed when the variable has more than one dereferencing operator.

ERROR_P if there were errors don't do semantic, just check syntax.

MOV_P test if the last generated instruction was **MOV**.

POPA_P test if the last generated instruction was **POPA**.

PUSHA_P test if the last generated instruction was **PUSHA_P**.

RESET_CODE_GENERATION_BEGINNING do not fill up the memory continuously. Reset to beginning of the memory, if possible.

SET_CODE_GENERATION_BEGINNING if variables are defined, set new memory beginning.

10.11 File 'pso.c'

Parser support functions.

lookup_tables lookup in tables of local variables, if the variable is not found in the table of global variables. Setting of addresses for code generation.

typedef_copy internal type for typedef must be copied, if there is no type specifier directly following the typedef (considering internal type tree (see fig. 9.7)), i.e. pointer, array, function, array, struct, etc.

set_memory_size for cast operator the memory size must be set.

10.12 File 's-conv.c'

Functions in the file manipulate format strings and check arguments to printf and scanf class of functions.

s_conv checks format string and emits error messages.

store_arg_type Types of arguments are stored in the array for later checking.

compare_format_args Checks if the argument is compatible with the specified format.

10.13 File 's-conv.h'

PRINTF_P

SCANF_P checks for printf and scanf class of function call parsing.

10.14 File 'tables.c'

Functions in this file set, process, change and clear information in hash tables.

hastab see 5.2

hastab_goto see 5.3.

hastab_type see 5.2.

typetab see 5.2.

identtab see 5.2.

identtab_loc see 5.2.

pi counter of globally declared variables, used as a subscript to the identtab array.

pi_type counter of globally declared types, used as a subscript to the typetab array.

integer_cons default integer for arit_class (see 9.1).

doub_cons default double for arit_class (see 9.1).

flt_cons default float for arit_class (see 9.1).

chr_cons default char for arit_class (see 9.1).

vid_cons default void for arit_class (see 9.1).

type the type used in expressions to control the generation of proper instructions. The variable has the same type as a arit_class in the internal type structure.

hastab_init initialization of the hash table for global variables.

hastab_goto_init initialization of goto label table.

identtab_init initialization of the table of global identifiers.

allocate_hastab_loc allocation of the hash table of local variables.

allocate_loc_tables allocation of table of identifiers of local variables. It is used each time a new block is entered.

clear_hash_tab_declaration

clear_hash_tab

clear_hash_tab_next_declaration

clear_hash_tab_next functions for initialization of different parts of hash table. These functions are used after block is exited.

point returns the address of global variable or null if it is not declared. It prints an error message as well.

point_call It is used

- to find if the identifier has a function type
- to fix the address (if the (formal) call was done prior to the definition of the function)
- to add the address to the list of addresses where to fix later.

has

putstruct the functions are used for storing global variables into the hash table. The allocation of the memory is also done in the functions.

putstruct_body sets appropriate variables that the function was defined (not only declared).

lookup returns pointer to the hash table where the variable is stored.

hash_code see 5.2.

hash_code_loc the same as above but for local variables.

point_loc returns pointer to the hash table of local variables.

has_loc

putstruct_loc functions are used for storing local variables into the hash table.

lookup_loc returns pointer to the hash table where the local variable is stored.

add_spec_to_has to the function, the specifiers (type information) of parameters are added. If the function was already declared it finds out if declaration and definition (or two declarations) match.

add_ident_to_has the information in hash table about function identifiers of formal parameters is added. If the function was already declared it finds out if declaration and definition (or two declarations) match.

link_function sets addresses of remote functions.

set_value gathers information for checking sets of local variables (used for issuing warnings). The information is checked after compilation of a function and/or block.

fix_and_clear_goto_table scans the goto label table; fix addresses of gotos. This is processed after the return to the level zero of parsing (just before the virtual machine runs).

has_goto adds a label into the hash table if it was not already in. It adds address of the goto to the list of gotos.

has_label adds a label into the hash table. If it was in, issues an error message. It adds address of the label to the internal representation.

lookup_goto_table returns address of the hash table where the label is stored.

hash_code_goto hash function for goto labels.

align_memory used for allocation of variables. Each type has to be properly aligned.

scope_offset_get

scope_offset_set Functions set and get global offset (valid for whole table). The functions are used if nesting scope is greater than zero, i.e. at least two different blocks exist.

move_offset_aligned adjusting offsets according to size of type.

lookup_type returns address of the hash table where the type is stored.

has_type

putstruct_type process tags and type names. Information is put into the hash table.

putstruct_type_body allocates space for the type if it is necessary.

add_spec_to_type adds information about type specifiers of struct, union or enum fields to the hash table.

add_ident_to_type adds identifiers of struct, union or enum fields to the hash table.

allocate_var allocates memory for global variable.

allocate_struct counts memory size of structures.

find_member returns offset in the currently parsed structure (dereference of it is parsed) or -1 if name is not a member.

offset_aggregate_member just a wrapper above the former function.

allocate_aggregate counts memory size of structures (calls allocate_struct) and aligns offsets of aggregate fields.

putstruct_static Static variables are put into tables.

enter_file_scope

exit_file_scope have special meaning. They are used in file scope static variables.

typedef_p If the name is declared as typedef return it as TYPENAME token - true (1); IDENT - false (0), otherwise.

get_declaration_line Returns the declaration line of a variable. Called from error_message function.

memory_size During initialization of variables, walk recursively down the type and set the size of the subtype of an aggregate.

check_init_bracket The right initialization bracketing has the same number of brackets as is the number of subtypes. The function checks the number of subtypes. The return value is in level variable.

get_memory_size For the first initializer of the variable, the number of subtypes is evaluated and returned from the check_init_bracket function. If the number of bracket is less than number of subtypes a warning is printed. The return value is the memory size of the aggregate.

get_field_size It is called during the variable initialization. The size of a sub-type of the aggregate is returned.

noninitialized_loc If the local variable was not initialized during declaration, the usage counter must be reset.

10.15 File 'type.h'

INTERNAL_TYPE macro for unfolding enum constants.

intern_arit_class enumeration type for internal arithmetical class (see 9.1).

PAR flag for formal parameter.

VAR flag for local variable.

intern_func_class enumeration type for internal function class (see 9.1).

YES flag for remote function with exported type of parameters.

NOT_DEFINED not defined range for arrays. It is used mainly for formal parameters. For example, `int z(int b[][3][3])`, the first subscript of the formal parameter `b` has undefined size.

GLOBAL_TYPE macro for unfolding enum constants.

global_type enumeration type for type flag used in remote functions with export types.

type_qual enumeration type for type qualifiers.

storage_class_specifier enumeration type for storage class specifiers.

POINTER_P

STRUCT_P

UNION_P

ENUM_P

ARRAY_P

SIMPLE_P

LOCAL_P

REMOTE_P predicates for the field function class in the internal type (see 9.1).

STATIC_P predicate for the field storage class specifier in the internal type (see 9.1).

VOID_P

CHAR_P

SHORT_P

INTEGER_P

LONG_P

FLOAT_P

DOUBLE_P

SIGNED_P

UNSIGNED_P

UNUSED_P predicates for the field attribute arit class in the internal type (see 9.1).

TYPES_EQ_P compares if subtype of the field attribute arit class is compatible.

CONST_P predicate for const type qualifier.

10.16 File 'virtual_machine.c'

exec wrapper to run the virtual machine. It checks if a virtual machine instruction was not interrupted.

exe all instructions of the virtual machine are coded in this function.

vtrue logical true. The value is stored on the temporary stack.

vfalse logical false. The value is stored on the temporary stack.

div_yes integer division.

divd_yes double division.

divf_yes float division.

divc_yes char division.

mod_yes modulus.

move_stack_aligned the temporary stack has to be always aligned.

10.17 File 'ys.h'

yyparse function prototype.

10.18 File 'ys.y'

Some rules in the grammar are not designed as stated in [4]. It is partially because of one pass compiler design and fully interpretative approach. Compilation of this file may cause problems (see section 12.1).

The biggest discrepancies between Standard ANSI C and Clif implementation is in parameter passing mechanism (see section 5.4). The bit-fields and preprocessing directives are not yet supported. The Clif has no preprocessing option. This stuff will be included as soon as possible. The goal is convergence of the framework to the Standard ANSI C specification.

Chapter 11

Some implementation details

11.1 Postfix operators (postfix increment and decrement)

Compilation of postfix increment and decrement is based on the following observation:

- On the top of the arithmetic stack is the address of the operand:
  - If the operand is an array, finish evaluation of the offset.
  - If the operand is a structure or union, finish evaluation of the offset.
  - If the operand is a pointer, add the size of the object it points to.
- Create a copy of the value, which is stored on the top of the arithmetic stack.
- Exchange two addresses of the top most operands on the arithmetic stack.
- Create address copy of the top most operand on the arithmetic stack.
- Push second operand on the arithmetic stack.
- Add two top most operands.
- Move the value on the top of the arithmetic stack to the top most but last address on the arithmetic stack.
- Pop the arithmetic stack.

11.2 Aggregate assigning

If the right side operand and the left side operand are aggregates of the same type, move a byte instructions are generated in the loop. The number of move instructions is generated accordingly to the memory size which the aggregate occupies.

Chapter 12

Bug report

If you have found a bug please report this bug to authors at the following e-mail address:

koren@vm.stuba.sk

Please, include in your bug report:

- Platform on which the Clif was running, i.e. machine, CPU, operating system.
- Version of the Clif.
- Source file that caused the problem. We will appreciate if the source file will be as short as possible and still consists the bug you want to report. If you don't know how to isolate the bug, send it anyway.
- Error message, if any, produced by the Clif.
- Indicate if you did any changes to the source of the Clif.

We will try to fix the bug if it is reproducible on platforms that are accessible to us.

12.1 Problems

Problems occur sometimes during compilation of the parser (c-parser.c). This problem is due to the large switch statement. Sometimes helps to specify -O2 optimization flag.

Bibliography

[1] A. V. Aho and S. C. Johnson. LR parsing. *Computing Surveys*, 6(2), June 1974.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

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

[4] ANSI. *American National Standard for Information Systems - Programming Language C.* ANSI, Dec. 1989. X3.159-1989.

[5] S. Bandyopadhyay, V.S. Begwani, and R.B. Murray. Compiling for the crisp processor. In *Proceedings of the IEEE Spring, COMPCON*, pages 96–100, 1987.

[6] A.D. Berenbaum, D.R. Ditzel, and H.R. McLellan. Introduction to the crisp instruction set architecture. In *Proceedings of the IEEE Spring, COMPCON*, pages 86–90, 1986.

[7] A.D. Berenbaum, D.R. Ditzel, and H.R. McLellan. Architectural innovations in the crisp microprocessor. In *Proceedings of the IEEE Spring, COMPCON*, pages 91–95, 1987.

[8] J.J. Dongarra, C.B. Moler, and et al. *LINPACK User's Guide.* SIAM, Philadelphia, 1979.

[9] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995. ISBN 0-8053-1670-1.

[10] D. Gries. *Kompilátory číslicových počítačov.* Alfa, 1981.

[11] D.G. Kafura and R.G. Lavender. Concurrent object-oriented language and the inheritance anomaly. In *Proceedings ISIPCALA'93*, pages 183–215, Prague, 1993.

[12] B. W. Kernighan and D. M. Ritchie. *Programovací jazyk C.* Alfa, 1988.

[13] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Technical report, Bell Laboratories, Murray Hill, NJ, 1975.

[14] R. Sedgewick. *Algorithms in C.* Computer Science. Addison-Wesley, 1990. ISBN 0-201-51425-7.

[15] B. Serpette, J. Vuillemin, and J.C. Hervé. BigNum: A portable and efficient package for arbitrary-precision arithmetic. Research report, INRIA, 1989. preprint.

[16] W. R. Stevens. *Advanced Programming in the UNIX Environment.* Professional Computing Series. Addison-Wesley, 1992. ISBN 0-201-56317-7.

[17] Ľ. Koreň. Small interpreter on the basis of C-language. Technical report, Slovak Technical University, Faculty of Mechanical Engineering, Bratislava, 1993.

[18] N. Wirth. *Algoritmy a štruktúry údajov.* Alfa, 1989.

Index

Contents

List of Figures