POET Reference Manual

Qing Yi
University of Texas At San Antonio
(qingyi@cs.utsa.edu)

July 24, 2010

Front Matter

POET is an interpreted program transformation language designed to apply source-to-source transformations to programs written in languages such as C, C++, Java and domain-specific ad-hoc languages such as an interface specification language. The POET language has been extensively used for the purpose of applying parameterized program optimizations to improve the performance (i.e., the runtime speed) of C programs in the context of empirical performance tuning, where the runtime performance of different program implementations are experimented and the feedback is used to guide further optimizations. The use of POET, however, is not limited to program optimizations. You can use POET to easily process any structured input, extract information from or apply transformations to the input, and then output the result.

The POET language was designed and implemented by Dr. Qing Yi at the University of Texas at San Antonio. Please directed all questions and feedbacks to her at qingyi@cs.utsa.edu.

Qing Yi
6/15/2009

Copyright (c) 2008, Qing Yi. All rights reserved.
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. Neither the name of UTSA nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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-1.02.06.tar.gz
> cd poet-1.02.06
> ./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 CVS

If you are a developer of POET and have access to the POET CVS repository, you can build POET using the following commands.

```
> cvs co POET
> 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 a variety of code templates and xform routines implemented using POET.

- The *test* directory, which contains some POET scripts used to test the correctness of the POET install.
- The *example* directory, which contains examples used in various POET tutorials.
- The *doc* directory which contains the manual and tutorials for POET.

1.4 Using POET

The main components of the POET implementation include the language interpreter, named *pcg*, and a library of POET code templates and xform routines for various purposes. 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 [-hv] {-dp[xp]} {-L<dir>} {-p<name>=<val>} <poet_file1> ... <poet_filen>
options:
     -h:      print out help info
     -v:      print out version info
     -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
     -dx: print out debugging info. for xform routine invocations
     -md: allow code template syntax to be multiply defined (overwritten)
```

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 != "";
      (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 11.5). 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 the empty string (“”).

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. In fact, the language is designed to support program optimizations, and a large library of program transformations have been provided to support this purpose. The following example from *POET/examples/applyopt.pt* shows overall structure of an optimization script in POET.

```
include opt.pi

<parameter inputFile type=STRING default="" message="input file name" />
<parameter outputFile type=STRING default="" message="output file name" />


<trace inputCode,nest1,nest1_1,nest1_2,nest1_3/>
<input from=inputFile syntax="Ffront.code" to=inputCode/>

<code SingleLoop/>
<eval
     SingleLoop#nest1_1=nest1;
     nest1_2=nest1_1[Nest.body];
     nest1_3=nest1_2[Nest.body];
     INSERT(nest1,inputCode);

     UnrollLoops[factor=2;cleanup=1;trace=nest1](nest1_3[Nest.body],nest1_3);
/>

<output to=outputFile syntax="Ffront.code" from=(inputCode)/>
```

NOTE13: The first line of the optimization script is an *include* directive, which reads in the entire content of file *opt.pi* before parsing any content of the current file. The file *opt.pi* is in POET/lib and contains the interface of a large collection of loop transformation routines, including the transformation routine *UnrollLoops* invoked in the current POET script.

NOTE14: The *trace* declaration declares a sequence of trace handles (see Section 7.4.2), which are global variables that can be embedded in the internal representation of the input code to support transformations to the input code. In POET, only trace handles can be modified through the side effects of xform routines (see Chapter 6 and POET transformation operations (see Section 11.12).

NOTE15: The file *Ffront.code* is in POET/lib directory and includes syntax definitions for parsing Fortran code. The code template *SingleLoop* is defined in *Ffront.code*. It needs to be declared again in *applyopt.pt* because the *Ffront.code* file is not read until the entire *applyopt.pt* file has been parsed.

NOTE16: The above script assumes that the input file contains an annotation which assigns a triply nested loop nest to be the value of the trace handle *nest*1. The three trace handles, *nest*1_1, *nest*1_2, *nest*1_3 are assigned with the each loop contained in the nest. Then, the *INSERT* operation (see section 11.11.2) inserts all trace handles into *inputCode*, which is itself a trace handle. Note that the *INSERT* operation can modify *inputCode* only because it is a trace handle.

NOTE17: After inserting all trace handles, the *xform* routine *UnrollLoops* is invoked to unroll the innermost loop in *nest*1_3. This routine is defined in *POET/lib/opt.pt*, and its interface has already been declared by the include file opt.pi.

Chapter 3

Language Overview

3.1 Overview of Concepts

The following briefly outlines the main concepts that will be covered by the rest of 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*; one special user-defined data type: *code templates*; and one function type: *xform routine handles*.

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). POET uses code templates exclusively to build the AST (Abstract Syntax Tree) internal representations of programs and to support the parsing/unparsing of the input code. 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 function calls to each other, use loops and if-conditionals to iterate over a body of computation, and systematically apply transformations or program analysis to internal representations of input code.

4. *Variables and Assignments (Chapter 7).* POET uses variables to hold values of intermediate evaluation and to reconfigure behaviors of the POET interpreter. Variable assignments can be used to modify the values of all types of variables. 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 code template object as value, but it cannot modify the content of the old code template object contained in x, as this object may be shared by other variables.

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

6. *Type specifications and pattern matching (Chapter 9).* POET is a dynamically typed language and provides a collection of type specifiers, one for each type of values supported by the

language, to allow the types of expressions to be dynamically tested and to allow different types of values to be converted to each other. The pattern matching operator (:) is used to dynamically examine the types of POET values, and the => and ==> operators are used to convert different types between one another.

7. *Parsing specifications and type conversion (Chapter 10).* POET provides a collection of parsing specifications to specify how to parse a stream of input tokens and convert them into an internal structured representation using code template objects. These parsing specifications are used inside code template definitions to guide the parsing/unparsing process and are used to dynamically convert different types of values from one to another.

8. *Expressions and built-in operations (Chapter 11).* POET provides a large collection of built-in operators to support all types of expressions, which are building blocks of program evaluation. POET expressions must be embedded within global commands or code template, xform routine, or variable declarations to be evaluated in POET programs.

9. *Statements (Chapter 12)*, which are different from expressions in that they do not have values. POET statements serve to provide support for debugging (side effects) and control ow needs such as sequencing of evaluation, conditional evaluation, loops, and early exit from a xform routines.

The POET/lib directory contains an extensive library of xform routines for applying various performance optimizations and a large 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 previous global declarations already processed. 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. Otherwise, the proper prefix, *CODE*, *XFORM*, or *GLOBAL*, must be used to qualify the use of these names.

3.3 Components of POET Programs

A POET program can be composed of an arbitrary number of different files, where each file is composed of a sequence of global declarations and commands of 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 include directives are used to start the POET/lib/Cfront.code file.

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

  All *include* directives must be at the start of a POET file, so that the specified external files are guaranteed to have been evaluated before reading the current file. Additionally, 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 together with the current file. That is, the *.pi* extension has been reserved by POET to indicate interface files of different libraries.

- 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 *> will be ignored by the POET interpreter, and all strings following <<* until the end of line will be ignored.

Except for the *include* directives, which must be put at the start of a POET file, all 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 start of a POET file.*

3.4 Notations

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

- <concept>, which specifies a concept equivalent to 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 production 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 concepts are used throughout the manual to specify syntax 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 9.1.
- <pattern>: all POET pattern specifications, defined in Section 9.2.
- <parse_spec>: all POET parsing specifications, defined in Section 10.1.
- <exp>: all POET expressions, defined in Chapter 11.

Chapter 4

Atomic and Compound Data Types

POET supports two types of atomic values: integers and strings; three types of compound data structures: tuples, lists, and associative maps; one user-defined compound data type: code templates; and one user-defined function type: xform handles.

Note that if cycles exist within the internal representation of a computation, attempting to traverse the entire representation would cause infinite recursion and thus break the POET interpreter. To prevent cycles from being formed, POET does not allow the modification of the content of compound data structures such as tuples, lists, and code templates, which can be used collectively to build internal representations of 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. The language may be extended in the future if the need of floating point values comes up. 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

An integer value is simply a sequence of digits, e.g., 12345, 27. POET provides built-in operations to support integer arithmetics (+,−,∗,/,%), integer comparisons (<,<=,>,>=,==,! =), and boolean arithmetics (!, && and ||). The semantic definition of these operations is straightforward and follows the C language. Except 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 meaning 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). Instead of providing any substring construction support, POET provides an operator $SPLIT$ which can be used to split strings into lists of substrings based on a specified separator. For example, $SPLIT($",", "abd, ade"$)$ returns a list of three strings ("abd" "," "ade"). Similarly, $SPLIT$ can also be applied to all strings contained in an input. For example, $SPLIT($"+", $Stmt\#($"$a+b+c$"$))$ will return $Stmt\#($"a" " + " "b" " + " "c"$)$. If the separator is an integer n, the $PLIT$ operator will split immediately after the nth character of the string. For example $(SPLIT(1,$ "abc"$) =$ ("a" "bc").

4.2 Compound Data Structures

POET supports three 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. Lists are extensively used in POET to conveniently store elements that will to be accessed sequentially (one element after another).

A POET list can be composed by simply listing elements together. For example, (a "<=" b) produces a list with three elements, a, "<=", and b. Additionally, the operator :: is provided to dynamically extend existing lists. 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. Because lists can be dynamically extended, they often contain an unknown number of elements.

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 the empty string ""). 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 the other without any space, e.g, a list ("a" "+" 3) is unparsed as "a+3".

4.2.2 Tuples

A POET tuple is simply a finite number of elements enumerated in a predetermined order. All elements within a tuple must be explicitly specified when constructing the tuple, so tuples cannot be built dynamically (e.g., using a loop). Because of their static nature, tuples are used to define a finite sequence of values, e.g, the parameters for a code template or a *xform* routines.

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. When being

unparsed to an external file, elements in a tuple are separated with commas.

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.

The LEN operator can be used obtain the size of an unknown tuple. For example, $LEN(1, 3, 4, 5) = 4$.

4.2.3 Associative Maps

POET uses associative maps to efficiently associate two arbitrary types of values. Each associate map is internally implemented using C++ STL maps, and they are the only compound data structure whose internal content can be 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 ( (amap bmap) :  (CLEAR from, CLEAR to) : FALSE) {
    PRINT ("MAPPING " from "=>" to);
}
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}
```

To create an empty map, use the MAP operator with an empty pair of {}. To create a map with a collection of pre-known entries, invoke the MAP operator followed by a tuple of entries enclosed within a pair of {}.

The elements within a map can be accessed using the “[]” operator and modified using assignments. 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 $foreach$ statement and the $PRINT$ operation, see Section 12.3.2 and 11.1.1.

4.3 Code Templates

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. Code template objects are essentially recursive pointer-based data structures and are mostly used

to build dynamically shaped trees and DAGs (directed acyclic graphs). 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 many parsing/unparsing specifications with each code template (see Chapter 5), so that input programs can be automatically parsed and converted to their internal code template representations.

The syntax for building a code template is

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

where <id> is the code template name, and each <exp> specifies a value for each parameter (i.e., each data field) of the code template. For example, $Loop\#(\text{“}i\text{”}, 0, \text{“}N\text{”}, 1)$ and $Exp\#(\text{“}abc/2\text{”})$ 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 the code template object, <id1> is the name of the code template, and <id2> is the name of the 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 $Loop\#(\text{“}ivar\text{”}, 5, 100, 1)$, then $aLoop[Loop.i]$ returns value “ivar”, and $aLoop[Loop.step]$ returns value 1.

4.4 Xform Routine Handles

A xform routine handle 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 routine handles are essentially function pointers that can be used as values of variables, which can then later be used in function invocations. An invocation succeeds, however, only if the function variable indeed contains a *xform* routine handle as value; otherwise, a list that contains two components will be returned as result (i.e., the invocation will be interpreted as the construction of a list data structure).

Each xform routine 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. Since a xform routine 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 routine handle can 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

In POET, code templates are user-defined compound data structures that are used to represent the input computation in a structured fashion. They are used to define how to parse an input computation, how to implement the internal representation of the computation, and how to unparse internal representations to external files.

5.1 Defining Code Templates

The syntax for defining a code template 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 BNF declares the interface of the code template; i.e., the data type used to implement internal representations of programs. The second BNF additionally defines the concrete syntax of code template; that is, how to parse the input file to build the internal representation and how to unparse the internal representation to external files.

Each POET code template is like a C struct, where the name of the code template defines a unique global data type. In particular, each code template defines a unique user-defined compound

data structure, 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 parsing/unparsing, 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 declaration are optional. The following explains the semantics of each optional component.

5.2 Template Parameters

In the following examples,

```
<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, the code template *Loop* has four required parameters (data fields): i, *start*, *stop*, and *step*; the code template If has a single parameter, *condition*, which is an EXP. Details of parsing specifications are explained in Section 10.1.

5.3 Template Body

The body of each code template is a POET expression that defines the concrete syntax of the code template both for parsing input code and for unparsing transformation results. The template body is typically a list of strings in another source-level language (e.g., C, C++). If evaluation of POET expressions are necessary within the body, the POET expressions need to be wrapped inside pairs of symbols. The following shows the definition of the *Loop* code template body in $POET/lib/Cfront.code$.

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

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 strings (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 parser 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. Note that 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 Optional attributes

Each code template declaration can optionally include a sequence 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 an optional 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 (*parse*, *lookahead*, *rebuild*, *match*, *output*, *unparse*, *rebuild*, *parse*, and *output*) are reserved and are used to specify how to parse, unparse, simplify, operate on objects of the code template.

5.4.1 The *parse* attribute

This attribute specifies how to parse the input computation against the particular code template. Specifically, when the *parse* attribute is defined, the template body is no longer used to parse a given input. 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*.

Another common use of the *parse* attributes is to utilize the built-in parsing capabilities defined in POET. For example, the following code template definition defines the *parse* attribute to be a list of *Name* :s separated by ",".

```
<code NameList pars=(content) parse=LIST(Name,",") />
```

Using the *parse* attribute can change how the code template is unparsed as well. For example, given the *NameList* code template definition above, when a *NameList* object needs to be unparsed to an external file, each component will be separated with a ",".

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 parser 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, $n = 1$ for all code templates.

5.4.3 The *match* attribute

This attribute specifies an alternative expression that can be used to substitute the current code template during pattern matching operations (see Section 9.2). In particular, if the code template fails to match against a given input during pattern matching, the POET interpreter uses the value

of code template's *match* attribute as an alternative target and tries again. For example, the following code template definition

```
<code Ctrl parse=If|While|For|Else match=Loop|If|While|For|Else />
```

specifies that the *Ctrl* code template can be matched against any of the *If*, *While*, *For*, or *Else* control flow concepts.

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 11.12) 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 optional attributes and can optionally 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 concrete syntax (i.e., the template body) of the code template 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* value

Each optional attribute in a code template definition must be given a constant value, e.g., an empty string, as its default value. A special value, named *INHERIT*, is provided by POET to provide context information when using code templates during parsing. In particular, the *INHERIT* value contains 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 as the default value for any of the optional attributes defined above. For example, the following code template definition (from *POET/lib/Cfront.code*)

```
<code Else ifNest=INHERIT>
else
</code>
```

specifies that the *ifNest* attribute should be initialized with the previous code template object (i.e., the *If* statement immediately before the *else* keyword).

Chapter 6

Xform Routines

Each xform routine in POET is a function that takes a number of input parameters and returns a result. POET xform routines can make recursive function calls to each other, use loops and if-conditionals to operate on the internal representation of some input code, and systematically apply transformations based on pattern matching or alternative program analysis results.

Note that POET xform routines operate on the internal representation (known in the compiler world as Abstract Syntax Tree) of the input code without knowing what input languages the ASTs are built from or what output languages they will be unparsed to. These xform routines are generic in the sense that they can be invoked to operate on code parsed from arbitrary programming languages. To switch to other programming languages, the user only needs to switch the language syntax descriptions, and the xform routines 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> }
               [output=( <id> {, <id>} ) ] "/>"
```

which declares the interface of a *xform* routine, or

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

which 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; the optional *output=...* specifies that the routine returns a tuple of values; and each *<id>=<type>* specifies an additional optional parameter (name defined by <id>) and the default value (defined by <type>) of the parameter. When each *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 these arguments 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; (cur=car p_input)==" " || cur == "\n" || cur == "\t";
     p_input=cdr p_input)  {""}
  p_input
</xform>
```

Here the first xform declaration of *ParseList* is a forward declaration of the routine interface without implementation. It 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. In particular, if a *ParseList* invocation returns x as result, the respective return values can be accessed using syntax $x[ParseList.result]$ and $x[ParseList.leftOver]$ respectively.

The body of the *xform* routine must return a tuple that has the same number of values as specified by the *output* attribute. If the output attribute is missing, a single value is returned by the routine.

6.2 Invoking Xform Routines

The syntax for invoking a *xform* routine is

```
<exp1> ( <exp2> {, <exp2>} )
```

Here <exp1> is the xform routine handle being invoked (for definition of xform routine handles, see Section 4.4), and the list of expressions inside the pair of () are values for the required parameters of the routine. For example, the following invocation

```
ScalarRepl[init_loc=nest1; trace=nest1]
("a_buf",alphaA, "i" * lda + "l", dim, nest1[Nest.body])
```

invokes a xform routine handle $ScalarRepl[init\_loc = nest1; trace = nest1]$ with actual parameters "a_buf", $alphaA, ...$ Within the xform handle, the optional parameter $init\_loc$ is set to $nest1$, and the optional parameter $trace$ is set to be $nest1$.

Chapter 7

Variables And Assignments

POET variables serve as place holders that store results of previous evaluations or reconfigurations of the POET interpreter. POET supports the following kinds of variables.

1. *Local variables*, whose lifetime span through a single code template and transformation routine definition.
2. *Static variables*, whose lifetime span through 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 throughout the entire POET program, and they can be accessed at the global scope across multiple POET files.

Each category of variables are created in symbol tables separate from the other kinds of variables. Further, since POET is dynamically typed, only global variables need to be declared before being used. The types of all variables are checked at runtime to ensure correctness of evaluation.

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 (attributes) 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 example,

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

the variables *input*, *p_input*, and *cur* are all 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/unparser or when an *xform* routine is invoked, and the storage goes away when the parsing/unparser or the routine invocation finishes. None of the storages is visible outside the respective code template or *xform* routine.

Note that code template or xform routine definitions cannot use any global variables. The reason for this is to avoid accidental naming conflict. Since local variables do not need to be explicitly declared, all variables used within the body of a code template or xform routine should be considered local variables whether or not there happens to be global variables declared with identical names.

7.2 Static Variables

Each POET file can have its own collection of static variables, which are used freely in the global commands (see Chapter 8) to store temporary results and to propagate information across different evaluation commands. While the lifetime of these static variables span through the entire program, their scope span only within a single POET file. Specifically, to avoid naming conflict across different POET files, static variables are constrained within the POET files that contain them (i.e., they are similar to the static variables in C). 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 10.2). 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 the a similar structure. Because all dynamic variables are created in a single symbol table throughout a POET program, name 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 throughout the entire POET program and can be accessed across different POET files. However, each global variable must be explicitly declared being used in the POET program. 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. *Macro variables*, which are global variables that can be used to reconfigure the behavior of the POET interpreter or the POET program being interpreted.

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

7.4.1 Command-line parameters

These are global variables whose values can be defined via command-line options. These command-line parameters act as configuration interfaces to POET programs. Their scope is the entire POET program; that is, once defined, they can be directly accessed across different POET files.

POET command-line parameters must be declared before used. To declare a parameter, use the following syntax.

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

Here <id> specifies the name of the parameter variable; <type> specifies the type of its value; <parse_spec> specifies how to parse the parameter from command-line strings; <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 examples of command-line parameter declaration.

```
<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 1. So the type of NB is a range (the *lb* .. *ub* specifier), which specifies that the parameter variable must have the integer type and must be within the lower and upper bound specified by *lb* and *ub* respectively. Since the parameter has only a lower bound, the special value _ can be used to denote the unknown upper bound.

- The *pre* parameter can have one of several 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 define parameter values is $-p$<id>=<val>, where <id> is the name of the parameter variable, and <val> is either an integer or a quoted 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 defines the value of the NB parameter to be 50. The given values for all command-line parameters will be checked against their type specifications to ensure the POET program is correctly invoked.

7.4.2 Trace handles

Trace handles are a special kind of global variables which can be embedded within POET expressions to trace transformations to fragments of the expression. In particular, as various transformations are applied to an input expression, the trace handles can be replaced with different

values. Each transformation can therefore operate on the trace handles without being concerned with whether or how many other transformations have already been applied.

Trace handles need to be explicitly declared in order to be embedded into POET expressions. The syntax for declaring trace handles is the following.

```
"<" trace  <id>  {,  <id> }  "/>"
```

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

```
<trace gemm,gemmDecl,gemmBody,nest3,loopJ,body3,
      nest2,loopI,body2,nest1,loopL,stmt1/>
```

Trace handles are different from other global variables as they can act as integral components of an expression and therefore can be modified within *xform* routines even if the routines cannot directly access them through their names. In fact, trace handles are the only kind of global variables that can be accessed (modified) inside xform routines. As different transformations are applied to an input expression, the xform routines can modify the values of trace handles by replacing them with new values.

NOTE that when a sequence of trace handles are declared in a single declaration, as illustrated above, these trace handles are assumed to be related, and the ordering of them in the declaration is assumed to be the same ordering that they should appear in a pre-order traversal of an expression when they are embedded in the expression. Subsequently, POET allows these trace handled to be inserted into a POET expression using a single operation (see the *INSERT* operation in Section 11.11.2). Therefore only related trace handles should be declared in a single declaration, and unrelated trace handles should be declared separately to avoid confusion.

7.4.3 Macros

POET macro variables are 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 involving only variables defined so far. The following shows some example macro definitions.

```
<define myVar1 "abc"/>
<define x  5 />
<define myFunc DELAY { x = 100; } />
```

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 a special meaning and can be used to modify the default behavior of the POET interpreter, as shown in Section 7.5.

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. These macros include:

7.5.1 The TOKEN Macro

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

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

This definition 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.

The *CODE.FLOAT* syntax indicates that *FLOAT* is the name of a code template, even if the code template name has not been explicitly declared.

7.5.2 The KEYWORD Macro

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

```
<define KEYWORDS ("case" "for" "if" "while" "float")/>
```

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

7.5.3 The PREP Macro

This macro reconfigures the internal recursive descent parser that POET invokes when reading files using the global *input* command (see Section 8.1). For example, the following definition 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 definitions configures the POET internal parser to invoke the *ParseLine* routine as a filter of the token stream before start the parsing process. In particular, the *ParseLine* routine filters out meaningless tokens based on specific meanings of column locations, e.g., only characters appearing between column 7-79 are meaningful tokens, and an entire line should be skipped if the comment-column is not empty. Such preprocessing is necessary for early languages such as Fortran and Cobol.

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 definition from POET-/lib/Cfront.code (the C language syntax) accomplishes exactly this task.

```
<define PARSE FALSE/>
```

Note that when backtracking is disabled, the POET internal parser uses the first token of each 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 files.

Therefore when backtracking is disabled, 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).

7.5.5 The PARSE Macro

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

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

This definition informs the POET interpreter that unless otherwise specified elsewhere, e.g., 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 adds a post-processor to POET's unparser when evaluating the *output* command (see Section 8.3). For example, the following definition appears in the Ffront.code (the Fortran language specialization) file in the POET/lib directory.

```
<define UNPARSE UnparseLine/>
```

This definition 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 token 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 internal support for parsing expressions, where programers only need to use the *EXP* parsing specifier to convert a sequence of tokens to an expression. Several macros are provided to define valid 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.

As example, the following EXP configurations 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 element access within the expression.

3. PARSE_BOP, which defines the code template to use to represent binary operations within the expression.

4. PARSE_UOP, which defines the code template to use to represent unary operations within the expression.

5. BUILD_BOP, which defines the xform routine to invoke to rebuild binary operations within an expression.

6. BUILD_UOP, which defines the xform routine to invoke to rebuild unary operations within an expression.

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

```
<define EXP_CALL FunctionCall/>
<define EXP_ARRAY ArrayAccess/>
<define PARSE_BOP Bop/>
<define PARSE_UOP Uop/>
<define BUILD_BOP BuildBop/>
<define BUILD_UOP BuildUop/>
```

Note that the above definitions are also used by the POET interpreter when parsing POET expressions. Specifically, if the *PARSE_BOP* macro is not defined, when seeing an expression "*abc*" + 3, the POET interpreter will report a runtime error (because integers cannot be added to an string). However, if the *PARSE_BOP* is defined, the expression *Bop#*("*abc*", 3) will be returned as the result of "*abc*" + 3. This feature allows internal representation of expressions to be more conveniently built and increases the readability of POET programs.

Chapter 8

Global Commands

The previous chapters introduce global declarations, e.g., macro, code template, xform routine, and trace handle declarations, which serve to specify attributes of global names (e.g., global macros, code templates, and xform routines). In contrast, the global commands, e.g., input, eval, and output commands, are actual instructions that are evaluated according to the order of their appearance in the POET program.

8.1 The Input Command

Each POET input command specifies a number of input code to be parsed and processed. The input code could be embedded within the command or could be contained in external files. The syntax of an input command is

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

which specifies that the input code is contained in external files, or

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

which includes the input computation (specified by <exp>) inside the body of the input command. The following describes the semantics of each optional attribute specification 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 allow annotations in the input code. By default, the *annot* has a true value, and the parsing process will utilize annotations within the input code; if *annot* is explicitly defined to be false, then annotations (if there is any) will be treated as part of the input code. For details of parsing annotations, see Section 10.3.
- *parse* =<parse_spec> specifies what start non-terminal (i.e., the top 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 POET programs. For more details on parsing specifications, see Section 10.1.

In the following example,

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

the file name that contains the input code is defined as the value of variable *inputFile*. The syntax of code templates are defined both in file *inputLang* and in $xformFile$, which are variables that contain names of the corresponding files. The command is evaluated only when $xformFile$! = “”, where the input code defined in *inputFile* will be parsed and translated into an internal code template representation based on the included input language syntax definitions.

In the following example,

```
<input from=xformFile cond=(xformFile!="") parse=POET />
```

the $xformFile$ file is parsed as a POET program.

8.2 The Eval Command

Each *eval* command specifies an expression or a sequence of statements to evaluate. The syntax of an eval command is

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

Here <exp> is a POET expression or a sequence of statements defined in chapters 11 and 12. **All POET expressions and statements must be embedded within an eval command to be evaluated at the global scope.**

8.3 The Output Command

POET *output* commands are used to write internal representations of computations to external files or standard output. The syntax for an output command is

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

Here each <exp> represents a POET expression. All the attribute specifications (*cond*,*syntax*, etc.) are optional and can be arbitrarily ordered. In particular,

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

The following shows some example output commands.

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

Here the code contained in *inputCode* is output to the file whose name is defined by *outputFile*. The syntax of code templates are defined in *inputLang*, which are variables that contain names of the language syntax files. When an empty string is specified as the output name (or when no name is specified), the result will be written to standard output.

Chapter 9

Type Specifications and Pattern Matching

Since POET is a dynamically typed language, the types of expressions often need to be dynamically checked to determine what operations should 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.

9.1 Type Expressions

POET type expressions (denoted by <type> in BNF formula) are used to dynamically categorize arbitrary unknown values during evaluation. Each type expression can be any of the following type specifiers.

- A constant value (i.e., an integer or a string literal), which includes the particular value specified.
- A code template name, which includes all objects of the particular code template.
- The *INT* specifier. The keyword *INT* specifies the atomic integer type, which includes all integer values.
- The range (..) Specifier. The special syntax *lb..ub*, where *lb* and *ub* are integer values or the *ANY* specifier _, specifies a *range* type which includes all integers >= *lb* and <= *ub*. For example, 3..7 includes all integers >= 3 and <= 7; 1.._ and _..1 include all integers >= 1 and <= 1 respectively (here the *ANY* specifier _ indicates an infinity bound).
- The *STRING* specifier. The keyword *STRING* specifies the atomic string type, which includes all string values.
- The *ID* specifier. The keyword *ID* specifies the identifier type, which 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. That is, all strings that can be treated as regular variable names.
- The ANY (_) Specifier. POET uses a single underscore character, _, to denote the universal type that includes all values supported by POET (see Chapter 4).

- The *TUPLE* Specifier. The keyword *TUPLE* specifies the tuple type, which includes all tuples (see Section 4.2.2).
- The *MAP* Specifier. The *MAP* keyword denotes the associative map type (see Section 4.2.3). It takes two type specifiers, $fromType$ and $toType$, as parameters to indicate the type of element pairs within the map. Specifically, $MAP(fromType, toType)$ includes all associative maps that associate values of $fromType$ to values of $toType$. In particular, $MAP(\_, \_)$ includes all maps that may contain arbitrary types of values.
- The *CODE* Specifier. The keyword *CODE* specifies the code template type, which includes all code template objects (see Section 4.3 and Chapter 5).
- The *XFORM* Specifier. The keyword *XFORM* specifies a xform routine type, which includes all *xform* routine handles (see Section 4.4 and Chapter 6).
- The *VAR* specifier. The keyword *VAR* specifies the *variable* type, which includes all trace handles (see Section 7.4.2) that can be embedded within POET expressions.
- The *EXP* Specifier. The keyword *EXP* specifies the expression type, which includes all POET expressions.
- Compound Type Specifiers. Type expressions can be organized into compound data structures using the type constructors for lists, tuples, and code templates. The syntax for building these compound type specifiers includes the following.
  - <id> # <type>, which specifies a code template type with <id> as the code template name and <type> as specifications for template parameters.
  - (<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>.
  - <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> ... and <type_1> ...., both of which specify a *list* type, where <type_1> specifies the type of elements within the list. Specifically, <type_1> ... specifies all lists that contain only <type_1> elements, and <type_1> .... specifies all lists that contain only <type_1> elements and contain at least one element (that is, the list is not empty). As a special case, (_....) and (_.....) specify lists that may contain arbitrary elements.
  - <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. In particular, when <type_1> and <type_2> are integer constants, the result of evaluation is returned; otherwise, a code template object is built based macro configurations of the POET interpreter (for more details, see Section 7.5.7).

  For example, *(INT INT)* includes all lists formed by two integers, (INT, INT, INT) includes all tuples that contain three integers, and $MyCode\#INT$ includes all objects of the $MyCode$ code template that contains a single integer as parameter value.

- The binary | operator Two type expressions can be combined using the binary | (alternative) operator.

  ```
  <type1> | <type2>
  ```

  Here the resulting type is alternatively either <type1> or <type2>. For example, *INT | (INT....)* specifies a type that is either a single integer or a list of integers.

- The unary operator The unary operation <type1> specifies the complement of <type1>; that is, it specifies all types of other values as long as the type is not <type1>.

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

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 pattern matching process.

The binary pattern matching operator determines whether or not an expression has a given type. The syntax of operation is

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

Here <exp> is an arbitrary expression, and <pattern> is a pattern specifier as defined in the following. The operation returns TRUE (integer 1) if <exp> matches the pattern specified and returns FALSE (integer 0) otherwise.

The pattern specification <pattern> may be in any of the following forms.

- A type specifier that could have any of the format defined in Section 9.1. 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>
  ```

  Here The CLEAR operator uninitializes an existing variable named <id> (for more details, see Section 11.9.1. The pattern matching always succeeds by assigning the unitialized variable with the value of <exp>.

- A previously defined 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 $pat1$, $pat2$, and $pat3$ respectively.

- A $xform$ routine handle, in which case the handle is invoked with <exp> as arguments, and the matching succeeds if the invocation returns TRUE (a non-zero integer). This capability 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 succeeds, the variable <id> is assigned with the value of <exp>.

- Two pattern specifications 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>. For example, $(exp : INT|STRING)$ returns TRUE if and only if *exp* is either an integer or a string value.

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 ...)                       <<* returns1
(3 4 5 "abc") : (INT ...)                 <<* returns0
(3 4 5 "abc") : (_ ...)                   <<* returns1
3 : (0 .. 2)                              <<* returns0
3 : (0 .. 5)                              <<* returns1
"a" : (0 .. 5)                            <<* returns0
"a" : CODE                                <<* returns0
MyCodeTemplate : CODE                     <<* returns1
MyCodeTemplate#123 : CODE                 <<* returns1
("abc" "." "ext") : CODE                  <<* returns0
MyCodeTemplate#123 : XFORM                <<* returns0
("abc" "." "ext") : XFORM                 <<* returns0
foo : XFORM                         <<* returns 1; here foo is a xform routine
"abc" : TUPLE                             <<* returns0
("abc",2) : TUPLE                         <<* returns1
MyCodeTemplate#123 : TUPLE                <<* returns0
```

In summary, a pattern specification may contain arbitrary type expressions as components and may additionally contain uninitialized variables. For example,

```
(2 3 4) : (first=INT second third)  <<* returns 1;  first =2; second = 3; third = 4
```

Here during the pattern matching evaluation, the uninitialized variables (*second* and *third*) are assigned with the necessary values in order to make the matching succeed. Further, since the assignment $first = INT$ is inside the pattern specification, the left-hand side of the assignment (i.e., the $first$ variable) is given the value of the matched expression.

Note that when uninitialized variables appear in the pattern specification, these variables are treated as place holders which can be matched to arbitrary expressions. If the matching is successful, all the uninitialized variables are assigned valid values as part of the evaluation. Therefore the pattern matching operation can be used not only for dynamic type checking, but also for initializing variables and for assigning values to variables (initialized or uninitialized).

Chapter 10

Parsing Specifications and Type Conversion

POET provides a collection of parsing specifications to guide the process of parsing and unparsing, where a stream of input tokens is converted into internal structured representations using code templates, and the internal representations are later unparsed to external files with proper syntax. These parsing specifications are used inside code template definitions to guide the parsing/unparsing process (see Section 5.4), inside expressions to dynamically convert different types of values from one to another, and inside annotations of the input code to improve the parsing efficiency.

10.1 Parsing Specifications

A parsing specifier specifies what targeting data structure should be used to parse and represent a given input. In particular, a parsing specification can be any of the following.

- Any of the type specifier defined in Section 9.1, where the leading input token will be converted to the specified value type as the parsing result. Note that all tokens can be converted to a string or the name of a variable, so all tokens can be successfully parsed using the *STRING* or *VAR* type specifier. But only integers (e.g., 134 or "134") can be successfully parsed using the *INT* specifier, and valid expressions can be parsed into an internal representation using the *EXP* specifier. The operations that are supported within expressions can be dynamically configured using macros, as discussed in Section 7.5.7.

- 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 expression is converted to an object of the given code template, and parsing continues with the rest of input tokens; otherwise, the parsing fails.

- A xform routine 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 strings of input, and $left\_over$ contains the rest of input after the parsing process. In this case, the xform routine is invoked with input 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 compound data structure in any of the following forms.

```
<code_template_name> # <parse_spec>,
(<parse_spec1>  <parse_spec2> ..... <parse_spec_n>), or
(<parse_spec1> , <parse_spec2>, ..... (parse_spec_n>)
```

Note that here <parse_spec> is recursively defined. In these cases, the parsing process not only tries to match the leading tokens in the input with the given compound data structure, it also tries to match each component of the data structure with the given <parse_spec>. In particular, if a code template data structure is given, all the data fields of the code template are constructed by invoking the corresponding <parse_spec>. If the parsing succeeds, an object of the given data structure will be constructed, and the parsing continues with the rest of the input.

- A variable assignment in the following format.

```
<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 following format.

```
TUPLE(<string_1> <parse_spec_1> ..... <string_n> <parse_spec_n> <string_n+1>)
```

Here each <string> operand is a single string constant, and each <parse_spec_1> ... <parse_spec_n> is a parsing specification. In this case, the parsing process tries to convert leading tokens of the input into a tuple of n elements, where <parse_spec_i> (i = 1,...,n) specifies how to parse the ith element. The tuple syntax starts with <string_1> and ends with the <string_n+1>, 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 syntax must start with "(" and end with ")", and each pair of elements must be separated with a ",". The parsing process will try to match the leading input tokens exactly to the specified syntax for the tuple, and fails if any component does not match.

- A *LIST* specification in the following format.

```
LIST ( <parse_spec_1>, <string>)
```

Here the parsing process will try to construct a list of elements from the leading tokens in the input, where <parse_spec_1> defines how to parse each element within the list, and <string> defines what must be the separator between each pair of elements. 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 (|) operator which connects two sub-specifications in the following format

`<parse_spec_1> | <parse_spec_2>`

Here the POET parser tries to parse the input using each of the parsing specifications in order (if the first one fails, the second one will be tried). For example, *INT | (INT,INT)* specifies a single integer or a pair of integers. Note that once the input is successfully matched against <parse_spec_1>, the parser will not try <parse_spec_2> at all. So the operands of the | operator need be listed in the increasing order of their restrictiveness. For example, if we use *(STRING | INT) (STRING | INT)* to parse a pair of integer/string values, the input will always be parsed as a pair of strings. Specifically, because all integers can be treated as strings, the *INT* specifier will never be tried in the parsing process.

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

POET uses two operators (=> and ==> operators) to convert an expression value from one type to another. The syntax for applying the operators are

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

Both the => and ==> operators have similar semantics in that they both take the given input <exp> , parse the expression against the structural definition contained in <parse_spec>, and store the parsing result into 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 experimentally parse an input expression using different type specifications.

For example, exp => (var = INT) converts an expression 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 arbitrary expressions to a single string. Here the conversion will always succeed because all expressions have a string representation. To check whether an expression is a string, use the pattern operation ($exp : STRING$). The following shows some more type conversion examples.

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

Note that when an expression is parsed using *VAR* parsing specifier, the expression 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 contains value 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.

10.3 Parsing Annotations

Based on a collection of code template definitions, POET can be invoked to dynamically parse an arbitrary input language. The input code can be annotated with type specifications 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(gemv=FunctionDecl) @*/
void ATL_/*@@*/__pre/*@@*/gemvT_a1_x1_b/*@@*/__BETA/*@@*/_y1(const int M, const
int N, const double alpha, const double *A, const int lda, const double *X, cons
t int incX, const double beta, double *Y, const int incY)
{
   int i, j;                                          //@=>gemvDecl=Stmt
   for (i = 0; i < M; i += 1)                         //@=>loopI=Loop BEGIN(gemvBody=Nest) BEGIN(nest2=Nest)
   {
     Y[i] = beta * Y[i];                              //@ =>stmt0=Stmt; BEGIN(nest1Wrap=Nest)
     for (j = 0; j < N; j += 1)                       //@=>loopJ=Loop BEGIN(nest1=Nest)
     {
         Y[i] += A[i*lda+j] * X[j];                   //@ =>ExpStmt
      }
   }
}
```

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. For other languages such as Fortran, POET annotations can be embedded within comments of the underlying language. Note that the empty annotations ($/ * @@ * /$) in the above example serve merely to separate an input string into separate tokens.

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 parse specification as defined in Section 10.1. For example, the annotation " int i, j; //@=>gemvDecl=Stmt" indicates that "int i, j;" is a statement that should be parsed using the *Stmt* code template, and the result should be stored in the global variable *gemvDecl*.

- 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 10.1). 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) //@ =>loopL=Loop BEGIN(nest1=Nest)" includes a multi-line annotation which starts from the *for* loop (a single-line annotated fragment stored in *loopL*). However, the "/*@; BEGIN(gemv=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 $gemv = FunctionDecl$), and the parsing results are saved in the corresponding global variables.

Chapter 11

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* routine handles, which are similar to function pointers in C.
- Variables, which are place holders for expression values.
- Invocation of POET *xform* routines, which are essentially calls to user-defined 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 expressions 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 one or more input operands, performs some internal evaluation, and returns a new value as result. These operations can be separated into the following categories.

11.1 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.

11.1.1 The PRINT operator

The syntax for invoking the PRINT operator is

```
PRINT <exp>
```

The PRINT operator takes an arbitrary expression <exp>, prints out the value of <exp> to standard error output, 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* operator.

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

11.1.2 The DEBUG operator

The syntax for 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 output. It prints out and returns the result of <exp>. The <int> is used to control how many levels of *xform* routine invocations to debug. By default, <int> is one, which means *xform* routines will be treated as regular expressions in debugging. If <int> is 2, then the debugger will step into each *xform* invocation within *exp* once. The following shows some examples of using the *DEBUG* operator.

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

11.1.3 The ERROR Operator

The syntax for 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);
RROR( "Cannot fuse different loops: " curLoop " and " pivotLoop);
```

11.2 Generic comparison of values

11.2.1 The == and != operators

The syntax for 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.

11.2.2 Integer and String comparison

A number of binary operators, including <, <=, >, and >=, are provided to support comparison of two integer or string values. These operators have the same meaning as those in C.

11.3 Integer Operations

11.3.1 Integer arithmetics

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

11.3.2 Boolean operations

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

11.4 String operations

11.4.1 The string 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".

11.4.2 The $SPLIT$ operator

The syntax for 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(",",input)      <<* result is "bc" "," "ade" "," "lkd" NULL
SPLIT("", "3,7+5")    <<* result is "3" "," 7 "+" 5 NULL
SPLIT(",",#(MyCodeTemplate,input))
                      <<* result is MyCodeTemplate#("bc" "," "ade" "," "lkd" NULL)
```

11.5 List operations

11.5.1 List construction

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

11.5.2 The Cons Operator (::)

The syntax for 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 at least two elements, "<=" and elements from b (if b is a list, the result contains all elements in b; otherwise, the resulting list contains b as an element). Because the type of b may be unknown, the result may contain arbitrary numbers of elements.

11.5.3 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 know as *TAIL*). The keywords *car* and *HEAD* can be used interchangeably, so can *cdr* and *TAIL*. The syntax for 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 the empty string "". 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 <<* returns1 2 3 4 5 NULL
HEAD(a2)        <<* returns1
TAIL(a2)        <<* returns2 (3 4 5 NULL) NULL
HEAD(a3)        <<* returns1 2 NULL
TAIL(a3)        <<* returns3 4 5 NULL
```

The syntax for 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.

11.6 Tuple operations

11.6.1 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).

11.6.2 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, $LEN(2, 3, 7) = 3$, $LEN(2, (\text{“} < \text{”}3), 4) = 3$.

11.7 Associative Map operations

11.7.1 Map Construction (the *MAP* Operator)

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

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

Here <type1> and <type2> are two type specifiers as defined in Section 9.1. Each invocation of $MAP(type1, type2)$ returns a new empty table that maps values of $type1$ to values of $type2$.

11.7.2 Map Access (the [] and *LEN* operators)

The elements within an associative map can be accessed using the "[]" operator and modified using assignments. The following illustrates how to create and operate on associative maps.

```
amap = MAP(_,_);
amap["abc"] = 3;
amap[4] = "def";
PRINT ("size  of amap is " LEN(amap));
foreach (amap :  (CLEAR from, CLEAR to) : FALSE) {
    PRINT ("MAPPING " from "=>" to);
}
PRINT amap;
```

The output of the above code is

```
size  of amap is  2 .
amap[4] is  "def" .
amap[2] is  "" .
MAPPING  4 => "def"
```

```
MAPPING "abc" => 3
(4->"def","abc"->3)
```

If a value e is stored as a key in a mapping table $amap$, then $amap[e]$ returns the value associated with e in $amap$; otherwise, an empty string is returned. The size of a mapping table $amap$ can be obtained using $LEN(amap)$. All the elements within a mapping table can enumerated using the built-in $foreach$ statement, where each entry within the table is placed into a pair of uninitialized variables. For more details about the $foreach$ statement, see Section 12.3.2.

11.8 Code Template Operations

11.8.1 Code Template Object Construction (the # operator)

POET treats each code template as a unique user-defined compound data 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, $Loop\#(\text{“}i\text{”}, 0, \text{“}N\text{”}, 1)$ and $Exp\#(\text{“}abc/2\text{”})$ build objects of two code templates named $Loop$ and Exp respectively.

11.8.2 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 the code template object, <id1> is the name of the code template type, and <id2> is the name of the data field (i.e., 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 $Loop\#(\text{“}ivar\text{”}, 5, 100, 1)$, then $aLoop[Loop.i]$ returns value “ivar”, and $aLoop[Loop.step]$ returns value 1.

11.9 Variable Operations

11.9.1 Un-initializing Variables (The CLEAR operator)

The syntax for invoking the $CLEAR$ operator is

```
CLEAR <id>
```

where <id> is the name of a variable. The $CLEAR$ operator takes a single variable v as parameter and clears the value contained in v so that v becomes uninitialized after the operation. The $CLEAR$ operator is provided to better support pattern matching, where un-initialized variables are treated as place holders that can be matched against arbitrary components of a compound data structure. For example, $input : Stmt\#(CLEAR\ content)$ will treat the uninitialized variable $content$ as a place holder and will assign a new value to it if the pattern matching succeeds. The $CLEAR$ operator needs to be applied to all pattern variables if the pattern matching operation is inside a loop, because a pattern variable can stay uninitialized only in the first iteration of the loop.

11.9.2 Variable assignment (the "=" operator)

In POET, pattern matching can be used to assign values to un-initialized variables. To modify variables already defined, regular variable assignment must be used, which has the following syntax.

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

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

- A single variable. 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 data structure of the <lhs>, and all the variables within <lhs> are assigned with the necessary values to make the matching successful. In contrast to the pattern matching operator, all variables in <lhs> are treated as uninitialized place holders in the assignment operation, and if the value of <exp> fails to match the type specification 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 $input$.

11.10 Delaying Evaluation of Expressions (the DELAY and APPLY operators)

POET provides a pair of special operators, $DELAY$ and $APPLY$, to support the delay of expression evaluations. Such delay is desired when the values for certain variables are yet unknown and when constructing a pattern expression that contains un-initialized variables for pattern matching (see Section 9.2). These operations can be used to delay evaluation of an arbitrary expression, whether the expression is local (inside a xform routine definition) or global (in the global scope).

11.10.1 The $DELAY$ operator

POET uses the $DELAY$ operator to save an expression in its original definition without evaluation. The delayed operations are in a way similar to *xform* routines except they are defined and invoked using a different syntax (defined using the $DELAY$ operator and invoked using the $APPLY$ operator), they don't have parameters, and they can operate on global variables (when defined in the global scope) or local variables (when defined inside a $xform$ routine). The syntax for defining a delayed expression is

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

Specifically, the DELAY operator takes a single expression (which could potentially be a sequence of POET statements and expressions) and saves it for later evaluation. The result is the internal representation of the saved expression. The delayed operations can be stored into an arbitrary variable and could be used in all situations that a regular expression may be used.

11.10.2 The *APPLY* operator

To evaluate a delayed expression saved in an arbitrary variable, say *foo*, use operator *APPLY*. Specifically, *APPLY foo* will evaluate the operations stored in *foo* and return the evaluation result. In this case, the *foo* variable acts like a parameterless function that may operate on global variables, and the *APPLY* operator serves to invoke the global function accordingly.

11.11 Trace Operations

POET supports a special kind of global variables called *trace* handles, which can be embedded within internal representation of computations to trace transformations to fragments of code. Once embedded inside an expression, *trace* handles become integral components of the expression value. Transformations to an input code therefore can be implemented by simply modifying the values of trace handles within the computation. Note that when inside a *xform* function, trace handles can modified on through built-in transformation operations, such as REPLACE, defined in Section 11.12. As various transformations are applied to the input code, these trace handles can be replaced with new values. The tracing capability makes the ordering of different code transformations extremely flexible, and one can easily adjust transformation orders as desired. The following POET operations can be used to set up trace handles.

11.11.1 TRACE (x, exp)

Here *x* is a single or a list of local/global 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[... traceRepl=Arepl;.... ](...))
```

the global variable *Arepl* is treated as a trace 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.

11.11.2 INSERT (x, exp)

This operation inserts all trace handles rooted at *x* (trace handles that are declared together with *x* in a single global trace handle declaration as defined in Section 7.4.2) to be embedded within expression *exp* if possible so that *x* may be used to trace transformations within *exp*. For example, the operation *INSERT*(*gemm*, *gemm*) (used in POET/test/gemmATLAS/gemm.pt) inserts all the trace handles declared together with *gemm* into the expression contained in *gemm*, so that all the trace handles become embedded within the internal representation of the input code.

In order for a number of trace handles to be successfully inserted inside a computation, the trace handles must have the correct values; that is, the values of the trace handles must indeed be part of the input expression to the *INSERT* operation. A special case of invoking the *INSERT* operation is *INSERT*(*tophandle*, *tophandle*), where *tophandle* is the first trace handle that was followed by a collection of other handles declared together. Note for the *INSERT* operation to work, the trace handles must have been declared in the save order as the order of encountering them in a pre-order traversal of the input expression.

11.11.3 ERASE(x, exp)

Here x is a single or a list of trace handles. This operation removes all the occurrences of trace handles in x from the input expression exp; that is, it returns a new expression that is equivalent to exp but no longer contains any trace handles in x. For example, if $input = Stmt\#(x)$, where x is a trace 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 trace handle). For example, if x is a trace handle and $x =$ "abc", then

```
ERASE(x) returns "abc".
```

11.11.4 COPY(exp)

While $ERASE(x, exp)$ explicitly specifies which trace handles to remove from the input expression exp, the operation $COPY(exp)$ replicates exp with a copy that has no trace handles at all (i.e., all trace handles are removed). For example, if $input = Assign\#(x, y)$, where both x and y are trace handles with values "var" and 4 respectively, then

```
COPY(input) returns Assign#("var", 4)
```

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

Here v1,v2, ..., vm is a tuple of trace handle names. This operation saves the current value of each trace handle v1,v2,...,vm so that the values of these trace handles can be restored later. After a sequence of transformations to the trace handles are finished and the results output to an external file, the original values of the trace handles can be restored so that a new sequence of transformations can start afresh. This operation therefore allows different transformations to be applied independently to a single input code.

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

Here v1,v2, ..., vm is a tuple of trace handle names. This operation restores the last value saved for the trace handles $v1, ..., vm$. 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.

11.12 Transformation Operations

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

11.12.1 DUPLICATE(c1,c2,input)

Here $c1$ is a single expression, $c2$ is a list of expressions, and *input* is the 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 *input* duplicates 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
```

11.12.2 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")
```

11.12.3 REBUILD(exp)

This operation takes a single POET expression *exp* and returns the result of rebuilding *exp*. Here the rebuilding process will invoke the *rebuild* attribute defined for each code template (see Section 5.4) and will eliminate obvious redundancies (e.g., empty strings) in *exp*.

11.12.4 REPLACE(c1,c2,input)

Here *input* is the input computation to transform, $c1$ is an expression embedded within *input*, and $c2$ is the new expression to replace $c1$, This operation replaces all occurrences of the code fragment $c1$ in *input* with $c2$. For example,

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

11.12.5 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 within *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))
```

11.13 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 12

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.

12.1 Single Statements

12.1.1 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.

12.1.2 The RETURN statement

The syntax for the RETURN statement is

```
RETURN <exp> ;
```

The RETURN statement must be inside a $xform$ routine definition (a runtime error is raised otherwise). When being evaluated, it exits the $xform$ routine with the result of <exp> as the result of the $xform$ routine call. The RETURN statement is provided to allow convenient early returns from a function call.

12.1.3 Statement Block

The syntax for a statement block is

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

Here $stmt1, stmt2, ..., stmtm$ 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 $stmtm$.

12.2 Conditionals

12.2.1 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.

12.2.2 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 specifications as defined in Section 9.2, 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 specification 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 9.2.

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> } ]
```

12.3 Loops

12.3.1 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 != ""; p_input = TAIL(p_input)) {
    PRINT ("seeing element: " HEAD(p_input));
}
```

12.3.2 The *foreach* Loop

The syntax of the *foreach* loop is

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

Here <exp> is the an arbitrary expression, <pattern> is a pattern specification as defined in Section 9.2, <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 specification <pattern>. If any pattern matching succeeds for a code fragment *subexp* in <exp>, it will evaluate the <body> statement and the <succ> expression. If <succ> evaluates to true (non-zero), the current *subexp* will be skipped, and the code fragment following *subexp* will be traversed next; otherwise, the foreach loop will continue traversing the *subexp* expression 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> specification needs to contain local variables that will be 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 (input : (curLoop = Loop) : TRUE)
{
  PRINT ("found a loop: " curLoop);
}
```

Note that the expression *curLoop* = *Loop* must be enclosed inside a pair of parentheses because the assignment operator has lower precedence than the : operator. The following loop collects all the loop nests within an *input* computation.

```
loopNests = "";
foreach (input : (curNest = Nest) : FALSE)
   loopNests = BuildList(curNest, loopNests);
```

Here because loop nests may be inside each other, the <succ> parameter of the foreach loop is set to *FALSE* so that the pattern matching will continue inside already located loop nests. Note that 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 11.12) after a foreach loop has finished, as the transformation operations may disrupt the traversal by the *foreach* loop.

12.3.3 The *foreach_r* Loop

The *foreach_r* Loop has the following syntax.

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

The *foreach_r* loop 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 traversing order allows the relevant information to be gathered and saved with more flexibility. For example, the following code

```
loopNests = "";
foreach_r (input : (curNest = Nest) : FALSE)
   loopNests = BuildList(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 12.3.2 saves all the loop nests in the reverse order of their appearances in *input*.

12.3.4 The BREAK and CONTINUE statements

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

```
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 "/>"
    | "<" "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
    |  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
```

```
    | "if" "(" exp ")" stmt  |  "if" "(" exp ")" stmt "else" stmt
    | "switch" "(" exp ")" "{" cases "}"
    | "for" "(" exp ";" exp ";" exp ")" stmt
    | "foreach" "(" exp ":" patternSpec ":" exp ")" stmt
    | "foreach_r" "(" exp ":" patternSpec ":" exp ")" stmt
    | "CONTINUE"  |  "BREAK"

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

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