# Epic and ARM -- User's Guide --

H.R.Walters

Software Engineering (SEN)

**SEN-R9724 December 31, 1997**

Report SEN-R9724
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB  Amsterdam
The Netherlands

# Epic and ARM
# – User's Guide –

H.R.Walters
pum@babelfish.nl

*Babelfish*

*Korenbloemweg 23, 2403 GA Alphen a/d Rijn, The Netherlands*

and

*CWI*

*P.O. Box 94079, 1090 GB Amsterdam, The Netherlands*

ABSTRACT

We give a brief introduction to Epic and to ARM (they are discussed in more detail elsewhere). We show how to use the Epic compiler and how to execute ARM code. Then we describe ARM's API (application programmer's interface) which allows ARM to be used as a plug-in library. We describe how to access external functions from ARM and how to add external data types.

*1991 Mathematics Subject Classification:* 68N17: Logic Programming; 68N20: Compilers and Generators; 68Q05: Models of Computation; 68Q40: Symbolic Computation; 68Q42: Rewriting Systems and 68Q65: Algebraic specification
*1991 Computing Reviews Classification System:* D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.4 [Programming Languages]: Processors - Compilers; D.1.6: Logic Programming; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages, *Algebraic approaches to semantics*; F.4.1 [Mathematical Logic and Formal Languages]: Logic Programming.
*Keywords and Phrases:* Epic, ARM, compiler, interpreter, term rewriting systems, abstract machines, program transformations
*Note:* Part of this work is carried out under project SEN1.2 "Software development"

# How to read this guide

This is the user's manual of Epic and ARM. If you know what term rewriting is and want to use ARM immediately, you should just read the introduction and Section 1, which explains how to use the Epic compiler and (stand-alone) ARM interpreter.

If you want to use ARM as a plug-in term-rewriting library in your application, you should also read Section 4 of this guide.

In chapter 5 we introduce Epic. It is discussed more formally in [KW96b]. In Appendix II we discuss the `given` construct, which is an extension of Epic. We describe a tool that translates programs using `given` to equivalent Epic programs. Note that the Epic compiler uses this construct.

Several aspects concerning Epic and ARM have already been published elsewhere, and will not be discussed in this guide. In particular we refer to [WK96], which presents the definition of Epic's syntax and semantics, and [FKW97], which presents a formal description of ARM.

# Table of Contents

# Introduction

ARM is an abstract machine model for an efficient term rewriting engine. Epic is a language in which unconditional, left-linear term (graph) rewriting systems can be expressed. At the time of writing this guide the current Epic/ARM version is 1.0.2.

Epic is mainly intended to be generated by other tools, such as compilers or theorem provers. It can be used by hand – the Epic compiler is in fact hand-written in Epic, – but it is spartan and not for the faint of heart.

Term (graph) rewriting systems are becoming increasingly important for the implementation of theorem provers [GL91, Fra94, GG91, KZ89, Bou94], verification tools, algebraic specifications [EM85, BHK89, Vis96, HM93], compiler generators [ESL89], language prototyping [vDHK96], program analyzers [BDHF96] and functional programming languages [PvE93].

The Epic compiler translates Epic sources into ARM object code. This code – which is represented textually and is vaguely reminiscent of assembler code, – is suitable input for the ARM interpreter. The ARM interpreter can be used stand-alone, in which case it loads a number of ARM files and proceeds to reduce an input term, or it can be used as a plug-in library in an application, in which case the application can direct it to load ARM object files, to reduce terms, etcetera.

Below is an Epic module in which addition and multiplication on binary numbers are defined. Note that the type names are misleading because (currently) the names of types are ignored. In Epic only the names of a function and the number of its arguments are significant.

```
module Numbers
types
  o: -> Nat;    i: -> Nat;
  ap     : Nat # Nat -> Nat;
  plus   : Nat # Nat -> Nat;
  times  : Nat # Nat -> Nat;
rules
  ap(o,X) = X;
ap(X,ap(Y,Z)) = ap(plus(X,Y),Z);

plus(o,X) = X;
plus(i,o) = i;
plus(i,i) = ap(i,o);
plus(i,ap(X,Y)) = ap(X,plus(i,Y));
plus(ap(X,Y),Z) = ap(X,plus(Y,Z));

times(o,X) = o;
times(i,X) = X;
times(ap(X,Y),Z) = ap(times(X,Z),times(Y,Z));
```

In this module, five function symbols are defined. The constants `o` and `i`, representing the binary digits 0 and 1 (Epic identifiers can not be numeric); the function `ap`, which represents appending a bit at the end of a string of bits; and the functions `plus` and `times` with their usual meaning. Further details (including a proof of correctness) of this program are discussed in [WZ95].

The module `Numbers` can be put in an Epic source file "numbers.ep". The Epic/ARM environment uses the extension `.ep` for Epic sources (and `.arm` for ARM files). Now, we can apply the Epic compiler:

```
$ epic numbers.ep
```

We will indicate what you type in a unix shell with "$" (you should not type the character "$" itself, though).

The command above invokes the Epic compiler, which produces the ARM code for `Numbers` in a file `numbers.arm`.

In addition to this ARM file, an executable unix script is produced which can invoke the ARM interpreter. The script is useful when many ARM files are used, or when non-standard compilation options are used. We will discuss this later. In our simple example the script will only read something like

```
/.../bin/arm -r/.../numbers.arm $*
```

We can invoke the script by typing

```
$ numbers
```

Or, assuming the ARM interpreter can be found in our current unix path, and the ARM file is in our working directory, by (note the absence of a space after `-r`):

```
$ arm -rnumbers.arm
```

This starts the interpreter which loads the ARM file and proceeds to wait for a term in prefix-bracketed textual notation on standard input (note that there is no prompt to indicate that ARM is waiting). We can type in `times(ap(ap(i,o),i),ap(ap(i,i,o)))`<cr>,

– where <cr> signifies pressing the return key, – and ARM will reduce this term, resulting in `ap(ap(ap(ap(i,i),i),i),o)` being produced on standard output. In some UNIX environments the output may actually be overwritten with the next prompt. This is due to the fact that ARM does not produce an additional newline character. In this case we could invoke the module as follows instead, which assures that we can see the output.

```
$ numbers; echo
```

# Chapter 1
# Using Epic and ARM

In this section we discuss how to use Epic and ARM. We refer to Chapter 5 for an introduction to Epic.

## 1. Using the Epic compiler

The Epic compiler is invoked as follows:

```
$ epic options epic-and-arm-files
```

The *epic-and-arm-files* are a sequence of files with extension `.ep` or `.arm`. Extension `.ep` signifies that the file should be compiled to ARM code and should be included in the tool; extension `.arm` signifies that the file has already been compiled, and should merely be included (e.g., library modules).

The Epic compiler compiles all Epic files, and produces a shell script which invokes the ARM interpreter with all indicated ARM files.

The Epic compiler accepts the following options:

`--version`
: print the current version

`-o`*foo*
: the generated shell script will be named `foo`. The default is the name of the last Epic or ARM file (without extension) appearing on the command line invoking the Epic compiler.

`-S`
: the tool uses lifted I/O. See Section 6 for details.

`-a`*bar*
: the parse-reduce-pretty-print function will be called `bar` instead of the default `ppp-foo`.

`-E`*dir*

use `dir` to locate executables and libraries, rather than the default (as defined during installation).

`-local`

use the location where Epic and ARM were created rather than where they were installed, to locate executables and libraries (i.e., use prefix rather than exec-prefix).

`-D`*dir*

use `dir` to find sources and produce objects and scripts, rather than the working directory.

Invoking the Epic compiler compiles all Epic files (resulting in ARM files) and produces a script with the name of the tool in which the ARM interpreter is invoked with the right ARM files and other options. This script can be used to activate the tool.

## 2. Using the ARM interpreter

The ARM interpreter is invoked as follows:

```
$ arm options
```

By default ARM reads its input from standard input (`stdin`), and produces its result on standard output (`stdout`). Error messages and debugging and tracing information are produced on the standard error stream (`stderr`).

ARM accepts the following options:

`-l`

lift the input and lower the output. See Section 6 for details.

`-a`*foo*

apply function `foo` to the input. Used mainly in conjunction with `-l` in order to indicate the parse-reduce-pretty-print function.

`-r`*file*

read the ARM file `file`.

`-m`*size*

limits ARM's maximal memory usage. `Size` is a digit from 0 to 9 and sets the memory limit to a value of 1 Mb, 2 Mb, 5 Mb, 10 Mb, 20 Mb, 50 Mb, 100 Mb, 200 Mb, 500 Mb and 1 Gb, respectively. Reaching the maximum is a fatal error for ARM. This option is used to debug programs suspected of infinite recursion.

`-R`*count*

print and limit the total number of used reduction steps. After a successful run, prints the number of (semi) reduction steps used. When this number reaches `count`, ARM quits with a fatal error. Use 0 for infinity (i.e., the number is printed but the limit is never reached). Note that the number printed is usually somewhat larger than the actual number of rewrite steps; ARM counts building a node when the outer-most function symbol can not be reduced as a separate step.

`-S`
: print ARM's memory usage as blocks are allocated.

`-G`
: print garbage collection information (total number of used and freed nodes at each garbage collection).

`-T`
: trace Epic functions. For each reduction print the reduced Epic function.

`-F`*function*
: start tracing only after the first reduction of the function `function`. Used to skip large traces before things get interesting.

`-A`
: (implies `-T`) Also print a number of arguments to that function. Note that ARM does not know the arity of functions. Hence a fixed number is displayed each time; see the `-L` and `-Y` options.

`-D`
: (implies `-T`) Also print a number of elements on top of each ARM stack (see `-L`, `-X` and `-Y` options).

`-M`
: trace individual ARM instructions.

`-L`
: limit the depth of stackdumps (otherwise the entire stack is displayed) and the depth of terms be printed (otherwise the entire terms are printed) when the `-A` or `-D` options are used.

`-X`*depth*
: set the depth of stacksdumps to `depth`.

`-Y`*depth*
: set the depth of the part of a term to be printed to `depth`.

`-P`
: produce profiling information on Epic functions and ARM instructions. This information concerns the number of times a function has reduced or an instruction has been executed. Note that this option requires a lot of execution time.

In Section 1.4 we will discuss the generated output when various tracing options are used.

## 3. Example

To illustrate the previous sections, we will present an example: Bincalc. Note that a full introduction to Epic is given in Chapter 5.

Bincalc is a binary calculator written in Epic. It consists of three modules, one of which is the module Numbers from the introduction. Here are the three Bincalc modules: Numbers, Io, and Bincalc.

## *3.1 Numbers*

In this module, five function symbols are defined. The constants `o` and `i`, representing the binary digits 0 and 1; the function `ap`, which represents appending a bit at the end of a string of bits; and the functions `plus` and `times` with their usual meaning. Further details (including a proof of correctness) of this program are discussed in [WZ95].

Note that Epic is single-sorted and that the use of sort-names is somewhat misleading; only the number of arguments of a function is significant.

```
module Numbers
types
  o      :           -> Nat;
  i      :           -> Nat;
  ap     : Nat # Nat -> Nat;
  plus   : Nat # Nat -> Nat;
  times  : Nat # Nat -> Nat;
rules
  ap(o,X) = X;
  ap(X,ap(Y,Z)) = ap(plus(X,Y),Z);
  plus(o,X) = X;
  plus(i,o) = i;
  plus(i,i) = ap(i,o);
  plus(i,ap(X,Y)) = ap(X,plus(i,Y));
  plus(ap(X,Y),Z) = ap(X,plus(Y,Z));
  times(o,X) = o;
  times(i,X) = X;
  times(ap(X,Y),Z) = ap(times(X,Z),times(Y,Z));
```

## *3.2 Io*

In the module Io a parser and pretty-printer for binary expressions with addition and multiplication are defined.

All characters allowed in the input are defined explicitly (Epic doesn't have pre-defined functions). Note that all characters and the string constructors `str`, `eos` and `cat` must be declared when they are use. Even though they are built-in into ARM (in order to be able to lift terms, see Section 6 for details), they are not built-in into Epic. The annotation `free` means that no rules will be defined for them in *any* module. See Section 3 for more details on this.

The functions defined in the module Numbers that will be used in this module must be declared as *external*, meaning that they are used here but are defined elsewhere. See Section 3 for more details on this.

Module Io uses one of Epic's short-hand features: the term `'foo'` is short-hand for the string-term `str('f, str('o, str('o, eos)))`. The term `'bar'+X` is short-hand for `str('b, str('a, str('r, X)))`.

The parser defined here maintains a tuple consisting of the expression recognized so-far, and the string of not-yet inspected characters. Function `nb` skips non-blanks. Function `parse-exp` parses either an expression enclosed in parentheses, or a constant. The function `parse-num` reads a sequence of zeroes and ones, and computes their proper value. The function `trail` handles infix operators (which are interpreted from right to left). The auxiliary function `enc-exp` reads an expression enclosed in parentheses, and the `aft-exp` skips the closing parenthesis of such an expression.

Note that the parser produces a term which consists of functions defined in the module Numbers. Such a term is implicitly reduced to its value. Hence, the Bincalc tool has a parser and a pretty-printer, but no explicit *reduce* function. The pretty-printer is trivial. Note the use of `cat`.

The annotations `free` and `external` signify functions used but not defined in this module. `Free` means that it isn't defined in *any* module; `external` means that it is defined in another module (or at least is implicitly or explicitly free there).

```
module Io
types
  \n:                    -> Char {free};
  ' :                    -> Char {free};
  '(:                    -> Char {free};
  '):                    -> Char {free};
  '*:                    -> Char {free};
  '+:                    -> Char {free};
  '0:                    -> Char {free};
  '1:                    -> Char {free};
  eos       :            -> Text {free};
  str       : Char # Text -> Text {free};
  cat       : Text # Text -> Text {free};
  o         :            -> Nat {external};
  i         :            -> Nat {external};
  ap        : Nat # Nat   -> Nat {external};
  plus      : Nat # Nat   -> Nat {external};
  times     : Nat # Nat   -> Nat {external};
  parse     : Text        -> Nat;
  get-val   : Tuple       -> Text;
  enc-exp   : Tuple       -> Text;
  aft-exp   : Num # Text  -> Tuple;
  plus-exp  : Num # Tuple -> Tuple;
  mul-exp   : Num # Tuple -> Tuple;
  nb        : Text        -> Text;
  parse-num : Text # Nat  -> Tuple;
  parse-exp : Text        -> Tuple;
  trail     : Text # Nat  -> Tuple;
  tuple     : Nat # Text  -> Tuple {free};
  print     : Num         -> Text;
rules
  parse(Txt) = get-val(parse-exp(nb(Txt)));
  get-val(tuple(Val,Rest)) = Val;
  parse-exp('('+Txt) = enc-exp(parse-exp(nb(Txt)));
  enc-exp(tuple(Val,Rest)) = aft-exp(Val,nb(Rest));
  aft-exp(Val,')'+Rest) = trail(nb(Rest),Val);
  parse-exp(Txt) = parse-num(Txt,o);
  parse-num('0'+Txt,Val) = parse-num(Txt,plus(Val,Val));
  parse-num('1'+Txt,Val) = parse-num(Txt,plus(plus(Val,Val),i));
  parse-num(Txt,Val) = trail(Txt,Val);
  trail('+'+Txt,Val1) = plus-exp(Val1,parse-exp(Txt));
```

```
        plus-exp(Val1,tuple(Val2,Rest)) = tuple(plus(Val1,Val2),Rest);
        trail('*'+Txt,Val1) = mul-exp(Val1,parse-exp(Txt));
        mul-exp(Val1,tuple(Val2,Rest)) = tuple(times(Val1,Val2),Rest);
        trail(Txt,Val) = tuple(Val,Txt);
        nb('\n'+Txt) = Txt ;
        nb(' '+Txt) = Txt ;
        nb(Txt) = Txt;
        print(ap(A,B)) = cat(print(A),print(B));
        print(o) = '0';
        print(i) = '1';
```

*3.3 Bincalc*

The module Bincalc is trivial.

```
      module Bincalc
      types
        ppp_bincalc: Text -> Text;
        parse: Text -> Nat {external};
        print: Nat -> Text {external};
      rules
        ppp_bincalc(Text) = print(parse(Text));
```

*3.4 Making the tool*

Bincalc is made by invoking Epic as follows

```
    $ epic -S numbers.ep io.ep bincalc.ep
```

This compiles the three modules, and produces the script **bincalc** shown below. If you are unfamiliar with shell scripts, please skip to the next section.

Note that the -l and -a options have been added due to the -S option.

```
  #! /bin/sh
  function=ppp_bincalc
  PREFIX=/.../epic
  EPREFIX=/.../epic
  EP=$EPREFIX
  case $1 in
  -local) EP=$PREFIX
  shift 1;;
  *) : ;;
  esac
  $EP/bin/arm  -r/.../epic/examples/bincalc/io.arm \
               -r/.../epic/examples/bincalc/numbers.arm\
               -r/.../epic/examples/bincalc/bincalc.arm\
               -l -a$function $*
```

## 4. Generating trace information

In this section we discuss the output generated by various tracing options of ARM. We use the Bincalc example, and we have used the following invocation of ARM.

```
  $ echo " txt" | arm  options -rio.arm -rnumbers.arm -rbincalc.arm -l -appp_bincalc
```

Here, *options* signify the various tracing options used in each example, and *txt* signifies the binary computation used for each example. We used two different computations: "1 + 1" and "1 * 10 * 11 * 100 * 101 * 110 * 111 * 1000 * 1001 * 1010 * 1011 * 1100 * 1101 * 1110 * 1111 * 10000 * 10001 * 10010 * 10011 * 10100 * 10101 * 10110 * 10111 * 11000 * 11001 * 11010 * 11011 * 11100 * 11101 * 11110 * 11111 * 100000 * 100001 * 100010 * 100011 * 100100 * 100101 * 100110 * 100111 * 101000 * 101001 * 101010 * 101011 * 101100 * 101101 * 101110 * 101111 * 110000 * 110001 * 110010 * 110011 * 110100 * 110101 * 110110 * 110111 * 111000 * 111001 * 111010 * 111011 * 111100 * 111101 * 111110 * 111111". The latter, which computes 63! (the faculty of 63) is used mainly to make sure the garbage collector is activated (once). We'll abbreviate this string to "<u>63!</u>" in this section (this is only used in our presentation, and is not significant for ARM).

Below we only give the tracing information, and do not show ARM's normal output (the result of the computation).

## *4.1 Number of reductions*

*description:*

> Print the number of semi rewrite steps. A semi rewrite step is a proper rewrite step or an internal step in which a sub-term has been found to be irreducible, and is built as a normal form. ARM can not distinguish these two situations.

*options:*

> -R0

*computation:*

> <u>63!</u>

*output:*

```
reductions: 144767
```

## *4.2 Garbage collection activity*

*description:*

> Trace garbage collector activity. The garbage collector is called once in this computation.

*options:*

> -G

*computation:*

> <u>63!</u>

*output:*

```
[gc:114 nds in use, 30606 nds freed]
```

## *4.3 Memory usage*

*description:*

Show type and sizes in bytes (individual and sum-total so far, labeled `sigma`) of allocated memory blocks. Note that ARM is designed to support individual threads of execution, but that this is not yet supported; hence a structure called Thread is allocated. The types include the following:

**clss:** class, used to store class information of (internal) types of memory blocks.

**body:** used to store the executable code of ARM functions

**uses,**

**hdrs:** both are used to store references between ARM functions

**id's:** used to store function names

**glob:** a global block to hold the entire ARM state

**thrd:** one ARM thread

**Astk,**

**Cstk,**

**Tstk,**

**Xstk:** one stack (argument, control, traversal and auxiliary). See Section 3 for further explanations.

**heap:** one chunk of heap storage

**flts:** one chunk of memory to hold floating point numbers

*options:*

-S

*computation:*

<u>63!</u>

*output:*

```
clss: block of       39 (sigma=      39)
clss: block of       39 (sigma=      78)
clss: block of       39 (sigma=     117)
clss: block of       39 (sigma=     156)
body: block of    10054 (sigma=   10210)
uses: block of    32058 (sigma=   42268)
hdrs: block of     6458 (sigma=   48726)
id's: block of    10267 (sigma=   58993)
clss: block of       39 (sigma=   59032)
clss: block of       39 (sigma=   59071)
clss: block of       39 (sigma=   59110)
clss: block of       39 (sigma=   59149)
glob: block of       27 (sigma=   59176)
thrd: block of       71 (sigma=   59247)
```

```
Cstk: block of     30759 (sigma=    90006)
Astk: block of     30759 (sigma=   120765)
Tstk: block of     30759 (sigma=   151524)
Xstk: block of     30759 (sigma=   182283)
heap: block of    491558 (sigma=   673841)
clss: block of        39 (sigma=   673880)
flts: block of    160058 (sigma=   833938)
hdrs: block of      6458 (sigma=   840396)
```

## *4.4 Trace functions*

*description:*

Trace reduced functions. This option results in all function names being printed when they are reduced. Note that the Epic compiler introduces many auxiliary functions during compilation. Such functions contain the symbol # in their name. The role of these functions falls beyond the scope of this report. See [FKW97] for more details. Ignoring lines without this symbol may produce clearer information, which we have shown in the second column

The auxiliary symbol `stop` is used by ARM to recognize the end of computation

*options:*

-T

*computation:*

1+1

*output:*

```
ppp_bincalc                                            ppp_bincalc
parse                                                  parse
"parse#N#0"                                            o
o                                                      parse-num
"parse-exp#0#1#1"                                      i
parse-num                                              plus
i                                                      plus
"parse-num#DS#str#DS#'1#0#RB#N#N#1#RB#2#RB#3#R"        parse-num
plus                                                   parse-exp
"parse-num#DS#str#DS#'1#0#RB#N#1#R"                    o
plus                                                   parse-num
"parse-num#DS#str#DS#'1#0#RB#1#R"                      i
parse-num                                              plus
parse-exp                                              plus
o                                                      parse-num
"parse-exp#0#1#1"                                      plus-exp
parse-num                                              plus
i                                                      o
"parse-num#DS#str#DS#'1#0#RB#N#N#1#RB#2#RB#3#R"        ap
plus                                                   print
"parse-num#DS#str#DS#'1#0#RB#N#1#R"                    print
```

```
plus
"parse-num#DS#str#DS#'1#0#RB#1#R"
parse-num
"trail#DS#str#DS#'+#0#RB#1#R"
plus-exp
plus
o
"plus#DS#i#DS#i#1#R"
ap
"plus-exp#1#DS#tuple#0#R"
"parse#0#R"
print
print
"print#DS#ap#N#0#RB#1#R"
print
"print#DS#ap#0#R"
stop
```

```
print
stop
```

## *4.5 Trace from*

*description:*

Start tracing at the first reduction of this symbol. Note that this can be used together with all other tracing options (T, A, M, D)

*options:*

-Fplus-exp

*computation:*

1+1

*output:*

```
plus-exp
plus
o
"plus#DS#i#DS#i#1#R"
ap
"plus-exp#1#DS#tuple#0#R"
"parse#0#R"
print
print
"print#DS#ap#N#0#RB#1#R"
print
"print#DS#ap#0#R"
stop
```

## *4.6 Trace with arguments*

*description:*

Trace reduced functions and show some arguments. Use default values (no more than four stack-items, terms no deeper than three) *options:*

-T -A -Fplus-exp

*computation:*

1+1

*output:*

```
plus-exp() i tuple(i,str(\n,eos))
plus() i i str(\n,eos)
o() str(\n,eos)
"plus#DS#i#DS#i#1#R"() o str(\n,eos)
ap() i o str(\n,eos)
"plus-exp#1#DS#tuple#0#R"() ap(i,o) str(\n,eos)
"parse#0#R"() tuple(ap(i,o),str(\n,eos))
print() ap(i,o)
print() o
"print#DS#ap#N#0#RB#1#R"() str('0,eos)
print() i str('0,eos)
"print#DS#ap#0#R"() str('1,eos) str('0,eos)
stop() cat(str('1,eos),str('0,eos))
```

## *4.7 Trace ARM instructions*

*description:*

Trace individual ARM instructions. This option is used mainly to debug ARM, compilers and the external functions. Each ARM cycle the instruction executed is printed.

The instruction such as `_dockD` are discussed in some detail in [FKW97].

*options:*

-M -Fplus-exp

*computation:*

1+1

*output:*

```
_usageD
_skipD
_tpushaD
_dockD
_tdropD
_retractD
_cpushD
_gotoD
_usageD
_tpushaD
_dockD
```

```
_tpushaD
_dockD
_tdropD
_cpushD
_gotoD


••• skipped a bit


_aabuildD
_aabuildD
_aabuildD
_recycleD
_usageD
_retractD
_cpushD
_gotoD
_usageD
_tpushaD
_dockD
_tdropD
_aabuildD
_aabuildD
_aabuildD
_recycleD
_aabuildD
_recycleD
f_stop
```

## *4.8 Trace ARM instructions with stackdump*

*description:*

Produce a stackdump at each ARM cycle. The top few elements of each of the three ARM stacks are shown.

The instruction such as `_dockD` are discussed in some detail in [FKW97].

*options:*

-D -Fplus-exp

*computation:*

1+1

*output:*

```
  CS     3:"parse#0#R"  print  stop
  AS     2: i  tuple(i,str(\n,eos))
  TS     0:
_usageD
  CS     3:"parse#0#R"  print  stop
  AS     2: i  tuple(i,str(\n,eos))
  TS     0:
_skipD
  CS     3:"parse#0#R"  print  stop
  AS     1: tuple(i,str(\n,eos))
  TS     1: i
_tpushaD
  CS     3:"parse#0#R"  print  stop
  AS     1: tuple(i,str(\n,eos))
  TS     2: tuple(i,str(\n,eos))  i
_dockD
  CS     3:"parse#0#R"  print  stop
  AS     2: i  str(\n,eos)
  TS     2: tuple(i,str(\n,eos))  i
_tdropD
  CS     3:"parse#0#R"  print  stop
  AS     2: i  str(\n,eos)
  TS     1: i
_retractD
  CS     3:"parse#0#R"  print  stop
  AS     3: i  i  str(\n,eos)
  TS     0:
_cpushD
  CS     4:"plus-exp#1#DS#tuple#0#R"  "parse#0#R"  print
  AS     3: i  i  str(\n,eos)
  TS     0:
_gotoD


••• skipped a large part


  CS     2:"print#DS#ap#0#R"  stop
  AS     2: str('1,eos)  str('0,eos)
  TS     0:
_recycleD
  CS     1:stop
  AS     2: str('1,eos)  str('0,eos)
  TS     0:
_aabuildD
  CS     1:stop
  AS     1: cat(str('1,eos),str('0,eos))
  TS     0:
_recycleD
  CS     0:
  AS     1: cat(str('1,eos),str('0,eos))
  TS     0:
f_stop
```

## *4.9 Produce profiling information*

*description:*

Produce profiling information concerning the use of Epic functions and of ARM instructions.

The output shown here is exactly as it is produced by ARM.

*options:*

-P

*computation:*

63!

*output:*

```
trs profiling:   steps       144767  %steps
                      parse         1   0.0
                "parse#N#0"         1   0.0
                "parse#0#R"         1   0.0
                       stop         1   0.0
                    mul-exp        62   0.0
  "mul-exp#1#DS#tuple#0#R"         62   0.0
                  parse-exp        62   0.0
"trail#DS#str#DS#'*#0#RB#1#R"        62   0.0
       "parse-exp#0#1#1"        63   0.0
"parse-num#DS#str#DS#'0#0#RB#1#R"       129   0.1
                          i       192   0.1
"parse-num#DS#str#DS#'1#0#RB#N#N#1#RB#2#RB#3#R"       192   0.1
"parse-num#DS#str#DS#'1#0#RB#N#1#R"       192   0.1
"parse-num#DS#str#DS#'1#0#RB#1#R"       192   0.1
"times#DS#ap#N#0#RB#1#RB#2#R"       253   0.2
"print#DS#ap#N#0#RB#1#R"       289   0.2
      "print#DS#ap#0#R"       289   0.2
                  parse-num       384   0.3
                      times       568   0.4
                      print       579   0.4
"plus#DS#i#DS#ap#0#RB#1#R"      7186   5.0
     "plus#DS#i#DS#i#1#R"      7377   5.1
                          o      7440   5.1
   "plus#DS#ap#0#RB#1#R"     24155  16.7
                       plus     31854  22.0
                         ap     63181  43.6


arm profiling:     cycles   1178677  %steps
                     tpusha    235848  20.0
                       dock    202751  17.2
                      adrop    134805  11.4
                    retract     99525   8.4
                       goto     79859   6.8
```

```
 recycle    64908   5.5
   cpush    64907   5.5
   tdrop    49599   4.2
 aabuild    47825   4.1
retractn    46986   4.0
  ausage    46796   4.0
    skip     7248   0.6
  cusage      193   0.0
```

## 5. How to use Traces

In this section we briefly sketch when trace information might be useful, and which trace information should then be generated. We do not attempt to be exhaustive.

### *5.1 Performance*

Number of reductions per second can be measured using the UNIX `time` command together with the 'number of reductions' trace. The fact that *semi*-reduction steps are counted introduces an bias which is generally not significant.

Complexity of algorithms (in number of rewrite steps) can also be computed using the 'number of reductions' trace, possibly together with the 'trace from' option.

Improving the performance of programs should be done by identifying (sets of) functions which are responsible for the largest amounts of time used. The profiling option results in an overview of reduced functions, ordered by number of reductions.

### *5.2 Undefined functions*

Epic's current type-checker does not flag undefined functions. Using the 'trace functions' option (`-T`) produces its name. The UNIX `tail` command can be used to skip the unwanted parts of the trace information.

### *5.3 Inter-module type-checking and external functions*

Epic has no inter-modular type checker. Also, external functions aren't type-checked. If a functions implementation (in C or in Epic) disagrees with its use in an (other) module, values will wrongly be introduced on or removed from the argument stack, leading to run-time errors or other inexplicable behavior. Tracing functions with arguments, and possibly tracing ARM instructions (possibly with stackdump) may help to locate the problem. Using the 'race from' option helps to keep the amount of information down; however, it is usualy advisable to dump trace information on file, and to go through it using an editors search command.

### *5.4 Infinite loops*

An infinite loop is very often the cause of "out of memory" errors. Inspecting memory usage (`-M` option) may indicate the problem in more detail (infinite recursion leads to stack blocks being created; infinite term building to heap blocks being added). Inspecting garbage collection also discloses infinite term building because no nodes are released. Use the `-T` option to identify infinite loops: the same sequence of functions is repeated ad infinitum.

# Chapter 2
# A brief introduction to Epic

In this chapter we will discuss Epic, focussing especialy on less standard aspects. We will discuss Epic, static semantics modularity, the API of the Epic compiler, and lifting of input and output.

1. Epic

*1.1 Identifiers*

In Epic, two kinds of identifiers are distinguished: upper-case identifiers and general identifiers. Upper-case identifiers start with an upper-case letter followed by zero or more alfanumeric characters, underscores, dashes and single quotes (i.e., `[A-Z][-_'A-Za-z0-9]*`).

General identifiers fall in three categories:

- Ordinary identifiers start with a lower-case letter or an underscore followed by zero or more alfanumerics, dashes, underscores or single quotes (i.e., `[_a-z][-_'A-Za-z0-9]`).
- Character identifiers start with a single quote followed by a printable ASCII character (e.g., `'a`, `' ` or `'\`), or with a backslash followed by another backslash, one of the letters t, n, r, or three digits (e.g., `\\`, `\n`, `\000` or `\255`).
- Quoted identifiers start with a double quote, followed by an arbitrary string in which double quotes have been escaped with a backslash, followed by a double quote (e.g., `"\"quotes galore!\""`).

Finally, comments consist of a dollar sign follwed by arbitrary text up to the end of the line.

*1.2 Program structure*

An Epic program consists of a set of modules. Each module has a name, a `types` section and a `rules` section.

The types section consists of the word typesa followed by a sequence of types. A type is the name of the function (a general identifier), followed by a colon, followed by zero or more upper-case identifiers separated by the symbol #, followed by and arrow (->), followed by an upper-case identifier, followed by zero or more attributes enclosed in braces and separated by comma's, followed by a semi-colon. The attributes are discussed in Section 3, below. Epic is single-sorted, which implies that only the number of arguments of a function is significant; the names can be used for documentary purposes but this can be misleading.

Examples:

```
'c                  :           -> Char {free};
plus                : Nat # Nat -> Nat;
trace               : Str # X   -> X {external};
"plus:Nat#Nat->Nat": X # X     -> X;
```

The rules section consists of the word rules followed by a sequence of rules. A rule consists of a term followed by the symbol =, followed by a term followed by a semi-colon. See also Appendix II.

A term can be one of the following:

- an upper-case identifier, in which case it is a variable;
- a general identifier (signifying a function) possibly followed by one or more terms separated by comma's and enclosed in parentheses;
- a back-quote character followed by a sequence of characters in which the back-quote character has been escaped with \ followed by a back-quote character, possibly followed by the character + and an upper-case identifier. This construct is short-hand for string-terms. For example, ‘hi!\n‘ is short for the term str('h, str('i, str('!, str(\n, eos)))) and ‘ ‘+X0 is short for str(' , X0). See also Section 4.1.2;
- a vertical bar followed by a class name followed by a colon followed by a string in which the vertical bar has been escaped with \ followed by a vertical bar. This signifies an external value. See also Chapter 5.

## 2. Intra-modular static semantics

Epic places only minimal restrictions on modules, ensuring that the module can be meaningfully implemented. Type-checking is insufficient to properly support human users.

The following restrictions apply and are checked by the compiler:

- Every function that occurs in the rules section should appear in the types section.
- The arity of a function in its declaration in the types section (i.e., the number of upper-case identifiers between : and ->) and in all its uses in the rules section must coincide. That is, if its arity is zero, it should always appear without following parentheses, and if it is some $n > 0$ the number of terms appearing between parentheses should be $n$;
- No function should appear more than once in the types section.
- A function with attribute free should not be defined (i.e., appear as the outer-most function symbol on the left-hand side of a rule);

- A function with attribute `external` should not be defined (i.e., appear as the outer-most function symbol on the left-hand side of a rule);
- The left-hand side of a rule should not be a sole variable;
- Rules must be left-linear (no variable should occur more than once in the left-hand side of any rule);
- Each variable occuring in the right-hand side of a rule must also occur in the left-hand side of that rule.

## 3. Modularity

Epic supports modularity mainly in order to allow for separate compilation. Some features commonly associated with modularity are not supported. Most notably: Epic has no local (hidden) functions; all function names are global.

A function is *defined* in a module if it occurs as the outer-most function symbol in the left-hand side of a rule. A function should be defined in at most one module. This requirement isn't checked by Epic or ARM. If a function is accidentaly defined in more than one module, and those modules are compiled and loaded in ARM, then the last definition loaded takes precedence. This feature can be used to re-define built-in functions, but apart from that it can be a source of problems.

A module can use functions defined in other modules (or functions built-in in ARM). Such a function must be declared with the `external` or `free` attribute. If a function is declared `free` it means that it is defined in no module and isn't built-in in ARM. If it is declared `external` it means that it is defined in another module, or is built-in in ARM.

Declaring a function `free`, or declaring it without an attribute and including no rule that defines the function, results in an internal definition being created for the function. This is the 'definition' responsible for building a node when the function application is irreducible. Declaring a function `external` suppresses this internal definition. Hence, aan `external` function should be defined in another module or in ARM, or it must be `free` in at least one module or it must occur without annotation in a module which does not define the function.

There are a few static semantic rules that an Epic program must adhere to. At the time of writing no tool exists which checks these constraints.

- A function that is declared `external` must be built-in in ARM, must be defined in another module in the program, or must occur without annotation in a module which does not define the function. It can also be declared `free` in some module, but this is misleading.
- The arity of a function must be used consistent in all modules that define or use it.

## 4. Features

In this section we discuss the two features in which Epic distinguishes itself from term rewriting in general: specificity order and right-most inner-most reduction. Whereas term rewriting in general is non-detereministic, Epic has deterministic operational semantics. One reason for this is the fact that debugging non-deterministic program is an order more complicated that debugging deterministic programs. Another reason is the fact that Epic's deterministic operational semantics can be implemented very efficiently using ARM.

### *4.1 Specificity*

The specificity order is a partial order on rules modulo *alpha*-conversion (i.e., we regard all variables as the same). Consider two rules $s = t$ and $u = v$ in which all variables have been replaced by $x$, and consider the pre-order traversals of $s$ and $u$, say, $s_1, s_2, ..., s_m$ and $u_1, u_2, ..., u_n$, where each $s_i$ and $u_j$ is a function symbol or $x$. Let $k$ be the smallest index such that for all $i < k$, $s_i$ equals $u_i$. Three possibilities could exist:

- both $s_k$ and $u_k$ are function symbols, which are unequal by definition. In this case the two rules are mutually exclusive: if one is applicable, the other is not. We regard the rules as unordered;
- one of $s_k$ and $u_k$ is a variable and the other is a function symbol. In this case the rule where the $k$-th symbol is a variable is said to be more general, and where it is a function symbol is said to be more specific.
- $m = n = k - 1$. In this case the left-hand sides are identical modulo $\alpha$-conversion. Formally this is forbidden to ensure that specificity is a partial order. In practice this isn't checked, and the Epic compiler will pick one of the rules at compile time.

When two rules are applicable to the same redex, one of them is more specific than the other. Rewriting using the specificity order means that in this case the most specific rule will be applied. Note that if a more general rule would be favored over of a more specific rule, the more specific rule would never be applied.

Specificity combines an expressive power available in most languages (if...then...else, or a case- or switch-statement with a default case) with pattern matching.

In addition, specificity can be implemented very eficiently: the partialy ordered set of rules with the same function symbol as outer-most function symbol is compiled into a single pattern-matching automaton. Recognizing a redex as an instance of one member of this set requires a single pass through the automaton, in an amount of time practically independent of the size of the set.

Examples:

- a simple case discrimination.

```
vowel('a) = true;
vowel('e) = true;
vowel('i) = true;
vowel('o) = true;
vowel('u) = true;
vowel(X) = false;
```

- a non-trivial case in which the rules are ordered from most specific to most general. Note that rule 1 is more specific than rule 3 due to the fact that specificity is based on pre-order.

```
f(g(a), X) = ...;       $1
f(g(X), g(Y)) = ...;    $2
f(X, g(a)) = ...;       $3
f(X,Y) = ...;           $4
```

### *4.2 Innermost strategy*

A term can contain many redexes. For Epic to be deterministic means that one of the redexes must be singled out. Epic uses the right-most inner-most strategy: the right-most redex which does not contain a redex is reduced at all times. Note that for inner-most strategies it doesn't matter whether we go from left to right or vice versa: the same reductions occur, but in a different order. There is a difference when compared to outer-most strategies, which have better properties. As an example, consider the following:

```
if(true, X, Y) = X;
if(false, X, Y) = Y;
```

In an inner-most strategy the values of X and Y have already been reduced before one is selected. Using an outermost strategy, this overhead is avoided. In Epic the overhead could be avoided by using an auxiliary function, or the given construct shown in Appendix II.

Outer-most strategies can be implemented less efficiently than inner-most strategies because more datastructures need to be built. In ARM the only data-structures built are those of normal forms. Furthermore, after a reduction, the place where to look for the next redex is within the instantiated right-hand side (values of variables are already in normal form). These two facts contribute to Epic's (ARM's) efficient implementation.

## 5. Epic's API

Epic is defined formally in [WK96]. Epic is in fact defined as an abstract datatype for the representation of left-linear non-conditional term rewriting systems. The language Epic we discuss currently is – strictly speaking – only one concrete instance of a textual representation of term rewriting systems; the one for which a map from textual representation to this abstract data type representation has been provided (namely, the Epic parser which is part of the Epic compiler). This situation has two important consequences:

- When term rewriting systems in another concrete language than Epic, or in some abstract (internal) representation, must be processed, they can be implemented without actually translating that language or that internal representation to concrete Epic. The following approaches are preferable:
  - To process another language than Epic, a parser for that language should be made which produces a data structure that can be handled by the Epic compiler. This approach avoids producing and analizing concrete Epic.
  - To process an internal representation, an abstract data type interface needs to be defined to allow the Epic compiler to handle the internal representation directly.

  To use these approaches, a user's manual of the API of the Epic compiler should be available. Currently that document does not exist, although the information can be obtained from [WK96] and Appendix A, pp. 127—162 of [Kam96].
- Inconveniences and awkwardnesses are often contained in the concrete Epic language rather than in Epic's structure. The most notable example is the misleading notation of sorts in function types (a remnant of earlier plans): it is trivial to alter the parser so

that only the number of arguments could be indicated and not their type. Alternatively a choice for a type system could be made (e.g., many sorted or order-sorted), and a type checker could be produced which checks for compliance according to that system.

## 6. Input and output

An important notion, especially when creating stand-alone applications, is that of *lifted I/O*. We will first discuss I/O.

Running an imperative program involves not only the program but also the data it operates on (files, including keyboard and screen). Functional, equational and logic programs (e.g., Epic programs) do not have side-effects[1], which means no files are modified during execution.

Instead, such programs describe how a term can be reduced to its value (which is also a term). The contents of files to be inspected could be contained in the initial term, and the contents of modified files could be part of the final term. In this sense it is reasonable to refer to the initial term to be reduced as 'the input' and to the normal form being produced as 'the output'.

### *6.1 Lifting*

As mentioned, Epic is primarily intended to be generated. For this reason Epic supports only prefix-bracketed notation of terms (that is, `plus(X,Y)` rather than `X + Y`). Clearly more freedom in notation would be preferable for humans; see [vDHK96] for an excellent presentation on this issue.

However, Epic and ARM are intended to create tools in many different circumstances, most of which use data in other representations. Even though the Epic programs are written using prefix-bracketed notation, the resulting tools must be able to process arbitrary texts.

Epic and ARM support this by offering you a choice: by default, prefix-bracketed notation is used (we have seen this in the introduction of this guide), but other texts can be handled as well.

In that case, the text offered as input is first lifted to a generic prefix-bracketed form consisting of the constant `eos`, representing the empty string (i.e., end-of-string); the two-place function `str`, which has a character and a string-term as arguments; and any ASCII character such as `'a`, `'$`, `' '`, \n, \000, etcetera. For example, the line of text "`Hello!`" followed by a new-line character, is lifted to the term `str('h, str('e, str('l, str('l, str('o, str('!, str(`\n`, eos)))))))` (see Section 4.2.1.2 for more details).

We refer to this phenomenon as *lifting* the input, and, similarly, lowering the output. For output one additional constructor is suitable: the two-place function symbol `cat` which has two strings as arguments. Using `cat` allows us to produce output in chunks (for example, lines of text) without having to transform the entire output into one huge string. We will see examples of this in Section 1.3.3.2.

When lifted input is used, the program must define a function that analyses the text and transforms it into a term suitable to the application. For example, the string "`1 + 1`" may

[1]Actually most functional languages – including Epic – do provide for side-effects, which is often regarded as an impure but pragmatic aspect. In Appendix 5 we will see how Epic offers the ability to define functions with side-effects, which can be used to read/write from/to files and I/O devices.

Recently, formal models have been developed which explain what I/O means in the context of side-effect free languages [AvGP92, P-JW93, Mog89, WK95]. No such model is implemented in Epic/ARM, and we will not go into it in this report.

initially be lifted to `str('1, str(' , str('+, str(' , str('1,eos)))))`, but may have to be transformed into `plus(one,one)` before it can be processed further. A function that analyzes text and translates it to an abstract syntax is called a *parser* or *scanner*.

Similarly, the output must be converted to a textual representation. A function that does this is called a *pretty-printer* (whether its product is pretty or not).

Epic assumes the existence of a function which parses the input, performs the action implemented by the tool, and pretty-prints the output (in this report we call this function the parse-reduce-pretty-print function). If the application is called `foo`, then the default name of this function is `ppp-foo`.

The default name is part of the following convention (which is not required by Epic). A tool called `foo` consists of a parser, a pretty-printer and functionality that implements the tools' proper action (i.e., *foo*). The parser is a one-place function which accepts a string as argument and produces a term in the abstract syntax of *foo*, and which is named `p-foo`. The pretty-printer is a one-place function that accepts a term in the abstract syntax of *foo* and produces a string, and which is called `pp-foo`. *Foo* could be implemented by applying some function (say, `foo`) to the result of the parse. In this case the tool is implemented by a function `ppp-foo`, which is defined as follows:

```
ppp-foo(X) = pp-foo(foo(p-foo(X)));
```

It is also possible that *foo* is implemented less operational, by reductions on its abstract syntax directly, in which case `ppp-foo` is defined as

```
ppp-foo(X) = pp-foo(p-foo(X));
```

In any case, when lifted I/O is used, Epic assumes the existence of a one-place function (which can have any name, but is by default assumed to be `ppp-foo`).

# Chapter 3
# ARM's Machine Model

In this chapter we sketch ARM's machine model so that the API described in the next chapter can be properly understood. In [FKW97] we describe the machine model in more detail, and also discuss the compilation process from Epic to ARM.

ARM uses three stacks and a heap. The heap is a repository where dags (directed, acyclic graphs) are kept which represent terms. In the remainder of this section we abstract from the heap: when we say that a term is put on the stack we actually mean that that term is represented in the heap as a dag, and that a pointer to that dag is put on the stack.

The three stacks are:

- the control stack (CS), which contains function symbols that have not yet been reduced;
- the argument stack (AS), which contains terms that have been normalized, but have not yet been matched;
- the traversal stack (TS), which contains the values of variables stored during matching.

ARM uses the right-most inner-most reduction strategy: the right-most redex which does not contain another redex is reduced at all times. From this it follows that when a function is reduced, its arguments have already been normalized.

In addition, ARM uses an automaton for redex recognition; the Epic compiler translates an Epic program to a so-called *minimal term rewriting system* (MTRS) in which the automaton is encoded using auxiliary functions (we have seen some of these pass by in Section 1.4).

As an example, we will consider the application of the rule `f(g(X),h(Y))=r(X,g(Y))` to the term `f(g(`$s$`),h(`$t$`))`, where $s$ and $t$ are two terms, and `g(`$s$`)` and `h(`$t$`)` are normal forms. This rule is translated by the Epic compiler to an MTRS which includes rules similar to those shown here (the "#" signs are part of the identifiers). Note that we will not go into this translation process (see [FKW97] for that).

```
f(g(X),Y) = f#g(X,Y);                $ 1
```

```
f#g(X,h(Y)) = f#g#h(X,Y);           $ 2
f#g#h(X,Y) = f#g#h#g(X,g(Y));       $ 3
f#g#h#g(X,Y) = r(X,Y);              $ 4
```

In our example the top of CS is `f`, and AS contains `g(`$s$`)` as top, and `h(`$t$`)` as second element.

- in the first cycle rule 1 is checked and (since it is applicable) is applied. Sub-term $s$ is moved from within the top of AS to TS, and symbol `f` is replaced by **`f#g`**.
- in the second cycle rule 2 is checked. Note that it is only interested in the second argument of **`f#g#h`**, which (and this is no coincidence) is conveniently located at the top of AS. The symbol `h` is matched, and the sub-term $t$ is moved from within the top of AS to TS. Symbol **`f#g`** is replaced by **`f#g#h`**.
- in the third cycle the top of TS is moved to AS, symbol **`f#g#h`** is replaced by **`f#g#h#g`** on CS, and symbol `g` is pushed on CS.
- in the fourth and subsequent cycles `g(`$t$`)` is normalized. When it is, it is left on AS and control passes automatically to the function **`f#g#h#g`**.
- in the final cycle of our example rule 4 is applied. Term $s$ (still on TS) is moved to AS, and `r` replaces **`f#g#h#g`** on CS.
- in subsequent cycles `r(`$s$`,g(`$t$`))` is normalized, leaving its result on AS.

In order to understand ARM's API, the following invariants are relevant:

- unevaluated function symbols go on CS;
- TS and AS hold normalized terms;
- when a function from an Epic program is evaluated, its arguments are on AS, the first argument on top (this is untrue for the auxiliary functions produced by the Epic compiler to encode the automaton);
- when matching for an Epic function has succeeded the values of arguments are stored (in pre-order) on the traversal stack;
- after a term has been normalized, its value is left on AS.

# Chapter 4
# ARM's API: using ARM as a plug-in library

In this chapter we discuss how ARM (via Epic) can be used as a plug-in library. This is useful for applications such as: theorem provers [GL91, Fra94, GG91, KZ89, Bou94], verification tools, algebraic specifications [EM85, BHK89, Vis96, HM93], compiler generators [ESL89], language prototyping [vDHK96], program analyzers [BDHF96] and functional programming languages [PvE93].

The stand-alone interpreter discussed in the Chapter 1 is in fact a fairly small program which uses ARM's API in one particular way. As a concrete example of how to use the API, this program is included in Appendix III.

## 1. Preliminaries

### *1.1 Types*

The API defines the following type-names:

`ARM_ref`: a pointer to the structure used to represent terms.

`ARM_Xclass`: a pointer to a structure representing a class, which must be used in the API to handle external data types such as floats.

### *1.2 Characters and strings*

ARM assumes the following convention for the representation of strings as terms.

The characters are represented as constants with the names `'a`, `'b`, etc. for all printable ASCII characters and the space character; \n, \t and \r for the remaining whitespace; and \ddd, where `ddd` is a three digit decimal number, for all remaining ASCII characters.

Strings are constructed of the constant `eos`, representing the empty string; the free function symbol `str`, which takes a character and a string as argument, and has type string; and the free function symbol `cat`, which takes two strings as argument, and has type string.

These conventions are used when lifting input and lowering output. With lifted I/O, the input text is represented as a string of characters as described above, using `eos` and `str`, and output is produced from the sequence of characters encountered in pre-order in the normal form, which must be a string as described above, containing `eos`, `str` and `cat`. The free constructor `cat` allows the production of a textual normal form in string segments without the need to concatenate all segments in one flat string.

Hereafter we will refer to strings as described above as string-terms in order to avoid confusion with C strings.

### *1.3 Loading terms*

The simplest way to load a term for normalization is to push its function symbols (in pre-order) onto the control stack. This is done in the READY state by a sequence of `push`-es.

There are several circumstances when parts of a term have already been normalized. To name a few:

- ARM is used by a tool which performs sequences with more or less related terms. Each subsequent term may contain (parts of) previous terms;
- Sharing is necessary or double work is to be avoided. Pushing the function symbols of, say, $f(s,s)$ would lead to $s$ being normalized twice, and to a dag in which two distinct copies of $s$'s normal form occur.

The result of a normalization can be protected from the garbage collector if it is to be used later on. (see functions `ARM_protect` and `ARM_unprotect`).

Terms that have been protected can be re-used in subsequent computation. This is done by pushing those terms on the argument stack (see function ARM_apush). Since the control stack holds unreduced function symbols and the argument stack holds normal forms, only certain terms can be reduced in one go: terms in which the pre-order traversal contains a contiguous sequence of uninterpreted function symbols followed by a contiguous sequence of normal forms. For example, if $s$ is a protected reference to an established normal form, then $f(s,s)$ can be reduced by pushing $f$ on the control stack, and pushing $s$ twice on the argument stack (see functions ARM_push and ARM_apush). However, $f(s,g(s))$ can not be handled in one go. Instead, $g(s)$ must be normalized, resulting in its normal form $u$, and then $f(s,u)$ can be normalized. Note that protecting $s$ is necessary while the normal form of $g(s)$ is computed because it is used again in $f(s,u)$.

# 2. API

### *2.1 Behavior*

At all times, ARM is in one of five states:

**INITIAL:** memory has not yet been allocated;

**CLEAN:** memory is allocated; no ARM programs have yet been loaded;

**SPECIFIC:** one or more ARM programs have been loaded;

**LINKED:** the ARM programs have been linked and are ready for execution;

**READY:** ARM has accepted (part of) a term and is ready to reduce it.

Note that ARM has no way of knowing when an entire term has been loaded. It is the responsibility of the application to make sure that a well formed term is loaded.

Actions can be performed in accordance with the following process expression. Here capital identifier signify states, and lower-case identifiers signify action performed by calling the related function in the API, or errors or warning that have been generated by ARM. The symbol "." signifies a transition to a new state and the symbol "+" signifies choice.

```
INITIAL   =  setup . CLEAN +
             ( warning . INITIAL +
               fatal_error )
CLEAN     =  load_arm . SPECIFIC +
             clear . CLEAN +
             ( warning . CLEAN +
               load_error . CLEAN +
               fatal_error )
SPECIFIC  =  clear . CLEAN +
             load_arm . SPECIFIC +
             link . LINKED +
             ( warning . SPECIFIC +
               load_error . CLEAN +
               fatal_error )
LINKED    =  load_arm . SPECIFIC +
             clear . CLEAN +
             ready . READY +
             (protect + unprotect) . LINKED +
             ( warning . LINKED +
               load_error . CLEAN +
               fatal_error )
READY     =  (push + apush + pushopq) . READY +
             reduce . LINKED +
             ( warning . READY +
               run_time_error . LINKED +
               load_error . CLEAN +
               fatal_error )
```

Note that `fatal_error`'s terminate ARM's execution.

## 3. API functions

The application programming interface of ARM offers the functions listed below. In the following sections we will discuss each function in detail.

| | |
|---|---|
| `ARM_set_up:` | set up memory structures for stacks, heap and (ARM) code. |
| `ARM_load_arm_file:` | load a single ARM file. |
| `ARM_link:` | link all loaded ARM code. |
| `ARM_clear:` | forget all previously loaded ARM code. |
| `ARM_ready:` | prepare for accepting an input term. |
| `ARM_reduce:` | reduce the pushed term to normal form. |
| `ARM_protect:` | protect a normal form from garbage collection during subsequent reduces. |
| `ARM_unprotect:` | allow the space occupied by a term to be recycled. |
| `ARM_lift_input:` | lift the entire input to a string-term representation. |

| | |
|---|---|
| `ARM_push:` | push one function symbol on the C stack. |
| `ARM_apush:` | push one normal form on the argument stack. |
| `ARM_pushopq:` | push an external value. |
| `ARM_class_of:` | given the name of an external type, produce the corresponding class. |
| `ARM_display:` | print the textual representation of a normal form (when indicated, lower the term, which must then be a string-term). |
| `ARM_ofs:` | given a normal form, yield its outermost function symbol. |
| `ARM_size:` | given a normal form, produce the number of immediate sub-terms (i.e., the arity of the ofs). |
| `ARM_child:` | given a normal form and an index i, produce the i-th immediate sub-term of the term. |
| `ARM_set_error_functions:` | Set the four error reporting functions. The functions take an error message and produce void. From left to right the functions handle warning; run-time error (to LINKED); arm-error (to CLEAN) and fatal error (must exit). |
| `ARM_DEBUG_STATUS:` | this variable controls whether tracing / debugging / profiling information is generated. |
| `ARM_set_debug_functions:` | Set the thirteen tracing/monitoring functions. The functions have different argument types and produce void. From left to right the functions control how tracing/monitoring information is collected or presented for: ARM mnemonics (-M option); Epic functions (-T option); each semi-rewrite step; *obsolete*; stacks display (-D option); arguments display (-A option); the moment the from-function (-F option) is found; start of a garbage collection; end of a garbage collection; allocation of a memory block; display of the name of a memory block; display of the number of reductions used; and display of profiling information. Default actions of these functions have been shown in Section 1.4. |
| `ARM_trace_gc:` | trace garbage collection activity. |
| `ARM_memstat:` | produce information of memory usage as blocks are allocated. |
| `ARM_count_rewr:` | count the number of (semi) reductions. Abort if the indicated maximum number is exceeded. |
| `ARM_profile:` | collect profiling information. |
| `ARM_dump_degree:` | set degree of stack dumping: 0 for none; 1 for (stackdepth) arguments (ARM doesn't know arities of function symbols, so a fixed number is displayed for every function); 2 for a full dump of all stacks (stackdepth deep). |
| `ARM_trace_degree:` | set degree of tracing: 0 for none; 1 for function symbols only; 2 for individual ARM instructions. |
| `ARM_stack_dump_depth:` | set maximal displayed depth of stacks during stackdump. |
| `ARM_term_dump_depth:` | set maximal displayed depth of terms during stackdump. |
| `ARM_trace_from` | start tracing from the first (semi) reduction of this symbol. |

We will now describe these functions in detail:

*3.1* `ARM_set_up` arguments: *none* returns: *void*
Initialize all data structures.

*3.2* `ARM_load_arm_file` arguments: `char *name` returns: *void*
Load the ARM file with the given name.

*3.3* `ARM_link` arguments: *none* returns: *void*
Link all loaded ARM code.

*3.4* `ARM_clear` arguments: *none* returns: *void*
Forget all previously loaded ARM code. Note that currently no garbage collection is available for code storage. Hence the occupied memory is not released.

*3.5* `ARM_ready` arguments: *none* returns: *void*
Prepare for accepting a subject term.

*3.6* `ARM_reduce` arguments: *none* returns: *void*
Reduce the term just (a)`push`-ed on the stacks using the loaded ARM programs. Leave the result on the Argument stack.

*3.7* `ARM_protect` arguments: `ARM_ref` returns: *void*
Protect term `t` from the garbage collector. This term must be a (sub-)term from a normal form that has just now been produced (or was protected earlier).

*3.8* `ARM_unprotect` arguments: `ARM_ref` returns: *void*
Remove the protection of a term `t` that was protected earlier.

*3.9* `ARM_lift_input` arguments: *none* returns: `ARM_ref`
Lift the entire input stream to a string-term.

*3.10* `ARM_push` arguments: `char *` returns: *void*
Push the indicated function symbol on the Control Stack.

*3.11* `ARM_apush` arguments: `ARM_ref` returns: *void*
Push the given term on the Argument Stack.

*3.12* `ARM_pushopq` arguments: `ARM_Xclass, char *` returns: *void*
Push an external value on the Control Stack. Note that an external value is somewhat comparable to a constant, and can accordingly be pushed on the Control Stack.

*3.13* `ARM_class_of` arguments: *char ** returns: `ARM_Xclass`
Return a pointer to the class structure with the given name.

*3.14* `ARM_display` arguments: `int, ARM_ref, int` returns: *void*
Print the textual representation of a term. If the first argument is zero, the term is printed in normal prefix-bracketed notation. Otherwise, the term must be a string-term and it is lowered. Sub-terms of the term that are not string-terms are printed in prefix-bracketed notation between square brackets.

The third argument indicates to which stream printing should occur. If it is zero, the standard output stream is used; otherwise the standard error stream is used.

*3.15* `ARM_ofs` arguments: `ARM_ref` returns: *char **
Given a term, yield the name of the outermost function symbol.

*3.16* `ARM_size` arguments: `ARM_ref` returns: *unsigned long*
Given a term, yield the arity (i.e., the number of immediate sub-terms).

*3.17* `ARM_child` arguments: `ARM_ref, unsigned long` returns: *ARM_ref*
Return the sub-term with the given index. The first sub-term has index 1.

*3.18* `ARM_set_error_functions` arguments: *see below* returns: *void*
This function is used to set the four error/warning functions. These functions all have type `void(*err1)(char *msg,...)`. They get a printf-compatible format followed by zero or more values as argument and return void. The functions are used for the following types of exceptions:

- Warning. I.e. all situations which are unexpected but not fatal (as far as ARM is concerned). Example: applying `ARM_child` with an index which is too large.
- Run-time error. I.e. all situations during rewriting which prohibit further reductions but which leave ARM in an otherwise repairable state. Example: out of memory.
- Load error. Recoverable errors encountered during loading. Example: ARM file not found.
- Fatal errors. All other caught errors. This fourth error-handler is not supposed to return.

*3.19* `ARM_DEBUG_STATUS` (this is a variable) type: *int*
This variable controls whether ARM produces any debugging/tracing/profiling information. If the variable is zero, no such information is produced.

*3.20* `ARM_set_debug_functions` arguments: *see below* returns: *void*
This function is used to set the functions that produce tracing and monitoring information. These function all produce void, and have arguments depending on their functionality. The functions are called when `ARM_DEBUG_STATUS` and when the individual flag that controls their behavior is set (see subsequent sections).

- `ARM_trace_show_mnemonic`. Arguments: `char *`. Called each ARM cycle. The argument is the name of the ARM instruction, which is printed by default.

- `ARM_trace_show_fun`. Arguments: `char *`. Called each (semi) rewrite step. The argument is the name of the function being reduced, which is by default printed.
- `ARM_trace_semi_step`. Arguments: *none*. Called each (semi) rewrite step. By default the counter is incremented.
- `ARM_trace_semi_step_fun`. Arguments: `ARM_fun`. *obsolete*
- `ARM_trace_display_stacks`. Arguments: `char *`. Called each ARM cycle in order to provide a stackdump (by default). The argument is the ARM instruction involved.
- `ARM_trace_display_args`. Arguments: `char *`. Called each ARM cycle in order to provide a dump of the arguments (by default). The argument is the ARM instruction involved.
- `ARM_trace_from_fun_found`. Arguments: *none*. Called when the from-fun is found.
- `ARM_trace_start_gc`. Arguments: *none*. Called when the garbage collector starts a collection. By default, "`[gc:  `" is printed.
- `ARM_trace_stop_gc`. Arguments: `long, long`. Called when the garbage collector finishes its collection. The arguments are the number of nodes currently in use and the number of freed nodes. By default they are printed.
- `ARM_trace_alloc`. Arguments: `long, long`. Called when a global block of memory is allocated. The arguments are the size of this block and the total size allocated so far.
- `ARM_trace_blockname`. Arguments: `char *`. Called when a global block of memory is allocated. The name is the type of block allocated (see Section 1.4 for details).
- `ARM_trace_show_reds`. Arguments: `long`. Called after normalization. The argument is the total number of semi rewrite steps used, which is printed by default.
- `ARM_trace_show_profiles`. Arguments: *none*. Called after normalization in order to display tracing information. Note that the raw data from which statistics are created are encoded in ARM's data structures.

*3.21* `ARM_trace_gc` arguments: *int* returns: *void*
Sets GC monitoring. Argument non-zero means "monitor GC activity". At the beginning and end of each garbage collection cycle the functions `ARM_trace_start_gc` and `ARM_trace_stop_gc` are called. By default they print a message, the number of nodes currently in use, and the number of nodes just collected.

*3.22* `ARM_memstat` arguments: *int* returns: *void*
Sets memory usage monitoring. A non-zero argument means "monitor memory usage". For every global block of memory that is allocated the two functions `ARM_trace_alloc` and `ARM_trace_blockname` are called.

*3.23* `ARM_count_rewr` arguments: *unsigned long, int* returns: *void*
Sets rewrite step monitoring and limitation. Second argument non-zero means "monitor memory usage".The first argument is the maximum number of rewrite steps allowed. Exceeding this number leads to a run-time error.

*3.24* `ARM_profile` arguments: *int* returns: *void*
Sets profiling monitoring. Argument non-zero means "collect and display profiling information". Note that collecting profiling information, due to limitations in the current implementation, takes a significant amount of time.

*3.25* `ARM_dump_degree` arguments: *int* returns: *void*
Set the degree of dump information that needs to be displayed. Value 0 means: no information; value 1 means arguments only (-A option); and value 2 means stackdump (-D option).

*3.26* `ARM_trace_degree` arguments: *int* returns: *void*
Set the degree of trace information that needs to be displayed. Value 0 means: no information; value 1 means Epic functions (-T option); and value 2 means ARM mnemonics (-M option).

*3.27* `ARM_stack_dump_depth` arguments: *unsigned long* returns: *void*
Set the depth to which stacks are dumped (`ARM_dump_degree` 1 or 2; options -A or -D).

*3.28* `ARM_term_dump_depth` arguments: *unsigned long* returns: *void*
Set the depth to which terms are dumped (`ARM_dump_degree` 1 or 2; options -A or -D). Any sub-term deeper than this level is shown as `...`, such as `f(a,g(...))`.

*3.29* `ARM_trace_from` arguments: *char ** returns: *void*
Start tracing/dumping after the first reduction of the given function. Note that the name should not be longer than 199 characters.

# Chapter 5
External values and functions

Adding externals to Epic and ARM is reasonably straightforward. However, the source code of the interpreter must be modified. The ARM distribution contains floats as an example of how to use externals.

## 1. Introduction and overview

Externals are used to represent data for which it would be inefficient or impractical to use ordinary term representation. For example, though floating point arithmetic can be expressed in Epic, the computations would probably be very inefficient compared to the calculations offered on most computer hardware. As a second example, the representation of files in Epic is ineffective, and representing them as external values is more appropriate.

In Epic, externals are trivially supported. Anything of the form `|smiley:(:-)|`, where `smiley` can be any name, and where `(:-)` can be an arbitrary string in which the character | has been escaped with \, is a constant and is passed on to ARM. Any function that is declared as {`external`} but isn't defined in any module must be defined in ARM. I.e., the external functions defined as described in this section can be used in Epic when they are declared {`external`}.

For example, consider the following Epic module.

```
module Smiley
types
  combine : Smiley \# Smiley -> Smiley {external};
  new     : -> Smiley;
rules
  new = combine(|smiley:(:-)|,|smiley:[;=]|);
```

In ARM, external values are represented as indirectly referenced entities; just as a term is represented in the heap as a dag, but is handled by way of a pointer into the heap, so is an external value represented by some data in a chunk of memory and is handled by way of a pointer to that memory.

ARM uses a notion of "classes" to handle external values. The class defines a class-variable (holding the name of the external type), and three methods. The first method (the reader) is used to interpret a string representation of the external value; the second (the printer) to print an external value, and the third (comparison) to determine equality of external values. When ARM code is read, a construct like `|foo:bar|` is unpacked. The name part (`foo`) is used to identify a class. Then that class's reader is applied to the second part, to produce the internal representation of the external value. Conversely, when an external value is produced on output, its printer is applied.

In the next sections we will discuss all aspects that must be covered in order to use externals in some detail. We will develop an external type `foo` which in C is represented as a type `fooType` (i.e., `fooType` might be an array, structure or other type).

## 2. Preliminaries

### 2.1 *Global memory management*

In the file `threads.h` a structure `thread` is defined which holds pointers to all memory in use (threads were at one point in time intended to facilitate multi-processor ARM; thence the name). In this structure a field of type `opq` (for opaque) must be present for each external type. For example, we could add the following line.

```
    opq fooSpace;
```

Secondly, space must be allocated. This is done during initialization, in the initializer for an external type. It is done by including the following line:

```
    Thread->fooSpace = (opq)new_array(20000,sizeof(Rfoo),"foos");
```

In this example, space is reserved for 20000 foo's. Note that ARM will extend this space as needed. However, no garbage collection occurs for external values. The string `"foos"` is used in memory usage monitoring, and is best laid-out if it is four characters long.

### 2.2 *The opaque structure*

A structure must be defined to hold external values. It contains a pointer to its class, and the actual value, and should look like:

```
typedef struct _foo Rfoo, *Xfoo;
struct _foo {
    ARM_Xclass class;
    fooType val;
};
```

### 2.3 *Methods*

Three methods should be defined for each external type: a reader, a writer and a comparison. The reader function is as follows:

```
opq str2foo(char *s)
{   fooType f;
    Xfoo o;

    /* interpret the string s and put the established value in f */
    ...

    o = (Xfoo)local_arr_entry((Xarray)Thread->fooSpace,fooClass);
    o->val = f;
    return (opq)o;
}
```

The call to `local_array_entry` allocates one `_foo` structure in the appropriate space.
The writer function is as follows:

```
void showfoo(opq q)
{   out2(stderr,"|float:");
    /* produce the textual representation of the foo value in ((Xfloat)q)->val */
    ...
    out2(stderr,"|");
}
```

The macro `out2` is defined in the file `basics.h`.
The equality function is as follows:

```
int eqfoo(opq q1,opq q2)
{   if ( /* (((Xfloat)q1)->val) and (((Xfloat)q2)->val) are equal */ )
        return 1;
    else
        return 0;
}
```

## *2.4 Class*

The class is a structure built by ARM to represent class-variable and specific functions for an external type. Classes are handled by pointers to that structure, which are of type `ARM_Xclass`. By convention, the class of a type `foo` is referred to by `ARM_Xclass fooClass`.
The class is created during initialization, in the initializer for an external type. It is created as follows:

```
    fooClass = new_class("foo",&str2fo,&showfoo,&eqfoo);
```

## *2.5 Initializer*

The initializer is called when ARM is initialized. In it, a class is created, space is allocated, and functions are made known for tracing and linking purposes. The initializer yields the external class.
The initializer must be called from the function `init_externals` as follows:

```
    new_external(foo);
```

This does some bookkeeping, and also calls the function `init_foos` shown below (note that the macro `new_external` appends a letter “s” after the type name `foo` to get `init_foos`)).
The initializer looks like

```
ARM_Xclass init_floos()
{   fooClass = new_class("foo",&str2foo,&showfoo,&eqfoo);

    Thread->fooSpace = (opq)new_array(20000,sizeof(Rfoo),"foos");

    /* definition of foo-functions */
    xfundef("addF",f_addF);

    /* declaration of use of other functions */
    xfunuse("true",v_true);

    return fooClass;
}
```

The definition and use of functions is discussed in Section 5.3, below.

### *2.6 Emergency handler*

The emergency handler is called when ARM has encountered a recoverable error. Since ARM doesn't know where the error originated, all data structures must be re-initialized (only the class is re-used). Previously allocated space is returned to the system, and new space is allocated. Otherwise it is similar to the initializer.

The emergency handler must be called from the function `emergency_externals` in the file `externals.c` as follows:

```
    emergency_foos();
```

In order to call it, the function must be declared immediately before `emergency_externals`.

```
void emergency_foos();
```

The emergency handler looks like

```
ARM_Xclass init_floats()
{   kill_array((Xarray)Thread->fooSpace);

    Thread->fooSpace = (opq)new_array(20000,sizeof(Rfoo),"foos");

    /* definition of foo-functions */
    xfundef("barQ",f_barQ);

    /* declaration of use of other functions */
    xfunuse("true",v_true);
}
```

The definition and use of functions is discussed in Section 5.3, below.

## 3. Functions

External values are manipulated from Epic via functions declared as `external`. These functions are written in C, and there existence must be made known in ARM for linking purposes. In addition, when (external or Epic) functions are used, their use must also be made known to ARM, so that linking can be done properly.

## *3.1 Preliminaries*

For every function defined elsewhere (as an external function or in Epic) that is used in this external package, a variable must be declared for linking purposes, and the use of that function must be declared.

The variable is declared by

```
    static ARM_fun v_true;
```

Its use is made known in the initializer, as follows.

```
    xfunuse("true",v_true);
```

This tells ARM that the Epic function **`true`** will be used and that in our external code we will refer to it by the variable **`v_true`** (examples follow below).

Every function that is defined must be declared as such in the initializer, as follows

```
    xfundef("barB",f_barB);
```

## *3.2 External functions*

By convention, external functions are called something like **`f_barB`**. Here, **`f_`** helps to avoid name-clashes with other functions in your application. In addition, it identifies the name as that of a function. With the function a control variable may be associated, in which case it is called **`v_barB`**. The letter `B` is appended to avoid name-clashes with similar functions for other external packages, such as **`f_addF`** for addition on floats, **`f_addI`** for addition on integers. In our example we use `B` for foo-functions.

Here is the general lay-out of external functions.

```
preamble(f_barB)
external f_barB()
{   constituent

    entryQ(f_barB,"f_barB");

    /* the actual body */

    dispatchQ(IP);
    xferQ;
}
```

In the body of the function, the arguments are available on the argument stack. They can be retrieved with the macro's **`APtop`**, **`APscnd`**, and **`APnth(n)`**, which retrieve the top, second, or n-th (for **`n`** $>= 0$) element. In addition **`APpop`** pops one element (and yields its value, and **`APdrop(n)`** drops **`n`** (for **`n`** $>= 0$) elements.

It is good practice to check if the arguments of an external function have the right type. This is done, for example, as follows:

```
Qtcopq(APtop,fooClass,"barB")
```

This statement checks that the current top of the argument stack belongs to the class `foo`. The string is the name of the function it is called from, for error reporting purposes.

The actual value of type fooType is retrieved by:

```
((Xfoo)APpop)->val
```

Suppose two foo-values can be bar-red by the function `bar`. Then the external version might look like:

```
preamble(f_barB)
external f_barB()
{   constituent

    Qtcopq(APtop,fooClass,"barB")
    Qtcopq(APscnd,fooClass,"barB")

    Ref1 = (ARM_ref)local_arr_entry((Xarray)Thread->fooSpace,fooClass); /*1*/
    ((Xfoo)Ref1)->val = bar(((Xfoo)APpop)->val, (((Xfoo)APpop)->val));  /*2*/
    APpsh = Ref1;                                                       /*3*/

    dispatchQ(IP);
    xferQ;
}
```

In the line marked `/*1*/` a new space is created for the new `foo` value. In the line marked `/*2*/` its value is set to the bar of the the top and the second value on the argument stack. Note that `APpop` implies their removal. In the line marked `/*3*/` the new value is pushed on the argument stack.

# Appendix I
References

REFERENCES

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine. A Tutorial Reconstruction.* The MIT Press, 1991.

[Aug85] Lennart Augustsson. Compiling pattern matching. In J.P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer-Verlag, 1985.

[AvGP92] P.M. Achten, J.H.G. van Groningen, and M.J. Plasmeijer. High-level specification of i/o in functional languages. In John Launchbury, editor, *Proceedings Glasgow Workshop on Functional Programming.* Springer-Verlag, 1992.

[BDHF96] J.A. Bergstra, T.B. Dinesh, J. Heering, and J. Field. A complete transformational toolkit for compilers. In Hanne Riis Nielson, editor, *Programming Languages and Systems – ESOP'96*, number 1058 in Lecture Notes in Computer Science, pages 92–107. Springer-Verlag, 1996.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[Bou94] Adel Bouhoula. Sufficient completeness and parameterized proofs by induction. In *Proceedings of the International Conference on Programming Language Implementation and Logic Programming, PLILP '94*, 1994.

[CCM85] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 50–64. Springer-Verlag, 1985.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specifications, Vol. I, Equations and Initial Semantics.* Springer-Verlag, 1985.

[ESL89] H. Emmelmann, F-W. Schröer, and R. Landwehr. BEG - a Generator for Efficient Back Ends. In *Proceedings of the Sigplan '89 Conference on Programming Language Design and Implementation.* In SIGPLAN Notices, Vol. 24, Number 7.

[FKW97] W.J. Fokkink and J.F.Th. Kamperman and H.R. Walters. Within ARM's Reach: Compilation of Left-Linear Rewrite Systems via Minimal Rewrite Systems. Technical Report SEN-R97xx, CWI, 1997. Submitted for publication elsewhere.

[Fra94] Ulrich Fraus. Inductive theorem proving for algebraic specifications - TIP system user's manual -. Technical report, Passau, 1994.

[FW87] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 34–45. Springer-Verlag, 1987.

[Gar90] Hubert Garavel. Compilation of lotos abstract data types. In S.T. Vuong, editor, *Formal Description Techniques, II*, pages 147–162. Elsevier Science Publishers B.V. (North-Holland), 1990. IFIP, 1990.

[GG91] S.J. Garland and J.V. Guttag. *A Guide to LP, The Larch Prover.* MIT, November 1991.

[GHM88] A. Geser, H. Hussmann, and A. Mück. A compiler for a class of conditional term rewriting systems. In S. Kaplan and J.-P. Jouannaud, editors, *Proceedings of the First International Workshop on Conditional Term Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 84–90. Springer-Verlag, 1988.

[GL91] K.B. Gallagher and J.R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[HF$^+$96] Pieter H. Hartel, Marc Feeley, et al. . Benchmarking implementations of functional languages with "pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4), 1996.

[HM93] B.M. Hearn and K. Meinke. ATLAS: A type language for algebraic specification. In Jan Heering, Karl Meinke, Bernhard Möller, and Tobias Nipkow, editors, *Higher-Order Algebra, Logic, and Term Rewriting*, number 816 in Lecture Notes in Computer Science, pages 146–168. Springer-Verlag, 1993.

[Kam96] J.F.Th. Kamperman. *Compilation of Term Rewriting Systems.* PhD thesis, Centrum voor Wiskunde en Informatica, 1996.

[Kap87] S. Kaplan. A compiler for conditional term rewriting systems. In P. Lescanne, editor, *Proceedings of the First International Conference on Rewriting Techniques*, volume 256 of *Lecture Notes in Computer Science*, pages 25–41. Springer-Verlag, 1987.

[Ken90] R. Kennaway. The specificity rule for lazy pattern-matching in ambiguous term rewrite systems. In N. Jones, editor, *ESOP '90 - Proceedings of the Third European Symposium on Programming*, volume 432 of *Lecture Notes in Computer Science*, pages 256–270. Springer-Verlag, 1990.

[KI89] H. Klaeren and K. Indermark. Efficient implementation of an algebraic specification language. In M. Wirsing and J.A. Bergstra, editors, *Proceedings of the METEOR workshop on Algebraic Methods: Theory, Tools and Applications. Passau 87*, volume 394 of *Lecture Notes in Computer Science*. Springer-Verlag, 1989.

[KW93a] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

[KW93b] J.F.Th. Kamperman and H.R. Walters. ARM – Abstract Rewriting Machine. In H.A. Wijshoff, editor, *Computing Science in the Netherlands*, pages 193–204, 1993.

[KW95] J.F.Th. Kamperman and H.R. Walters. Minimal term rewriting systems. Technical Report CS-R9573, CWI, december 1995. Available as http://www.cwi.nl/epic/articles/CS-R9573.ps.Z. To appear in the proceedings of the 11th Workshop on Abstract Data Types, published by Springer-Verlag.

[KW96a] J.F.Th. Kamperman and H.R. Walters. Minimal Term Rewriting Systems. In Magne Haveraaen, Olaf Owe, and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*, volume 1130 of *Lecture Notes in Computer Science*, pages 274–290. Springer Verlag, 1996.

[KW96b] J.F.Th. Kamperman and H.R. Walters. Simulating TRSs by Minimal TRSs: a simple, efficient, and correct compilation technique. Technical Report CS-R9605, CWI, january 1996. Available as http://www.cwi.nl/epic/articles/CS-R9605.ps.Z.

[KZ89] Deepak Kapur and Hantao Zhang. RRL: Rewrite rule laboratory user's manual. Technical Report 89-03, The University of Iowa, 1989.

[MOI95] Aart Middeldorp, Satoshi Okui, and Tesuo Ida. Lazy narrowing: Strong completeness and eager variable elimination. In *Proceedings of the 20th Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1995.

[Mog89] E. Moggi. Computational lambda calculus and monads. In *Logic in Computer Science*. IEEE, 1989.

[P-JW93] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *ACM Symposium on Principles Of Programming Languages (POPL)*, pages 71–84, 1993.

[PvE93] M.J. Plasmeijer and M.C.J.D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison Wesley, 1993.

[Ram92] Norman Ramsey. Literate programming tools need not be complex. Technical Report TR-351-91, Department of Computer Science, Princeton University, October 1991, revised September 1992.

[Sch88] Ph. Schnoebelen. Refined compilation of pattern-matching for functional languages. *Science of Computer Programming*, (11):133–159, 1988.

[SG90] Wolfram Schulte and Wolfgang Grieskamp. Generating efficient portable code for a strict applicative language. In *Phoenix Seminar and Workshop on Declarative Programming, Hohritt (Sasbachwalden, Germany)*, Lecture Notes in Computer Science. Springer-Verlag, 1990. to appear.

[SSD91] David Sherman, Robert Strandh, and Irène Durand. Optimization of equational programs using partial evaluation. *ACM SIGPLAN Notices*, 26(9):72–82, september 1991.

[Str89] Robert Strandh. Classes of equational programs that compile into efficient machine code. In N. Dershowitz, editor, *Rewriting Techniques and Applications, third international conference*, Lecture Notes in Computer Science, pages 449–461. Springer-Verlag, 1989.

[vDHK96] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific Publishing Inc., april 1996.

[Vis96] Eelco Visser. Multi-level specifications. In A. van Deursen, J. Heering, and P. Klint, editors, *Language Prototyping. An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing.* World Scientific Publishing Inc., april 1996.

[Wal94b] H.R. Walters. Implementing tools by algebraic specification. In R.Giegerich and J.H.Hughes, editors, *Functional programming in the Real World*, volume 89 of *Dagstuhl Seminar Report.* Schloss Dagstuhl, 1994.

[War77] D.H.D. Warren. Implementing prolog - compiling predicate logic programs. Technical Report DAI Research Reports 39 and 40, Department of Artifical Intelligence, Edinburgh University, 1977.

[WB90] Dietmar Wolz and Paul Boehm. Compilation of lotos data type specifications. In E. Brinksma, G. Scollo, and C.A. Vissers, editors, *Protocol Specification, Testing, and Verification, IX*, pages 187–202. Elsevier Science Publishers B.V. (North-Holland), 1990. IFIP, 1990.

[WK96] H.R. Walters and J.F.Th. Kamperman. Epic 1.0 (unconditional), An equational programming language. Technical Report CS-R9604, CWI, january 1996.

[WK95] H.R. Walters and J.F.Th. Kamperman. A model for I/O in Equational Languages with Don't Care Non-determinism. In Magne Haveraaen, Olaf Owe and Ole-Johan Dahl, editors, *Recent Trends in Data Type Specification*, number 1130 in Lecture Notes in Computer Science, pages 522—535. Springer-Verlag, 1995.

[WZ95] H.R. Walters and H. Zantema. Rewrite systems for integer arithmetic. In Jieh Hsiang, editor, *Rewriting Techniques and Applications*, number 914 in Lecture Notes in Computer Science, pages 324—338. Springer-Verlag, 1995.

# Appendix II
# The given construct

## 1. Introduction

"Given" is a construct that provides you with some of the expressive power of conditions, without the (model-theoretic) problems of conditions. Before looking at given, let's first consider conditions.

## 2. Conditions

A condition is a test that must be verified before a rule can be applied. That is, even when the left-hand side of a rule matches, it is taken to be inapplicable if its condition(s) fail. For example, consider the following rule:

```
lookup(Name1,list(Name2,Data,List)) = Data <=  Name1 == Name2;
```

Here, `lookup(Name1,list(Name2,Data,List)) = Data` is only applied when `Name1` and `Name2` are equal.

Conditions are an expressive mechanism in practice, but their use can lead to difficulties when inequality is considered. Note that even when explicit inequalities aren't allowed, they occur implicitly due to the specificity ordering, as the following example illustrates:

```
f(a,Y,Z) = a <= Y == Z;
f(X,Y,Z) = b;
```

Clearly, the second rule is more general than the first, but any term of the form `f(a,`*s*`,`*t*`)` reduces to `b` if and only if $s$ and $t$ are unequal. Testing for `f(a,S,T) == b` is identical to testing for the inequality of `S` and `T`.

The difficulties stemming from inequality are also easily illustrated. Consider the following program (we use explicit inequality for brevity here):

```
a = c <= a != b
a = b <= a != c
```

Intuition doesn't help to tell us what this program means, and many implementations which support inequalities get stuck in an infinite loop, when reducing "a". Note that formal models do exist for specifications such as this, but they don't really help us. In this case the program has two (equally reasonable) models: one in which `a` and `c` are equal, but unequal to `b`, and one the other way around.

*2.1 Case determination and auxiliary values*
Two further uses of conditions are

- case-determination: the condition is used to check the outermost function symbol of an argument (possible sub-terms are assigned to auxiliary variables);
- auxiliary values: the condition is used to compute some auxiliary value (and possibly assign it to an auxiliary variable).

*Case-determination* An example of the first use is the following. We assume integers have been defined consisting of function symbols `zero`, `succ` (for plus one) and `pred` (for minus one), and the functions `div` (for integer division) and `mod` (for modulo). We define a function `print` which computes a textual representation of a number.

```
print(zero) = '0';
print(X) = cat(print(div(X,ten)),print(mod(X,ten))) <= X == succ(Y);
print(X) = cat('-',cat(print(div(minus(X),ten)),
                       print(mod(minus(X),ten)))) <= X == pred(Y);
print(succ(zero)) = '1';
...
print(succ(... succ(zero)...)) = '9';
ten = succ(... succ(zero)...);
```

*Auxiliary values* We define sets consisting of the function symbols `emptyset` and `set`, and assume there is a function "`has`" which can determine if a set contains an element. We define a function "`ins`" which inserts an element in a set. Note that the condition "`has(Set,Name) == true`" tests for equality, but isn't harmful in the sense indicated in the previous section, as we will discuss in Section II.4.

```
ins(Name,Set) = Set <= has(Set,Name) == true;
ins(Name,Set) = set(Name,Set) <= has(Set,Name) == false;
ins(Name,emptyset) = set(Name,emptyset);
```

## 3. Given

`Given` is a construct that has expressive power somewhat comparable to conditions, but does not lead to the difficulties sketched in the previous section. With the given construct case determination and auxiliary values are available, but not general term comparison.

As a single example, we will show how the rules defining `plus` from Section 1.3 (which we will first show as a reminder) could be defined using the `given` construct. Note that we do not suggest this to be a good idea in general. In many cases it is an improvement because auxiliary functions do not have to be made explicit; in some cases (such as the one below) it appears to be a matter of taste.

```
plus(o,X) = X;
plus(i,o) = i;
plus(i,i) = ap(i,o);
plus(i,ap(X,Y)) = ap(X,plus(i,Y));
plus(ap(X,Y),Z) = ap(X,plus(Y,Z));
```

These rules are equivalent to:

```
plus(X,Y) = given X,Y as
    o,Z:        Y
    i,o:        i
    i,i:        ap(i,o)
    i,ap(P,Q):  ap(P,plus(i,Q))
    ap(P,Q),Z:  ap(P,plus(Q,Y));
```

Note that `Z` is used twice as a place-holder which matches always and the value of which isn't used (since it is equal to that of `Y`, which *is* used).

The **`given`** construct can be used in the right-hand side of a rule, in any position where a term can be used. It has the form "`given` *terms* `as` *clauses*", where *terms* are a list of one or more terms separated by comma's, and clauses is a sequence of one or more clauses separated by whitespace, which have the form "*terms* `:` `term`". The number of terms between **`given`** and **`as`** must be equal to that before "`:`" in each of the clauses. Of course, the given construct can be nested. Due to the absence of a terminator of the sequence of clauses, an ambiguity arises. This can be solved by terminating the sequence with the (optional) keyword **`nevig`**, as the following example shows.

```
plus(X,Y) = given X as
    o: Y
    i: given Y as
           o:       i
           i:       ap(i,o)
           ap(P,Q): ap(P,plus(i,Q)) nevig
    ap(P,Q):  ap(P,plus(Q,Y));
```

## 4. Semantics

So far we haven't explained why the given construct could not lead to the difficulties surrounding conditions. The answer is simple: **`given`** does not allow us to check for equality of arbitrary terms. It allows us to compute auxiliary value and to do case determination. If the case is a normal form, this constitutes checking equality, but this is also possible in Epic itself. Note that in practice conditions are used very often to compute auxiliary values or to do case determination.

The meaning of the **`given`** construct is defined by a trivial map to pure Epic. We will sketch this definition. The meaning of the rule

```
s = given X,... as
    t1,...: u1
    t2,...: u2
    ...
```

is defined as the meaning of the set of rules

```
s = f(X,...);
f(t1,...) = u1;
f(t2,...) = u2;
```

```
...
```

Here, `f` is a new function symbol that is not otherwise used.

## 5. Degiven

`Degiven` is a tool that translates Epic programs in which the `given` construct is used into equivalent Epic programs without that construct. The tool is a filter, which means that it reads a text (an Epic module with the `given` construct) from standard input and produces and equivalent module without that construct on output.

The tool doesn't have any options, so it is called as follows:

```
degiven < module.epg > modul.ep
```

Note that we use the extension `.epg` for Epic programs with the `given` construct.

Finally, note that this filter is written in Epic with the `given` construct, and that it uses lifted I/O to handle the input language.

# Appendix III
# The stand-alone interpreter

## 1. Literate Programming in NoWeb

This appendix is a literate program, which means that program and documentation are derived from a single source. The program is divided in chunks, whose definition may be distributed over the document. As an example, consider the first part of the chunk *example text*:

55a ⟨*example text* 55a⟩≡
```
    This is example text A
```

The label in the left margin (consisting of the page number, 55, and possibly a letter) can be used to quickly find the definitions of this chunk.

A chunk may be used in the definition of another chunk:

55b ⟨*example.file* 55b⟩≡
  ⟨*example text* 55a⟩
```
    Above, we have texts A and B
```
This code is written to file `example.file`.

Chunks with names that do not contain spaces are written to files with the same name as the chunk. So, for this example, the file `example.file` will contain the text:

```
This is example text A
This is example text B
Above, we have texts A and B
```

The second line is also part of the chunk 'example text', but this part of the chunk is defined later:

55c ⟨*example text* 55a⟩+≡
```
    This is example text B
```

## 2. THE FUNCTION main

In this section we present the function main, which uses the API in a straightforward manner. This function is the 'default' Arm interpreter as used by the Epic/Arm environment.

56 ⟨*the function main* 56⟩≡

```
main(int argc, char *argv[])
{  int i;
   ARM_ref result;
   ⟨comline arg analysis 62⟩
   ARM_set_error_functions(
         &ARM_warning,       &ARM_run_time_error,
         &ARM_load_error,    &ARM_fatal_error);

   ARM_set_debug_functions(
         &ARM_default_show_mnemonic,   &ARM_default_show_fun,
         &ARM_default_semi_step,       &ARM_default_semi_step_fun,
         &ARM_default_display_stacks,  &ARM_default_display_args,
         &ARM_default_from_fun_found,  &ARM_default_start_gc,
         &ARM_default_stop_gc,         &ARM_default_alloc,
         &ARM_default_blockname,       &ARM_default_show_reds,
         &ARM_default_show_profiles);
   ARM_set_up();
   for (i=0; i<libi; i++) {
      ARM_load_arm_file(library[i]);
   }
   ARM_link();
   ARM_ready();
   if (preapply[0]) {
      ARM_push(preapply);
   }
   if (lifted_io) {
      ARM_apush(ARM_lift_input());
   } else {
      do_term();
   }
   result = ARM_reduce();
   ARM_display(lifted_io,result,0);
   return 0;
}
```

57 ⟨*arm.h** 57⟩≡

```
/***                    ***\
***  The Arm interpreter:   ***
** a high-performance engine **
*  for hybrid term rewriting  *
    Design & implementation
*        H.R. Walters         *
**    (c) 1995, 1996 CWI     **
***   (c) 1997 Babelfish    ***
\***                    ***/

#ifndef arm
#define arm

#pragma export on

typedef unsigned long nat32;
typedef nat32 ARM_fun;

extern nat32 ARM_MAXSIZE;
extern nat32 ARM_MAXREWR;
extern long ARM_generation;

typedef struct _ARM_class *ARM_Xclass;
typedef struct _ARM_node *ARM_ref;

void ARM_set_up();
void ARM_load_arm_file(char *name);
void ARM_link();
void ARM_ready();
void ARM_clear();
void ARM_push(char *name);
void ARM_apush(ARM_ref term);
void ARM_pushopq(ARM_Xclass class,char *name);
ARM_ref ARM_lift_input();
ARM_Xclass ARM_class_of(char *name);
ARM_ref ARM_reduce();
ARM_ref ARM_protect(ARM_ref term);
void ARM_unprotect(ARM_ref term);
void ARM_display(int lifted,ARM_ref t,int err);
char *ARM_ofs(ARM_ref t);
unsigned long ARM_size(ARM_ref t);
ARM_ref ARM_child(ARM_ref t,unsigned long i);

void ARM_set_error_functions(
        void(*err1)(char *msg,...),void(*err2)(char *msg,...),
        void(*err3)(char *msg,...),void(*err4)(char *msg,...));

extern void (*ARM_Warn)(char *msg,...);
extern void (*ARM_RTErr)(char *msg,...);
extern void (*ARM_LDErr)(char *msg,...);
extern void (*ARM_Fatal)(char *msg,...);

extern void (*ARM_trace_show_mnemonic)(char *fn);
extern void (*ARM_trace_show_fun)(char *name);
extern void (*ARM_trace_semi_step)();
extern void (*ARM_trace_semi_step_fun)(ARM_fun f);
extern void (*ARM_trace_display_stacks)(char *name);
extern void (*ARM_trace_display_args)(char *name);
extern void (*ARM_trace_from_fun_found)();
extern void (*ARM_trace_start_gc)();
extern void (*ARM_trace_stop_gc)(long count1, long count2);
extern void (*ARM_trace_alloc)(long chunk, long total);
extern void (*ARM_trace_blockname)(char *n);
extern void (*ARM_trace_show_reds)(long l);
extern void (*ARM_trace_show_profiles)();

void ARM_default_show_mnemonic(ARM_fun fn);
void ARM_default_show_fun(char *name);
void ARM_default_semi_step();
void ARM_default_semi_step_fun(ARM_fun f);
void ARM_default_display_stacks(char *f);
void ARM_default_display_args(char *f);
void ARM_default_from_fun_found();
void ARM_default_start_gc();
void ARM_default_stop_gc(long count1, long count2);
void ARM_default_alloc(long chunk, long total);
void ARM_default_blockname(char *n);
void ARM_default_show_reds(long count);
void ARM_default_show_profiles();



extern nat32 ARM_count,ARM_MAXREWR;
extern int ARM_DEBUG_STATUS;
extern int ARM_DBGMSK;
extern nat32 ARM_count;
extern struct tms *ARM_tmx;
extern long ARM_termdpth, ARM_stckdpth;
extern char ARM_frmfid[200];
extern nat32 ARM_frmfun;
extern int ARM_frmmsk;

void ARM_trace_gc(int i);
void ARM_memstat(int i);
void ARM_count_rewr(unsigned long max,int i);
void ARM_profile(int i);
void ARM_dump_degree(int i);
void ARM_stack_dump_depth(unsigned long l);
void ARM_term_dump_depth(unsigned long l);
void ARM_trace_degree(int i);
void ARM_trace_from(char *funsym);

/* here starts previous debug.h */

extern unsigned long ARM_count;

extern int ARM_DBGMSK;
extern unsigned long ARM_count;
extern long ARM_termdpth, ARM_stckdpth;
extern char ARM_frmfid[200];
extern unsigned long ARM_frmfun;
extern int ARM_frmmsk;

#define FRMTRC 1  /* produce statistics only after reducting this */
#define GCINFO 2  /* gc statistics */
#define STKDMP 4  /* produce stackdump each cycle */
#define MNMTRC 8  /* print uarm instruction each cycle */
#define FUNTRC 16 /* print TRS fun being evaluated each reduction */
#define SZINFO 32 /* print initial memory size */
#define RCOUNT 64 /* count (semi) rewrite steps */
#define PRFTIM 128 /* count (semi) rewrite steps */
#define PRFCNT 256 /* count (semi) rewrite steps */
#define DBGLMT 512 /* limit the depth of terms and the depth of stackdumps */
#define ARGDMP 1024  /* produce stackdump each cycle */

void tracefunction(ARM_fun f,void *IP,ARM_ref *AP,ARM_ref *CP);

#define Qshowname(n) \
        if (ARM_DEBUG_STATUS && ARM_DBGMSK & SZINFO) { \
                ARM_trace_blockname(n); \
        }
#define Qtcopq(r,c,m) \
        if (tag(r) != opqtag || ((opq)r)->class != c) { \
                ARM_Warn("bad opaque in %s",m); \
          }


  #endif
```
This code is written to file `arm.h`.

61 ⟨*main.c** 61⟩≡

```
/***                       ***\
***  The Arm interpreter:   ***
** a high-performance engine **
*  for hybrid term rewriting  *
    Design \& implementation
*        H.R. Walters         *
**    (c) 1995, 1996 CWI     **
***   (c) 1997 Babelfish    ***
\***                       ***/

#include "stdio.h"
#include "stdarg.h"
#include "arm.h"
#include "sys/time.h"

#define Kb 1024L

unsigned long units[10] = {1,2,5,10,20,50,100,200,500,Kb};

#define LIBBUFSZ 4000
#define LIBRARIES 200
char libbuf[LIBBUFSZ], *libhere=libbuf;
char *library[LIBRARIES];
int libi=0;

char preapply[200];

int lifted_io;

int TheChar;
char TheBuf[1024];

⟨default error functions 64⟩

⟨external value scanner do_opq 65⟩

⟨function symbol scanner do_fun 66⟩

⟨input term parser do_term 67⟩

⟨the function main 56⟩
```

This code is written to file `main.c`.

62 ⟨*comline arg analysis* 62⟩≡

```
while (--argc > 0) {
   unsigned long OPTVAL;
   char OPT[2];

   if (sscanf(*++argv,"-%1s",OPT) == 1) {
      switch (OPT[0]) {
      case 'm': if (sscanf(*argv,"-%1s%lu",OPT,&OPTVAL) != 2) goto huh;
            if (OPTVAL<0||OPTVAL>9) goto huh;
            ARM_MAXSIZE = units[OPTVAL]*Kb*Kb;
            break;
      case 'a': if (sscanf(*argv,"-%1s%200s",OPT,preapply) != 2) goto huh;
            break;
      case 'l': lifted_io = 1;
            break;
      case 'r': if (sscanf(*argv,"-%1s%200s",OPT,libhere) != 2) goto huh;
            library[libi++] = libhere;
            while (*libhere) {
               libhere++;
            }
            libhere++;
            if (libhere-libbuf > LIBBUFSZ-200 || libi > LIBRARIES) {
               ARM_Warn("too many libraries");
            }
            break;
      case 'L': ARM_DBGMSK |= DBGLMT;   ARM_DEBUG_STATUS = 1;   break;
      case 'A': ARM_DBGMSK |= ARGDMP;   ARM_DEBUG_STATUS = 1;   break;
      case 'G': ARM_DBGMSK |= GCINFO;   ARM_DEBUG_STATUS = 1;   break;
      case 'D': ARM_DBGMSK |= STKDMP;   ARM_DEBUG_STATUS = 1;   break;
      case 'M': ARM_DBGMSK |= MNMTRC;   ARM_DEBUG_STATUS = 1;   break;
      case 'T': ARM_DBGMSK |= FUNTRC;   ARM_DEBUG_STATUS = 1;   break;
      case 'S': ARM_DBGMSK |= SZINFO;   ARM_DEBUG_STATUS = 1;   break;
      case 'R': if (sscanf(*argv,"-%1s%9u",OPT,&OPTVAL) != 2) goto huh;
            ARM_MAXREWR = OPTVAL;
            ARM_DBGMSK |= RCOUNT;   ARM_DEBUG_STATUS = 1;   break;
      case 'P': ARM_DBGMSK |= PRFTIM | PRFCNT; ARM_DEBUG_STATUS = 1; break;
      case 'F': if (sscanf(*argv,"-%1s%200s",OPT,ARM_frmfid) != 2) goto huh;
              ARM_DEBUG_STATUS = 1; break;
      case 'X': if (sscanf(*argv,"-%1s%lu",OPT,&OPTVAL) != 2) goto huh;
            ARM_termdpth = OPTVAL; ARM_DEBUG_STATUS = 1; break;
      case 'Y': if (sscanf(*argv,"-%1s%lu",OPT,&OPTVAL) != 2) goto huh;
            ARM_stckdpth = OPTVAL; ARM_DEBUG_STATUS = 1; break;
      default:
        huh:
         fprintf(stderr,
            "   -m# size 0-9 for max space 1 Mb - 1 Gb; default is 1 Gb\n"
            "   -aid apply the function ‘id’ to input\n"
            "   -l lift input, lower output\n"
            "   -rfile read ‘file’ as library (additional arm file)\n"
            "   -L  limit the stackdepth and termdepth in debug tracing\n"
            "   -Xn  set the term depth in tracing to n (4)\n"
            "   -Yn  set the stack depth in tracing to n (4)\n"
            "   -G  display GC info\n"
            "   -A  produce a dump of args to TRS function\n"
            "   -D  produce stackdump each cycle\n"
            "   -M  trace uarm machine instructions\n"
            "   -T  trace TRS functions\n"
            "   -Fid produce trace after first reduction for this function\n"
            "   -S  print initial memory size\n"
            "   -R#  count (semi) rewrite steps; quit if > # (0 for never)\n"
            "   -P  produce profiling information\n"
            );
            exit(0);
         }
      }
   }
```

64 ⟨*default error functions* 64⟩≡

```
void ARM_warning(char *fmt,...)
{       va_list args;

        fprintf(stderr,"\n*** WARNING\n");
        va_start(args,fmt);
        vfprintf(stderr,fmt, args);
        va_end(args);
        fprintf(stderr,"\n");
        exit(1);
}

void ARM_run_time_error(char *fmt,...)
{       va_list args;

        fprintf(stderr,"\n*** RUN TIME ERROR\n");
        va_start(args,fmt);
        vfprintf(stderr,fmt, args);
        va_end(args);
        fprintf(stderr,"\n");
        ARM_emergency_linked();
        exit(1);
}

void ARM_load_error(char *fmt,...)
{       va_list args;

        fprintf(stderr,"\n*** LOAD ERROR\n");
        va_start(args,fmt);
        vfprintf(stderr,fmt, args);
        va_end(args);
        fprintf(stderr,"\n");
        ARM_emergency_clean();
        exit(1);
}

void ARM_fatal_error(char *fmt,...)
{       va_list args;

        fprintf(stderr,"\n*** FATAL ERROR\n");
        va_start(args,fmt);
        vfprintf(stderr,fmt, args);
        va_end(args);
        fprintf(stderr,"\n");
        exit(1);
}
```

65 ⟨*external value scanner* do_opq 65⟩≡

```
void do_opq()
{   ARM_Xclass class;
   char *buf;

   buf=TheBuf;
   while ((TheChar = getc(stdin))!=':') {
      *buf++ = TheChar;
   }
   *buf = '\0';
   class = ARM_class_of(TheBuf);
   buf=TheBuf;
   while ((TheChar = getc(stdin))!='|') {
      *buf++ = TheChar;
   }
   TheChar = getc(stdin);
   *buf = '\0';

   ARM_pushopq(class,TheBuf);
}
```

66 ⟨*function symbol scanner* do_fun 66⟩≡

```
void do_fun()
{   char *buf=TheBuf;

   if (TheChar == '\'') {
      *buf++=TheChar;   *buf++=getc(stdin);   TheChar = getc(stdin);
   } else if (TheChar == '\\') {
      *buf++=TheChar;   TheChar = getc(stdin);
      switch (TheChar) {
         case 'n': case 'r': case 't': case '\\':
            *buf++=TheChar;   break;
         default:
            *buf++=TheChar;   *buf++=getc(stdin);   *buf++=getc(stdin);
      }
      TheChar = getc(stdin);
   } else if (TheChar == '"') {
      *buf++=TheChar;
         while ((TheChar = getc(stdin)) != '"') {
            if (TheChar == '\\') {
               *buf++=TheChar;   *buf++=getc(stdin);
            } else *buf++=TheChar;
         }
         *buf++=TheChar;   TheChar=getc(stdin);
   } else {
      while ((TheChar>='a' && TheChar<='z')
         || (TheChar>='A' && TheChar<='Z')
         || (TheChar>='0' && TheChar<='9')
         || TheChar=='_' || TheChar=='\'' || TheChar=='-') {
         *buf++=TheChar;
         TheChar = getc(stdin); }
      }
   *buf = '\0';

   if (TheBuf[0]=='\0') {
      ARM_LDErr("illegal char in id: (%d)\n",(long)TheChar);
   }
   ARM_push(TheBuf);
}
```

67 ⟨*input term parser* do_term 67⟩≡

```
void do_term()
{  int balance=0;

   TheChar = getc(stdin);
   for (;;) {
       switch (TheChar) {
         case ' ':
         case '\t':
         case '\r':
         case '\n':  TheChar = getc(stdin); break;
         case '(':   balance++; TheChar = getc(stdin); break;
         case ',':   TheChar = getc(stdin); break;
         case ')':   balance--;
                  if (balance==0) return; else TheChar = getc(stdin); break;
         case EOF:   if (balance) ARM_RTErr("unexpected end of file");
                  return;
         case '|':   do_opq();
                  if (balance==0) return; else break;
         default:   do_fun();
                  if (balance==0 && TheChar != '(')
                     return; else break;
         }
     }
}
```