stichting mathematisch centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 94/78 JANUARI

A.P.W. BÖHM

THE INSTALLATION OF ALICE ON THE PDP11/45 UNDER UNIX

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

AMS(MOS) subject classification scheme (1970): 68A05, 68A10

ACM-Computer Review Categories: 4.12, 4.12, 4.22

The installation of ALICE on the PDP11/45 under UNIX

by

A.P.W. Böhm

ABSTRACT

This report documents the installation of ALICE on the PDP11/45 under UNIX. It describes the ALICE to assembly language translator and the run-time system. The performance of the implementation is compared to C, the systems implementation language of UNIX.

KEY WORDS & PHRASES: portability, intermediate code, code generation.

INDEX

0	Introduction	3
1	Outline of the ALICE implementation	4
2	The ALICE to AS translator	4
3	The run-time system	12
4	Performance	13
5	References	17
6	Appendix 1: The ALICE to AS translator	18
7	Appendix 2: The run-time system	51
8	Appendix 3: An ALICE character input-output program and its AS translation	56
9	Appendix 4: An ALICE version of the Ackermann function and its AS translation	59

0 Introduction

This report documents the installation of ALICE [1] on the PDP11/45 under UNIX. The installation took less than four man weeks. Even though this is a satisfying result, it does not show that ALICE is sufficiently portable. First, the designer and the installer were the same person, and second, while designing ALICE a first implementation was already being developed which helped in deciding how things were not to be done. And, last but not least, the ALEPH compiler generating ALICE still has to be finished. Only then can the ALICE implementation be fully tested and used.

1 Outline of the ALICE implementation

The ALICE implementation consists of:

A translator from ALICE to AS(sembler) [2,3], written in C [4], the systems implementation language of UNIX [5].

A run-time system, written in AS.

A driver, activating the ALICE translator, the assembler and the linkage editor.

When the user types: al <filename>
the file <filename> containing the ALICE program is translated and an executable object program is created in a file named "a.out".

2 The ALICE to AS translator

2.1 Skeleton of the ALICE translator

Roughly speaking, the translator (Appendix 1) performs the following loop:

```
while (getm()) expand();
```

The function "getm" reads a macro from the input file and puts it in the character array "mac". The function "expand" scans the macro and generates code from it. "expand" calls scan functions ("sym" for a macro-name or an ALICE-tag, "ch" for a character, "digs" for a digit sequence, "par" for any parameter, and "spar" for a string parameter), and it stores information in global variables and a value table called "valtab".

2.2 The value table

While processing value macros, the value table is built up. The size of "valtab" is determined by the status macro. The macros that put information in the value table are:

int denotation	(int),
manifest constant	(mcn),
char denotation	(chd),
string length	(sln),
external table length	(etl),
calculation	(add, sub, mul, dvd).

2.3 Data

2.3.1 Storage allocation

Integers

For an integer value (and consequently for a virtual machine word) one PDP11 word (16 bits) is allocated. Problems may arise when a program needs a very big address space. They can only be solved by choosing more than one word for every integer value in the program. This requires rewriting parts of the translator and the run-time system. It is not likely, however, that such a program will turn up on the PDP11. One word per integer value is enough for a program of the size of the ALEPH compiler compiling itself.

Strings

Strings are allocated as follows:
the characters in the string are put in bytes and are directly followed by at least one and at most two zero-bytes (word boundary). The row of characters is followed by an integer field containing the number of characters in the string. A pointer to a string points at the right-most word of the string (ALEPH convention). In this implementation a string pointer points at the "number of characters" field.

Example: the string "ALICE the MALICE" is represented by:

AL	IC	E	th	e	MA	LI	CE	\0\0	16

Quotes in a string are represented in ALICE by quote-images (double quotes). The quote-images are translated to single quotes.

Example: The string containing one quote ("""" in ALICE) is represented by:

"\0	1

The "\0" delimiter makes it easy to pass strings to system routines such as "open". The number of characters field is needed to allow all ASCII characters in a string (such as the ASCII-NUL-character with character code zero), to find the beginning of the string, and to avoid a time consuming implementation of the external "string length".

Lists

On the PDP11/45 under UNIX it is possible to separate instructions and data. Both instruction space and data space are allocated in up to 32K words of main storage. A data space of 32K is big enough to contain all lists of a program as big as the ALEPH compiler compiling itself. The lists are therefore kept in core. A reallotment program ([1] section 3.2.2) written in ALEPH will be one of the first big ALICE programs to test both ALICE implementation and ALEPH compiler.

Files

Every file gets a 512 byte buffer. This is done to speed up input-output even though it makes interaction with the program hard. If this problem turns out to be serious (which it doesn't yet), special non-buffering routines will be written for terminal input-output.

In the sequel "@" stands for "the address of" and "#" stands for "the number of".

2.3.2 Constant sources

Constant source macros don't generate any code. When a constant source must be loaded (loadv constant macro), an AS-literal will be generated from the value retrieved from valtab. So the constants are allocated in instruction space.

2.3.3 Variables

Variable macros generate a label and an AS data-declaration:

```
t<repr>:        <value>.
```

Debugging information is not generated (yet) so the debugging parameters (repr and string) of the variable macro are ignored.

2.3.4 Lists

A list macro generates a label and an empty word. This empty word corresponds to the minimal virtual address of the list.

For every list filling macro an AS data declaration is generated (int fill, string fill, fallow). While doing this, the number of words allocated for the list area is computed.

If the list is breathing, the end list macro generates a number of uninitialized words. This number is proportional to the number of virtual addresses associated to the list.

A list adm macro generates the following data structure:

```
a<repr>:
        type,
        virtual min lim,
        virtual max lim,
        virtual left lim,
        virtual right lim,
        key,
        @next list adm,
        "name of the list"
```

The "key" field determines the conversion from virtual address to core address:

```
        core address = virtual address + key
```

This key field will be updated by the reallotment program.

2.3.5 Files

A file administration generated by a begin file adm macro, a number of pointer macros, a number of numerical macros, and an end file administration macro looks as follows:

@(list adm)$_1$
. .
. .
. .
@(list adm)$_n$

lower bound$_1$
upper bound$_1$
. .
. .
. .
lower bound$_m$
upper bound$_m$

n
m

a<repr>:
UNIX file descriptor
#characters in buffer
@next char in buffer
buffer[512 bytes]

Only input character-files and output character-files are implemented.

2.4 Rules

The run-time stack of the ALICE ABSTRACT MACHINE is allocated on the hardware run-time stack manipulated by the stack-pointer (SP) register. The hardware stack will only be used for this purpose to facilitate (future) symbolic dump routines.

Every time an ALICE rule is called it gets a piece of run-time stack for its parameters and return address. The parameters are pushed directly on the run-time stack by the caller.

The implementation of the externals comes in two flavours:

a) If the external requires a few instructions, in-line code is generated. In that case the role of the ALICE gate is played by registers.

The externals implemented this way are:

```
less      (les),
equal     (eql),
mreq      (mrq),
lseq      (lsq),
more      (mor),
incr      (inc),
decr      (dcr),
minus     (min),
plus      (pls),
transport (trp),
next      (nxt).
```

b) If an external requires more than a few instructions, it is implemented as a subroutine. The gate and the return address are allocated in registers.

The externals implemented this way are:

```
put char  (pch),
get char  (gch),
put int   (pnt),
get int   (gnt),
random    (rnd).
```

The rest of the externals will be implemented later.

2.4.1 Parameter passing

The way parameters are passed depends on the kind of rule that is called.

In case of an ALICE rule, the way from memory via v-register (or a-register) via gate to run-time stack in the ALICE abstract machine, is cut short in the PDP11. A simple parameter is moved directly from memory to run-time stack. The caller sets up a stack frame for the rule to be called (actual stack frame macro and stack frame macro). Restoring from run-time stack via gate via w-register to actual output parameter is cut short in the same manner.

For an external the run-time stack is not used: the parameters are loaded on the gate which is allocated in registers. The a-register, v-register, w-register, and the gate are all mapped on the same hardware registers. Two global variables ("av_reg" and "w_reg") administrate which hardware register plays the role of an ALICE abstract machine device.

The calculation of the address of an indexed element is always done in registers. Code for bound checking is generated in-line. There is no facility to switch off bound checking, although it can be implemented easily.

2.4.2 Calling a rule or external

From a call to an ALICE rule that cannot fail (scall macro) code to push the return address on the run-time stack and a simple jump are generated. In case of a call to an ALICE rule that can fail (fcall macro) a jump to the false address of the call is added.

In case of an in-line external there is no need for a return address. A jump to the false address (if any) will be incorporated in the code for the external. In case of an external, implemented as subroutine, return address and false address (if present) are passed in registers.

2.4.3 Returning from a rule

Returning from an ALICE rule proceeds as follows:

If the rule cannot fail (rule type parameter), a simple jump to the return address is generated by the succ tail macro.

If the rule can fail there are two tails:

In the success tail, a jump to the instruction following the instruction on the return address is generated.
In the fail tail code to scratch the run-time stack frame of the failing rule and a jump back to the return address is generated. On this return address a jump to the false address is generated by the fcall macro.

2.5 ALICE primitives

Jump, label, exit, source line

The code generated for jump and label macros is trivial: a jump macro generates a jump; a label macro generates an label. An exit macro generates a jump to a run-time routine that closes the files. A source line macro generates no code at all (because the run-time error handling still is in a rudimentary stage).

Class

A class box macro puts the translator in such a state, that the role of the v-register will be played by a register. A class box end macro puts the translator back in the normal state and generates a jump. A class begin macro generates a label. A zone bounds macro generates a test and a jump to a true address. Care has been taken to generate reasonable code in case of special bounds (min-int or max-int). A zone value generates another test and a jump to a true address. A class end macro generates a jump to a run-time error routine. From the above it is clear that the simplest implementation of classes has been chosen.

Extension

The PDP registers cannot be used to play the role of the gate in an extension, because the size of the gate may become too great for that. The sources are therefore put on the run-time stack just as input parameters of an ALICE rule. An extension call macro generates a call to a run-time routine. This routine checks whether the extension is allowed and possible, updates the list administration, and returns with the core address of the new top of the list in a register. An extension copy macro generates a move from the run-time stack to the new block on top of the list. An extension end macro generates code to reset the run-time stack pointer.

2.6 Comments and messages

From an ALICE comment line an AS comment is generated, that is: "xxx" is replaced by "/".

A non standard (but useful) ALICE macro has been used while writing the translator and run-time system:

message	: mess symbol, sp, string, el.
mess symbol	: "mss".
sp	: space.
el	: end of line.

From a mess macro code to put the string parameter on the terminal is generated.

3 The run-time system

With every particular program a (small) run-time system (Appendix 2) is linked. The execution of a program starts in the run-time opening routine.

The routines making up the run-time system are:

Opening

The chain of file administrations is scanned. Files are either opened (input) or created (output). If a file cannot be opened or created an error message is put on the terminal and execution is stopped. After opening the files the particular program is started.

Error messages

Run-time error messages (bounds, class) are put on the terminal and followed by a jump to the closing routine.

Extension

The extension routine consists of three parts:

a) Ensuring whether the extension fits in the physical address space of the list;
   if it does not fit the reallotment routine is called.
b) Updating the list administration.
c) Returning to the program with the new (physical) top address of the list.

Reallotment

This routine will be implemented as soon as the ALEPH compiler generates ALICE.

Closing

The chain of file administrations is scanned. Every buffer of an output file is flushed, that is: it is written on the output file.

4 Performance

A set of test programs (hand written) in ALICE accompanies the implementation. Most of these programs were written to test the correctness of the code. These programs will not be discussed. Two ALICE programs were written to measure the performance of this implementation. Their running times and sizes were compared to equivalent programs written in C (compiled with the optimizing version of the C compiler).

4.1 Input-output

The ALICE equivalent (Appendix 3) of the following ALEPH program performs character input-output:

```
'variable' char = /?/.

'charfile' inp = >"input".

'charfile' outp = "output">.

'root' copy characters.

'action' copy characters:
        get char + inp + char, put char + outp + char, :copy characters;
        +.

'end'
```

This program was compared to the following C program:

```
struct  buffer {
        int fd;
        int nlft;
        char *nextp;
        char buff[512];
        };
struct buffer ibuffer, obuffer;


main()
{
if (fopen("input",&ibuffer) < 0)
        {
        printf("can't open input\n");

        exit(1);
        }
if (fcreat("output",&obuffer) < 0)
        {
        printf("can't open output\n");
        exit(1);
        }
copy_characters();
fflush(&obuffer);
}

copy_characters()
{
register char ch;
        while ( !((ch = getc(&ibuffer)) < 0) )
        putc(ch,&obuffer);
}
```

Copying a file containing 40128 characters gave the following results:

	ALICE	C	
user time	1.26	2.20	seconds
system time	2.86	0.82	seconds
size	1708	2844	bytes

where "size" is the total size of the object program, which is of course independent of the size of the input file. Although C is faster, ALICE seems fast enough.

4.2 Calling mechanism

To measure the implementation of the calling mechanism, the Ackermann function was programmed in both ALICE and C. The ALICE version (Appendix 4) is a translation of the following ALEPH program:

```
'action' ack + >m + >n + r>:
        m=0, plus + n + 1 + r;
        n=0, decr + m, ack + m + n + r;
        decr + n, ack + m + n + r,
         decr + m, ack + m + r + r.

'root' ackermann.


'action' ackermann - i - j - r:
        0 -> i,
         (l1: more + i + 3;
              0 -> j, (l2: more + j + 7, incr + i, :l1;
                           ack + i + j + r,
                            put int + pr + i,
                            put int + pr + j,
                            put int + pr + r,
                            incr + j,
                            :l2
                      )
         ).

'charfile' pr = "output">.

'end'
```

This program was compared to a C version of the Ackermann program (which also buffers its files). The Ackermann function itself is programmed as follows in C:

```
ackermann(m,n)
int m,n;{
        return ( (m == 0) ? (n + 1)
                          : ((n == 0) ? ackermann(m-1,1)
                                      : ackermann(m-1, ackermann(m,n-1))
                            )
               );
}
```

The results were:

	ALICE	C	
user time	29.02	36.08	seconds
system time	0.28	0.62	seconds
size	2556	2614	bytes

From this measurement it can be concluded that the calling mechanism has been implemented sufficiently efficient.

5 References

[1] A.P.W. Böhm
ALICE: an exercise in program portability
Report IW 91/77
Mathematisch Centrum Amsterdam 1977

[2] D.M. Ritchie
UNIX Assembler Reference Manual

[3] Digital
PDP11/45 Processor Handbook

[4] D.M. Ritchie
C Reference Manual

[5] D.M. Ritchie & K. Thompson
The UNIX Time-Sharing System
Communications of the ACM, july 1974, Volume 17, Number 7

6 Appendix 1: The ALICE to AS translator

```
#define macro(l1,l2,l3) 676*('l1'-'a')+26*('l2'-'a')+'l3'-'a'
#define tag(l1,l2,l3) 676*('l1'-'a')+26*('l2'-'a')+'l3'-'a'


#define buffsize        512
#define macwidth        72
#define fraction        4
#define Bool            int
#define TRUE            1
#define FALSE           0
#define MIN_int         -32768
#define MAX_int         32767
#define NL              10




/* global arrays and variables  */


char mac[macwidth];     /* macro buffer */
char    *macp   mac;

struct {
        int fd;
        int nlft;
        char *nextp;
        char buffer[buffsize];
        }buffer;        /*input buffer maintained by getc in getm */

int     *valtab;        /* symbol table; size determined by status macro */




int     vald;           /* value of scanned digit sequence (digs)*/
int     valsym;         /* conversion value of a symbol scanned by sym */
int     stlen;          /* number of characters in a string
                         * scanned by spar */



int     chcode;         /* character code */
int     pointer;        /* number of pointers in a file adm */
int     numeric;        /* number of numerics in a file adm */
```

```
int     av_reg  1;      /* indicates which pdp register plays
                         *  the role of v register or a register */

int     w_reg;          /* same for w register */

int     posos;          /* position on stack    */
int     rule;           /* repr of rule  */
int     target;         /* repr of target rule in a call */


int     lsize   0;      /* size of a list area
                         * set by list (lst) macro */


Bool    status  FALSE;  /* a Boolean to determine whether
                         *  the status macro has been expanded
                         */
Bool    extcall FALSE;  /* TRUE while an extcall is scanned     */
Bool    index   FALSE;  /* TRUE while indexed param is scanned */
Bool    extend  FALSE;  /* TRUE while an extension is scanned */
Bool    ruletype;       /* of scanned rule */
```

```
/*
*  Scan functions sym, ch, digs, par, spar
*  These scan functions perform a little testing on the
*  parameters they read.
*  After execution macp points at the first character of the
*  next parameter or at the end of the macro ('\0').
*/



sym()
/*
*  This function converts 3 characters, pointed at by
*  macp, to an int (valsym) just as the macros at the first two
*  lines of this program do.
*
*  If the characters are no letters, sym returns 0.
*
*  If the character following the 3 characters is not
*  ' ' or '\0' or ','  sym returns 0,
*  otherwise sym returns 1.
*
*/
{ int i,j,k;
  i= *macp++ - 'a';
  if (noletter(i)) return(0);
  j= *macp++ - 'a';
  if (noletter(j)) return(0);
  k= *macp++ - 'a';
  if (noletter(k)) return(0);


  valsym = (i*676)+(j*26)+k;
  if (*macp == ' '| *macp == ',')
        {*macp++; return(1);}
  else if (*macp == '\0')
        return(1);
  else return(0);
}

noletter(l)     int l;
{
if (l < 0 | l > 25)
        return(1);
else    return(0);
}
```

```
ch()
/*
*  this function puts the character code of
*  a character parameter in the global variable chcode.
*/
{
chcode = *macp++;
}
```

```
digs()
/*
* This function converts a string of digits, pointed at
* by macp to an int (vald). If all went well digs
* returns 1, otherwise digs returns 0.
* If characters other than digits are encountered, digs
* writes a message and returns 0.
*/
{ int d;
if (*macp == '\0') return(0);
for (vald=0; *macp != '\0' && *macp != ',' ;)
 {d = *macp++ - '0';
  if ( d < 0  |  d > 9 ) {printf("don't dig: %s\n", mac); return(0);}
  vald = 10*vald+d;
 }
if (*macp == ',') *macp++;
return(1);
}
```

```
par(t)          char *t;
/*
*  This function delivers the parameter
*  into a string (t) and returns 1;
*  if something is wrong par returns 0.
*
*/
{
 if (*macp == '\0') return(0);
 while (*t++ = *macp++)
 {
  if (*macp == ','){*t = '\0'; *macp++; return(1);}
  if (*macp == '\0'){*t = '\0'; return(1);}
 }
}
```

```
spar(t)         char *t;
/*
*  spar peels the quotes from a string param,
*  converts quote-images ("") to quotes ("),
*  and ">" tokens (AS string delimiters) to their escaped versions.
*  The length of the string is put in stlen.
*/




{
*macp++;
stlen = 0;
while (TRUE)
 {
 if (*macp == '"')
  {
  *macp++;
  if (*macp++ == '"') /* quote-image */
   {
   *t++ = '"';
   stlen++;
   }
  else   /* end of string */
   {
   *t = '\0';
   break;
   }
  }
 else if (*macp == '>')
   {
   *t++ = '\\';
   *t++ = '>';
   *macp++;
   stlen++;
   }
      else
   {
   *t++ = *macp++;
   stlen++;
   }
 }
}
```

```
/*      ALICE macro processor           */


main(argc,argv)         int argc;       char **argv;
{
if (argc < 2)
        {printf("arg count\n");
         exit(1);
        }
if (fopen(argv[1],&buffer) < 0)
        {printf("can't open %s\n", argv[1]);
         exit(1);
        }
while(getm()) expand();
}

getm()
{
char *macin;
char c;
macin = mac;
do {
nextch: c = getc(&buffer);
        if (c < 0) return(0);
        if (c != '\n')
                {
                *macin++ = c;
                goto nextch;
                }
   } while (macin == mac);
*macin = '\0';
return(1);
}

gen_mess(p,n)   char *p;        int n;
/*
* this routine generates code to print
* a message. It will be used for debugging
* purposes. It is activated when a "mss" macro
* is read
* p points at the string to print
* n is equal to the length of the string
*/
{
printf("\n/ message\n");
printf("\nmov $1,r0");
printf("\nsys write; 8f; %d.", n);
printf("\n.data");
printf("\n8: <%s\\n>", p);
printf("\n.text");
printf("\n/ egassem\n\n\n");
}
```

```
/*
the following routines generate code for input/output

they all handle an i/o buffer generated
by the file administration macros:



t<repr>:        file descriptor
                #characters
                address of next char to be put in or out
                256 words (512 characters i/o buffer)



comments are given in an ALEPH-like language
*/

gen_pch()
/*  code for the external rule putchar  */

/*

putchar + >char:
        decr + #characters,
        (less + #characters + 0, write + buffer; +),
        char -> buffer[address of next char],
        incr + address of next char,
        return.


*/

{
printf("\ndec 2(r1)");
printf("\nbge 6f");
printf("\nmov r1,r4");
printf("\nadd $6,r4");
printf("\nmov r4,0f");
printf("\nmov 4(r1),0f+2");
printf("\nbeq 1f");
printf("\nsub r4,0f+2");
printf("\nmov (r1),r0");
printf("\nsys 0;2f");
printf("\n.data");
printf("\n2: sys write; 0: ..; ..");
printf("\n.text");
printf("\n1: mov r4,4(r1)");
printf("\nmov $512.,2(r1)");
printf("\n6: movb r2,*4(r1)");
printf("\ninc 4(r1)");
printf("\njmp (r3)\n");
}
```

```
gen_gch()

/*  code for the external rule getchar */



/*


getchar + char>:
        decr + #characters,
        (less + #characters + 0, read + buffer; +),
        buffer[address of next char] -> char,
        incr + address of next char,
        return(success addr).


read + buffer:
        sys read + number of characters read,
        (less + number of characters read + 0, return(fail addr);
                number of characters read -> #chars).



*/
{
printf("\ndec 2(r1)");
printf("\nbge 1f");
printf("\nmov r1,r0");
printf("\nadd $6,r0");
printf("\nmov r0,4(r1)");
printf("\nmov r0,0f");
printf("\nmov (r1),r0");
printf("\nsys 0;6f");
printf("\n.data");
printf("\n6: sys read; 0: ..; 512.");
printf("\n.text");
printf("\ndec r0");
printf("\nbmi 4f");
printf("\nmov r0,2(r1)");
printf("\n1: movb *4(r1),r5");
printf("\ninc 4(r1)");
printf("\nmov r5,r1");
printf("\njmp (r2)");
printf("\n4: jmp (r3)\n");
}
```

```
gen_gnt()
/* code for getint */
/*
get int + rl>: get sign + r5, digits + r3,
               (r5 = /-/, -r3 -> rl; r3 -> rl).

get sign: get char + r5,
                ( =r5=  [/+/;/-/], 0 -> r3;
                        [/0/:/9/], r5 -> r3;
                        :get sign
                );
          fail.

digits: get char + r4,
                ( =r4=  [/0/:/9/], 10*r3 + r4 -> r3, :digits;
                        reset info in file administration
                );
        succeed. $eof



*/
{
printf("\ngis: dec 2(rl)");
printf("\nbge lf");
printf("\nmov rl,r0");
printf("\nadd $6,r0");
printf("\nmov r0,4(rl)");
printf("\nmov r0,0f");
printf("\nmov (rl),r0");
printf("\nsys 0;6f");
printf("\n.data");
printf("\n6: sys read; 0: ..; 512.");
printf("\n.text");
printf("\ndec r0");
printf("\nbmi gifl");
printf("\nmov r0,2(rl)");
printf("\nl: movb *4(rl),r5");
printf("\ninc 4(rl)");
printf("\ncmpb $'+,r5");
printf("\nbeq gifd");
printf("\ncmpb $'-,r5");
printf("\nbeq gifd");
printf("\ncmp $'9,r5");
printf("\nbmi gis");
printf("\nsub $'0,r5");
printf("\nbmi gis");
printf("\nmov r5,r3");
printf("\nbr gids");
printf("\ngifd: clr r3");
printf("\ngids:");
printf("\ndec 2(rl)");
```

```
printf("\nbge 1f");
printf("\nmov r1,r0");
printf("\nadd $6,r0");
printf("\nmov r0,4(r1)");
printf("\nmov r0,0f");
printf("\nmov (r1),r0");
printf("\nsys 0;6f");
printf("\n.data");
printf("\n6: sys read; 0: ..; 512.");
printf("\n.text");
printf("\ndec r0");
printf("\nbmi gisc");
printf("\nmov r0,2(r1)");
printf("\n1: movb *4(r1),r4");
printf("\ninc 4(r1)");
printf("\ncmpb $'9,r4");
printf("\nbmi girs");
printf("\nsub $'0,r4");
printf("\nbmi girs");
printf("\nmul $10.,r3");
printf("\nadd r4,r3");
printf("\nbr gids");
printf("\ngirs:");
printf("\ndec 4(r1)");
printf("\ninc 2(r1)");
printf("\ngisc:");
printf("\nmov r3,r1");
printf("\ncmpb $'-,r5");
printf("\nbne 1f");
printf("\nneg r1");
printf("\n1: jmp (r2)");
printf("\ngifl:");
printf("\njmp (r3)\n");
}
```

```
gen_pnt()

/*  code for the external rule putint  */



/*

putint + >int:

        convert + int + string,
        put 6 characters + string.

convert + >int(r2,r3) + string>(3f,3f+2,3f+4) - sign:
        clear + string,
        (less + int + 0, /-/ -> sign, complement + int;
                / / -> sign),
        (div: divide + int + 10 + rest,
                conv + rest + char,
                stack + char + next right-most pos of string,
                (int = 0; :div)
        ),
        stack + sign +next right most pos of string.



*/



{
printf("\nmov $4f,r4");
printf("\nmov $\"  ,r5");
printf("\nmov r5,3f");
printf("\nmov r5,3f+2");
printf("\nmov r5,3f+4");
printf("\nmov r3,4f");
printf("\nmov r2,r3");
printf("\nbge 1f");
printf("\nneg r3");
printf("\nmovb $'-,r5");
printf("\n1: clr r2");
printf("\ndiv $10.,r2");
printf("\nadd $'0,r3");
printf("\nmovb r3,-(r4)");
printf("\nmov r2,r3");
printf("\nbne 1b");
printf("\nmovb r5,-(r4)");
printf("\n.data");
printf("\n3: <      >");
printf("\n4: 0");
printf("\n.text");
printf("\nmov $3b,r3");
```

```
printf("\n3: dec 2(r1)");
printf("\nbge 6f");
printf("\nmov r1,r2");
printf("\nadd $6,r2");
printf("\nmov r2,0f");
printf("\nmov 4(r1),0f+2");
printf("\nbeq 1f");
printf("\nsub r2,0f+2");
printf("\nmov (r1),r0");
printf("\nsys 0;2f");
printf("\n.data");
printf("\n2: sys write; 0: .. ; ..");
printf("\n.text");
printf("\n1: mov r2,4(r1)");
printf("\nmov $512.,2(r1)");
printf("\n6: movb (r3)+,*4(r1)");
printf("\ninc 4(r1)");
printf("\ncmp r3,$4b");
printf("\nbne 3b");
printf("\njmp *4b\n");
}



gen_rnd()
/* code for the external random */

/*
random + >min + >max + res>:

        times + 13077 + ran + ran,
        plus + 6925 + ran + ran,
        fiddle + ran,
        trim + min + max + ran,
        ran -> res.

fiddle + >ran>: swap and clear sign bit.

trim + >min + >max + >ran> - diff:
        minus + max + min + diff,
        incr + diff,
        divrem + ran + diff + ? + ran,
        plus + min + ran + ran.

register allocation: input: r1 = min, r2 = max, r3 = ret addr;
                       output: r1 = res.
                      scratch: r4, r5.
*/

{
printf(".data\n");
printf("ran: 12345.\n");
printf(".text\n");
```

```
printf("mov ran,r5\n");
printf("mul $13077.,r5\n");
printf("add $6925.,r5\n");
printf("mov r5,ran\n");
printf("swab r5\n");
printf("bic $100000,r5\n");
printf("clr r4\n");
printf("sub r1,r2\n");
printf("inc r2\n");
printf("asr r5\n");
printf("div r2,r4\n");
printf("add r5,r1\n");
printf("jmp (r3)\n");
}
```

```
expand()
{

int loc, source, value, left, right, min, max, cal, type;
char  pbf[10], pbf2[10], pbf3[10];
macp = mac;
if (!sym())
printf("incorrect format: %s\n",mac);
else
{ switch(valsym)
{


case macro(a,d,d):
/* add          location, valref, valref  */
        digs(); loc = vald;
        digs(); value = vald;
        digs();
        valtab[loc] = valtab[value] + valtab[vald];
        break;


case macro(b,c,k):
/* background */
        break;


case macro(b,f,a):
/* begin file adm       repr, ch or data, i/o, next file, file name */
        pointer = 0;
        numeric = 0;
        break;


case macro(c,s,b):
/* class begin                      repr */
        par(pbf);
        printf("t%s:\n", pbf);
        break;


case macro(c,s,e):
/* class end */
        printf("jmp clserr\n");
        break;


case macro(c,h,d):
/* char         location, char denotation  */
        digs();
        ch();
        valtab[vald] = chcode;
        break;
```

```
case macro(c,l,l):
/* call id      repr, type, recursion */
        digs();
        target = vald;
        break;


case macro(c,m,m):
/* communication        first list, first file  */
        printf("comma: ");
        digs();
        if (vald)
                printf("a%d;", vald);
        else    printf("nil;");
        digs();
        if (vald)
                printf("a%d\n",vald);
        else    printf("nil\n");
        printf(".text\n");
        break;


case macro(c,a,r):
/* copy a reg           formal */
        if (!extcall)
                {
                par(pbf);
                digs();
                printf("%d.(sp)\n", 2*vald);
                }
        break;


case macro(c,i,g):
/* copy from input gate         formal */
        break;


case macro(c,s,s):
/* constant source */
        break;


case macro(c,v,r):
/* copy v reg           formal */
        digs();
        if (!extend) digs();
        if (!extcall)
                if (index)
                printf("mov r%d,%d.(sp)\n", av_reg-1, 2*vald);
                else printf("%d.(sp)\n", 2*vald);
        index = FALSE;
        break;
```

```
case macro(d,v,d):
/* divide        location, valref, valref  */
        digs(); loc = vald;
        digs(); value = vald;
        digs();
        valtab[loc] = valtab[value] / valtab[vald];
        break;


case macro(d,m,p):
/* dump */
        break;


case macro(e,f,a):
/* end file adm          repr, type, next adm, file name */
        printf("%d.\n%d.\n", pointer, numeric);
        par(pbf);
        digs(); type = vald;
        printf("%d\n", type);
        digs();
        if (vald)
                printf("a%d\n", vald);
        else    printf("nil\n");
        printf("a%s: 0;0;0\n", pbf);
        printf(". = . + 512.\n");
        spar(pbf);
        printf("<%s\\0>\n.even\n",pbf);
        break;


case macro(e,l,s):
/* end list area         repr, type, #virt addresses */
        par(pbf);
        digs();
        if (vald > 1)   /* breathing */
                printf(". = . + %d.\n", lsize);
        break;


case macro(e,n,d):
/* end program           program name */
        break;


case macro(e,v,a):
/* end values   */
        printf(".data\n");
        break;
```

```
case macro(e,x,t):
/* exit          int deno        */
        printf("jmp cl\n");
        break;


case macro(e,f,c):
/* ext fcall            repr, tag, f addr */

        av_reg = 1;
        par(pbf);
        sym();
        par(pbf2);

        switch(valsym)


        {case tag(g,c,h):
        /* getchar */
        printf("mov $1f,r2\nmov $t%s,r3\njbr t%s\n1: ", pbf2, pbf);
        break;


        case tag(g,n,t):
        /* getint */
        printf("mov $1f,r2\nmov $t%s,r3\njbr t%s\n1: ", pbf2, pbf);
        break;


        case tag(l,e,s):   /* short jump used   */
        printf("cmp r1,r2\nbge t%s\n", pbf2);
        break;


        case tag(e,q,l):   /* short jump used     */
        printf("cmp r1,r2\nbne t%s\n", pbf2);
        break;


        case tag(m,r,q):   /* short jump used     */
        printf("cmp r1,r2\nbmi t%s\n", pbf2);
        break;


        case tag(l,s,q):   /* short jump used     */
        printf("cmp r2,r1\nbmi t%s\n", pbf2);
        break;


        case tag(m,o,r):   /* short jump used     */
        printf("cmp r2,r1\nbge t%s\n", pbf2);
        break;
```

```
        default:
        printf("unknown tag: %s\n", mac);
        }
        break;


case macro(e,s,c):
/* ext scall            repr, tag       */
        av_reg = 1;
        par(pbf);
        sym();
        switch(valsym)


        {case tag(p,c,h):
        /* putchar */


        case tag(p,n,t):
        /* putint */


        case tag(r,n,d):
        /* random */
        printf("mov $1f,r3\njbr t%s\n1: ", pbf);
        break;


        /* incr */
        case tag(i,n,c):   /* no overflow check is generated */
        printf("inc r1\n");
        break;


        /* decr */
        case tag(d,c,r):   /* no overflow check is generated */
        printf("dec r1\n");
        break;


        /* minus */
        case tag(m,i,n):   /* no overflow check is generated */
        printf("sub r2,r1\n");
        break;


        /* plus */
        case tag(p,l,s):   /* no overflow check is generated */
        printf("add r2,r1\n");
        break;
```

```
        /*  ->  */
        case tag(t,r,p):
        /* empty */
        break;


        case tag(n,x,t):
        /* next */
        printf("mov 14.(r1),r1\nadd r2,r1\n");
        break;


        default:
        printf("unknown tag: %s\n", mac);
        }
        break;


case macro(e,c,e):
/* extcall end          TRUE address    */
        av_reg = 1;
        digs();
        if (vald)
                printf("jbr t%d\n", vald);
        extcall = FALSE;
        index = FALSE;
        break;


case macro(e,f,i):
/* ext fcall id         repr, tag, f addr */
        extcall = TRUE;
        break;


case macro(e,s,i):
/* ext scall id         repr, tag       */
        extcall = TRUE;
        break;


case macro(e,x,c):
/* extension copy               formal */
        digs(); source = 2*vald;
        digs(); vald=2*vald;
        printf("mov %d.(sp),%d.(r3)\n", source, vald);
        break;
```

```
case macro(e,x,e):
/* extension end */
        printf("mov r1,sp\n");
        extend = FALSE;
        av_reg = 1;
        break;


case macro(e,x,i):
/* extension id */
        extend = TRUE;
        break;


case macro(e,t,c):
/* extension call */
        printf("r2\nmov $1f,(sp)\njmp extend\n1: ");
        break;


case macro(e,r,l):
/* (standard) external rule decl        repr, tag       */
/* only externals implemented as subroutines generate code
   from this macro
*/
        par(pbf);
        sym();
        switch(valsym)


        {case tag(p,c,h):
        /* put char */
        printf("\nt%s:",pbf);
        gen_pch();
        break;


        case tag(g,c,h):
        /* get char */
        printf("\nt%s:",pbf);
        gen_gch();
        break;


        case tag(g,n,t):
        /* get int */
        printf("\nt%s:",pbf);
        gen_gnt();
        break;
```

```
        case tag(p,n,t):
        /* put int */
        printf("\nt%s:",pbf);
        gen_pnt();
        break;

        case tag(r,n,d):
        /* random */
        printf("\nt%s:\n", pbf);
        gen_rnd();
        break;

        default:
        printf("unknown tag: %s\n", mac);
        }
        break;

case macro(f,t,i):
/* fail tail id         repr */
        par(pbf);
        printf("t%s:\n", pbf);
        break;

case macro(f,l,w):
/* fallow       valref */
        digs();
        value = 2 * valtab[vald];
        printf(". = . + %d.\n", value);
        lsize = lsize - value;
        break;

case macro(f,c,l):
/* fcall                 repr, f addr */
        par(pbf);
        par(pbf2);
        printf("mov $1f,(sp)\njbr t%s\n1: jmp t%s\n", pbf, pbf2);
        break;

case macro(c,b,i):
/* classifier box id */
        extcall = TRUE;
        break;
```

```
case macro(e,b,x):
/* classifier box end          class address */
        digs();
        if (vald)
                printf("jbr t%d\n", vald);
        extcall = FALSE;
        av_reg = 1;
        break;


case macro(f,r,w):
/* free w_reg */
        break;


case macro(i,i,p):
/* indexed input parameter */
        index = TRUE;
        if (!extcall) av_reg = 2;
        break;


case macro(i,g,t):
/* input gate          size of gate */
        if (extend)
                {
                digs();
                printf("mov sp,r1\nsub $%d.,sp\n", 2*(vald+1));
                }
        break;


case macro(i,n,t):
/* int          location, int denotation */
        digs(); loc = vald;
        digs();
        valtab[loc] = vald;
        break;


case macro(i,o,p):
/* indexed output parameter */
        index = TRUE;
        if (extcall)    av_reg = w_reg + 1;
        else            av_reg = 2;
        break;
```

```
case macro(i,t,f):
/* int fill     valref */
        digs();
        printf("%d.\n", valtab[vald]);
        lsize = lsize - 2;
        break;


case macro(j,m,p):
/* jump         repr    */
        par(pbf);
        printf("jbr t%s\n", pbf);
        break;


case macro(l,a,b):
/* label        repr    */
        par(pbf);
        printf("t%s:\n", pbf);
        break;


case macro(l,d,m):
/*  list adm    repr, type, min, max, left, right, cal, next, name */
        par(pbf);
        digs(); type = vald;
        printf("a%s: %d\n", pbf, type);
        digs(); min = valtab[vald];
        digs(); max = valtab[vald];
        digs(); left = valtab[vald];
        digs(); right = valtab[vald];
        printf("%d.\n%d.\n%d.\n%d.\n", min, max, left, right);
        digs(); cal = valtab[vald];
        /* calculate bump */
        if (type > 1)   /* breathing */
                value = min + (max - min + 1) / fraction;
        else    value = max;
        printf("%d.\n%d.\n", cal, value);
        printf("l%s-[%d.]\n", pbf, min*2);
        digs(); /* next adm */
        if (vald)
                printf("a%d\n", vald);
        else    printf("nil\n");
        spar(pbf);
        printf("<%s\\0>\n.even\n", pbf);
        break;
```

```
case macro(l,s,t):
/* list area            repr,type,#virt addresses */
        par(pbf);
        printf("l%s: 0\n", pbf);
        digs();
        value = valtab[vald];
        if (vald > 1)   /* breathing */
                {
                digs();
                value = valtab[vald];
                lsize = (value / fraction) * 2; /* bytes */
                }
        break;


case macro(l,a,g):
/* loada glob   repr    */
        par(pbf);
        if (index)
                printf("mov $a%s,r%d\n", pbf, av_reg);
        else    if (extcall)
                        {
                        printf("mov $a%s,r%d\n", pbf, av_reg);
                        av_reg++;
                        }
                else    printf("mov $a%s,", pbf);
        break;


case macro(l,v,c):
/* loadv constant       repr,valref     */
        par(pbf);
        digs();
        if (extcall | index)
                {
                printf("mov $%d.,r%d\n", valtab[vald], av_reg);
                av_reg++;
                }
        else    printf("mov $%d.,", valtab[vald]);
        break;
```

```
case macro(l,v,l):
/*loadv limit            limit */
        digs();
        if (vald == 0) /* left */ vald = 6;
        if (vald == 1) /* cal  */ vald = 10;
        if (vald == 2) /* right*/ vald = 8;
        if (index)
                {
                printf("mov %d.(r%d),r%d\n", vald, av_reg, av_reg);
                av_reg++;
                }
        else    if (extcall)
                        printf("mov %d.(r%d), r%d\n",
                                vald, av_reg-1, av_reg-1);
                else    printf("r2\nmov %d.(r2),", vald);
        break;


case macro(l,v,i):
/* loadv indexed element                selector */


/* generate bounds check */
        printf("cmp r%d,6.(r%d)\n", av_reg-1, av_reg);
        printf("blt 0f\n");
        printf("cmp r%d,8.(r%d)\n", av_reg-1, av_reg);
        printf("ble 1f\n");
        printf("0: mov $1f,r0\n");
        printf("jmp bounds\n");


/* generate index operation */
        printf("1: asl r%d\nadd 14.(r%d),r%d\n",
                av_reg-1, av_reg, av_reg-1);
        digs();
        if (vald)
                printf("sub %d.,r%d\n", 2*vald, av_reg-1);
        printf("mov (r%d),r%d\n", av_reg-1, av_reg-1);
        break;
```

```
case macro(l,a,s):
/* loada stack var       position on stack */
        digs(); vald = 2*vald;
        if (index)
                if (extcall)
                        printf("mov %d.(sp),r%d\n", vald, av_reg);
                else
                        printf("mov %d.(r1),r%d\n", vald, av_reg);
        else
                if (extcall)
                        {
                        printf("mov %d.(sp),r%d\n", vald, av_reg);
                        av_reg++;
                        }
                else
                        printf("mov %d.(r1),", vald);
        break;


case macro(l,v,s):
/* loadv stack var       position on stack */
        digs();
        if (extcall | index)
                {
                printf("mov %d.(sp),r%d\n", 2*vald, av_reg);
                av_reg++;
                }
        else    printf("mov %d.(r1),", 2*vald);
        break;


case macro(l,v,v):
/* loadv var             repr    */
        par(pbf);
        if (extcall | index)
                {
                printf("mov t%s,r%d\n",pbf,av_reg);
                av_reg++;
                }
        else
                printf("mov t%s,",pbf);
        break;
```

```
case macro(l,d,w):
/* loadw        formal  */
        digs();
        if (extcall)
                w_reg = vald;
        else
                {
                digs();
                posos = vald;
                }
        break;


case macro(m,c,n):
/* manifest constant    location, tag   */
        digs();
        sym();
        switch(valsym)
        {case tag(n,l,n):
        /* newline */
        value = NL;
        break;


        case tag(m,n,i):
        /* min int */
        value = MIN_int;
        break;


        case tag(m,x,i):
        /* max int */
        value = MAX_int;
        break;


        case tag(m,n,a):
        /* min addr */
        value = 1;
        break;


        case tag(m,x,a):
        value = MAX_int;
        break;


        default:
        printf("unknown tag: %s\n", mac);

        }
        valtab[vald] = value;
        break;
```

```
case macro(m,s,s):
        spar(pbf);
        stlen++;
        gen_mess(pbf,stlen);
        break;


case macro(m,u,l):
/* mul           location, valref, valref  */
        digs(); loc = vald;
        digs(); value = vald;
        digs();
        valtab[loc] = valtab[value] * valtab[vald];
        break;


case macro(n,u,m):
/* numeric       valref, valref */
        digs(); value = vald;
        digs();
        printf("%d.\n%d.\n",valtab[value],valtab[vald]);
        numeric++;
        break;


case macro(o,g,t):
/* output gate           size of output gate */
        break;


case macro(p,t,r):
/* pointer       repr    */
        par(pbf);
        printf("t%s\n",pbf);
        pointer++;
        break;


case macro(p,i,d):
/* prog id       program name    */
        printf(".globl start, comma\n");
        break;


case macro(r,l,i):
/* rule id       repr, rule type, recursion */
        digs();
        printf("t%d:\n", vald);
        rule = vald;
        digs();
        ruletype = vald;
        break;
```

```
case macro(r,o,g):
/* restore to output gate                formal */
        break;


case macro(r,u,t):
/* root          program name    */
        printf("start:\n");
        break;


case macro(s,r,l):
        break;


case macro(s,c,l):
/* scall         repr */
        par(pbf);
        printf("mov $lf,(sp)\njbr t%s\nl: ", pbf);
        break;


case macro(s,f,r):
/* stack frame           #parameters, #locals, #actuals */
        digs(); digs();
        printf("loc%d = %d.\n", rule,  vald * 2);
        break;


case macro(s,t,s):
/* status        max stack frame, max gate, #expressions, #lists */
        if (!status)
                {
                par(pbf);
                par(pbf2);
                digs();
                valtab = alloc(vald);
                status = TRUE;
                }
        break;


case macro(s,w,v):
/* storew var            repr */
        par(pbf);
        if (extcall)
                printf("mov r%d,t%s\n", w_reg, pbf);
        else
                printf("mov %d.(sp),t%s\n", 2*posos, pbf);
        break;
```

```
case macro(s,w,i):
/* storew indexed                selector */
/* administration:      r(av_reg),
*  index:               r(av_reg - 1),
*  value to store:      if (extcall)  r(w_reg)  [= r(av_reg - 2)]
*                       else R.T.S position posos
*/


/* generate bounds check  */
        printf("cmp r%d,6.(r%d)\n", av_reg-1, av_reg);
        printf("blt 0f\n");
        printf("cmp r%d,8.(r%d)\n", av_reg-1, av_reg);
        printf("ble 1f\n");
        printf("0: mov $1f,r0\n");
        printf("jmp bounds\n");


/* generate index operation */
        printf("1: asl r%d\nadd 14.(r%d),r%d\n",
                        av_reg-1, av_reg, av_reg-1);
        digs();
        if (vald)
                printf("sub %d.,r%d\n", 2*vald, av_reg-1);
        if (extcall)    printf("mov r%d,(r%d)\n", w_reg, av_reg-1);
        else            printf("mov %d.(sp),(r%d)\n", 2*posos, av_reg-1);
        index = FALSE;
        break;


case macro(s,w,s):
/* storew stack var     position on stack */
        digs();
        if (extcall)
                printf("mov r%d,%d.(sp)\n", w_reg, 2*vald);
        else    printf("mov %d.(sp),%d.(r1)\n", 2*posos, 2*vald);
        break;


case macro(s,l,n):
/* str length   location, integer        */
        digs(); loc = vald;
        digs();
        vald = (vald + 2) / 2  +  1;
        valtab[loc] = vald;
        break;
```

```
case macro(s,t,r):
/* string fill          string */
        spar(pbf); value = ((stlen+4)/2)*2;
        printf("<%s\\0>\n", pbf);
        printf(".even\n%d.\n", stlen);
        lsize = lsize - value;
        break;


case macro(s,u,b):
/* subtract     location, valref, valref  */
        digs(); loc = vald;
        digs(); value = vald;
        digs();
        valtab[loc] = valtab[value] - valtab[vald];
        break;


case macro(s,t,i):
/* succ tail id         repr */
        par(pbf);
        printf("t%s:\n", pbf);
        break;


case macro(a,c,f):
/* actual stack frame           #params */
        digs();
        vald = (vald + 1) * 2;
        printf("mov sp,r1\nsub $[%d.+loc%d],sp\n", vald, target);
        break;


case macro(u,n,l):
/* unstack and link             true address */
        printf("mov r1,sp\n");
        digs();
        if (vald)
                printf("jbr t%d\n", vald);
        av_reg = 1;
        break;
```

```
case macro(u,n,r):
/* unstack and return          #params, #locals, true or empty */
        digs(); value = vald;
        digs();
        vald = (vald + value + 1) * 2;
        sym();
        switch(valsym)
        {


        case tag(t,r,u):
        if (ruletype)  /* success return from rule that can fail */
                printf("mov sp,r1\nadd $%d.,r1\nmov (sp),r2\
\nadd $4,r2\njmp (r2)\n", vald);
        else           /* success return from rule that cannot fail */
                printf("mov sp,r1\nadd $%d.,r1\njmp *(sp)\n", vald);
        break;


        case tag(f,l,s):        /* fail return */
        printf("mov (sp),r1\nadd $%d.,sp\njmp (r1)\n", vald);
        break;


        default:
        printf("unknown tag: %s\n", mac);
        }
        break;


case macro(v,a,r):
/* variable     repr, valref */
        par(pbf);
        digs();
        printf("t%s: %d.\n",pbf,valtab[vald]);
        break;
```

```
case macro(x,x,x):
/* comment      comment */
        par(pbf);
        printf("\/ %s\n", pbf);
        break;


case macro(z,n,b):
/* zone bounds          repr, valref (min), repr, valref (max), true add */
        par(pbf);
        digs(); min = valtab[vald];
        par(pbf);
        digs(); max = valtab[vald];
        par(pbf);
        if (min == MIN_int && max == MAX_int)
                printf("jbr t%s\n", pbf);
        else
        if (min == MIN_int)
                printf("cmp r1,$%d.\nbgt 1f\njbr t%s\n1: ", max, pbf);
        else
        if (max == MAX_int)
                printf("cmp r1,\$%d.\nblt 1f\njbr t%s\n1: ", min, pbf);
        else
        printf("cmp r1,$%d.\nblt 1f\ncmp r1,$%d.\nbgt 1f\njbr t%s\n1: ",
                        min, max, pbf);
        break;


case macro(z,n,v):
/* zone value           repr, valref, true addr */
        par(pbf);
        digs();
        par(pbf);
        printf("cmp r1,$%d.\nbne 1f\njmp t%s\n1: ", valtab[vald], pbf);
        break;
default:
printf("unidentified macro: %s\n",mac);
}   /* switch  */
}  /* fi      */
} /* expand  */
```

7 Appendix 2: The run-time system

```
indir = 0
.globl cl, nil, bounds, shuffle, extend, clserr
nil:
/open or create a user file
mov $comma,r2           / communication area
mov 2(r2),rl            / rl points at file administration
cmp rl,$nil             / if there is a file open it
bne openf
jmp start               / else start the execution of the program
openf: mov -4(rl),r3    / get the file type
mov rl,r4
add $518.,r4            / and the file name


/file type = 2:  input charfile, open with mode read
/file type = 4:  output charfile, create with mode write

cmp r3,$2
beq inpf
cmp r3,$4
bne filerr

/output file
outpf:
mov r4,lf+2     / begin address of the filename
sys indir ; lf
br nextfl
.data
l: sys creat ; 0 ; 666
.text

/input file
inpf:
mov r4,2f+2     / begin address of the filename
sys indir ; 2f
.data
2: sys open ; 0 ; 0
.text


nextfl:
bcs filerr      / if something has gone wrong goto filerr
mov r0,(rl)     / put the filedescriptor in the file adm
jbr nextfile    / if not in testmode this instruction is absent
mov $2,r0       / put the name of the opened file on the terminal
sys indir ; lf
.data
l: sys write; 2f; 8.
2: <opened: >
.text
```

```
.globl start, comma
.data
t1: 63.
0.
0.
2
a12
a11: 0;0;0
. = . + 512.
<input0>
.even
0.
0.
4
nil
a12: 0;0;0
. = . + 512.
<output0>
.even
comma: nil;a11
.text

t100:
dec 2(r1)
bge 6f
mov r1,r4
add $6,r4
mov r4,0f
mov 4(r1),0f+2
beq 1f
sub r4,0f+2
mov (r1),r0
sys 0;2f
.data
2: sys write; 0: ..; ..
.text
1: mov r4,4(r1)
mov $512.,2(r1)
6: movb r2,*4(r1)
inc 4(r1)
jmp (r3)

t102:
dec 2(r1)
bge 1f
mov r1,r0
add $6,r0
mov r0,4(r1)
mov r0,0f
mov (r1),r0
sys 0;6f
.data
6: sys read; 0: ..; 512.
.text
dec r0
bmi 4f
mov r0,2(r1)
```

```
clr r0
mov r4,r5
0: tstb (r5)
beq 1f
inc r0          / count the number of chars in the file name
inc r5
br 0b
1: mov r0,2f+4
mov r4, 2f+2
mov $2,r0
sys indir; 2f
.data
2: sys write; 0; 0
.text
mov $2,r0
sys indir; 2f
.data
2: sys write; 3f; 1
3: <\n\n>
.text
nextfile:
mov -2(r1),r1   / get next file administration
cmp r1,$nil     / if there are more files
bne openf       / handle them
jmp start       / else start the program
/ end of opening routine

filerr:
mov $2,r0       / put name of file in trouble on the terminal
sys indir ; 1f
.data
1: sys write ; 2f ; 12.
2: <can't open: >
.even
.text
clr r0
mov r4,r5
0: tstb (r5)
beq 1f
inc r0
inc r5
br 0b

1: mov r0,2f+4
mov r4,2f+2
mov $2,r0
sys indir ; 2f
.data
2: sys write ; 0 ; 0
.text
mov $2,r0
sys indir ; 2f
.data
```

```
2: sys write ; 3f ; 1
3: <\n\n>
.text
sys exit       / and stop



/ extension routine

extend:
/ r2 points at list administration


/ ensure extension
mov 8.(r2),r3  / right limit
add 10.(r2),r3 / new right limit := calibre + right limit
mov 12.(r2),r4 / bump
cmp r3,r4      / if new right limit > bump
bgt shuffle    / goto shuffle

/ update administration
update:
mov r3,8.(r2)  / right limit := new right limit

/ calculate physical top
asl r3
add 16.(r2),r3

/ return
jmp *(sp)


shuffle:
/ plug implementation of reallotment program
/ (ALEPH mobile system)
/ in here
mov $2,r0
sys write; 0f; 8.
.data
0: <shuffle\n>
.text
br cl


bounds:
mov $2,r0              / report bounds error on the terminal
sys write; 0f; 14.
.data
0: <bounds error \n>
.text
br cl                  / and close the files
```

```
/ class error
clserr:
mov $2,r0                / report a class error on the terminal
sys write; 8f; 12.
.data
8: <class error\n>
.text                    / and close the files


/ closing the files
cl: mov $comma,r2        / get communication area
mov 2(r2),r1             / get file administration
tfl: cmp r1,$nil         / if there is a file
bne flush                / check whether it must be flushed
sys exit                 / else stop
flush:
mov -4(r1),r3            / file type
cmp r3,$4                / if no output file
bne fn                   / get next file
fo:                      / else flush
mov r1,r2
add $6,r2                / address of first character in buffer
mov r2,0f
mov 4(r1),0f+2           / number of characters in buffer
beq fn                   / if empty buffer: no flush
sub r2,0f+2
mov (r1),r0
sys 0;2f
.data
2: sys write; 0: 0; 0
.text
fn: mov -2(r1),r1        / next file administration
br tfl
```

8 Appendix 3: An ALICE character input-output program and its AS translation

```
pid "copy characters"
sts 0,0,1
chd 1,?
eva
var 1,1,0,"char"
bfa 11,2,12,"input"
efa 11,2,12,"input"
bfa 12,4,0,"output"
efa 12,4,0,"output"
cmm 0,11
erl 100,pch
erl 102,gch
rut "copy characters"
cll 200
igt 0
acf 0
scl 200
unl 0
ext 0
rli 200,0,0,"copy characters"
sfr 0,0,2
lab 210
efi 102,gch,220
xxx                     get char
lag 11
car 1,1
xxx                     + input
efc 102,gch,220
ldw 1,2
swv 1
xxx                     + char
frw
ece 0
esi 100,pch
xxx                     put char
lag 12
car 1,1
xxx                     + output
lvv 1
cvr 2,2
xxx                     + char
esc 100,pch
ece 210
sti 220
ogt 0
unr 0,0,tru
end "copy characters"
```

```
.globl start, comma
.data
t1: 63.
0.
0.
2
a12
a11: 0;0;0
. = . + 512.
<input\0>
.even
0.
0.
4
nil
a12: 0;0;0
. = . + 512.
<output\0>
.even
comma: nil;a11
.text

t100:
dec 2(r1)
bge 6f
mov r1,r4
add $6,r4
mov r4,0f
mov 4(r1),0f+2
beq 1f
sub r4,0f+2
mov (r1),r0
sys 0;2f
.data
2: sys write; 0: ..; ..
.text
1: mov r4,4(r1)
mov $512.,2(r1)
6: movb r2,*4(r1)
inc 4(r1)
jmp (r3)

t102:
dec 2(r1)
bge 1f
mov r1,r0
add $6,r0
mov r0,4(r1)
mov r0,0f
mov (r1),r0
sys 0;6f
.data
6: sys read; 0: ..; 512.
```

```
.text
dec r0
bmi 4f
mov r0,2(r1)
1: movb *4(r1),r5
inc 4(r1)
mov r5,r1
jmp (r2)
4: jmp (r3)
start:
mov sp,r1
sub $[2.+loc200],sp
mov $1f,(sp)
jbr t200
1: mov r1,sp
jmp c1
t200:
loc200 = 0.
t210:
/                     get char
mov $a11,r1
/                     + input
mov $1f,r2
mov $t220,r3
jbr t102
1: mov r1,t1
/                     + char
/                     put char
mov $a12,r1
/                     + output
mov t1,r2
/                     + char
mov $1f,r3
jbr t100
1: jbr t210
t220:
mov sp,r1
add $2.,r1
jmp *(sp)
```

9 Appendix 4: An ALICE version of the Ackermann function and its AS translation

```
pid "ackermann"
sts 0,0,5
int 1,0
int 2,1
int 3,7
int 4,3
mcn 5,nln
eva
css 21,1
css 22,2
css 23,3
css 24,4
css 25,5
bfa 101,4,0,"output"
efa 101,4,0,"output"
cmm 0,101
erl 9,pls
erl 3,dcr
erl 7,mor
erl 5,inc
xxx            put int + file + >int:
erl 12,pnt
xxx            put char + file + >char:
erl 11,pch
erl 14,eql
erl 13,trp
xxx            ack + >m + >n + r>:
rli 1,0
sfr 3,0
cig 1,1
cig 2,2
xxx            m = 0,
efi 14,eql,502
lvs 1
cvr 1,1
lvc 21,1
cvr 2,2
efc 14,eql,502
ece 0
xxx            plus + m + 1 + r;
esi 9,pls
lvs 2
cvr 1,1
lvc 22,2
cvr 2,2
esc 9,pls
ldw 1,3
sws 3
frw
```

```
ece 503
lab 502
xxx             n = 0,
efi 14,eql,504
lvs 2
cvr 1,1
lvc 21,1
cvr 2,2
efc 14,eql,504
ece 0
xxx             decr + m,
esi 3,dcr
lvs 1
cvr 1,1
esc 3,dcr
ldw 1,1
sws 1
frw
ece 0
xxx             ack + m + n + r;
cll 1
igt 2
acf 3,0
lvs 1
cvr 1,1
lvc 22,2
cvr 2,2
scl 1
ldw 1,3
sws 3
frw
unl 503
lab 504
xxx             decr + n,
esi 3,dcr
lvs 2
cvr 1,1
esc 3,dcr
ldw 1,1
sws 2
frw
ece 0
xxx             ack + m + n + r,
cll 1
igt 2
acf 3,0
lvs 1
cvr 1,1
lvs 2
cvr 2,2
scl 1
ldw 1,3
sws 3
```

```
unl 0
xxx            decr + m,
esi 3,dcr
lvs 1
cvr 1,1
esc 3,dcr
ldw 1,1
sws 1
frw
ece 0
xxx            ack + m + r + r.
cll 1
igt 2
acf 3,0
lvs 1
cvr 1,1
lvs 3
cvr 2,2
scl 1
ldw 1,3
sws 3
unl 0
xxx            end of ack
sti 503
ogt 1
rog 1,3
unr 3,0,tru
rut "ackermann"
cll 2
igt 0
acf 0,0
scl 2
unl 0
ext 0
xxx            ackermann - i - j - r:
rli 2,0
sfr 0,3
xxx            0 -> i,
esi 13,trp
lvc 21,1
cvr 1,1
esc 13,trp
ldw 1,1
sws 2
frw
ece 0
lab 602
xxx            more + i + 3;
efi 7,mor,607
lvs 1
cvr 1,1
lvc 24,4
cvr 2,2
```

```
efc 7,mor,607
ece 608
lab 607
xxx                0 -> j,
esi 13,trp
lvc 21,1
cvr 1,1
esc 13,trp
ldw 1,1
sws 2
frw
ece 0
lab 603
xxx                more + j + 7;
efi 7,mor,610
lvs 2
cvr 1,1
lvc 23,3
cvr 2,2
efc 7,mor,610
ece 0
xxx                incr + i,
esi 5,inc
lvs 1
cvr 1,1
esc 5,inc
ldw 1,1
sws 1
frw
ece 602
lab 610
xxx                ack + i + j + r,
cll 1
igt 2
acf 3,0
lvs 1
cvr 1,1
lvs 2
cvr 2,2
scl 1
ldw 1,3
sws 3
frw
unl 0
xxx                put int + output + i,
esi 12,pnt
lag 101
car 1,1
lvs 1
cvr 2,2
esc 12,pnt
ece 0
xxx                put int + output + j,
```

```
esi 12,pnt
lag 101
car 1,1
lvs 2
cvr 2,2
esc 12,pnt
ece 0
xxx             put int + output + r,
esi 12,pnt
lag 101
car 1,1
lvs 3
cvr 2,2
esc 12,pnt
ece 0
xxx             putchar + output + newline,
esi 11,pch
lag 101
car 1,1
lvc 25,5
cvr 2,2
esc 11,pch
ece 0
xxx             incr + j,
esi 5,inc
lvs 2
cvr 1,1
esc 5,inc
ldw 1,1
sws 2
frw
ece 603
xxx             end of ackermann
sti 608
ogt 0
unr 0,0,tru
end "ackermann"
```

```
.globl start, comma
.data
0.
0.
4
nil
a101: 0;0;0
. = . + 512.
<output\0>
.even
comma: nil;a101
.text

/               put int + file + >int:
t12:
mov $4f,r4
mov $"  ,r5
mov r5,3f
mov r5,3f+2
mov r5,3f+4
mov r3,4f
mov r2,r3
bge 1f
neg r3
movb $'-,r5
1: clr r2
div $10.,r2
add $'0,r3
movb r3,-(r4)
mov r2,r3
bne 1b
movb r5,-(r4)
.data
3: <      >
4: 0
.text
mov $3b,r3
3: dec 2(r1)
bge 6f
mov r1,r2
add $6,r2
mov r2,0f
mov 4(r1),0f+2
beq 1f
sub r2,0f+2
mov (r1),r0
sys 0;2f
.data
2: sys write; 0: .. ; ..
.text
1: mov r2,4(r1)
mov $512.,2(r1)
6: movb (r3)+,*4(r1)
```

```
inc 4(r1)
cmp r3,$4b
bne 3b
jmp *4b

/               put char + file + >char:
t11:
dec 2(r1)
bge 6f
mov r1,r4
add $6,r4
mov r4,0f
mov 4(r1),0f+2
beq 1f
sub r4,0f+2
mov (r1),r0
sys 0;2f
.data
2: sys write; 0: ..; ..
.text
1: mov r4,4(r1)
mov $512.,2(r1)
6: movb r2,*4(r1)
inc 4(r1)
jmp (r3)
/               ack + >m + >n + r>:
t1:
loc1 = 0.
/               m = 0
mov 2.(sp),r1
mov $0.,r2
cmp r1,r2
bne t502
/               plus + m + 1 + r;
mov 4.(sp),r1
mov $1.,r2
add r2,r1
mov r1,6.(sp)
jbr t503
t502:
/               n = 0
mov 4.(sp),r1
mov $0.,r2
cmp r1,r2
bne t504
/               decr + m
mov 2.(sp),r1
dec r1
mov r1,2.(sp)
/               ack + m + n + r;
mov sp,r1
sub $[8.+loc1],sp
mov 2.(r1),2.(sp)
```

```
mov $1.,4.(sp)
mov $1f,(sp)
jbr t1
1: mov 6.(sp),6.(r1)
mov r1,sp
jbr t503
t504:
/               decr + n
mov 4.(sp),r1
dec r1
mov r1,4.(sp)
/               ack + m + n + r
mov sp,r1
sub $[8.+loc1],sp
mov 2.(r1),2.(sp)
mov 4.(r1),4.(sp)
mov $1f,(sp)
jbr t1
1: mov 6.(sp),6.(r1)
mov r1,sp
/               decr + m
mov 2.(sp),r1
dec r1
mov r1,2.(sp)
/               ack + m + r + r.
mov sp,r1
sub $[8.+loc1],sp
mov 2.(r1),2.(sp)
mov 6.(r1),4.(sp)
mov $1f,(sp)
jbr t1
1: mov 6.(sp),6.(r1)
mov r1,sp
/               end of ack
t503:
mov sp,r1
add $8.,r1
jmp *(sp)
start:
mov sp,r1
sub $[2.+loc2],sp
mov $1f,(sp)
jbr t2
1: mov r1,sp
jmp c1
/               ackermann - i - j - r:
t2:
loc2 = 6.
/               0 -> i
mov $0.,r1
mov r1,4.(sp)
t602:
/               more + i + 3;
```

```
mov 2.(sp),r1
mov $3.,r2
cmp r2,r1
bge t607
jbr t608
t607:
/               0 -> j
mov $0.,r1
mov r1,4.(sp)
t603:
/               more + j + 7;
mov 4.(sp),r1
mov $7.,r2
cmp r2,r1
bge t610
/               incr + i
mov 2.(sp),r1
inc r1
mov r1,2.(sp)
jbr t602
t610:
/               ack + i + j + r
mov sp,r1
sub $[8.+locl],sp
mov 2.(r1),2.(sp)
mov 4.(r1),4.(sp)
mov $1f,(sp)
jbr t1
1: mov 6.(sp),6.(r1)
mov r1,sp
/               put int + output + i
mov $a101,r1
mov 2.(sp),r2
mov $1f,r3
jbr t12
1: /            put int + output + j
mov $a101,r1
mov 4.(sp),r2
mov $1f,r3
jbr t12
1: /            put int + output + r
mov $a101,r1
mov 6.(sp),r2
mov $1f,r3
jbr t12
1: /            putchar + output + newline
mov $a101,r1
mov $10.,r2
mov $1f,r3
jbr t11
1: /            incr + j
mov 4.(sp),r1
inc r1
```

```
mov r1,4.(sp)
jbr t603
/               end of ackermann
t608:
mov sp,r1
add $2.,r1
jmp *(sp)
```