POET Reference Manual

Qing Yi
University of Texas At San Antonio
(qyi@uccs.edu)

March 30, 2017

Front Matter

POET is an interpreted program transformation language designed to apply source-to-source transformations to programs written in arbitrary languages such as C, C++, Java or any domain-specific ad-hoc languages. The POET language has been extensively used for the purpose of applying parameterized compiler optimizations to improve the performance (i.e., the runtime efficiency) of C programs, so that differently optimized code can be automatically generated and empirically tuned. The use of POET, however, is not limited to compiler optimizations. You can use POET to easily process any structured input code, extract information from or apply transformations to the input, and then output the result.

The POET language was designed and implemented by the research group lead by Dr. Qing Yi at the University of Texas at San Antonio during 2007-2012. The project has been moved to the University of Colorado, Colorado Springs since 2012. Please directed all questions and feedbacks to her at qyi@uccs.edu.

Qing Yi
9/1/2011

Contents

Chapter 1

Building and Using POET

1.1 Building POET From Distribution

After downloading a POET distribution, say poet-1.02.06.tar.gz, you can build POET using the following commands.

```
> tar -zxf poet-1.02.07.01.tar.gz
> cd poet-1.02.07.01
> ./configure --prefix=<your install directory>
>  make     (make and then run the POET interpreter on a few tests)
> make check (test whether the POET interpreter works correctly)
> make install (install POET interpreter and libraries on your local machine)
```

1.2 Building POET From SVN repository

If you have access to the POET internal git repository, you can build POET using the following commands.

```
> git clone git.machine://git.directory/POET/trunk
> cd POET
> aclocal
> automake -a
> autoconf
> ./configure
>  make     (make and then run the POET interpreter on a few tests)
> make check (test whether the POET interpreter works correctly)
```

1.3 Directory Structure of POET Source Distribution

The POET interpreter is implemented in C++. The distribution of POET includes the following sub-directories.

- The *src* directory, which contains the C/C++/YACC/LEX code used to implement POET.
- The *lib* directory, which contains libraries implemented using POET.

- The *test* directory, which contains POET scripts used to test POET releases.
- The *examples* directory, which contains examples used in various POET tutorials.
- The *doc* directory which contains the manual and tutorials for POET.

1.4 Using POET

Each POET release includes the language interpreter, named *pcg*, and various POET library files. After running *make install*, the binary interpreter *pcg* is copied to directory <your install directory>/*bin*, and the POET libraries are copied to <your install directory>/*lib*. The command line options for running *pcg* are as follows.

```
Usage: pcg [-bhv] {-L<dir>} {-p<name>=<val>} <poet_file1> ... <poet_filen>
options:
     -h:      print out help info
     -v:      print out version info
     -c<i>:   verify the first i conditions only (exit without evaluating the program)
     -L<dir>: search for POET libraries in the specified directory
     -p<name>=<val>: set POET parameter <name> to have value <val>
     -dp:     print out debugging info for parsing operations
     -dl:     print out debugging info for lexical analysis
     -dx:     print out debugging info for xform routines
     -dm:     print out debugging info for pattern matching
     -md:     allow global names to be multiply-defined (overwritte)
     -dt:     print out timing information for various components of the POET interpreter
     -dy:   print out debugging info from yacc parsing
```

Chapter 2

Building Translators: Getting Started

This chapter goes over some example POET translators in the POET/examples directory.

2.1 Hello World

The following simple POET program (POET/exampls/helloworld.pt) prints out the string "hello world" to standard output.

```
<************* Hello World Script *********>
<output from="hello world"/>
```

NOTE1: All POET comments are either enclosed inside a pair of <* and *>, or from <<* until the end of the current line. Specifically, all strings enclosed within $< *$ and $* >$ will be ignored by the POET interpreter, and all strings following $<< *$ until the end of line will be ignored.

2.2 The Identity Translator

First and foremost, POET is designed to build translators. The easiest kind of translator is an identity translator, which reads the input code from an arbitrary file, does nothing, and then writes the input code to a different file. The following POET code (POET/examples/IdentityTranslator.pt) does exactly this.

```
<********** The Identity Translator ****************>
<parameter inputFile default="" message="input file name" />
<parameter outputFile default="" message="output file name" />

<input from=inputFile annot=0 to=inputCode/>
<output to=outputFile from=inputCode/>
<********** The Identity Translator ****************>
```

NOTE2: Each *parameter* declaration declares a global variable whose value can be modified via command-line options. For example, the identity translator can be invoked using the following command.
> *pcg -pinputFile=myFile1 -poutputFile=myFile2 IdentityTranslator.pt*

The command-line options are optional as long as a default value (declared using the *default* keyward) is given for each parameter.

NOTE3: Each *input* command opens a list of input files and saves the content of the files as the content of a global variable (here the *inputCode* variable). The *annot* = 0 specification ensures that the input files will be read as a sequence of integer/string tokens and all annotations in the file will be ignored. to use when parsing/unparsing the input/output code.

NOTE4: Each *output* command opens an external file and then outputs the content of an expression into the external file.

NOTE5: When the input file name is an empty string, the *input* command will read from the standard input (the user will be prompted to type in the input); when the output file name is an empty string, the *output* command will write to the standard output (the screen).

2.3 The String Translator

In general, after reading some input files, we would like to apply some transformation and then output the transformed code. The following POET program (POET/examples/StringTranslator.pt) serves to substitute a pre-defined set of strings with other strings.

```
<********** The String Translator ****************>
<parameter inputFile default="" message="input file name" />
<parameter outputFile default="" message="output file name" />
<parameter inputString type=(STRING...) default="" message="string to replace" />
<parameter outputString type=(STRING...) default="" message="string to replace with" />

<input from=inputFile annot=0 to=inputCode/>

<eval
return = inputCode;
for ((p_input = inputString,p_output=outputString); p_input != NULL;
      (p_input = TAIL(p_input); p_output=TAIL(p_output)))
   { return = REPLACE(HEAD(p_input), HEAD(p_output), return);}
/>

<output to=outputFile from=return/>
```

NOTE6: The *type* attribute within a parameter declaration ensures that only a value of proper type can be assigned to the parameter. In particular, the type ($STRING$...) specifies a list of strings.

NOTE7: The *eval* command is used to evaluate expressions and statements at the global scope. All POET expressions must be embedded within an eval command to be evaluated at the global scope.

NOTE8: POET supports assignment statements and for loops in a similar fashion as the C language. Because POET is dynamically typed, variables in POET do not need to be declared.

NOTE9: The HEAD and TAIL keywords are operators that extract values from a list (see section 9.6). Specifically, $HEAD$ returns the first element in the list, $TAIL$ returns the tail of elements in the list (excluding the first one). If the operand *list* is actually a single value, then $HEAD(list)$ returns the single value, and $TAIL(list)$ returns $NULL$.

NOTE10: The REPLACE keyword is a built-in operator for systematically applying transformations (replacements) to an input expression.

Transformation Logic: The example loop first initializes two variables, *p_input* and *p_output*, with the values of global variables *inputString* and *outputString* respectively (each variable has a list of strings as content). It then examines whether *p_input* is an empty string. As long as *p_input* is not empty, the body of the for loop is evaluated, which modifies the value of the *return* variable by invoking the built-in $REPLACE$ operator. Each time the $REPLACE$ operator is invoked, it replaces all the occurrences of $HEAD$(*p_input*) with $HEAD$(*p_output*) in the input code contained in the *return* variable, where $HEAD$(*_input*) and $HEAD$(*p_output*) returns the first string contained *p_input* and *p_output* respectively. At the end of each iteration, both *p_input* and *p_output* are modified with the $TAIL$ of their original values.

2.4 Language Translators

Instead of reading an input file as a sequence of strings, we frequently need to discover the syntactical structure of the input file. This is called parsing. For example, the following translators (POET/examples/C2C.pt and C2F.pt) reads and parses the syntax of a C program, and then unparses the code to an external file in either C syntax or Fortran syntax.

```
<*************** C to C Translator *************>
<parameter inputFile type=STRING default="" message="input file name" />
<parameter outputFile type=STRING default="" message="output file name" />

<input from=inputFile syntax="Cfront.code" to=inputCode/>
<* You can add transformations to the inputCode here *>
<output to=outputFile syntax="Cfront.code" from=inputCode/>


<*************** C to Fortran Translator *************>
<parameter inputFile type=STRING default="" message="input file name" />
<parameter outputFile type=STRING default="" message="output file name" />

<input from=inputFile syntax="Cfront.code" to=inputCode/>
<output to=outputFile syntax="C2F.code" from=inputCode/>
```

NOTE11: The *syntax* attribute in the *input* and *output* commands specifies what language syntax to use when parsing/unparsing the input/output code.

NOTE12: The syntax files *Cfront.code* and *C2F.code* are stored in the POET/lib directory. The *Cfront.code* file contains syntax definitions for parsing/unparsing C programs. The *C2F.code* file contains corresponding Fortran syntax for translating C code to Fortran.

2.5 Program Optimizations

Translating between the syntax of different languages is only a small part of what POET can be used for. The language was designed to the parameterization and empirical tuning of compiler optimizations, and a large library of optimizations have been provided to support this purpose. The interface of these library routines is declared in POET/lib/opt.pi, and their implementations are in POET/lib/opt.pt. For more details on how to use POET to support compiler optimizations, see additional POET documentations which are downloadable from the POET web page at `www.cs.uccs.edu/~qyi/poet/docs.php`.

Optimizing And Tuning Scientific Codes Qing Yi. In SCALABLE COMPUTING AND COMMUNICATIONS: THEORY AND PRACTICE. Samee U. Khan, Lizhe Wang, and Albert Y. Zomaya. John Wiley&Sons. 2011.

POET: A Scripting Language For Applying Parameterized Source-to-source Program Transformations. Qing Yi. Software Practice & Experience. Accepted For Publication. 2011. (The article has already been published online since MAY 11, 2011 with DOI: 10.1002/spe.1089).

Chapter 3

Language Overview

3.1 Overview of Concepts

The following briefly outlines the main concepts that will be covered by this manual.

1. *Atomic and compound values* (Chapter 4). POET supports two types of atomic values: *integers and strings*; three types of compound data structures: *tuples, lists, and maps*; and one user-defined data type, *code templates*. It uses a special type, *xform handle*, to support function pointers.

2. *Code templates (Chapter 5).* POET code templates are essentially pointer-based data structures that can be used to build arbitrarily shaped trees and DAGs (directed acyclic graphs). They are used to build the AST (Abstract Syntax Tree) internal representations of different input programs and to support the parsing/unparsing of the input codes. In particular, code templates are used in the parsing phase to recognize the structure of the input code, in the program evaluation phase to represent the internal structure of programs, and in the unparsing phase to output results to external files.

3. *Xform routines (Chapter 6)*, which are functions that each takes a number of input parameters and returns a result. POET xform routines can make recursive invocation of each other, use loops and if-conditionals to iterate over a body of computation, and systematically apply transformations and program analyses to the internal representations of various input codes.

4. *Variables and Assignments (Chapter 7).* POET uses variables to hold values of intermediate evaluation and to adapt behaviors of the POET interpreter. Variable assignments can be used to modify variables to contain different values. However, they cannot modify the internal content of existing compound data structures. For example, variable assignment can modify a variable x to contain a new value, but it cannot modify the content of an old data object contained in x, as this object may be shared by other variables.

5. *Global commands (Chapter 8)*, which are declarations and evaluations at the global scope of POET programs. The collection of global commands are evaluated in the order of their appearance, where each command can use the result of previous evaluations.

6. *Expressions and built-in operations (Chapter 9).* POET provides a large collection of built-in operators to support different types of expressions, which are building blocks of program

evaluation. POET expressions must be embedded within global commands to be evaluated in POET programs.

7. *Statements (Chapter 10)*, which are different from expressions in that they do not have values. POET statements serve to provide support for debugging and control flow such as sequencing of evaluation, conditional evaluation, loops, and early exit from a xform routines.

The POET/lib directory contains a library of routines for applying various compiler optimizations and a collection of code templates which specialize the xform routines for different programming languages such as C. The transformation libraries are typically named using the ".pt" extension, where their header files (which declare only the interfaces of the libraries) using the ".pi" extension. The code template files are typically named using the ".code" extensions.

3.2 Categorization of POET Names

When the POET interpreter sees an identifier, it categorizes it into one of the following kinds.

- Code template names, which are names that have been declared as code templates in global declarations. If a name, say $MyCode$, has not been declared, it must be written as *CODE.MyCode* in order to be parsed by the POET interpreter as a code template name.
- *Xform* routine names, which are names that have been declared as xform routines in previous global declarations. If a name, say *MyRoutine*, has not been declared, it must be written as *XFORM.MyRoutine* to be treated by the POET interpreter as a xform routine name.
- Global variable names, which are names declared in the global scope. When a global variable, say $MyName$, is used within a local scope (e.g., inside a code template or xform routine body), it must be written as $GLOBAL.MyName$ to avoid being treated as a local variable.
- Static or local variable names. In POET, variables don't need to be declared before used. Therefore, unless an identifier has been explicitly declared as a code template name, a xform routine name, or a global variable name, it will be treated as a local variable variable if used within a code template/xform routine and will be treated as a static variable if used in the global scope.

Because a POET program can include multiple files, the ordering of processing different files may impact how the names in POET file are interpreted. To avoid misinterpreting code template, xform routine, and global variable names, these names need to be properly declared before used in each POET file. Alternatively, a proper prefix, $CODE$, *XFORM*, or $GLOBAL$, can be used to qualify the use of each name.

3.3 Components of POET Programs

A POET program is comprised of an arbitrary number of different files, where each file contains a sequence of global declarations the following kinds.

- Include directives, each specifies the name of an external file that should be evaluated before reading the current file. For example, the following directives are used to start the POET-/lib/Cfront.code file.

```
include ExpStmt.incl
include Loops.incl
...
```

All *include* directives must be placed at the start of a POET file, so that all the specified external files are evaluated before reading the current file. If a file name with extension *.pi* is included, a corresponding library file with the same name but with extension *.pt* will loaded and evaluated after processing the current file.

- Xform routine declarations (see Section 6.1), which define global functions that can be invoked to operate on arbitrary input code.
- Global variable declarations(See section 7.4), which define global names that can be accessed across different POET files.
- Code template declarations (see Section 5.1), which define global code template types and their syntax in various source languages.
- Global commands (see Chapter 8) which define what input computations to parse and process, what expressions to evaluate, and what results to output to external files.
- Comments, which are either enclosed inside a pair of <* and *>, or from <<* until the end of the current line. Specifically, all strings enclosed within <* and *> are ignored by the POET interpreter, so are all strings following <<* until the end of line.

Except for the *include* directives, which must be placed at the beginning of a POET file, the other POET global declarations and commands can appear in arbitrary order. The global declarations serve to specify attributes of global names (e.g., global variables, code templates and xform routines). In contrast, the global commands are actual instructions that are evaluated according to the order of their appearance in the POET program.

As explained in Section 3.2, the ordering of processing global declarations may impact how different names used in a POET file are interpreted, e.g., as a code template name or a local variable name. It is important to note that each POET file is first parsed and saved in an internal representation before being evaluated. Therefore, although the global *input* command can include external POET files, the declarations contained in these files are *not* visible to the current POET file being processed. *The only way to make visible the global declarations of other files is to use the include directives at the beginning of a POET file.*

3.4 Notations

The following notations will be used throughout the rest of this manual when using BNF (Backus-Naur Form) to specify the context-free grammar of POET.

- <concept>, which specifies a non-terminal in BNF. Each *concept* has its own syntax and semantics explained elsewhere. Examples of concepts include <exp> (all POET expressions), <type> (all POET type specifiers).
- [*syntax*], which specifies that the appearance of *syntax* (could be any syntax definition) is optional. For example, [$default$ =<exp>] indicates that the definition of the default value (using the $default$ keyword) can be optionally skipped.

- $\{syntax\}$, which specifies that the appearance of *syntax* (could be any syntax definition) can be repeated arbitrary times (include 0 times, which means *syntax* can be optionally skipped). For example $\{, < id >$ [=<exp>]$\}$ indicates that additional variable initializations (separated by ",") can appear arbitrary times.

- $/ * comments * /$, which is not a part of the BNF but is a comment that explains the BNF concept that immediately precedes it.

Because the above notations have given $<, >, [,], \{, \}$ special meanings, these characters are quoted with "" when they are part of the language syntax.

The following notations are used throughout the manual to specify various components of the POET language.

- <id>: all variable names.

- <pos_int>: all positive integer values, e.g., 1,2,3,....

- <type>: all POET type specifications, defined in Section 9.11.

- <pattern>: all POET pattern specifications, defined in Section 9.12.

- <parse_spec>: all POET parsing specifications, defined in Section 5.5.

- <exp>: all POET expressions, defined in Chapter 9.

Chapter 4

Atomic and Compound Data Types

POET supports two types of atomic values: integers and strings; three types of built-in compound data structures: tuples, lists, and associative maps; one user-defined data type: code templates; and one global function type: xform handles. POET does not allow the modification of compound data structures such as tuples, lists, and code templates, which can be used collectively to build internal representations of the input computation. The only compound data structure that can be modified is *associative maps*, which cannot be used inside other compound data structures and therefore do not affect the traversal of program internal representations.

4.1 Atomic Values

POET supports two types of atomic values, integers and strings. It does not support floating point values under the assumption that code transformation and analysis do not need floating point evaluations. Like C, POET uses integers to represent boolean values: the integer value 0 is equivalent to boolean value $false$, and all the other integers are treated as the boolean value *true*. It provides two boolean value macros, *TRUE* and *FALSE*, to denote the corresponding integer values 1 and 0 respectively.

4.1.1 Integers

POET provides built-in support for integer arithmetics ($+$, $-$, $*$, /, %), integer comparison ($<$, $<=$, $>$, $>=$, $==$, $!=$), and boolean operations (!, && and ||). The semantics of these operations are straightforward and follow those of the C language. Except for the $==$ and $!=$ operators, which apply to all types of values, the other arithmetic and comparison operations are defined for integer values only. When evaluating boolean operations, all input values are converted to integers 1 and 0, where empty strings are converted to 0 and all other non-integer values to 1.

4.1.2 Strings

A string value is defined by enclosing the content within a pair of double quotes, e.g., “hello”, “123”. The escaped strings “\n”, “\r” and “\t” have the same meanings as those in C. POET additionally provides a special string, ENDL, to denote line-breaks in the underlying language.

POET treats strings as atomic values and does not allow modifications to the contents of strings. It provides a binary operator $\wedge$ to support string concatenation, e.g., “abc” $\wedge$ 3 $\wedge$ “def” returns “$abc3def$” (note that integer operands are automatically converted to strings before used in the concatenation). The operator $SPLIT$ can be used to split a string into a list of substrings based on a specified separator. For example, $SPLIT(\text{“},\text{”}, \text{“}abd, ade\text{”})$ returns a list of three strings (“*abd*” “,” “*ade*”). It can also be applied to strings contained in compound data structures. For example, $SPLIT(\text{“}+\text{”}, Stmt\#(\text{“}a+b+c\text{”}))$ returns $Stmt\#(\text{“}a\text{”}\ \text{“}+\text{”}\ \text{“}b\text{”}\ \text{“}+\text{”}\ \text{“}c\text{”})$. If the separator is an integer n, the input string is split immediately after the nth character of the string. For example $(SPLIT(1, \text{“}abc\text{”}) = (\text{“}a\text{”}\ \text{“}bc\text{”})$.

4.2 Compound Data Structures

POET provides three built-in types of compound data structures: lists, tuples, and associative maps.

4.2.1 Lists

A POET list is simply a singly linked list of elements and can be composed by simply listing elements together. For example, (a “<=” b) produces a list with three elements, a, “<=”, and b. A special keyword, $NULL$, is used to denote an empty list, which is equivalent to an internally null pointer. The operator :: is provided to dynamically extend an existing list. For example, if b is a list, $a :: b$ inserts a into b so that the value of a becomes the first element of the new list. If b is $NULL$, then the result is a list that contains a single element, a.

Because lists can be dynamically extended, they can contain an arbitrary number of elements at runtime. Two operations are provided to access elements in a list ℓ: $HEAD(\ell)$ (or $car(\ell)$), which returns the first element of ℓ (if ℓ is not a list, it simply returns ℓ); and $TAIL(\ell)$ (or $cdr(\ell)$), which returns the tail of the list (if ℓ is not a list, it returns $NULL$). For example, if $\ell = (a$ “<=d” 3), then $HEAD(\ell)$ (or $car(\ell)$) returns a, $HEAD(TAIL(\ell))$ (or $car(cdr(\ell))$) returns “<=”, and $HEAD(TAIL(TAIL(\ell)))$ (or $car(cdr(cdr(\ell)))$) returns 3; The number of elements in a list may be obtained using the LEN operator. For example, $LEN(123) = 3$.

When being unparsed to external files, the elements within a list are output one after another without any space, e.g, a list (“a” “+” 3) is unparsed as “a+3”.

4.2.2 Tuples

A POET tuple is a finite number of elements and is composed by connecting a predetermined number of elements with commas. For example, (“i” , 0, “m” , 1) produces a tuple t with four elements, “i”,0,“m”, and 1. Each element in a tuple t is accessed via the syntax $t[i]$, where i is the index of the element being accessed (like C, the index starts from 0). For example, if $t = (i, 0, \text{“}m\text{”}, 1)$, then $t[0]$ returns “i”, $t[1]$ returns 0, $t[2]$ returns “m”, and $t[3]$ returns 1. Because all elements within a tuple must be explicitly specified when constructing the tuple, tuples are used to define a finite sequence of values, e.g, the parameters for a code template or a *xform* routines. The LEN operator can be used obtain the size of a tuple. For example, $LEN(1, 3, 4, 5) = 4$.

When being unparsed to an external file, elements within a tuple are output one after another separated by commas.

4.2.3 Associative Maps

POET uses associative maps to associate pairs of arbitrary types of values. To create an map, use the *MAP* operator followed by a tuple of entries enclosed within a pair of {}. For example, MAP{}

creates an empty map, and MAP{1 => 5, 2 => 6} creates a map with two entries that map 1 to 5 and 2 to 6 respectively. The elements within a map can be accessed using the "[]" operator and modified using assignments. For example, if e is an associative key inside the map *amap*, then *amap*[*e*] returns the mapping result; otherwise, the empty string "" is returned. The size of a map *amap* can be obtained using *LEN*(*amap*). All the elements within the map can enumerated using the built-in *foreach* statement. For more details about the *foreach* operation, see Section 10.3.2.

Each associate map is internally implemented using C++ STL maps, and they are the only compound data structure whose internal content can be dynamically modified. The following illustrates how to create and operate on associative maps.

```
amap = MAP{};
amap["abc"] = 3;
amap[4] = "def";
bmap = MAP{1 => 5, 2 => 6};
print ("size  of amap is " LEN(amap));
foreach cur=(from=_, to=_)  \in (amap bmap)  do
    print ("MAPPING " from "=>" to);
enddo
print amap;
print bmap;
```

The output of the above code is

```
size  of amap is  2 .
MAPPING  4 => "def"
MAPPING  "abc" => 3
MAP{4=>"def","abc"=>3}
MAP(1=>5,2=>6}
```

4.3 Code Templates

POET code templates are essentially pointer-based data structures and can be used to build recursive data structures such as dynamically shaped trees and DAGs (directed acyclic graphs). To a certain extent, POET code templates are just like C structs, where each code template name specifies a different type, and each component of the data structure is given a specific field name. POET uses code templates extensively to build the AST (Abstract Syntax Tree) internal representations of programs for transformation or analysis. POET offers integrated support for associating parsing/unparsing specifications with each code template (see Chapter 5), so that input programs can be automatically parsed, converted to their internal code template representations, and unparsed with proper syntax.

The syntax for building a code template is

```
<id> # (<exp> {, <exp>})
```

where <id> is a code template name, and each <exp> specifies a value for each parameter (i.e., each data field) of the code template. For example, $Loop\#(``i", 0, ``N", 1)$ and $Exp\#(``abc/2")$ build objects of the code templates *Loop* and *Exp* respectively. Section 5.1 presents details on how to define different code template types.

To get the values of data fields within a code template object, use syntax

```
<exp>[ <id1> . <id2>]
```

where <exp> is a code template object, <id1> is the name of the code template, and <id2> is the name of a template parameter. For example, *aLoop*[*Loop.i*] returns the value of the *i* data field in *aLoop*, which is an object of the code template type *Loop*. Similarly, *aLoop*[*Loop.step*] returns the value of the *step* field in *aLoop*. If *aLoop* is built via *Loop*#(“*ivar*”, 5, 100, 1), then *aLoop*[*Loop.i*] returns value “ivar”, and *aLoop*[*Loop.step*] returns value 1.

4.4 Xform Handles

Each POET xform handle is a global function pointer. It is similar to a C function pointer except that it includes not only the global name of a routine but also a number of values for the optional parameters of the routine. Similar to C functions, POET xform routines can be defined only at the global scope. So each name uniquely identifies a xform routine. POET xform handles can be used as values of variables, which can then later be invoked by following it with a tuple of actual parameter values. Each xform handle can be defined using the following syntax.

```
<id> [ "["<id>=<exp> { ;.<id>=<exp>} "]" ]
```

Here each <*id*>=<*type*> defines a new value for an optional parameter (name defined by <id>) of the xform routine. Therefore, optional parameters of an xform routine can be given values before the routine is actually invoked. For example, $ParallelizeLoop[trace\_include = x]$ is a function pointer to the $ParallelizeLoop$ routine, where the optional parameter $trace\_include$ of the routine is set to the value of x. Since a xform handle can be used in arbitrary places where an expression is expected and can be invoked at an arbitrary time in the future, an xform handle can be used to pre-configure behaviors of an xform routine before it is invoked. To take advantage of this support, parameters that can be used to reconfigure the behavior of an *xform* routine should always be defined as optional parameters. Only pure input parameters (parameters that define the input of the routine) should be declared as required parameters.

Chapter 5

Code Templates

POET code templates are user-defined data types that can be used to implement arbitrary acyclic data structures. In particular, they are typically used to implement the internal representation of arbitrary input codes. Additionally, they can be associated with syntax specifications that specify how to parse an input computation and how to unparse internal representations to external files.

5.1 Defining Code Templates

The syntax for declaring code templates is

```
  "<" code  <id>  [pars = (<id> [: <parse_spec> ] {, <id> [: <parse_spec]}) ]
                  [parse = <parse_spec>]
                  [lookahead = <pos_int>]
                  [rebuild = <exp>]
                  [match = <type>]
                  [output = <type>]
                  {<id> = <type> } "/>"
```

or

```
  "<" code  <id>  [pars = (<id> [: <parse_spec> ] {, <id> [: <parse_spec]})]
                  [parse = <parse_spec>]
                  [lookahead = <pos_int>]
                  [rebuild = <exp>]
                  [match = <type>]
                  [output = <type>]
                  {<id> = <type> } ">"
<exp>          /* body of code template*/
"</"code">"
```

The first format declares the interface of a code template; i.e., the data type name and components of each data object. The second format additionally defines the associated concrete syntax of the code template; that is, how to build its internal representation from parsing an input file and how to unparse the internal representation to external files.

Each code template name uniquely identifies a user-defined data type, where the template parameters are data fields within the structure. A code template can be declared an arbitrary number of times. However, its concrete syntax definition can be encountered only once during each evaluation of the global parsing/unparsing command (i.e., the *input* and *output*), unless the POET interpreter is invoked with the command-line option $-md$. Except the code template name, all the other components of a code template are optional. The following explains the semantics of each optional component.

5.2 Template Parameters

In the following example,

```
<code Loop pars=(i,start,stop,step)/>
<code If pars=(condition:EXP) >
if (@condition@)
</code>
```

the *pars=...* syntax specifies the required parameters of the code templates. Each template parameter is specified either using a single name or a <id> : <parse_spec> pair, which specifies both the parameter name and how to obtain the parameter value via parsing. For example, as declared above, the code template *Loop* has four required parameters: *i*, *start*, *stop*, and *step*; the code template *If* has a single parameter, *condition*, which is an expression and can be obtained via parsing by matching the input code against the *EXP* type specifier, explained in Section 9.11.

5.3 Template Body

In the following example,

```
<code Loop pars=(i:ID,start:EXP, stop:EXP, step:EXP) >
for (@i@=@start@; @i@<@stop@; @i@+=@step@)
</code>
```

The list of strings between the pair of <code...> and </code> defines the body of the *Loop* code template. Each template body is typically a list of strings in a POET input/output language such as C, C++. POET expressions, e.g., the template parameters *i*, *start*, *stop*, and *stop* in the above example, need to be wrapped inside pairs of @s within the body to be properly evaluated.

By default, POET uses the bodies of code templates as the basis both for constructing internal representations of programs from parsing and for unparsing internal representations to external files. When trying to parse the tokens of an input program against a specific code template, the POET parser first substitutes each template parameter with its declared type in the code template body (if no type is declared for the parameter, the *ANY* type specifier, which can be matched to an arbitrary string, will be used). The substituted template body is then matched against the leading strings of the input. Whenever a code template type within the template body is encountered, the POET interpreter tries to match the program input against the new code template. This strategy essentially builds a recursive-descent parser on-the-fly by interpreting the code template bodies. Since POET uses recursive descent parsing, the code templates cannot be left-recursive; that is, the starting symbol in the template body cannot recursively start with the same code template type. If a left-recursive code template is encountered during parsing evaluation, segmentation fault will occur as is the case for all recursive descent parsers.

5.4 Template Attributes

Each code template can include a number of attributes, each defined using syntax *<id>=<type>*, where <id> is the attribute name, and <type> specifies the default value of the particular attribute. For example, the following code template declaration

```
<code Loop pars=(i,start,stop,step)  maxiternum=""/>
```

specifies a template attribute named *maxiternum*, which remembers the maximal number of iterations that a loop may take.

While users can define arbitrary attribute names for a code template, several keywords, including *parse*, *lookahead*, *rebuild*, *match*, *output*, *unparse*, *rebuild*, *parse*, and *output*, are reserved and are used to specify how to parse, unparse, simplify, and operate on objects of the code template.

5.4.1 The *parse* attribute

When the *parse* attribute is defined, the template body is no longer used to parse a given input and unparse a code template object. Instead, the value of the *parse* attribute is used.

A common use for the *parse* attribute is to specify an alternative xform routine to convert input tokens to code template objects. Such xform routines are called *parsing functions*. Each parsing function must take a single parameter, the *input* token stream to parse, and return a pair of values ($result, leftOver$), where *result* is the result of parsing the leading strings in *input*, and $leftOver$ is the rest of the input token stream to continue parsing. A number of commonly used parsing functions are defined in the *POET/lib/utils*.

5.4.2 The *lookahead* attribute

The POET interpreter uses a recursive descent parser to dynamically match input tokens against the concrete syntax of code templates. Since multiple alternative code templates may be specified to match the input, the POET interpreter uses the leading input tokens to determine which code template to choose. Specifically, if the *lookahead* attribute is given value n for a code template, the next n input tokens is used to determine whether to select the particular code template for parsing. By default, the value of *lookahead* is 1 for all code templates.

5.4.3 The *match* attribute

This attribute specifies a set of parent code template types which can be viewed as a base type of the current code template. During pattern matching operations (see Section 9.12), if the code template fails to match against a given pattern, the POET interpreter uses the value of it's *match* attribute as an alternative target and tries again. For example, the following code template definition

```
<code Loop pars=(i:CODE.Name,start:EXP, stop:EXP, step:EXP) match=CODE.Ctrl />
```

specifies that the *Loop* code template can be considered a derived instance of the *Ctrl* code template and therefore can match the pattern *CODE.Ctrl*.

The *rebuild* attribute This attribute specifies an alternative expression that should be used to substitute the resulting code template object during parsing or when the REBUILD operator (see Section 9.16) is invoked. For example, the following code template definition

```
<code StmtBlock pars=(stmts:StmtList) rebuild=stmts>
{
   @stmts@
}
</code>
```

specifies that after parsing the input computation using the *StmtBlock* code template, the value of the template parameter *stmts* should be returned as result of parsing. The rebuild expression can use both the template parameters and other attributes and can invoke existing xform routines to build the desired result. The result of evaluating the expression will be returned as the result of parsing or the result of the REBUILD operator.

5.4.4 The *output* attribute

This attribute specifies an alternative expression that should be evaluated when a code template object needs to be unparsed to an external file. By default, the template body is used both for parsing and unparsing. However, for some code templates, this may not be the right choice. For example, the following code template definition (taken from *POET/lib/Cfront.code*) invokes the *UnparseStmt* routine to unparse all C statements, where a pair of { } is wrapped around a statement block if more than one statements are unparsed.

```
<code Stmt pars=(content:GLOBAL.STMT_BASE) output=(XFORM.UnparseStmt(content)) >
@content@
</code>
```

5.4.5 The *INHERIT* attribute

This attribute is used to remember the previous code template object constructed by the POET recursive descent parser immediately before the current code template is used to match the input token stream. It can be used within the parse attribute to flexibly satisfy type checking and AST construction needs. For example, the following code template definition (from *POET/lib/Cfront.code*)

```
<code Else pars=(ifNest) parse=("else" eval(return(CODE.Else#INHERIT))) >
else
</code>
```

specifies that when parsing an *else* statement, its *ifNest* member should be initialized with the previous code template object, i.e., the *true* branch of an *if* statement before the *else* branch.

5.5 Parsing Specifications

POET provides a collection of parsing specifiers to guide how to parse a stream of input tokens into a structured hierarchy of code template objects. These parsing specifiers are used inside code template declarations to specify how to parse each template parameter (see Section 5.4), inside global variable declarations to specify how to parse each command-line parameter (see Section 7.4.1), inside expressions to dynamically convert different types of values from one to another (see Section 9.13), and inside input code annotations to improve the parsing efficiency.

A parsing specifier specifies what targeting data structure should be used to parse and represent a sequence of input tokens and can take any of the following forms.

- A type specifier defined in Section 9.11, where the leading input token will be converted to the specified value type as the parsing result. More details on type conversion can be found in Section 9.13.

- The name of a code template. In this case, the leading tokens in the input are matched against the syntax definition of the given code template; if the matching is successful, the matched tokens are converted to an object of the given code template, and parsing continues with the rest of input tokens; otherwise, the parsing fails.

- A xform handle which takes a single parameter (the input tokens to parse) and returns a pair of two values: $(result, left\_over)$, where $result$ contains the resulting structural representation from parsing the leading tokens of input, and $left\_over$ contains the rest of input after parsing. In this case, the xform routine is invoked with input tokens as argument, the $result$ from invoking the routine is saved, and the parsing continues with the $left\_over$ value returned by the routine invocation.

- A list of parsing specifiers in the form (<parse_spec1> <parse_spec2> ...... <parse_specn>). Here the parsing process tries to match each of the parsing specifiers in turn. If the parsing succeeds, a list of the parsing results is constructed, and the parsing continues with the rest of the input.

- A variable assignment in the form <id> = <parse_spec1>. Here the input is parsed against the given <parse_spec1>, and the parsing result is saved in the given variable <id>.

- A *tuple* specification in the form *TUPLE(<string1> <parse_spec1> ...... <string_n> <parse_spec_n> <string_n+1>)*. In this case, the targeting data structure is a tuple of n elements, where the input tokens must start with <string1>, end with the <string_n+1>, and for each i = 1,...,n, the ith tuple element is parsed using <parse_spec_i>, and the ith and $i+1$th elements must be separated by <string_i+1>. For example, $TUPLE($“(”INT“,”INT“,”INT“)”$)$ specifies a tuple of three integers, where the tuple starts with “(” and end with “)”, and each pair of elements is separated with a “,”. The parsing fails if any component does not match.

- A *list* specification in the form *LIST(<parse_spec_1>, <string>)*. Here the targeting data structure is a list of elements, with each element parsed using <parse_spec_1>, and each pair of elements separated by <string> in the input tokens. For example, $LIST(INT,$ “;”$)$ specifies a list of integers separated by “;”s; and *LIST(INT, “ ”)* specifies a list of integers separated by spaces.

- The binary alternative (|) operation in the form <parse_spec_1> | <parse_spec_2>. Here the input tokens are matched against each of the parsing specifiers in order (if the first one fails, the second one will be tried). For example, *INT* | *ID* specifies a single integer or an identifier. Note that once the input is successfully matched against <parse_spec_1>, the parser will not try <parse_spec_2>. So the operands of the | operator need be listed in the increasing order of their restrictiveness.

5.6 Parsing Annotations

Based on a collection of code template definitions, POET can invoke its internal top-down recursive descent parser to dynamically parse an arbitrary input language. The input code can be annotated to speed up the parsing process. Annotations can also be used to partially parse fragments of an input code, in which case only those fragments with annotations are parsed into the desired code template representations, and the rest of the code is represented as a stream of tokens.

The following shows a fragment of input code with parsing annotations (taken from POET/test/gemvATLAS/gemv-T.pt).

```
/*@; BEGIN(gemm=FunctionDecl)@*/
void ATL_USERMM(const int M, const int N, const int K,
          const double alpha, const double *A, const int lda,
          const double *B, const int ldb, const double beta,
          double *C, const int ldc)
{
   int i,j,l;                                                     //@=>gemmDecl=Stmt
   for (j = 0; j < N; j += 1)                                     //@ BEGIN(gemmBody=Nest) BEGIN(nest3=Nest)
   {
      for (i = 0; i < M; i += 1)                                  //@ BEGIN(nest2=Nest)
      {
         C[j*ldc+i] = beta * C[j*ldc+i];
         for (l = 0; l < K; l +=1)                                //@ BEGIN(nest1=Nest)
         {
            C[j*ldc+i] += alpha * A[i*lda+l]*B[j*ldb+l];
         }
     }
   }
}
```

Each POET parsing annotation either starts with "//@" and lasts until the line break, or starts with "/*@" and ends with "@*/". The special syntax allows programmers to naturally treat these annotations as comments in C/C++ code, so that the source input is readily accessible for other uses.

POET supports two kinds of parsing annotations.

- Single-line annotations. A single-line annotation applies to a single line of program source. It has the format "=> T", where T is a parsing specification as defined in Section 5.5. For example, the annotation " int i, j, l; //@=>gemmDecl=Stmt" indicates that "int i, j,l;" is a statement that should be parsed using the *Stmt* code template, and the result should be stored in the global variable *gemmDecl*.

- Multi-line annotations. Multi-line annotations are used to help parse compound language constructs such as functions and loop nests, which may span multiple lines of the input code. Each multi-line annotation has the format "BEGIN(T)" or "; BEGIN(T)" where T is a parsing specification (see Section 5.5). If there is a ";" before the $BEGIN$ annotation, the parsing annotation refers to the source code immediately following the annotation; otherwise, it starts from the left-most position of the current line. For example, the annotation "for (l = 0; l < K; l += 1) //@ BEGIN(nest1=Nest)" includes a multi-line annotation which starts from the *for* loop. However, the "/*@; BEGIN(gemm=FunctionDec) @*/" starts from the following line; that is, it does not include source code at the same line as the annotation. In both cases, the relevant source code in the underlying language is parsed based on the given parsing specification ($nest1 = Nest$ or $gemm = FunctionDecl$), and the parsing results are saved in the respective global variables.

Chapter 6

Xform Routines

POET xform routines are global functions that each takes a number of input parameters and returns a result. They can make recursive function calls of each other, use loops and if-conditionals to operate on the internal representation of input codes, and systematically apply transformations based on pattern matching or different program analysis results. These xform routines are generic in the sense that they can be invoked to operate on the internal code template representation of the input code parsed from arbitrary programming languages and thus can be reused across different input/output languages.

6.1 Xform Routine Declarations

The syntax for defining a *xform* routine is

```
"<" xform  <id>  pars = (<id>[:<type>]{,<id>[:<type>]}) {<id> = <type>} "/>"
```

or

```
"<" xform  <id>  pars = (<id>[:<type>]{,<id>[:<type>]}) {<id> = <type>} ">"
<exp>         /* body of xform routine */
"</"xform">"
```

The first syntax declares the interface of a *xform* routine, while the second one additionally defines the implementation, i.e., the body, of the routine. Here, the first <id> specifies the name of the *xform* routine; *pars=...* specifies the required input parameters of the routine; and each *<id>=<type>* specifies an additional optional parameter (name defined by <id>) and the default value (defined by <type>) of the parameter. When a *xform* routine is invoked, a value must be given for each required parameter (defined using "pars=..."). Additional arguments may be supplied to replace the default values of the optional parameters, but they are not required.

The following shows two example xform routine declarations in POET/lib/utils.incl.

```
<xform ParseList pars=(input) stop="" continue="" output=(result, leftOver)/>
<xform SkipEmpty pars=(input) >
  for (p_input=input;
     p_input != NULL && ((car p_input) : " "|"\n"|"\t");
     p_input=cdr p_input)  {""}
  p_input
</xform>
```

Here the first declaration of *ParseList* specifies that the routine takes a single parameter named *input*, two optional parameters named *stop* and *continue* (both with “” as their default values), and produces a tuple of two values as result. The names in the output specification serve to document the meaning of each value returned by the routine.

6.2 Invoking Xform Routines

The syntax for invoking a *xform* routine is

```
<exp1> ["["<id1>=<exp2>{; <id2>=<exp3>}"]"] ( <exp4> {, <exp5>} )
```

Here <exp1> is an xform handle (e.g., the name of an xform routine) being invoked, <id1> and <id2> are names of optional parameters of the xform routine, and <exp2>,...,<exp5> are values for the xform routine parameters. In particular, the list of expressions inside the pair of () are values for the required parameters, while values for optional parameters are assigned to their respective parameter names. For example, the following invocation

```
ParallelizeLoop[threads=par;private=nest1_private](nest1)
```

invokes a xform routine *ParallelizeLoop* with actual parameter *nest*1 while assigning the optional parameter *threads* with the value of *nest*1, and the optional parameter *private* with *nest*1*_private*.

Chapter 7

Categorization Of Variables

POET variables serve as place holders that can store arbitrary types of values at runtime. In particular, they can be categorized into the following different kinds.

1. *Local variables*, whose lifetime span through the body of a single code template or a single xform routine definition.
2. *Static variables*, whose lifetime span the entire program but can be accessed only within a single POET file; i.e., their scope is restrained within a single file.
3. *Dynamic variables*, whose lifetime span the entire program, and they can be dynamically created and operated on at any point in the program.
4. *Global variables*, whose lifetime span the entire POET program and can be accessed at any point across different POET files.

Each category is maintained in a separate symbol table distinct from the other categories.

7.1 Local Variables

The lifetime and scope of each local variable is restricted within a single code template or xform routine. Local variables are introduced by declaring them as parameters of a code template or xform routine, or by simply using them in the body of a code template or xform routine. For example, in the following,

```
<code Loop pars=(i:ID,start:EXP, stop:EXP, step:EXP) >
for (@i@=@start@; @i@<@stop@; @i@+=@step@)
</code>
```

the variables *i*, *start*, *stop*, and *step* are local variables of the code template *Loop*. In the following,

```
<xform SkipEmpty pars=(input) >
  for (p_input=input;
     p_input != NULL && ((car p_input) : " "|"\n"|"\t");
     p_input=cdr p_input)  {""}
  p_input
</xform>
```

the variables *input* and *p_input* are local variables of the xform routine *SkipEmpty*.

Code template or *xform* routine parameters are given values when the respective code template is being used to build an object or when the respective xform routine is invoked to operate on some input. Other local variables are entirely contained within code templates or xform routines and are invisible to the outside. A storage is created for each local variable when a code template is used in parsing/unparsing or when an *xform* routine is invoked, and the storage goes away when the parsing/unparsing or the routine invocation finishes. None of the storages is visible outside the respective code template or *xform* routine.

7.2 Static Variables

Each POET file can have its own collection of static variables, which are used by its commands (see Chapter 8) to store temporary results and to propagate information across different evaluation commands. While the lifetime of these static variables span the entire program, their scopes are constrained only within a single POET file avoid naming conflict across different POET files. Static variables do not need to be declared before used.

7.3 Dynamic Variables

These are variables dynamically created on the fly by converting an arbitrary string to a variable name (for more details, see Section 9.13). Dynamic variables are provided mainly to support dynamic pattern matching. For example, a list of dynamic variables can be created to replace all the integers in an unknown expression. The substituted expression can then be used as a pattern to match against other expressions that have a similar structure. Because all dynamic variables are created in a single symbol table throughout a POET program, naming collision can easily occur. Therefore it is strongly discouraged to use dynamic variables for purposes other than dynamic pattern matching, e.g., using dynamic variables as a way for implicit parameter passing is considered very dangerous.

7.4 Global Variables

These are variables whose lifetime span an entire POET program and can be accessed across different POET files. However, each global variable must be explicitly declared in each POET file before being used. There are three categories of global variables.

1. *Command-line parameters*, which are global variables whose values can be redefined via command-line options.

2. *Macros*, which are global variables that can be used to reconfigure the behavior of the POET interpreter or the POET program being interpreted.

3. *Tracing handles*, which are global variables that can be embedded inside the internal representation of computations to keep track of selected fragments as they go through different transformations.

7.4.1 Command-Line Parameters

POET command-line parameters act as the configuration interface of a POET program and their initial values can be specified via command-line options. POET command-line parameters must be declared before used. Once defined, they can be directly accessed across different POET files. To declare a command-line parameter, use the following syntax.

```
"<"parameter <id> type=<type> parse=<parse_spec> default=<exp> message=<string>"/>"
```

Here <id> specifies the name of the variable; <type> specifies the type of its value; <parse_spec> specifies how to obtain the value of the variable from parsing command-line options of invoking the POET interpreter; <exp> specifies the default value of the parameter when the command-line option does not specify an alternative value; and <string> is a string literal that documents the meaning of the parameter in the declaration. The following shows several example command-line parameter declarations.

```
<parameter NB type=1.._ default=62 message="Blocking size of the matrices" />
<parameter pre type="s"|"d" default="d"
           message="Whether to compute at single- or double- precision" />
```

The *NB* parameter is a single integer that must be greater than or equal to 1 (the special value _ is used to denote an unknown upper bound). Its default value is 62 if not reset by command-line options. The *pre* parameter can have two alternative values. Here the type of the parameter uses the | operator to enumerate all the possible values ("s" or "d"). The default value of the above *pre* parameter is string "d".

The command line option to assign a new value to a POET command-line parameter is $-p$<id>=<val>, where <id> is the name of the variable, and <val> is a string that defines the value of the parameter. If necessary, the given parameter value will be parsed, and the parsing result checked against the given <parse_spec>, before the final result is assigned as value of the parameter. For example, $-pNB$=50 assigns 50 the new value of the NB parameter.

7.4.2 Macros

POET macros can be used to reconfigure the behavior of the POET interpreter or the POET program being interpreted. The syntax for defining a macro is:

```
"<" define  <id> <exp> "/>"
```

Here <id> is the name of the macro, and <exp> is a POET expression which defines the value of the macro. The following shows some example macro declarations.

```
<define myVar1 "abc"/>
<define x  5 />
```

The above macro variables are all user-defined names that do not have any special meaning to POET. However, POET does provide some built-in macros that have special meanings and can be used to modify the default behavior of the POET interpreter, as discussed in Section 7.5.

7.4.3 Tracing Handles

POET tracing handles can be viewed both as global variables (i.e., they can be modified via variable assignments) and as a special data type as they can be embedded inside other data structures to trace transformations to various components of the data structure. In particular, as various transformations are applied to a compound data structure, the tracing handles embedded within can be automatically modified to always contain the most up-to-date values of the respective components, so that different transformations can be applied independently of each other irrespective of how many other transformations have been applied.

Tracing handles need to be explicitly declared in each POET file before being used. The syntax for declaring tracing handles is:

```
 "<" trace  <id>[=<exp>]  {,  <id>=<exp> }  "/>"
```

For example, the following declares a long sequence of global trace handles.

```
<trace gemm,gemmDecl,gemmBody,nest3,nest2,nest1/>
```

NOTE that when a sequence of tracing handles are declared in a single declaration, as illustrated above, these handles are assumed to be related, and their ordering in the declaration is assumed to be the same ordering that they should appear in a pre-order traversal of the data structure that they are embedded inside. Therefore only related tracing handles should be declared in a single declaration, and unrelated handles should be declared separately to avoid confusion.

7.5 Reconfiguring POET via Macros

POET provides a number of built-in macros to modify the default behavior of the POET interpreter, specifically the behavior of the internal lexer, parser, and unparser when used to parse input files and to unparse results to external output files.

7.5.1 The TOKEN Macro

This macro reconfigures the POET internal lexer (tokenizer) when reading files using the global *input* command (see Section 8.1). For example, the following declaration appears in the Cfront.code (the C language syntax) file in the POET/lib directory.

```
<define TOKEN (("+" "+") ("-" "-") ("=""=") ("<""=") (">""=") ("!""=")
               ("+""=") ("-""=") ("&""&") ("|""|") ("-"">") ("*""/")
               CODE.FLOAT CODE.Char CODE.String)/>
```

This declaration configures the POET interpreter to replace every pair of "$+$" "+" into a single "$++$" token, every pair of "$-$" "$-$" into "$--$", and so forth. Additionally, the tokenizer will also recognize the syntax of code templates $FLOAT$, $Char$, and $String$ as tokens.

7.5.2 The KEYWORD Macro

This macro reconfigures the POET internal recursive descent parser when reading files using the global *input* command (see Section 8.1). For example, the following declaration appears in the Cfront.code (the C language syntax) file in the POET/lib directory.

```
<define KEYWORDS ("float" "int" "unsigned" "long" "char" "struct" "union"
"extern" "static" "const" "register" "if" "else" "switch" "case" "default"
"continue" "break" "for" "while" "case")/>
```

This declaration configures the POET internal parser to treat strings "case", "for", "if", "while", etc. as reserved words of the language, so that these strings are not treated as regular identifiers; i.e., they cannot be matched against the ID or STRING token type during parsing.

7.5.3 The PREP Macro

This macro reconfigures the POET internal recursive descent parser when reading files using the global *input* command (see Section 8.1). For example, the following declaration appears in the Ffront.code (the Fortran language syntax) file in the POET/lib directory.

```
<define PREP ParseLine[comment_col=7;text_len=70] />
```

This declaration configures the POET internal parser to invoke the *ParseLine* routine as a filter of the token stream before starting the parsing process. In particular, the *ParseLine* routine filters out meaningless characters based on their column locations. For example, only characters appearing between column 7-79 are meaningful characters, and an entire line should be skipped if the comment-column is not empty.

7.5.4 The BACKTRACK Macro

This macro can be used to disable backtracking in the POET's internal parser when reading files using the global *input* command (see Section 8.1). The following declaration from POET-/lib/Cfront.code (the C language syntax) accomplishes exactly this task.

```
<define BACKTRACK TRUE/>
```

Note that when backtracking is enabled. By default, the POET internal parser uses the first token of each code template body to determine which code template to use when multiple options exist to parse an input code. When multiple code templates start with the same token, only the first one will be tried. If the syntax of the chosen code template fails to match the input tokens, the entire parsing process fails. As a result, by disabling backtracking, the parsing process becomes faster (because only one code template will be tried when multiple choices are available) but also more restrictive (the parsing fails immediately instead of trying out other options). Similarly, by enabling backtracking, the parsing is slower, but more likely to succeed.

7.5.5 The PARSE Macro

This macro specifies the top-level start non-terminal used to match the entire input code by the POET's internal parser when reading files using the global *input* command (see Section 8.1). For example, the following declaration appears in the Cfront.code (the C language syntax) file in the POET/lib directory.

```
<define PARSE CODE.DeclStmtList/>
```

This declaration informs the POET interpreter that unless otherwise specified in the input command, the POET internal parser should use the code template *DeclStmtList* as the start non-terminal when parsing program input.

7.5.6 The UNPARSE macro

This macro specifies an additional post-processor of the POET internal unparser when evaluating the *output* command (see Section 8.4). For example, the following declaration appears in the Ffront.code (the Fortran language specialization) file in the POET/lib directory.

```
<define UNPARSE XFORM.UnparseLine/>
```

This declaration instructs the POET interpreter to invoke the *UnparseLine* routine (defined in the POET/lib/utils.incl file) after the POET unparser has produced a token stream to output to an external file. The *UnparseLine* routine takes two parameters: the tokens to output, and the location (column number) of the current line in the external file. It will be invoked with the correct parameters when each token needs to be output to the external file. The *UnparseLine* routine will filter the token stream by inserting line breaks and empty spaces as required by the column formatting requirements of the source language (e.g., Fortran or Cobol).

The UNPARSE macro can also be redefined to contain a code template as value. For example, the following definition appears in POET/lib/Cfront.code (the C language syntax file).

```
<define UNPARSE CODE.DeclStmtList/>
```

Here the code to output will be treated as an object of the *DeclStmtList* code template (which inserts line breaks between different statements) before being unparsed to external files.

7.5.7 The Expression Macros

POET provides built-in support for parsing expressions so that developers only need to use the *EXP* parsing specifier to convert a sequence of tokens to an expression. Several macros are provided to define the accepted terms and operations within an expression, including

1. EXP_BASE, which defines all the base terms accepted within an expression.
2. EXP_BOP, which defines all the binary operators (in decreasing order of precedence) accepted within an expression.
3. EXP_UOP, which defines all the unary operators (in decreasing order of precedence) accepted within an expression.

For example, the following declarations are used in POET/lib/Cfront.code (the C syntax file).

```
<define EXP_BASE INT|FLOAT|String|Char|CODE.VarRef />
<define EXP_BOP ( ("=" "+=" "-=" "*=" "/=" "%=") ("&" "|") ("&&" "||")
             ("==" ">=" "<=" "!=" ">" "<") ("+" "-") ("*" "%" "/") ("." "->")) />
<define EXP_UOP ("++" "*" "&" "~" "!" "+" "-" "new")/>
```

The following additional macros are provided to specify how to construct internal representations of the parsed expressions.

1. EXP_CALL, which defines the code template to use to represent a function call within the expression.
2. EXP_ARRAY, which defines the code template to use to represent an array access within the expression.

3. EXP_MATCH, which defines all the code template types that can be considered instances of expressions when performing pattern matching.
4. PARSE_BOP, which defines the code template to use to represent binary operations within the expression.
5. PARSE_UOP, which defines the code template to use to represent unary operations within the expression.
6. BUILD_BOP, which defines the xform routine to invoke to rebuild binary operations within an expression.
7. BUILD_UOP, which defines the xform routine to invoke to rebuild unary operations within an expression.

For example, the following declarations are contained in POET/lib/ExpStmt.incl, which is included by the Cfront.code file.

```
<define PARSE_CALL FunctionCall/>
<define PARSE_ARRAY ArrayAccess/>
<define PARSE_BOP Bop/>
<define PARSE_UOP Uop/>
<define BUILD_BOP BuildBop/>
<define BUILD_UOP BuildUop/>
```

Note that if defined, these macros are also used by the POET interpreter when parsing expressions contained in POET scripts. Specifically, if the *PARSE_BOP* is defined, the expression $Bop\#(``abc", 3)$ will be returned as the result of evaluating a POET expression $``abc" + 3$.

Chapter 8

Top-Level Commands

At the top level, a POET program can include four different kinds of executable commands, input, condition, evaluation, and output commands. which serve to parse the input code, check the validity of input parameters, compute the output results, and unparse the results to external files. Each command is evaluated one after another according to their order of appearance in each POET file.

8.1 The Input Command

The syntax of the POET input command is

```
"<" input [cond=<exp>] [DEBUG=<int>] [syntax=<exp>] [from=<exp>] [to=<exp>]
          [annot=<exp>] [parse=<parse_spec>] "/>"
```

or

```
"<" input [cond=<exp>] [DEBUG=<int>] [syntax=<exp>] [from=<exp>] [to=<exp>]
          [annot=<exp>] [parse=<parse_spec>] ">"
   <exp>
</input>
```

Each POET input command specifies a number of input code to be parsed and processed. The first form specifies that the input code should be obtained by parsing external files, while the second form includes the input computation (specified by <exp>) to be parsed inside the body of the input command. The following describes the semantics of each attribute within the *input* command.

- *cond* =<exp> specifies a pre-condition that must be satisfied before reading the input code (if the *cond* expression evaluates to false, no input will be read); In particular, it enforces that the *input* command is evaluated only when *exp* is evaluated to true.

- *DEBUG* =<int> specifies that the parsing of input code needs to be debugged at a given level (defined by the constant integer). In particular, the higher the level is, the more debugging information is output.

- *syntax* =<exp> specifies a list of POET file names that contain syntax definitions for code templates required to parse the input; A single file name instead of a list of names can also be used.

- *from* =<exp> specifies a list of external file names that collectively contain the input code. A single file name instead of a list of names can also be used.
- *to* =<exp> specifies the name of a global variable that should be used to stored the parsed input code. If a special keyword, "*POET*", is used in place of a variable name, the input code will be stored as part of the current POET program.
- *annot* =<exp> specifies whether or not to recognize POET annotations in the input code. By default, the *annot* has a true value, and the parsing process will recognize and interpret POET annotations within the input code; if *annot* is explicitly defined to be false, then POET annotations (if there is any) will be treated as part of the input code. For details of POET parsing annotations, see Section 5.6.
- *parse* =<parse_spec> specifies what start non-terminal (i.e., the start code template) to use to parse the input computation. When left unspecified, the value of the macro *PARSE* is used as the parsing target. If specified using a special keyword, "*POET*", the input code should be parsed as a POET program. For more details on parsing specifications, see Section 5.5.

The following are two example input commands which read input codes file external files.

```
<input from=inputFile to=inputCode syntax=(inputLang)/>
<input from=xformFile cond=(xformFile!="") parse=POET />
```

In the first example, the file name that contains the input code is contained in a global variable *inputFile*, the file name that contains syntax definitions of the input language is contained in a global variable *inputLang*, and the parsing result will be saved as the value of global variable *inputCode*. In the second example, the file named contained in $xformFile$ is parsed iff $xformFile! =$ "", and the parsing result will be parsed as a POET program.

8.2 The Condition Command

The syntax of the POET condition command is

```
"<" cond <exp> "/>"
```

This command evaluates the value of the boolean expression <exp>, which serves to declare constraints among values of global variables that must be true for the POET program to be correct. The program continues as usual if <exp> evaluates to *true* and abort the whole evaluation with an error message otherwise.

8.3 The Evaluation Command

The syntax of the POET evaluation command is

```
"<" eval <exp>  "/>"
```

This command triggers <exp>, which is a POET expression or a sequence of statements defined in chapters 9 and 10, to be evaluated. **All POET expressions and statements must be embedded within an eval command to be evaluated at the global scope.**

8.4 The Output Command

The syntax for the POET output command is

```
 "<" output  [cond=<exp>] [syntax=<exp>] [from=<exp>] [to=<exp>] "/>"
```

Each POET *output* command unparses the internal representation of a computation to an external file or standard output. Here each <exp> represents a POET expression. The following defines the semantics of each attribute specification.

- *cond* =<exp> specifies a pre-condition that must be satisfied before performing the unparsing task (if the *cond* expression evaluates to false, no result will be unparsed);
- *syntax* =<exp> specifies a list of POET file names that contain syntax definitions required to unparse the result; A single file name instead of a list of names can also be used.
- $from$ =<exp> specifies the resulting computation that should be unparsed to an external file (or *stdout*).
- *to* =<exp> specifies the name of an external file to output the specified computation.

The following are two examples demonstrating the use of the output command.

```
<output cond=(inputLang=="") to=outputFile from=inputCode/>
<output cond=(inputLang!="") syntax=inputLang to=outputFile from=inputCode/>
```

Here the code contained in *inputCode* is unparsed to the file whose name is defined by $outputFile$. The file names that contain the syntax definitions of the output language are contained in the global variable *inputLang*. If an empty string is specified as the output name (or when no name is specified), the resulting computation will be unparsed to standard output.

Chapter 9

Expressions

POET expressions are the building blocks of all evaluations and could take any of the following forms.

- Atomic values, which include integers and strings. POET use integers to represent boolean values.
- Compound data structures, which include lists, tuples, associative maps, and code template objects.
- *Xform* handles, which are similar to global function pointers in C.
- Variables, which are place holders for arbitrary types of values.
- Invocation of POET *xform* routines, which are essentially calls to global functions.
- Invocation of built-in POET operators, including both arithmetic operations and other operations provided to support efficient code transformation.

Except for variables and associative maps, none of the other POET compound data structures can be modified. Transformations are performed by constructing new values to replace the old ones. POET provides a large collection of built-in operators, each of which takes a number of input values, performs some internal evaluation, and returns a new value as result. These operations can be separated into the following categories.

9.1 Debugging Operations

POET provides the following operations to support debugging and error reporting. These operations produce side effects by printing out information and exiting the program if necessary.

9.1.1 The PRINT (print) operator

The syntax of invoking the PRINT operator is

```
PRINT <exp>
```

or

```
print <exp>
```

The PRINT or print operator takes an arbitrary expression <exp>, prints out the value of <exp> to standard error, and returns an empty string "" as result. It can be used to print out the value of an arbitrary expression for debugging purposes. The following shows some examples of using the *PRINT (print)* operator.

```
print("x=" x);
PRINT("Warning: cannot resolve " cur_sub "-" cpstart
          ": permuteDim=" permuteDim "; left_offset = " left_offset);
```

9.1.2 The DEBUG operator

The syntax of invoking the DEBUG operator is

```
DEBUG ["["<int>"]"] "{" <exp> "}"
```

The DEBUG operator takes an arbitrary expression <exp> and prints out debugging information for evaluating <exp> to standard error and then returns the result of evaluating <exp>. The <int> is used to control how many levels of *xform* routine invocations to debug. By default, <int> is one, which means *xform* invocations will be treated as a built-in operators in debugging. If <int> is 2, then the debugger will step into each *xform* invocation once. The following shows some examples of using the *DEBUG* operator.

```
DEBUG {x = 56;}
DEBUG [3] {UnrollLoops(inputCode)}
```

9.1.3 The ERROR Operator

The syntax of invoking the ERROR operator is

```
ERROR  <exp>
```

The ERROR operator takes an arbitrary expression <exp> and prints out the value of <exp> as an error message to inform the user what has gone wrong before quitting the entire POET evaluation. A line number and the file name that contains the ERROR invocation are also printed out to inform the location that the error has occurred. The ERROR operator therefore should be invoked only when an erroneous situation has occurred and the POET program needs to exit. The following shows some examples of using ERROR operator.

```
ERROR( "Expecting input to be a sequence: " input);
ERROR( "Cannot fuse different loops: " curLoop " and " pivotLoop);
```

9.2 Generic comparison of values

9.2.1 The == and != operators

The syntax of invoking these binary operators are

```
<exp1> == <exp2>
<exp1> != <exp2>
```

Both operators take two expressions, <exp1> and <exp2>, and return a boolean (integer) value indicating whether the two operands are equal or not equal respectively.

9.2.2 Integer and String comparison

Four binary operators, including <, <=, >, and >=, are provided to support the partial ordering of integer and string values.

9.3 Integer arithmetics

POET integer arithmetic operations include the following binary operators: +, -, *, /, %, +=, -=, *=, %=; and a single unary operator: -. These operators have the same meanings as those in C.

9.4 Boolean operations

POET provides two binary operators, && and ||, to support the conjunction and disjunction of boolean values respectively. It provides a single unary operator, !, to support the inversion of boolean values.

9.5 String operations

9.5.1 The concatenation $\wedge$ operator

It applies to two operands, each of which is a string, an integer, or a list of strings and integers, and compose the operands into a single string. For example, "abc" $\wedge$ 3 $\wedge$ "def" returns "$abc3def$". Note that integer operands are automatically converted to strings when used in the concatenation. The $\wedge$ operator can also be applied to a list of strings, e.g. ("abc" "def") $\wedge$ 3 returns "abcdef3".

9.5.2 The $SPLIT$ operator

The syntax of invoking the $SPLIT$ operator is

```
SPLIT "("<exp1> ","  <exp2>")"
```

Here <exp2> is an arbitrary input expression that may contain strings, and <exp1> is either a string that specifies the separator that should be used to split the strings in <exp2>, or an integer that specifies how many characters to count before splitting <exp2> into two substrings. If <exp1> is an empty string, the POET internal tokenizer (lexical analyzer) will be invoked to split the given string. The operation returns a list of substrings obtained from splitting <exp2>. For example,

```
SPLIT(1,"abc")        <<* result is "a" "bc"
SPLIT(",","bc,ade,lkd")      <<* result is "bc" "," "ade" "," "lkd" NULL
SPLIT("", "3,7+5")    <<* result is "3" "," 7 "+" 5 NULL
```

9.6 List operations

9.6.1 List construction

List is the most commonly used data structure in POET. Building a list simply requires that the components be placed together one after another. For example, *(1 2 3)* builds a list that contains three elements: 1, 2, and 3.

9.6.2 The Cons Operator (::)

The syntax of invoking the :: operator is

```
<exp1> :: <exp2>
```

Here the operator returns a new list that inserts <exp1> before all elements in <exp2>. For example, " <= " :: *b* produces a list with "<=" and elements from b. If *b* is *NULL* (the empty list), the result is a list that contains a single element; If *b* is a list, the result contains all elements in b; otherwise, the resulting list contains two elements, " <= " and *b*.

9.6.3 List Access (The *car/HEAD*, *cdr/TAIL*, and *LEN* operators)

Elements in a list are accessed through two unary operators: *car* (also known as *HEAD*) and *cdr* (also known as *TAIL*). The keywords *car* and *HEAD* can be used interchangeably, so can *cdr* and *TAIL*. The syntax of invoking the *car* and *cdr* operators are

```
car <exp>
cdr <exp>
```

Here the *car* operator returns the first element of the list <exp>; if <exp> is not a list, it simply returns <exp>. The *cdr* operator returns the tail of the list; if <exp> is not a list, it returns *NULL*. The following are some examples illustrating the use of these operators.

```
a1 = (3 4 5)     <<* returns 3 4 5 NULL
a2 = (1 2 a1)     <<* returns 1 2 (3 4 5 NULL) NULL
a3 = (1 2) :: a1  <<* returns (1 2 NULL) 3 4 5 NULL
a4 = 1 :: 2 :: a1 <<* returns 1 2 3 4 5 NULL
HEAD(a2)         <<* returns 1
TAIL(a2)         <<* returns 2 (3 4 5 NULL) NULL
HEAD(a3)         <<* returns 1 2 NULL
TAIL(a3)         <<* returns 3 4 5 NULL
```

The syntax of invoking the *LEN* operator is

```
 LEN <exp>
```

When given a list as operand, the *LEN* operator returns the number of elements within the list. For example, *LEN(2 3 7)* returns 3, *LEN(2 7 "abc" "")* returns 4.

9.7 Tuple operations

9.7.1 Tuple Construction (the "," operator)

A tuples is composed by connecting a predetermined number of elements with commas. For example, "i" , 0, "m" , 1 produces a tuple t with four elements, "i", 0, "m", and 1. All elements within a tuple must be explicitly specified when constructing the tuple, so tuples cannot be built dynamically (e.g., using a loop).

9.7.2 Tuple Access (The [] and LEN operators)

Tuples provide random indexed access to their elements. Each element in a tuple t is accessed by invoking $t[i]$, where i is the index of the element being accessed (like C, the index starts from 0). For example, if $t = (i, 0, \text{“}m\text{”}, 1)$, then $t[0]$ returns “i”, $t[1]$ returns 0, $t[2]$ returns “m”, and $t[3]$ returns 1.

When given a tuple as operand, the LEN operator returns the number of elements within the tuple. For example, $LEN(2, 3, 7) = 3$, $LEN(2, (\text{“} < \text{”}3), 4) = 3$.

9.8 Associative Map Operations

9.8.1 Map Construction (the MAP Operator)

POET uses associative maps to associate pairs of arbitrary types of values. The syntax of building an associative map using the MAP operator is the following.

```
MAP "(" <type1> , <type2> ")"
```

or

```
MAP "{" [ <exp1> "=>" <exp2> {"," <exp3> "=> <exp4>} ] "}"
```

Here <type1> and <type2> are two type specifiers as defined in Section 9.11, and <exp1>,...,<exp4> are expressions. The first form returns a new empty table that maps values of *type*1 to values of *type*2, while the second form returns a new table with a number of pre-specified entries mapping <exp1> to <exp2>, etc.

9.8.2 Map Access (the [] and LEN operators)

The elements within an associative map can be accessed using the “[]” operator and modified using the assignment operator (i.e., =). The following illustrates how to create and operate on maps.

```
amap=MAP(_,_);
bmap=MAP{"abc"=>3,"def"=>4};
cmap=MAP{};
amap["abc"] = 3;
amap[4] = "def";
abc = amap["abc"];

LEN(amap) ;
foreach cur=(from=_,to=_) \in amap do
    (from) "=>" (to);
enddo
```

If a value e is stored as a key in a map *amap*, then *amap*[*e*] returns the value associated with e in *amap*; otherwise, an empty string is returned. The size of a map *amap* can be obtained using $LEN(amap)$. All the elements within a map can enumerated using the built-in $foreach$ statement. For more details about the $foreach$ statement, see Section 10.3.2.

9.9 Code Template Operations

9.9.1 Object Construction (the # operator)

POET treats each code template name as a unique user-defined type, where the template parameters are treated as data fields within the structure. To build an object of a code template, use the following syntax.

```
<id> "#" (<exp1>, <exp2>, ..., <expn>)
```

where <id> is the code template name. For example, *Loop*#("*i*", 0, "*N*", 1) and *Exp*#("*abc*/2") build objects of two code templates named *Loop* and *Exp* respectively.

9.9.2 Code Template Access (the [] operator)

To get the values of individual data fields stored in a code template object, use syntax

```
<exp>[ <id1> . <id2>]
```

where <exp> is a code template object, <id1> is the name of the code template type, and <id2> is the name of the data field (i.e., the name of the code template parameter) to be accessed. For example, *aLoop*[*Loop.i*] returns the value of the *i* data field in *aLoop*, which is a variable that contains a *Loop* code template object. Similarly, *aLoop*[*Loop.step*] returns the value of the *step* field in *aLoop*. If *aLoop* contains value *Loop*#("*ivar*", 5, 100, 1), then *aLoop*[*Loop.i*] returns value "ivar", and *aLoop*[*Loop.step*] returns value 1.

9.10 Variable Operations

9.10.1 Variable Assignment (the "=" operator)

All POET variables can be modified via the assignment operator using the following syntax.

```
 <lhs>  = <exp>
```

Here <exp> is an arbitrary expression, and <lhs> has one of the following forms.

- A single variable name. In this case, the value of the <exp> is assigned to <lhs>.
- A compound data structure (e.g., a list, tuple, or code template) that contains variables as components. In this case, the value of the <exp> is matched against the structure of the <lhs>, and all the variables within <lhs> are assigned with the necessary values to make the matching successful. If the value of <exp> fails to match the structure in <lhs>, the evaluation exits with an error.

For example, after the following two assignments,

```
     a = Stmt#input;
    Stmt#a = a;
```

the variable *a* should have the same value as *input*.

9.10.2 Un-initializing Variables (The CLEAR operator)

The syntax of invoking the *CLEAR* operator is

```
CLEAR <id>
```

where <id> is the name of a variable. The operation clears the value contained in v so that v becomes uninitialized after the operation. The *CLEAR* operator is provided to support pattern matching, where un-initialized variables are treated as place holders that can be matched against arbitrary components of a compound data structure (see Section 9.12).

9.11 Type Expressions

Since POET is a dynamically typed language, the types of variables often need to be dynamically checked to determine what operations could be applied to them. POET provides a collection of type specifiers, one for each type of atomic and compound values, to allow the types of expressions to be dynamically tested using pattern matching. Each type specifier can take any of the following forms.

- The ANY (_) Specifier, which uses a single underscore (_) character to denote the universal type that includes all values supported by POET.
- A code template name, which includes all objects of the particular code template.
- The *INT* specifier, which specifies the atomic integer type and includes all integer values.
- The *STRING* specifier, which specifies the atomic string type and includes all string values.
- The *ID* specifier, which specifies the identifier type and includes all string values that start with a letter (‘A’-‘Z’,‘a’-‘z’) or the underscore (‘_’) character and are composed of letters, the underscore (‘_’) character, and integer digits.
- The *TUPLE* Specifier, which specifies the tuple type and includes all POET tuple.
- The *MAP* Specifier, which denotes the associative map type and includes all POET maps. It can optionally take two parameters to indicate the types of element pairs within the map. Specifically, $MAP(fromType, toType)$ includes all associative maps that associate values of $fromType$ to values of $toType$.
- The *CODE* Specifier, which specifies the code template type and includes all code template objects.
- The *XFORM* Specifier, which specifies the xform type and includes all *xform* handles.
- The *VAR* specifier, which specifies the *variable* type and includes all tracing handles as values.
- The *EXP* Specifier, which specifies the expression type and includes all POET expressions.
- The range Specifier *lb..ub*, where *lb* and *ub* are integer values or the *ANY* (_) specifier, is a *range* type and includes all integers >= *lb* and <= *ub*. If the *ANY* (_) specifier is used as the lower or upper bound, it indicates an infinity bound.

- (<type_1>,<type_2>,......,<type_n>), which specifies a tuple of n elements, where the type of the ith element (i = 1,...,n) is specified by <type_i>.
- (<type_1> <type_2> ...... <type_n>), which specifies a list of n elements, where the type of the ith element (i = 1,...,n) is specified by <type_i>.
- <id> # <type>, which specifies a code template type with <id> as the code template name and <type> as types of template parameters.
- <type_1> :: <type_2>, which specifies a list type with the type of the first element specified by <type_1> and the rest of the list specified by <type_2>.
- <type_1>..., which specifies the *list* type with <type_1> as the type of all elements within the list. As a special case, (_....) specifies lists that may contain arbitrary elements. Note that an empty list is a value of the <type_1>... type.
- <type_1>...., which is identical to <type_1>... except that it does not include the empty list.
- <type_1> + <type_2>, <type_1> - <type_2>, <type_1> * <type_2>, <type_1> / <type_2>, and <type_1> % <type_2>, which specify expression types composed of the binary operators $+$, $-$, $*$, /, and % respectively.
- <type_1> | <type_2>, which specifies the union of two types, <type_1> and <type_2>. In particular, it includes all values that belong to either <type_1> or <type_2>.
- ˜<type_1>, which specifies the complement of <type1>; that is, it includes all values that do not belong to <type_1>.

9.12 The Pattern Matching Operator (the ":" operator)

The syntax of invoking the pattern matching operator is

```
<exp> ":" <pattern>
```

which determines whether or not an arbitrary expression <exp> has a given type. POET uses pattern matching to dynamically test the type and structure of arbitrary unknown values. Further, uninitialized variables can be used to save the structural information (component values) of the data of interest during the process. The operation returns TRUE (integer 1) if <exp> matches the pattern specified and returns FALSE (integer 0) otherwise. The pattern specifier <pattern> may be in any of the following forms.

- A constant value (i.e., an integer or a string literal), where pattern matching succeeds if <exp> has the given value.
- A type specifier that could have any of the forms defined in Section 9.11. Here the pattern matching succeeds only if <exp> has the specified type.
- An uninitialized variable name or an operation in the following format,

  ```
  CLEAR <id>
  ```

where <id> is a variable name which is reset by the CLEAR operator to become uninitialized (see Section 9.10.2). The pattern matching always succeeds by assigning the unitialized variable with the value of <exp>. Note that the *CLEAR* operator typically needs to be applied to all pattern variables if the pattern matching operation is inside a loop, because all uninitialized variables become initialized after the first successful pattern matching.

- The name of an already initialized variable. The pattern matching succeeds if <exp> has the same value as the value of the variable, and fails otherwise.
- A compound data structure, e.g., a list, a tuple, or a code template, that contains other pattern expressions as components. The pattern matching succeeds if <exp> has the specified data structure and its components can be successfully matched against the sub-patterns. For example, <exp> can be successfully matched to *(pat1 pat2 pat3)* if it is a list with three components, each of which can be matched to *pat*1, *pat*2, and *pat*3 respectively.
- A $xform$ handle, in which case the handle is invoked with <exp> as argument, and the matching succeeds if the invocation returns TRUE (a non-zero integer). This feature allows a function to be written to perform complex pattern matching tasks, and the function can be used as a pattern specifier in all pattern matching operations.
- An assignment operator in the format of <id> = <pattern>, where <id> is a single variable name, and <pattern> is a pattern specification. Here the pattern matching succeeds if <exp> can be successfully matched against <pattern>; and if successful, the variable <id> is assigned with the value of <exp>.
- Two pattern specifiers connected by the binary | operator, in the format of
  <pattern1> | <pattern2>.
  Here the pattern matching succeeds if <exp> could be matched to either <pattern1> or <pattern2>.

The following illustrates the results of applying pattern matching to check the types of various expressions.

```
"3" : STRING                           <<* returns 1
3 : STRING                             <<* returns 0
"3" : ID                               <<* returns 0
"A3" : ID                              <<* returns 1
MyCodeTemplate#"123" : STRING          <<* returns 0
MyCodeTemplate#123 : MyCodeTemplate    <<* returns 1
3 : MyCodeTemplate                     <<* returns 0
MyCodeTemplate#123 : MyCodeTemplate#INT <<* returns 1
("abc" "." "ext") : STRING             <<* returns 0
("3" "4" "5") : (INT ....)              <<* returns 1
3 : (INT ...)                          <<* returns 0
(3 4 5) : (INT ...)                    <<* returns 1
(3 4 5 "abc") : (INT ...)              <<* returns 0
(3 4 5 "abc") : (_ ...)                <<* returns 1
3 : (0 .. 2)                           <<* returns 0
3 : (0 .. 5)                           <<* returns 1
```

```
"a" : (0 .. 5)                        <<* returns 0
"a" : CODE                            <<* returns 0
MyCodeTemplate : CODE                 <<* returns 1
MyCodeTemplate#123 : CODE             <<* returns 1
("abc" "." "ext") : CODE              <<* returns 0
MyCodeTemplate#123 : XFORM            <<* returns 0
("abc" "." "ext") : XFORM             <<* returns 0
foo : XFORM                    <<* returns 1; here foo is a xform routine
"abc" : TUPLE                         <<* returns 0
("abc",2) : TUPLE                     <<* returns 1
MyCodeTemplate#123 : TUPLE            <<* returns 0
```

9.13 Type Conversion (The => and ==> Operators)

POET uses two operators (=> and ==> operators) to convert a value from one type to another. The syntax of invoking the operators is

```
<exp>  =>   <parse_spec>
<exp>  ==>  <parse_spec>
```

Both the => and ==> operators have similar semantics in that they both take the given input <exp> , parse it against the structural definition contained in <parse_spec>, and store the parsing result into the variables contained in <parse_spec>. The difference between the => and ==> operators is that when parsing fails, the => operator reports a runtime error, while the ==> operator simply returns *FALSE* (the integer 0) as result of evaluation. Therefore the ==> operator can be used to experiment with parsing an input expression using different type specifiers.

For example, *exp* => (*var* = *INT*) converts the value contained in *exp* to an integer and saves the integer value to variable *var*. Note that here the type conversion succeeds only if *exp* can be successfully converted to an integer; a runtime error is reported otherwise. In contrast, *exp* ==> (*var* = *INT*) returns false if the conversion fails. Similarly, *exp* => (*var* = *STRING*) can be used to convert an arbitrary expression to a single string. Note that all values can be converted to a string or the name of a variable, so all expressions can be successfully converted to the *STRING* or *VAR* type specifier. The following shows some examples of string conversion.

```
3 => STRING                     <<* returns  "3"
MyCodeTemplate#123 => STRING    <<* returns  "MyCodeTemplate#123"
```

Similarly, when an input value is converted to *VAR* parsing specifier, it is first converted to a string, and the string is then used as the name to create a dynamic variable on the fly. For example, 5 => *VAR* returns a new dynamic variable that has an internal name created from the number 5. The *VAR* parsing specifier therefore allows place-holder variables to be dynamically created for convenient pattern matching. The lifetime and scope of these dynamic variables span the entire program, and they can be dynamically created and operated on at any point in the program. Section 7.3 further discusses the concept of dynamic variables.

9.14 The DELAY and APPLY Operators

POET provides two operators, *DELAY* and *APPLY*, to support the delay of expression evaluation. Such delay is desired when incrementally constructing pattern expressions that contain un-initialized variables (see Section 9.12).

The syntax of invoking the *DELAY* operator is

```
DELAY "{" <exp> "}"
```

This operation saves the input expression, which could potentially be a sequence of POET statements and expressions, in its original form and saves it for later evaluation. The result is the internal representation of the saved expression and can be stored into an arbitrary variable, e.g., passed as parameters to an xform routine invocation, to be evaluated later.

The syntax of invoking the *APPLY* operator is *APPLY <exp>*, which triggers all delayed expressions contained in <exp> to be evaluated and returns the evaluation result.

9.15 Operations On Tracing Handles

POET supports a special kind of global variables called *tracing handles*, which can be embedded within the internal representations of various computations to trace transformations to fragments of code. Once embedded inside a compound data structure, tracing handles become integral components of the data structure. Transformations to the data structure therefore can be implemented by simply modifying the values of the embedded tracing handles. As various transformations are applied to the data structure, these tracing handles can be automatically replaced with new values, thus making the ordering of different transformations extremely flexible, and one can easily adjust transformation orders as desired.

The following POET operations can be used to set up and maintain tracing handles.

9.15.1 TRACE (x, exp)

Here x is a single or a list of variables. These variables become tracing handles during the evaluation of the *exp* expression, so that they may be used to trace transformations performed by *exp*. For example, in the following evaluation in POET/test/gemmATLAS/gemmKernel.pt,

```
TRACE (Arepl, ScalarRepl[... trace_vars=Arepl;.... ](...))
```

the variable *Arepl* is treated as a tracing handle during the invocation of the *xform* routine *ScalarRepl*, so that the routine can modify *Arepl* to contain the names of new variables created by the routine.

9.15.2 INSERT (x, exp)

This operation inserts tracing handle x, together with all the other tracing handles that are declared together with x and following x in the same declaration, to be embedded inside expression *exp* if possible so that x may be used to trace transformations within *exp*. A special case of invoking the *INSERT* operation is *INSERT*(*tophandle*, *tophandle*), where *tophandle* is the first tracing handle that was followed by a collection of other handles declared together. Note for the *INSERT* operation to work, the tracing handles must have been declared in the same order as the order of encountering them in a pre-order traversal of the input computation.

9.15.3 ERASE(x, exp)

Here x is a single or a list of tracing handles, and *exp* is an arbitrary expression. This operation returns a new computation that is equivalent to *exp* but no longer contains any trace handles in x. For example, if *input* = *Stmt#*(x), where x is a tracing handle and contains value 3, then *ERASE(x,input)* returns *Stmt#3*. As a special case, the invocation *ERASE*(x, x), returns the value contained in the variable x (i.e., the resulting value is no longer a tracing handle). For example, if x is a trace handle and x = "*abc*", then *ERASE(x)* returns "abc".

9.15.4 COPY(exp)

Instead of explicitly specifying which tracing handles to erase from an input expression *exp*, the operation *COPY*(*exp*) replicates *exp* with a copy that has no tracing handles at all (i.e., all tracing handles are erased). For example, if *input* = *Assign#*(x, y), where both x and y are tracing handles with values "*var*" and 4 respectively, then *COPY(input)* returns *Assign#("var", 4)*.

9.15.5 SAVE (v1,v2,...,vm)

Here v1,v2, ..., vm is a tuple of tracing handle names. This operation saves the current value of each tracing handle so that the values of v1,v2,...,vm can be restored later.

9.15.6 RESTORE (v1, v2, ..., vm)

Here v1,v2, ..., vm is a tuple of tracing handle names. This operation restores the last value saved for each tracing handle. The *SAVE* and *RESTORE* operations are usually used together for saving and restoring information relevant to trace handles. Both the *SAVE* and *RESTORE* operations return the empty string as result.

9.16 Transformation Operations

POET provides several built-in operations, including replication, permutation, and replacement of code fragments, to apply a wide variety of transformations to input computations. All built-in operations support the update of tracing handles embedded within their input computations; that is, each tracing handle embedded within the input will be modified to contain the transformation result of its original value. Note that except for modifying tracing handles, all built-in operations return their transformation results without any other direct modifications to the input code.

9.16.1 DUPLICATE(c1,c2,input)

Here $c1$ is a single expression, $c2$ is a list of expressions, and *input* is the input computation to transform. This operation replicates *input* with multiple copies, each copy replacing the code fragment $c1$ in *input* by a different component in the list $c2$. It returns a list of the copies as result. For example,

```
   input = Stmt#"var";
   print ("DUPLICATE(\"var\", (1 2 3), input) = " DUPLICATE("var", (1 2 3), input));
```

produces the following output.

```
DUPLICATE("var", (1 2 3), input) =  Stmt#1 Stmt#2 Stmt#3 NULL
```

9.16.2 PERMUTE(config,input)

Here *input* is a list of expressions, and *config* is a list of integers that specify the index of permutation location for each component in *input*. This operation reorders elements in the *input* list based on *config*, which defines a position for each element in *input*. For example,

```
PERMUTE((3 2 1), ("a" "b" "c")) returns ("c" "b" "a")
```

9.16.3 REBUILD(exp)

This operation takes a single POET expression *exp* and returns the result of rebuilding *exp*. Here for each code template object contained in *exp*, the rebuilding process replaces the object in *exp* by invoking the *rebuild* attribute defined for the corresponding code template type (see Section 5.4) if appropriate. The operation therefore can be used to automatically eliminate redundancies (e.g., empty strings) in *exp* based on customizable definitions for each relevant code template type.

9.16.4 REPLACE(c1,c2,input)

Here *input* is the input computation to transform, *c*1 is an expression embedded inside *input*, and *c*2 is the new expression to replace *c*1, This operation replaces all occurrences of the code fragment *c*1 in *input* with *c*2. For example,

```
REPLACE("x","y",SPLIT("","x*x-2"))  returns  "y" "*" "y" "-" 2 NULL
```

9.16.5 REPLACE(config, input)

Here *input* is the input computation to transform, and *config* is a list of pairs in the format of (*orig*, *repl*), where *orig* is an expression embedded inside *input*, and *repl* is the expression to replace *orig*. This operation traverses the *input* to locate the *orig* component of each pair in *config* and replaces each *orig* with *repl* in *input*. Each (*orig*, *repl*) pair in *config* is expected to be processed exactly once, in the order of their appearances in *config*, during a pre-order traversal of the *input*. If there is any pair never processed in *config*, the rest of the specifications in *config* will be ignored, and a warning is issued. For example

```
REPLACE( (("a",1) ("b",2) ("c",3)), Bop#("+","a",Bop#("-","b","c")))
    =  Bop#("+",1,Bop#("-",2,3))
```

9.17 The Conditional Expression (The "?:" operator)

POET supports conditional evaluation of expressions using the following syntax (same as C).

```
<cond> ? <exp1> : <exp2>
```

Here the <cond> expression is first evaluated, which should return a boolean (integer value). If the return value of <cond> is true, the result of evaluating <exp1> is returned; otherwise, the result of evaluating <exp2> is returned.

Chapter 10

Statements

In POET, statements are considered special expressions whose results may be ignored when composed into a sequence. For example, when a collection of statements $s_1, s_2, ..., s_m$ is composed into a sequence, the evaluation results of the previous $s_1, s_2, ..., s_{m-1}$ statements are thrown away, and only the result of the last statement is returned. In contrast, when a collection of expressions $e_1, e_2, ..., e_m$ is composed into a sequence, the evaluation result is a list that contains the result of all expressions $e_1, e_2, ..., e_m$ as components (in POET, when expressions are simply listed together, they are considered operands in a list construction operation. See Section 4.2.1).

POET statements serve to provide control flow support such as sequencing of evaluation, conditional evaluation, loops, and early returns from xform routines.

10.1 Single Statements

10.1.1 The Expression statement

The syntax for the expression statement is

```
<exp> ;
```

An expression statement is composed by following any POET expression with a semicolon (i.e., ";"). If an expression is followed by a ";", its evaluation result is always an empty string, and the result is ignored when composed with other statements. Expression statements are used to support sequencing of statements — that is, only the result of the last expression is returned, and the results of all previous evaluations are ignored.

10.1.2 The RETURN (return) statement

The syntax for the RETURN statement is

```
RETURN <exp> ;
```

or

```
return <exp> ;
```

The RETURN or return statement must be inside the body of a $xform$ routine (a runtime error is raised otherwise). When evaluated, it exits the $xform$ routine with the result of evaluating <exp>. The RETURN (return) statement is provided to allow convenient early returns from xform invocations.

10.1.3 Statement Block

The syntax for a statement block is

```
{ stmt1  stmt2  ... stmtm }
```

Here *stmt*1, *stmt*2, ..., *stmtm* are a sequence of statements. So a statement block merely combines a sequence of statements into a single one. The value of the statement block is the value of the last statement *stmtm*.

10.2 Conditionals

10.2.1 The If-else Statement

The syntax for the *if-else* statement is

```
if ( <cond> )  <stmt1>  [ else  <stmt2> ]
```

Here <cond> is a POET boolean expression, and <stmt1> and <stmt2> are single statements (including statement blocks). If <cond> evaluates to *true* (a non-zero integer), <stmt1> is evaluated, and the value of the last expression in <stmt1> is returned; otherwise, <stmt2> is evaluated, and the value of the last expression in <stmt2> is returned. If <cond> evaluates to $false$ and the *else* branch is missing, then an empty string is returned as result of evaluation.

10.2.2 The Switch Statement

The syntax for the *switch* statement is

```
switch (<cond>)
{
case <pattern1> : <stmts1>
case <pattern2> : <stmts2>
......
case <patternm> : <stmtsm>
[ default :  <default_stmts> ]
}
```

Here <cond> is an arbitrary expression, <pattern1>,<pattern2>,..., <patternm> are pattern specifiers as defined in Section 9.12, and <stmts1>,<stmts2>,...,<stmtsm> and <default_stmts> are sequences of statements or expressions. The switch statement first evaluates <cond> and then matches the result of <cond> against each pattern specifier in order. Specifically, if <cond> : <pattern1> succeeds, then <stmts1> is evaluated and the result of <stmts1> becomes the result of the switch statement; otherwise, the result of <cond> is matched against <pattern2>, and so forth. If none of the patterns can successfully match the value of <cond>, the <default_stmts> is evaluated and returned as result. If no pattern matching succeeds and no default statements are specified (the default branch is optional), an error message is issued.

Note that when evaluating the switch statement, only one pattern will be successfully matched with the given <cond> throughout the evaluation. Once a pattern matching succeeds, the corresponding statements are evaluated and the result is returned immediately (no statements in the following patterns will be evaluated). If two patterns need to be combined, they should be combined into a single pattern specification using the | operator, shown in Section 9.12.

The *switch* statement syntax is equivalent to the following syntax using if-else statements.

```
var = <cond>;
if (var : <pattern1>)  { <stmts1> }
else if (var : <pattern2>) { <stmts2> }
......
else if (var : <patternm>) { stmtsm> }
[else { <default_stmts> } ]
```

10.3 Loops

10.3.1 The *for* Loop

The syntax of the *for* loop is as the following.

```
for ( <init> ; <cond> ; <incr>)
     <body>
```

Here <init> and <incr> are arbitrary expressions, <cond> is a boolean expression, and <body> is a single statement (could be a statement block) that comprises the loop body. First, the <init> expression is evaluated to initialize the loop. Then, <cond> is evaluated. If <cond> returns TRUE, <body> and <incr> are evaluated, and <cond> is evaluated again to determine whether to repeat the evaluation of <body> and <incr>.

As example, the following loop prints out each element contained within a list *input*.

```
for (p_input = input; p_input != NULL; p_input = TAIL(p_input)) {
    print ("seeing element: " HEAD(p_input));
}
```

10.3.2 The *foreach* Loop

The syntax of the *foreach* loop is

```
foreach <pattern> \in <exp> s.t. <succ> do
    <body>
enddo
```

or the older syntax

```
foreach (<exp> : <pattern> : <succ> )
    <body>
```

Here <exp> is the an arbitrary expression, <pattern> is a pattern specifier as defined in Section 9.12, <succ> is a boolean expression, and <body> is a single statement (or a statement block). The *foreach* statement traverses the input computation <exp> and matches each component contained in <exp> against the pattern specifier <pattern>. If any pattern matching succeeds for a code fragment *subexp* in <exp> and if <succ> evaluates to true, it evaluates the <body> statement. If <succ> is set to be the boolean constant value FALSE, the foreach loop will continue traversing the *subexp* in order to find additional matches. The foreach statement therefore serves as the built-in operation for collectively applying pattern matching analysis to an input computation.

Note that in order to process each fragment that matches a given pattern, the <pattern> specifier needs to contain local variables that are assigned with the matched fragment when the matching succeeds. The following example illustrates how to print out all the loop controls inside an *input* computation.

```
foreach curLoop = Loop \in input do
  print ("found a loop: " curLoop);
enddo
```

The above is equivalent to

```
foreach (input : (curLoop = Loop) : TRUE)
{
  print ("found a loop: " curLoop);
}
```

The expression *curLoop* = *Loop* is enclosed inside a pair of parentheses in the first syntax because the assignment operator has lower precedence than the : operator. The following loop collects all the loop nests within an *input* computation.

```
loopNests = "";
foreach curNest = Nest \in s.t. FALSE do
   loopNests = BuildList(curNest, loopNests);
enddo
```

Here because loop nests may be inside one another, the <succ> parameter of the foreach loop is set to *FALSE* so that the pattern matching can continue inside already located loop nests.

Since each foreach loop makes a traversal over the entire input, it is recommended to use the foreach loop to collect information only. If the input computation needs to be transformed, it is better to invoke a *REPLACE* operation (see Section 9.16) after a foreach loop has finished, as the transformation operations may disrupt the traversal by the *foreach* loop.

To traverse an input in reverse order, use the following

```
foreach <pattern> \in reverse(<exp>) s.t. <succ> do
    <body>
enddo
```

or the older syntax using like the following.

```
foreach_r (<exp> : <pattern> : <succ> )
    <body>
```

The above essentially has the same syntax and semantics as the *foreach* loop, except that it traverses the input <exp> in the reverse order of the traversal by the corresponding *foreach* loop. The different traversal order allows the relevant information to be gathered and saved with more flexibility. For example, the following code

```
loopNests = NULL;
foreach curNest = Nest \in  reverse(input)  s.t. FALSE do
   loopNests = curNest :: loopNests;
enddo
```

or in older syntax

```
loopNests = NULL;
foreach_r (input : (curNest = Nest) : FALSE)
   loopNests = curNest :: loopNests;
```

collects all the loop nests inside *input* and saves the loop nests in a list in the same order of their appearances in the original code. In contrast, the almost identical loop in Section 10.3.2 saves all the loop nests in the reverse order of their appearances in *input*.

10.3.3 The BREAK (break) and CONTINUE (continue) statements

Just like the *break* and *continue* statements in the C language, POET provides *break* and *continue* statements to jump to the continuation and exit of a loop. The syntax for both statements are

```
BREAK
CONTINUE
```

or

```
break
continue
```

These two statements have the same meaning as those in C, and can be used to break out of (or back to the start) of *for*, *foreach*, and *foreach_r* loops.

Appendix A. Context-free grammar of the POET language

```
poet :  commands ;
commands :  commands command | ;

command :  "<" "parameter" ID  paramAttrs "/>"
    | "<" "define" ID exp "/>"
    | "<" "eval" exp "/>"
    | "<" "cond" exp "/>"
    | "<" "trace" traceVars "/>"
    | "<" "input" inputAttrs inputRHS
    | "<" "output" outputAttrs "/>"
    | "<" "code" ID codeAttrs codeRHS
    | "<" "xform" ID xformAttrs xformRHS

paramAttrs :  paramAttrs paramAttr | ;
paramAttr : "type" "=" typeSpec  | "default" "=" expUnit
    |  "parse" "=" parseSpec  |  "message" "=" STRING
traceVars : ID |  traceVars "," traceVars
inputAttrs : inputAttr inputAttrs  | ;
inputAttr : "debug" "=" expUnit  |  "annot" "=" expUnit  |  "cond" "=" expUnit
    |  "syntax" "=" expUnit  |  "parse" "=" "POET"  |  "parse" "=" parseSpec
    |  "from" "=" expUnit   |  "to" "=" ID  |  "to" "=" "POET"
inputRHS : ">" inputCodeList "</input>"  |  "/>"
inputCodeList : inputCode | inputCode inputCodeList

outputAttrs : outputAttr outputAttrs  | ;
outputAttr :   "cond" "=" expUnit  |   "syntax" "=" expUnit
    |   "from" "=" expUnit   |  "to" "="  expUnit
codeAttrs :  codeAttrs codeAttr | ;
codeAttr : "pars" "=" "(" codePars ")"  | ID "=" typeSpec
    | "cond" "=" expUnit  | "rebuild" "=" expUnit
    | "parse" "=" parseSpec  | "output" "=" typeSpec
    | "lookahead" "=" INT  |  "match" "=" typeSpec
codeRHS : ">" exp "</code>   |   "/>"
xformAttrs : xformAttrs xformAttr  |   ;
xformAttr: "pars" "=" "(" xformPars ")" | "output" "=" "(" xformPars ")"
    |  ID "=" typeSpec
xformRHS : ">" exp "</xform>"  |  "/>"
codePars :  ID  | ID ":" parseSpec  | codePars "," codePars
xformPars : ID | ID ":" typeSpec | xformPars "," xformPars

typeSpec :  INT  |  STRING  |  "_" |  ID
    | "INT" | "STRING"  | "ID"  |  "VAR" | "CODE"  |  "XFORM"  |  "TUPLE"
    | "MAP" "(" typeSpec "," typeSpec ")" |  "EXP"
    | typeSpec "..." | typeSpec "...."
    | typeSpec ".." typeSpec  | ID "#" typeSpec
    | "(" typeList ")"  |  "(" typeTuple ")"
    | typeSpec "|" typeSpec | "~" typeSpec
    | typeSpec "+" typeSpec | typeSpec "-" typeSpec | typeSpec "*" typeSpec
    | typeSpec "/" typeSpec | typeSpec "%" typeSpec | typeSpec "::" typeSpec
typeList :  typeSpec  |  typeSpec typeList
typeTuple :  typeSpec "," typeSpec  |  typeTuple "," typeSpec

patternSpec : typeSpec | "CLEAR" ID
     | "(" patternSpecList ")" | "(" patternSpecTuple" ")"  |  patternSpec "|" patternSpec
     |  ID "[" xformConfig "]"  | ID "#" patternSpec | ID "=" patternSpec
patternSpecList :  patternSpec patternSpec | patternSpec patternSpecList
patternSpecTuple :  patternSpec "," patternSpec | patternSpecTuple "," patternSpecTuple

parseSpec : typeSpec
    | "TUPLE" "(" parseSpecList ")"  | "LIST" "("  parseSpec ","  singleType ")"
    | "(" parseSpecList ")" | "(" parseSpecTuple" ")"  |  parseSpec "|" parseSpec
    |  ID "[" xformConfig "]"  | ID "#" parseSpec
    | ID "=" parseSpec
parseSpecList :  parseSpec parseSpec | parseSpec parseSpecList
parseSpecTuple :  parseSpec "," parseSpec | parseSpecTuple "," parseSpecTuple
xformConfig : ID "=" parseSpec  | xformConfig ";" xformConfig

exp : expUnit | exp exp | exp "::" exp | exp ","  exp
    | "car" expUnit | "cdr" expUnit | "HEAD" expUnit | "TAIL" expUnit | "LEN" expUnit
    | "ERROR" expUnit  |  "PRINT" expUnit | "print" expUnit
    |  DEBUG "[" INT "]" "{" exp "}"  |  DEBUG "{" exp "}"
    | exp "=" exp  | exp "+=" exp | exp "-=" exp
    | exp "*=" exp | exp "/=" exp | exp "%=" exp
    | exp "=>" parseSpec | exp "==>" parseSpec
    | exp "?" exp ":" exp
    | exp "&&" exp | exp "||" exp | "!" exp  | exp "|" exp
    | exp "<" exp | exp "<=" exp | exp "==" exp
    | exp ">" exp | exp ">=" exp | exp "!=" exp
    | exp ":" patternSpec | "-" exp
    | exp "+" exp | exp "-" exp | exp "*" exp | exp "/" exp | exp "%" exp
    | exp "^" exp     | "SPLIT" "(" exp "," exp ")"
    | "REPLACE" "(" exp "," exp ")" | "REPLACE "(" exp "," exp "," exp ")"
    | "PERMUTE" "(" exp "," exp ")"  | "DUPLICATE" "(" exp "," exp "," exp ")"
    | "COPY" expUnit  |  "REBUILD" expUnit
    | "ERASE" "(" exp "," exp ")" | INSERT "(" exp "," exp ")"
    | "DELAY" "{" exp "}"  |  "APPLY" expUnit  | "CLEAR" expUnit
    | "SAVE" expUnit  |  "RESTORE"  expUnit  | "TRACE" "(" exp "," exp ")"
    | expUnit "..." | expUnit "...." | expUnit ".." expUnit
    | "MAP" "(" typeSpec "," typeSpec ")"
    | exp "[" exp "]"  |  exp "#" expUnit
    | stmt
```

```
stmt:  exp ";" | "{" exp "}"  |  "RETURN" expUnit | "return" expUnit
    | "if" "(" exp ")" stmt  |  "if" "(" exp ")" stmt "else" stmt
    | "switch" "(" exp ")" "{" cases "}"
    | "for" "(" exp ";" exp ";" exp ")" stmt
    | "foreach" id = patternSpec \in exp "s.t." exp do stmt enddo
    | "foreach" id = patternSpec \in reverse(exp) "s.t." exp do stmt enddo
    | "foreach" "(" exp ":" patternSpec ":" exp ")" stmt
    | "foreach_r" "(" exp ":" patternSpec ":" exp ")" stmt
    | "CONTINUE" | "continue" |  "BREAK"  | "break"

expUnit: "(" exp ")" | ID | "XFORM" | "CODE" | "TUPLE" | "STRING" | "INT" | "VAR"
    | INT | STRING | "_"

cases : cases "case" patternSpec ":" exp
    |  cases "default" ":" exp
    | "case" patternSpec ":" exp
```