I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

9401
NOTE
TECHNIQUE
N°

THE ALPHA COMPILER AND UNCOMPILER
TECHNICAL REPORT

DORAN K. WILDE AND ANDREW SNODDY

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 99 84 71 00 – Fax : (33) 99 84 71 71

The ALPHA Compiler and Uncompiler Technical Report

Doran K. Wilde* and Andrew Snoddy

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet API

Note technique n ° 9401 — May 1994 — 25 pages

Abstract: The ALPHA parser translates an ALPHA source program into an abstract syntax tree (AST). The ALPHA unparser, or pretty printer, does the opposite translation from an ALPHA AST back to a source program. These two translators are an integral part of the ALPHA environment. This report is both a user's guide and techical documentation for these two programs.

Key-words: recurrence equations, systolic arrays, hardward design language

(Résumé : tsvp)

*email: wilde@irisa.frThis work was partially supported by the Esprit Basic Research Action NANA 2, Number 6632 and by NSF Grant No. MIP-910852.

Centre National de la Recherche Scientifique Institut National de Recherche en Informatique

Le compilateur et décompilateur de ALPHA
Rapport Technique

Résumé : Le compilateur d'ALPHA traduit un programme écrit en ALPHA vers un arbre de syntaxe abstract (AST). Le décompilateur d'ALPHA fait l'inverse traduction à partir d'un ALPHA AST en retour vers un programme de source. Ces deux traducteurs font un rôle integral dans l'environnement d'ALPHA. Ce rapport est á la fois, un guide d'utiliseur et la documentation technique pour ces deux programmes.

Mots-clé : équations récurrentes, réseaux systoliques, langage de description de matériel

1 Introduction

The ALPHA parser translates an ALPHA source program into an abstract syntax tree (AST). The ALPHA unparser, or pretty printer, does the opposite translation from an ALPHA AST to a source program (see figure 1). These two translators are an integral part of the ALPHA environment which is illustrated in figure 2.

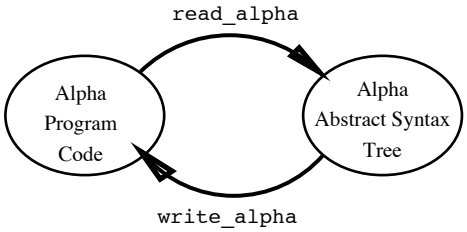

Figure 1: The `read_alpha` and `write_alpha` programs

2 User Guide

This section presents information on how to use the programs described in this document.

2.1 The `read_alpha` program

This program reads and ALPHA source code and produces an abstract syntax tree. The current version at the time of this writing is version 3. The version number can be checked by looking at the first line of the output which says: `(* ALPHA Tree produced by read_alpha v.3 *)`. By

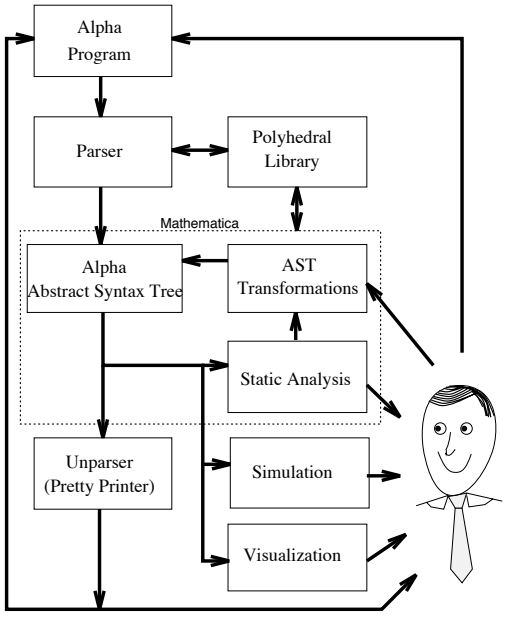

Figure 2: The ALPHA environment

default, input is from *stdin* and output is to *stdout*, which can be overriden with switches. Parsing errors are always reported to *stderr*.

Switches

–i filename	open *filename* as the source file
–I integer	open file descriptor *integer* as the source file
–o filename	open *filename* as the destination file
–O integer	open file descriptor *integer* as the destination file
–d	turn on debugging of parser (generates lots of output)
–E	parse source file as an Alpha expression
–D	parse source file as an Alpha domain

Description

Normally, the compiler tries to compile an entire ALPHA program. However, the –E and –D switches allow the compilation of small pieces of ALPHA source code, by entering the parser at the *expression* and *domain* parse rules, respectively. This is used by the ALPHA environment to do mini-compilations of domains and expressions written in ALPHA code which are used as arguments to transformation functions.

The –d switch is generally used to locate a bug in the compiler.

Error messages

The parse error messages are very terse. However, the line number of the error is reported. This is an area where the compiler needs a lot of improvement.

2.2 The `write_alpha` program

Switches

–i filename	open *filename* as the source file
–I integer	open file descriptor *integer* as the source file
–o filename	open *filename* as the destination file
–O integer	open file descriptor *integer* as the destination file
–d	turn on debugging of parser (generates lots of output)
–a	write ALPHA code in *array* notation
–p	write ALPHA code using *parameters*

Description

The switch –a produces a program in *array notation*, where the equation: A = *expression*.$(z \rightarrow Az + b)$; is written as: A$[z]$ = *expression*$[Az + b]$; Array notation is described more in section 6.2.2.

The switch –p produces a program using parameterized notation, where parameter index variables listed in the `<param_domain>` of the AST are defined in the system header using the keyword `parameter` and are used in the body of the system (in domains and affine functions) without having to be explicitly declared. For example, if N and M are parameters, then an affine function $(i, j, N, M \rightarrow N - i, N + M - j, N, M)$ can be written as: $(i, j \rightarrow N - i, N + M - j)$. Parameter notation is described more in section 6.2.3.

3 AST

This section contains brief descriptions of the files containing general procedures to read, represent and operate on ALPHA Abstract Syntax Trees (AST).

It is organised into four files. One file gives the node type structure, and the other three files contain libraries of related utility procedures. For each of the library files, there is a source file (.c), and two files created by the makefile (.h and .o).

The procedures in any of the library files may be used in another program by including the required .h file in the source code and linking using the .o file.

Below is a list of the files created by the makefile in this directory.

```
/AST/nodeprocs.h
/AST/nodeprocs.o
/AST/node2item.h
/AST/node2item.o
/AST/nodepoly.h
/AST/nodepoly.o
```

3.1 /AST/node.h

A node tree is used to represent an ALPHA AST in memory. The data structure of the node tree is contained in the file `node.h`, which is reproduced here.

```
/* node types */
typedef enum { none,
               /* Number and Id */  num, id,
               /* Lists */          list,
               /* System */         system, decl,
               /* Equation */       equation, cases, restrict, var, affine,
                                    iconst, bconst, rconst,
               /* binop subtypes */ addop, subop, mulop, divop, idivop,
                                    modop, eqop, leop, ltop, gtop, geop, neop,
                                    orop, andop,
               /* unop subtypes */  negop, notop,
               /* ifop */           ifop,
               /* Domains */        dom, pol,
               /* Matrices */       mat
             } nodetype;

/* data types */
typedef enum { inttype, booltype, realtype, notype } datatype;

typedef struct node_
        {   nodetype type; /* node type */
            /* all nodes are listable, and have the following fields */
            struct node_ *next; /* next node in list */
            int index;
            union
            {
/* I.  General Specifications */
                struct
                {   int value;
```

```
                } number;                       /* <number> */
                struct
                {   char name[2];
                } id;                           /* <id> */
                struct
                {   struct node_ *first;
                    struct node_ *last;
                    int count;
                } list;                         /* <list> */
/* II. System specifications */
                struct
                {   struct node_ *id;           /* <system.id> */
                    struct node_ *in;           /* <system.in_var> */
                    struct node_ *out;          /* <system.out_var> */
                    struct node_ *local;        /* <system.local_var> */
                    struct node_ *equations;    /* <system.equation_list> */
                } system;                       /* <system> */
                struct
                {   struct node_ *id;           /* <decl.id> */
                    datatype type;              /* <decl.data_type> */
                    struct node_ *domain;       /* <decl.domain> */
                    int          *l, *w;        /* <decl.l> <decl.w> */
                                                /* abv used in C simulator only */
                } decl;
/* III. Equation and Expression specifications */
                struct
                {   struct node_ *id;           /* <equation.id> */
                    struct node_ *cases;        /* <equation.case> */
                } equation;                     /* <equation> */
                struct
                {   struct node_ *list;         /* <case.list> */
                } cases;                        /* <case> */
                struct
                {   struct node_ *domain;       /* <restrict.domain> */
                    struct node_ *exp;          /* <restrict.exp> */
                } restrict;                     /* <restrict> */
                struct
                {   struct node_ *id;           /* <var.id> */
                } var;                          /* <var> */
                struct
                {   struct node_ *var;          /* <affine.var> */
                    struct node_ *matrix;       /* <affine.matrix> */
                } affine;                       /* <affine> */
                struct
                {   int value;                  /* <const.number> */
                } iconst;                       /* <const> */
                struct
                {   int value;                  /* <const.number> */
                } bconst;                       /* <const> */
                struct
                {   int value;                  /* <const.number> */
                    int fraction;               /* <const.fraction> */
```

```
                } rconst;                       /* <const> */
                struct
                {   struct node_ *lexp;         /* <binop.left_exp> */
                    struct node_ *rexp;         /* <binop.left_exp> */
                } binop;                        /* <binop> */
                struct
                {   struct node_ *exp;          /* <unop.exp> */
                } unop;                         /* <unop> */
                struct
                {   struct node_ *exp1;         /* <if.exp1> */
                    struct node_ *exp2;         /* <if.exp2> */
                    struct node_ *exp3;         /* <if.exp3> */
                } ifop;
/* IV. Domain specifications */
                struct
                {   int dim;                    /* <domain.dimension> */
                    struct node_ *index;        /* <domain.id_list> */
                    struct node_ *pol;          /* <domain.pol_list> */
                } dom;                          /* <domain> */
                struct
                {   int conum;                  /* <pol.constraints_num> */
                    int raynum;                 /* <pol.ray_num> */
                    int eqnum;                  /* <pol.eqn_num> */
                    int linum;                  /* <pol.line_num> */
                    struct node_ *constraints;
                    struct node_ *rays;
                        /* constraints,rays = list of list of iconst */
               } pol;                           /* <pol> */
/* V. Matrix specifications */
                struct
                {   int rows;                   /* <matrix.rows> */
                    int cols;                   /* <matrix.cols> */
                    struct node_ *id;           /* <matrix.id_list> */
                    struct node_ *matrix;       /* <matrix.matrix> */
                              /* matrix = list of list of const */
                } mat;                          /* <matrix> */
            } the;
        } node;
```

3.2 /AST/nodeprocs.c

This section describes procedures used to create node tree.

```
extern node * add_to_list ( node *n, node *e );
extern void free_node ( node *n );
extern node * new_id ( char *s );
extern node * new_list ( node *e );
extern node * new_node ( nodetype t );
```

add_to_list(n,e)

This procedure takes two input parameters. The first must be a pointer to a node of type list and the second must be a pointer to a node. It returns a pointer to the list, with the new node appended to the end of the list. If the list is empty, a new list containing only the new node is returned. If the new node is empty, then the original list is returned.

free_node(n)

This procedure takes a node pointer and if memory has been allocated for it, frees the memory for re-use. Nothing is returned.

new_id(s)

This procedure takes a string as it's input parameter. It returns a new node of type 'id' where the input string becomes the textual content of the node.

new_list(e)

The procedure takes a node pointer as it's input parameter and returns a list type node, with the input node as the sole item on the list. If the node pointer is empty, then an empty list is created and returned.

new_node(t)

This procedure takes a node type as it's input parameter. It allocates the required memory, initialises the type to the input type and returns this node. Memory overflow will result in an error message being displayed.

3.3 /AST/nodepoly.c

This section describes the procedures used to do computation with domains. These procedures rely on functions in the polyhedral library [2].

```
extern Polyhedron **compute_loop ( Polyhedron *D, Polyhedron *C );
extern void compute_W ( Polyhedron *D, int numP, int *valP,
int **l1, int **w1 );
extern Polyhedron *node2Polyhedron ( node *D );
extern node *Polyhedron2node ( Polyhedron *p );
```

compute_loop(D,C)

This procedure does loop synthesis on the domain D in the context of domain C [1]. The domain D is factored into a list of domains which are suitable as nested loop domains. This list is returned by the function.

compute_W(D,numP,valP,l1,w1)

This procedure computes the doping vector W used to compute the offset into a memory array storing the values in domain D [4]. The variable $numP$ and the vector $valP$ contain the number and values of the parameters respectively. The result is returned in the integer vectors w and l.

Using these two results, both the amount of memory needed and a linear offset function for a given point can be determined.

node2Polyhedron(D)

This function takes node pointer D as it's input parameter, and converts it into a Polyhedron pointer. The same data is contained in both but in different formats. The conversion is effected to enable calculations on data by functions in the Polyhedral library.

Polyhedron2node(P)

This does the inverse translation to the procedure above.

3.4 /AST/lex.l

This is a lexical analyzer to read AST files as described in section 5.

4 Pretty

This section contains brief descriptions of the files used for reading, manipulating and printing item trees, along with a description of the item structure. Item trees represent a hierarchy of text formatting commands.

It is organised into four main files. Firstly a file to describe the item structure. Secondly a file containing the relevant procedures, and finally two files to deal with reading and writing. The makefile creates from the source files .h and .o files for the procedure and write files, and also creates a file called *pretty*, which is the executable pretty printer. The file *pretty* may be used separately on any box language program to produce pretty printed output.

The following files are created by the makefile

```
/Pretty/itemprocs.h
/Pretty/itemprocs.o
/Pretty/writeitem.h
/Pretty/writeitem.o
/Pretty/pretty
```

To use any of the procedures in another piece of source code, include the .h file and link using the .o file.

4.1 Pretty Printer Description

This section describes the pretty printer formatting commands and the implementation of the program to produce pretty–printed ALPHA code. The main objectives in the design of the pretty printer engine were to :

1. Facilitate the translation of an ALPHA AST into ALPHA code format. This meant that the language needed to be able to format block structured text and be able to show blocks using indentation and enclosure in keywords. The pretty printer also needed to be able to print lists in a variety of ways.

2. Format the output, using a small set of commands, to ensure flexibility of output format, ease of implementation and aesthetically pleasing output.

3. Increase the maintainability of the pretty printer in view of the continuing developement of the Alpha language.

It should be noted, that the scope of the box language is not limited to formatting ALPHA code. The box language is highly flexible, and is capable of formatting text to satisfy a wide variety of desired output requirements. The pretty printer is suitable for formatting and printing any block and list oriented text.

4.1.1 The Item Box Language

The item box language is used to format ASCII programs and consists of a small number of print templates (patterns, or stencils) which are called "boxes" (to denote a place to put things). The box language consists of ten different items, each with it's own parameters.

The item types are :

item	::=	Text(*string*)	|
		Number(*value*)	|
		Hspa(*value*)	|
		Vspa(*value*)	|
		Hsep(*indent*, *sep*, *item_list*)	|
		Hlis(*indent*, *sep*, *item_list*)	|
		Vsep(*indent*, *sep*, *item_list*)	|
		Vlis(*indent*, *sep*, *item_list*)	|
		Venc(*indent*, *open*, *close*, *body*)	|
		Henc(*indent*, *open*, *close*, *body*)	
item_list	::=	*item*, ⋯	
value	::=	NUMBER	
indent	::=	NUMBER	
sep	::=	"a quoted list separator"	
string	::=	"a quoted text"	
open	::=	*item*	
close	::=	*item*	
body	::=	*item*	

Notice that item_list, open, close and body, are all themselves *items*. Thus, the arguments of an item may themselves be items, recursively. Therefore, the associated item processing procedures must also be recursive.

4.1.2 Description of item boxes

In this section, each kind of item box is described and it's parameter types and function are defined.

Text(*string*)

A *text* box is an item which contains only a *string* as parameter. It may contain anything from an empty string to a string of many characters, such as "qwertyuiop". All combinations of digits, letters and keyboard symbols are valid contents for the text box.

Examples of text items are: Text("case") and Text(";").

Figure 3: Text Item Box

Number(*value***)**

A *number* box is an item for displaying any integer *value*. It does not allow for the representation of real numbers. This item is similar to the text box, except it prints a number instead of a string.

Examples of number items are Number(12), Number(0), and Number(-18765).

Hspa(*value***)**

The horizontal space item prints horizontal blank space on a line. This box type has as it's single parameter *value*, an integer variable, which defines the number of spaces which are to be printed. If *Hspa* overflows the line, a newline is forced, the rest of the spaces are ignored by the formatter, and printing continues with the next item at the margin of the next line. The arrow shows a horizontal

Figure 4: Hspa Item

space between two items.

Vspa(*value***))**

The vertical space item is used when one or more blank *lines* need to be printed. The item takes a parameter *value*, an integer representing the number of whole lines to be left blank. The arrow

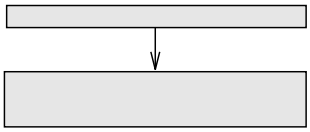

Figure 5: Vspa Item

shows a vertical space between two items.

Hsep(*indent***,** *sep***,** *item_list***)**

The horizontally separated item is a means of printing lists of items, separated by a constant string *sep*. For example, the item tree:

```
Hsep( 0, " | ", Hsep( 0, ",", Text("Z"), Text("t"), Text("p") )
                Hsep( 8, "; ", Text("2Z-t-p+4=0"), Text("-p+4>=0"),
                               Text("p>=0"), Text("-t-p+14>=0")
                    )
    )
```

would be printed as:

```
Z,t,p | 2Z-t-p+4=0; -p+4>=0; t+p-4>=0; p>=0;
        -t-p+14>=0
```

This example illustrates the recursive nature of item trees. In this case, we have Text items inside of Hsep items inside of an Hsep item.

The problem of what to do when a list overflows onto a new line is resolved by specifying *indent*, an integer parameter which gives the relative indention of additional lines of the list, if they are required. On all subsequent new lines, the list text would be indented to this value, until the end of the list was reached.

This produced a box which would have a pattern similar to where the arrow signifies the indent,

Figure 6: Hsep Item Box

after the first new line. There is great flexibility about the exact shape of a box, as the contents will decide the specific dimensions and pattern.

Hlis(*indent***,** *sep***,** *item_list***)**

The horizontal list item is only slightly different from the Hsep item. It formats the list with a separator *sep* like Hsep with the addition of one last separator after the final item on the list. Hlis(*indent*, *sep*, *item_list*) is roughly equal to Hsep(0, "", Hsep(*indent*, *sep*, *item_list*), Text(*sep*)).

Vsep(*indent***,** *sep***,** *item_list***)**

The vertically separated item is similar to the horizontally separated item, except each item on the list appears on a new line and thus *vertically* separated. The separator *sep* is printed after and on the same line as each item on the list except the last item.

The general box shape is as follows:

Figure 7: Vsep Item Box

Vlis(*indent***,** *sep***,** *item_list***)**

The vertical list item is similar to the horizontal list box (Hlis) in that also it prints a separator after the last item in the list. It is similar to the vertically separated item in that each item and its separator are printed on a new line.

Venc(*indent***,** *open***,** *close***,** *body***)**

Vertically enclosed items are used to format blocks of text delimited by *open* and *close* items. The *body* item is placed vertically in between the delimiters with an indentation of *indent* relative to

the delimiters. Examples of such a box is: Venc(2, Text("case"), Text("esac"), Vlis(0, ";", ...)) which is printed as:

```
case
  {Z,t,p | Z=0}: True.(Z,t,p->);
  {Z,t,p | Z-1>=0}: False(Z,t,p->);
esac
```

The box shape for a Venc item is shown below. As usual the arrow denotes the indent.

Figure 8: Venc Item Box

Henc(*indent*, *open*, *close*, *body*)

The horizontally enclosed item is similar to the vertically enclosed items and is also used to format blocks of text delimited by *open* and *close* items. The difference is that the *body* item is placed horizontally in between the delimiters. If the line overflows, new lines will be indented with a relative indentation of *indent*. As can be seen from the diagram, the body continues on the same

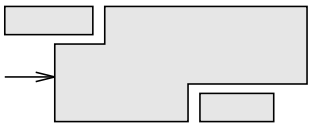

Figure 9: Henc Item Box

line as the open, and the close item follows the body on the same line if possible. No new lines separate the open, close and body. If one of the items length's is such that it requires starting a new line, it simply continues to the next line (with suitable indent), and continues as normal.

4.2 /Pretty/item.h

This data structure defines the internal representation used by the print engine. It is contained in the file `item.h` which is shown below :

```
#define TEXTLENGTH 128
#define SEPLENGTH 8

typedef struct item_
{       int             type;
        int             prec;
        struct item_   *next;
        union { struct { struct item_   *item_list;
                         char            sep[SEPLENGTH];
                         int             width;
                         int             indent;            } hsep;
                struct { struct item_   *item_list;
```

```
                        char            sep[SEPLENGTH];
                        int             width;
                        int             indent;             } hlis;
                struct { struct item_   *item_list;
                        char            sep[SEPLENGTH];
                        int             indent;             } vsep;
                struct { struct item_   *item_list;
                        char            sep[SEPLENGTH];
                        int             indent;             } vlis;
                struct { int            indent;
                        struct item_    *open;
                        struct item_    *close;
                        struct item_    *body;              } venc;
                struct { int            indent;
                        struct item_    *open;
                        struct item_    *close;
                        struct item_    *body;              } henc;
                struct { char           string[TEXTLENGTH]; } text;
                struct { int            value;              } number;
                struct { int            n;                  } hspa;
                struct { int            n;                  } vspa;
        } the;
} item;

typedef enum { hsep, hlis, vsep, vlis, venc, henc, text, number, hspa, vspa
            } item_type;
```

4.3 /Pretty/itemprocs.c

This section describes procedures used to create an item tree.

```
extern item *add_to_ilist ( item *i1, item *i2 );
extern item *check_prec ( item *a, int b );
extern item *copy_item ( item *anitem );
extern item *new_henc ( int indent, item *open, item *close,
item *body );
extern item *new_hlis ( int indent, char sep[8], item *body );
extern item *new_hsep ( int indent, char sep[8], item *body );
extern item *new_hspa ( int val );
extern item *new_item ( item_type spec_type );
extern item *new_list1 ( item *i1 );
extern item *new_list2 ( item *i1, item *i2 );
extern item *new_list3 ( item *i1, item *i2, item *i3 );
extern item *new_list4 ( item *i1, item *i2, item *i3, item *i4 );
extern item *new_list5 ( item *i1, item *i2, item *i3, item *i4,
item *i5 );
extern item *new_list6 ( item *i1, item *i2, item *i3, item *i4,
item *i5, item *i6 );
extern item *new_text ( char *txt );
extern item *new_venc ( int indent, item *open, item *close,
```

```
item *body );
extern item *new_vlis ( int indent, char sep[8], item *body );
extern item *new_vsep ( int indent, char sep[8], item *body );
extern item *new_vspa ( int val );
```

add_to_ilist(i1,i2)

This procedure takes two items as input parameters. It appends the second item to the first item, before returning a pointer to the first.

check_prec(a,b)

This function takes two items as input parameters. It compares their respective precedences to decide whether or not the first needs to be placed in parentheses, with respect to the second. If so, then a new item is created.

copy_item(anitem)

This function takes an item and by recursive techniques, copies the item. Finally it returns a pointer to the copied item.

new_henc(...) new_venc(...) etc.

These functions allocate memory for various kinds of item nodes. They take the input parameters specific to that kind of item and return an item of the particular type suitable for inclusion in the item tree.

new_list2(..) new_list3(...) etc.

Each of these functions are used to create lists suitable as the last parameter in Hsep, Vsep, Hlis and Vlis items.

new_item(spec_type)

This functions allocates memory for a new item node. It takes as it's input parameter an item type (Vsep, Henc, etc.) and returns an empty item of this particular type.

4.4 /Pretty/writeitem.c

4.4.1 The item tree printer

The item tree printer is a single procedure called `print_item` capable of printing any kind of item. It is defined in the file `pretty.c` along with all of the utility procedures that it calls. Since an item tree is recursivly defined, `print_item` is a recursive procedure to print the item tree. A large switch statement is used to decide what type of item to print and the code for each case prints the item according to the specification. The print format of all types of items are encoded into this one procedure, and this is where the definition of the output format is realised in code.

The printing is controlled by the current context, which includes:

- a current position marker which keeps a count of where the text is currently being positioned on a line,

- a left margin to position the first character of a new line, and
- a right margin value to limit the size of each line.

All indentations are made relative to the current left margin. Thus, an indentation is "nested" within the current context.

One of the main features of the pretty printer, was it's ability to handle large items i.e. those which ran over several lines, whilst maintaining the desired aesthetic qualities. The guiding rules are (in priority):

1. if an item will fit entirely on the current line, then place it there, else
2. if an item will fit entirely the next line then force a new line and place the item on the next line, else
3. start to place the item on the current line. It will need to be split on the boundaries between the items contained within it, if possible.

Depending on the items and their lengths, the largest possible amount of text is fit on to each line. Breaks in text are only made when necessary, and at the highest possible level in the item tree. This result in breaks at *logical* points, which improves not only the asthetic quality, but also readability.

```
extern void print_item ( item *x );
```

print_item(item pointer)

This procedure takes the item pointer declared as the input parameter, and prints the item according to its item format and according to the current context. A large switch statement is used to decide what type of item to print and the code for each case prints the item according to the specification.

First, all the necessary changes to the left margin are made by simply adding the indent value found in the item to the current left margin. This ensures that on recursive calls, the left margin will be correct.

Then according to the item type:

Vspa item: The specified number of newlines are printed and the cursor is left at the current left margin.

Number or Text item: The item length is tested (forcing a newline if the item doesn't fit), then the item is printed and the current position counter augmented by the value of the length of the item.

Hspa item: If the specified number of spaces will fit on the current line, then print the spaces and update the current position counter, else force a newline.

Hsep and Hlis item: The length of each item and its separater are tested by the `test` procedure to see if they will fit on the current line. If not, then a new line is printed. Each item on the list is printed by a recursive call to the `print_item` procedure. This was repeated for each item in the list. After the final item, Hlis prints a separator and Hsep does not. Finally, the left margin variable is restored to its original value.

Vsep and Vlis item: Each item on the list is printed on a separate line by a recursive call to the `print_item` procedure, followed by its separator. This was repeated for each item in the list. Vlis prints a separator after the final item, whilst Vsep does not.

Henc item: The open item is printed by a call to the *print_item* procedure. The left margin variable was augmented by the indent value to ensure that subsequent new lines will begin at the correct margin value. Then the body and close items are printed by recursive calls to the *print_item*. Finally, the left margin variable is restored to its original value.

Venc item: If the curser it not at the current margin, it is aligned by calling the `align_to_margin` procedure. Then, the open item is printed followed by a new line. Then, the left margin is adjusted to indent the body. Next, the body is printed followed by a new line. Then the left margin is restored to its original value, and the close item is printed, leaving the cursor at the end of the close item. The open, body, and close items are printed using a recursive call to the `print_item` procedure.

The following internal procedures are used, but are not accessible outside the library which contains the main printing procedure.

len_item(item pointer)

This procedure takes as it's input an item pointer, and returns the length of the item, in terms of characters required. For all items, except the Venc, Vsep and Vlis type, the value returned is the sum of the constituent parts, including all appropriate instances of separators etc.

For a Venc type, the value considered for length purposes, is simply the length of the open item, as a new line will be, by definition, forced after this. In the case of both Vsep and Vlis, the value returned is the length of the first item on the list plus the separator value, as again a new line will be forced after this.

newline()

This procedure takes no input parameters and returns nothing. It's purpose is merely to conditionally print a new line, provided the current position counter is not equal to the current left margin already. After printing a new line, the number of spaces necessary to bring the current cursor position to the left margin are printed, and the current position counter is set to the left margin.

test(integer)

This procedure inputs an integer value and tests whether an item with that length will fit on a line. This is done by comparing the value of the current position variable plus the length with the right margin. If the value *is* greater than the right margin, a call is made to the newline procedure, otherwise nothing is done.

When used in conjunction with the `len_item` procedure, this procedure tests if a particular item will entirely fit on the current line.

align_to_margin(integer)

This function takes an input integer margin and tests if the current position is equal to this margin. If it is greater, a new line is forced. Then the current position is spaced to the specified margin.

4.5 /Pretty/lex.l

This is a lexical analyzer to read the pretty command language which is described in detail in section 4.1.1.

4.6 /Pretty/readitem.c

This is the source code for the program to read and print a pretty command file. After compilation, the executable file is called *pretty*.

5 The ALPHA abstract syntax tree (external format)

This is the definition of the ascii form of the abstract syntax tree (AST) which is produced by the parser and read and written by the symbolic algebra system. Semantic notes are placed in parentheses. Non-terminals are enclosed in angle braces <> and the vertical bar | is used for alternation. The starting non-terminal is <program>.

Terminal and General specifications

Numbers, ids, and lists are defined generally.

```
<*_number>   ::= <number> (with <*> representing any nonterminal)
<*_id>       ::= <id>     (with <*> representing any nonterminal)
<*_list>     ::= { <*>, <*>, ... , <*> }

<number>     ::= [0-9][0-9]*
<real>       ::= <number>.<number>
<boolean>    ::= True | False
<id>         ::= "a name"
<comment>    ::= (* blah blah blah *)                          (3)
```

Main System Specifications

```
<program>    ::= system [<system_id>,                          (9)
                         <param_domain>,                       (4)
                         <in_vars>,
                         <out_vars>,
                         <local_vars>,
                         <equation_list> ]
<param_domain>::= <domain>
<in_vars>    ::= <declare_list>
<out_vars>   ::= <declare_list>
<local_vars> ::= <declare_list>
<declare>    ::= decl [ <id>, <data_type>, <domain> ]          (5)
<data_type>  ::= integer | boolean | real | notype             (10)
```

Equation and Expression specifications

```
<equation>   ::= equation [ <id>, <exp> ]                      (1)
<var>        ::= var[<id>]
<exp>        ::= <var>
                 | const[<number>]
```

```
                | const[<boolean>]
                | const[<real>]
                | binop [ <bop>, <exp>, <exp> ]                (6)
                | unop  [ <uop>, <exp> ]                       (6)
                | if [ <exp>, <exp>, <exp> ]                   (6)
                | affine [ <exp>, <matrix> ]                   (7)
                | restrict [ <domain>, <exp> ]                 (13)
                | case [ <exp_list> ]                          (14)
<bop>       ::= add | sub | mul | div | idiv | mod | min | max
                | eq | le | lt | gt | ge | ne | or | and | xor
<unop>      ::= neg | not
```

Domain specifications

The domain specification closely follows the internal format of the domain defined in the polyhedral library [2] to minimize the overhead of domain storage and of making library calls.

```
<domain>     ::= domain [ <dimension_number>,                  (2)
                          <id_list>,                           (12)
                          <pol_list> ]
<pol>        ::= pol [ <constraints_number>,
                       <rays_number>,
                       <equations_number>,
                       <lines_number>,
                       <constraint_list>,
                       <ray_list> ]
<constraint> ::= { <const_type>, <number>, ... , <number> }  (8)
<const_type> ::= 0 (=equality) | 1 (=inequality)
<ray>        ::= { <ray_type>, <number>, ... , <number> }     (11)
<ray_type>   ::= 0 (=line) | 1 (=vertex/ray)
```

Matrix specifications

```
<matrix>     ::= matrix [ <rows_number>,
                          <cols_number>,
                          <id_list>,                           (12)
                          {{ <number>,<number>, ... ,<number> },
                           { <number>,<number>, ... ,<number> },
                           ...
                           { <number>,<number>, ... ,<number> }}]
```

5.1 Semantics

1. Single assignment semantics
2. Domains are unions of finite convex polyhedra
3. Comments may appear anywhere
4. Parameter domain defines the parameter space over which the system is defined.
5. Scalar domains are domains of dimension 0

6. Semantics of operators:

Operators	Source Types→	Destination Type
add sub div mul neg	int/real	int/real
eq le lt gt ge ne	int/real	boolean
and or not	int/boolean	int/boolean
if	boolean, int/real/bool, int/real/bool	int/real/boolean

7. All dependencies are of the form: `var1[i,j,k,...] --> var2[affine_function(i,j,k,...)]` This affine function may be represented as a transformation matrix T with d rows and d+1 columns, for example:

```
[i]    [ t11 t12 t13 ] [i]
[j] =  [ t21 t22 t23 ] [j]
                       [1]
```

   represents dependency:
   `[i,j] --> [t11*i + t12*j + t13, t21*i + t22*j + t23]`

8. Constraints may be either affine equalities or affine inequalities. Constraints represented as a list of coefficients, for example:

```
{ 0, c1, c2, c3, c4 }  ::= ( c1*i + c2*j + c3*k + c4 = 0 )
{ 1, c1, c2, c3, c4 }  ::= ( c1*i + c2*j + c3*k + c4 >= 0 )
```

9. Single (main) function programs, subsystems not supported.

10. Precision is unspecified.

11. "Rays" may be either vertices, rays, or lines. They are represented as a list of coefficients, for example:

```
{ 0, c1, c2, c3, 0 }  ::=   line(c1,    c2,    c3)
{ 1, c1, c2, c3, 0 }  ::=    ray(c1,    c2,    c3)
{ 1, c1, c2, c3, c4 } ::= vertex(c1/c4, c2/c4, c3/c4), c4!=0
```

12. `<id_list>` is a list of names of the indices, each of which is limited to be 8 characters long.

13. An expression `<exp>` is only defined within the restricted domain `<dom>`

14. The domains of the expressions within a case do not intersect each other.

6 The ALPHA Parser — read_alpha

The `read_alpha` program, described in this section, reads ALPHA source code and produces ALPHA AST (whose format is described in section 5. The syntax and semantics of an ALPHA program are fully described in [3].

6.1 Technical Guide

The `Read_Alpha` directory contains the following source files:

```
lex.l
yacc.y
node.c
nodetypes.h
read_alpha.c
```

The file `lex.l` is a lexical analyzer and `yacc.y` is the parser to read ALPHA source programs. The file `nodetypes.h` defines the internal data structure to represent the AST of an ALPHA program and the file `node.c` is a library of procedures to operate on that data structure. This data structure is similar to the node tree described earlier in section 3, but has some important differences and is more complex. (It actually is the forerunner to the node tree data structure.) And finally, the file `read_alpha.c` is the source code of the main program, which simply reads in the command line parameters and calls the parser.

6.2 ALPHA syntax

This section describes the most recent features added to the `read_alpha` program, version 3.0. The new parser recognizes the following constructs: array notation, min and max lists in domain constraints, an "include" facility, nesting of equations in restrictions, parameterized system support, and the reduce function. These features will be described and examples given based on the convolution program:

```
system convolution (a : { j | 1<=j<=4 } of integer;
                    x : { i | i>=1    } of integer)
      returns     (y : { i | i>=1    } of integer);
  var
    Y : { i,j | 0<=j<=4 ; i>=1; i>=j }  of integer;
  let
    Y = case
         { i,j | j=0 }     : 0.(i,j->);
         { i,j | 1<=j<=4 } : Y.(i,j->i,j-1)
                           + a.(i,j->j) * x.(i,j->i-j+1);
        esac;
    y = case
         { i | i<=3 } : Y.(i->i,i);
         { i | i>=4 } : Y.(i->i,4);
        esac;
  tel;
```

6.2.1 Constraints

The parser recognizes lists of expressions between relational operators ($<, \leq, =, \geq, >$). These lists can be interpreted as minimum and maximum operations, depending on the context. For example, the declaration for Y in the above program, can be rewritten as:

```
  Y : {i,j | (1,j)<=i; 0<=j<=4} of integer;
```

meaning that i is greater than or equal to $max(1, j)$. The words *min* and *max* are not explicitly written.

In general, $(a, b) < (c, d) < (e, f)$ would mean $max(a, b) < min(c, d)$ and $max(c, d) < min(e, f)$. It can also be interpreted as the list of constraints: $a < c$; $a < d$; $b < c$; $b < d$; $c < e$; $c < f$; $d < e$; $d < f$.

6.2.2 Array Notation

Array notation is parsed and interpreted as follows:

$$\mathrm{A}[z] = \cdots \{ \mid \textit{constraints}\} : \textit{expression}[Az + b] \cdots;$$
$$\downarrow$$
$$\mathrm{A} = \cdots \{\mathrm{z} \mid \textit{constraints}\} : \textit{expression}.(z \rightarrow Az + b) \cdots;$$

where *expression* is in normal (non–array) format.

The convolution example can be written in *array notation* as:

```
system convolution (a : {j | 1<=j<=4} of integer;
                    x : {i | 1<=i} of integer)
      returns (y : {i | 1<=i} of integer);
var
  Y : {i,j | (1,j)<=i; 0<=j<=4} of integer;
let
  Y[i,j] =
      case
        {|j=0} : 0[];
        {|1<=j<=4} : Y[i,j-1] + a[j] * x[i-j+1];
      esac;
  y[i] =
      case
        {|i<=3} : Y[i,i];
        {|4<=i} : Y[i,4];
      esac;
tel;
```

6.2.3 Parameterized Systems

The parser recognizes a parameter list and associated constraints as the first entry in the input declarations part of the system header. All parameters must be presented in a single parameter declaration and the context domain constraining the parameters with respect to one another must match the parameter list. The syntax for a parameter declaration is:

$$z : \{z \mid \textit{constraints}\} \textbf{ parameter} ;$$

The convolution example can be extended from 4 to N, using the *parameter notation* as follows:

```
system convolution (N : { N | N>=0 } parameter;
                    a : { j | 1<=j<=N } of integer;
                    x : { i | i>=1    } of integer)
      returns     (y : { i | i>=1    } of integer);
  var
    Y : { i,j | 0<=j<=N ; i>=1; i>=j }  of integer;
  let
    Y = case
         { i,j | j=0 }     : 0.(i,j->);
         { i,j | 1<=j<=N } : Y.(i,j->i,j-1)
                           + a.(i,j->j) * x.(i,j->i-j+1);
        esac;
    y = case
         { i | i<=N-1 } : Y.(i->i,i);
         { i | i>=N }   : Y.(i->i,N);
        esac;
  tel;
```

6.2.4 Reduction Operator

The reduction operator is parsed in its usually accepted format:

reduce(*commutative-associative-op*, *projection*, *expression*)

The commutative and associative operations allowed at are: $+$, $*$, **and**, **or**, **xor**, **min**, and **max.** Using the reduction operator, the convolution can be specified (parametrically) as:

```
system convolution (N : { N | N>=0 } parameter;
                    a : { j | 1<=j<=N } of integer;
                    x : { i | i>=1    } of integer)
      returns     (y : { i | i>=1    } of integer);
  var
    Y : { i,j | 0<=j<=N; i>=1; i>=j }  of integer;
  let
    y = reduce(+, (i,j->i), a.(i,j->j)*x.(i,j->i-j+1) );
  tel;
```

6.2.5 File include directive

The compiler recognizes and obeys an "include" compiler directive. For example, the ALPHA program consisting of the single lines:

```
--include convolution.alpha
```

Will cause the file "convolution.alpha" to be included and parsed. The include directive must start at the beginning of the line and may be nested to any depth.

6.2.6 Prefix notation for binary operators

Binary operations can be written in either infix notation (e.g. $A+B$, $A \max B$) or in prefix notation (e.g. $+(A,B)$, $\max(A,B)$). The commutative binary operators are: $+$, $*$, **and**, **or**, **xor**, **min**, and **max** and the non-commutative binary operations are: **−**, **div** and **mod.**

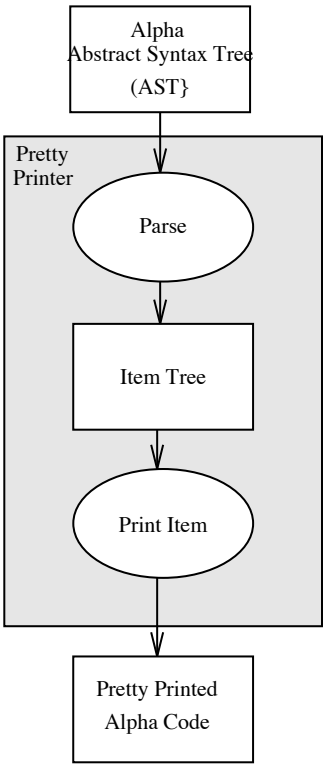

Figure 10: Steps in Unparser

6.3 Parser operation

The parser creates a data structure called a *node tree* while parsing the source file. The node tree is the internal memory representation of an ALPHA AST (see section 5) and it is described in the next subsection. After the node tree has been created, it is printed using the node tree print procedure.

7 The ALPHA Unparser — (write_alpha)

The unparser (pretty printer) operates in two steps as illustrated in figure 10. First, the *generator* parses an ALPHA AST (defined in section 5) and at the same time, it creates an internal data structure called an *item tree* which contains text and formatting commands. Each node in this item tree is a template telling how to print that branch of the tree.

Secondly, the print engine scans the tree produced by the generator, and writes the formatted ALPHA code to the output. The print engine is a general purpose formatter which carries out the formatting commands contained in the nodes of the item tree. The kind of formatting done is universal to the general problem of formatting structured code, and thus the pretty printer engine is the "back end" of many of our programs that have to generate structured textual programs. The print engine is described in detail in section 4.1.

7.1 /Write_Alpha/yacc.y (The tree generator)

This is the main program for the unparser. It uses a yacc parser to generate an item tree using procedures in Pretty (see section 4.3). The generator parses the AST into a formatted *item tree*, the internal data structure of the pretty printer engine. This data structure is defined in the file `item.h` (see section 4.2). The main body of the generator is a parser found in the `yacc.y` file. It gives the specific actions for creating the item tree for each grammatical rule needed to parse an ALPHA AST.

The item tree is then printed using the *print_item* procedure, as detailed in section 4.4.

7.2 Other Pretty Printers

Two other similarly structured pretty printers exist. These are called Write_C and Write_Tex.

Write_C outputs a C program capable of simulating an Alpha program.

Write_Tex outputs a LaTeX version for an Alpha program, i.e. a piece of code that may be included in a Latex document to print Alpha code in LaTeX style.

References

[1] H. Le Verge, V. Van Dongen, and D. Wilde. *Loop Nest Synthesis using the Polyhedral Library.* Technical Report Internal Publication 830, IRISA, Rennes, France, May 1994. Also published as INRIA Research Report 2288.

[2] D. Wilde. *A Library for Doing Polyhedral Operations.* Technical Report Internal Publication 785, IRISA, Rennes, France, Dec 1993. Also published as INRIA Research Report 2157.

[3] D. Wilde. *The* ALPHA *Language.* Technical Report Internal Publication 828, IRISA, Rennes, France, May 1994. Also published as INRIA Research Report.

[4] D. Wilde and S. Rajopadhye. *Allocating memory arrays for polyhedra.* Technical Report Internal Publication 749, IRISA, Rennes, France, July 1993. Also published as INRIA Research Report 2059.