

POET Reference Manual

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

October 14, 2008

Front Matter

POET is a dynamic scripting language designed to apply transformations to source code written in an arbitrary programming language. POET was originally designed for the purpose of parameterizing performance optimizations so that the optimized code may be empirically tuned to attain the best performance on a wide variety of different HPC platforms. The use of POET, however, is not limited to code optimizations. You can use POET to easily read in any structured input, extract information from or apply transformations to the input, and then output the result. Check other tutorials for existing examples and applications for the POET language. If you are a new user of the POET language, please send us a brief description of your project and what roles POET has played in your project. We appreciate your input and any feedback you may have in the design and implementation of POET. The POET language was designed and implemented by Dr. Qing Yi at the University of Texas at San Antonio. All questions and feedbacks should be directed to her at qingyi@cs.utsa.edu.

The current release of POET includes five directories: *src*, *lib*, *doc*, *test*, and *examples*. The *src* directory contains C/C++/Yacc/Lex code for the POET language interpreter; the *lib* directory contains the existing POET libraries that include an extensive collection of language descriptions and program transformation routines useful both for software optimization and evolution; the *doc* directory contains documentations about the POET language; the *test* directory contains some example applications of the POET code; and the *examples* directory contains some stand-alone POET code that serve as examples in POET tutorials.

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

Table Of Content	2
1 Building and Using POET	7
1.1 Building POET From Distribution	7
1.2 Building POET From CVS	7
1.3 Directory Structure of POET Source Code Distribution	7
1.4 Using POET	8
2 Tutorials and Examples	9
2.1 The Identity Translator	9
2.1.1 The <i>parameter</i> Declaration	9
2.1.2 The <i>input</i> command	10
2.1.3 The <i>output</i> command	10
2.2 The String Translator	10
2.2.1 The <i>eval</i> command	11
2.3 The Translator Driver	12
2.4 The Language Translator	13
3 Language Overview	15
3.1 Comments	16
3.2 Language Specialization	16
3.3 Transformation Routines	17
3.4 Driver Scripts	18
3.4.1 Input Specification	19
3.4.2 Evaluation and Output specifications	20
3.5 Summary	22
4 Notations and Concepts	23
5 POET Expressions	25
5.1 Atomic Values	25
5.1.1 Integers	26
5.1.2 Strings	26
5.2 Compound Data Structures	26
5.2.1 Lists	26
5.2.2 Tuples	27
5.2.3 Mapping Tables	27
5.2.4 Code Templates	28

5.3	<i>xform</i> routine handles	28
5.4	Type Specifications	28
5.4.1	The <i>INT</i> specifier	29
5.4.2	The <i>STRING</i> specifier	29
5.4.3	The <i>VAR</i> specifier	29
5.4.4	The <i>CODE</i> Specifier	29
5.4.5	The <i>XFORM</i> Specifier	29
5.4.6	The <i>TUPLE</i> Specifier	29
5.4.7	The ... Specifier	29
5.4.8	The .. Specifier	30
5.4.9	The <i>MAP</i> Specifier	30
5.4.10	The Special ANY (.) Specifier	30
5.4.11	Code Template Names	30
5.5	Variables	30
5.5.1	Local Variables	30
5.5.2	Global Variables	31
5.5.3	Dynamic Variables	31
5.5.4	Parameters	31
5.5.5	Trace Handles	31
5.6	Invoking Transformation Routines	31
5.7	Invoking Built-in Operations	32
5.8	Delaying the evaluation of POET Expressions	32
6	Built-in POET operations	33
6.1	Debugging Operations	33
6.1.1	The PRINT operator	33
6.1.2	The DEBUG operator	33
6.1.3	The ERROR Operator	34
6.2	Generic comparison of values	34
6.2.1	The == operator	34
6.2.2	The != operator	34
6.2.3	Integer and String comparison	34
6.3	Integer Operations	34
6.3.1	Integer arithmetics	34
6.3.2	Boolean operations	34
6.4	String operations	34
6.4.1	The string concatenation ^ operator	34
6.4.2	The <i>SPLIT</i> operator	35
6.5	List operations	35
6.5.1	List construction	35
6.5.2	The :: Operator	35
6.5.3	List Access (The <i>car/HEAD</i> , <i>cdr/TAIL</i> , and <i>LEN</i> operators)	35
6.6	Tuple operations	36
6.6.1	Tuple Construction (the “,” operator)	36
6.6.2	Tuple Access (The [] and <i>LEN</i> operators)	36
6.7	Table operations	36
6.7.1	Table Construction (the <i>MAP</i> Operator)	36

6.7.2	Table Access (the [], <i>LEN</i> operators)	36
6.8	Code Template Operations	37
6.8.1	Code Template Object Construction (the # operator)	37
6.8.2	Code Template Access (the [] operator)	37
6.9	Type operations (the operator)	37
6.10	Pattern Matching And Assignment	37
6.10.1	The Pattern Matching Operator (the “:” operator)	38
6.10.2	Un-initializing Variables (The <i>CLEAR</i> operator)	39
6.10.3	Variable assignment (the “=” operator)	40
6.11	Parsing and Type Conversion	40
6.11.1	The => and ==> Operators	40
6.11.2	Parsing Annotations (The =>, <i>BEGIN</i> and <i>END</i> operators)	43
6.11.3	Type Conversion Between Expressions	43
6.12	Delaying Evaluation of Instructions (the <i>DELAY</i> and <i>APPLY</i> operators)	44
6.12.1	The <i>DELAY</i> operator	44
6.12.2	The <i>APPLY</i> operator	44
6.13	Trace Operations	44
6.13.1	<i>TRACE</i> (x, exp)	45
6.13.2	<i>INSERT</i> (x, exp)	45
6.13.3	<i>ERASE</i> (x, exp)	45
6.13.4	<i>COPY</i> (exp)	45
6.13.5	<i>SAVE</i> (v1,v2,...,vm)	45
6.13.6	<i>RESTORE</i> (v1, v2, ..., vm)	46
6.14	Transformation Operations	46
6.14.1	<i>DUPLICATE</i> (c1,c2,input)	46
6.14.2	<i>PERMUTE</i> (config,input)	46
6.14.3	<i>REBUILD</i> (exp)	46
6.14.4	<i>REPLACE</i> (c1,c2,input)	46
6.14.5	<i>REPLACE</i> (config, input)	47
6.15	The Conditional Expression (The “?:” operator)	47
7	POET Statements	49
7.1	Single Statements	49
7.1.1	The Expression statement	49
7.1.2	The <i>RETURN</i> statement	49
7.1.3	Statement Block	50
7.2	Conditionals	50
7.2.1	The If-else Statement	50
7.2.2	The Switch Statement	50
7.3	Loops	51
7.3.1	The for Loop	51
7.3.2	The foreach Loop	51
7.3.3	The foreach_r Loop	52
7.3.4	The <i>BREAK</i> and <i>CONTINUE</i> statements	52

8 Global Declarations and Commands	53
8.1 Global Macro Definitions and Reconfiguring POET	53
8.1.1 The TOKEN Macro	54
8.1.2 The PARSE Macro	54
8.1.3 The UNPARSE macro	55
8.1.4 The PARSE_BOP Macro	55
8.2 Code Template and Parsing/Unparsing Definitions	55
8.3 Xform Routine Declarations	59
8.4 Global Variable Declarations	60
8.4.1 Trace Handle Declarations	60
8.4.2 Parameter Declarations	60
8.5 Input Commands	62
8.6 Evaluation Commands	63
8.7 Output Commands	63
Append A. Context-free grammar of the POET language	65

Chapter 1

Building and Using POET

1.1 Building POET From Distribution

Once having downloaded a POET distribution, say `poet-1.02.01.tar.gz`, you can build POET using the following commands.

```
> tar -zxf poet-1.02.01.tar.gz
> cd poet-1.02.01
> ./configure --prefix=<your install directory>
> make      (make and then run the POET interpreter on a few tests)
> make check ( test whether the POET interpreter works correctly)
> make install (install POET interpreter and libraries on your local machine)
```

1.2 Building POET From CVS

If you are a developer of POET and have access to POET CVS, 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 Code Distribution

The POET language is implemented through interpretation, and the interpreter is implemented in C++. The source code 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 the predefined library of language syntax definitions and transformation routines.

- The *test* directory, which contains some applications of POET and is used to test correctness of POET works.
- The *example* directory, which contains some examples used as bases for various POET tutorials.
- The *doc* directory, which contains the manual for using POET (this document).

1.4 Using POET

The POET package will build a language interpreter named *pcg* (a binary executable) and will provide a collection of predefined library of POET code templates and transformation routines. After running *make install*, the binary interpreter *pcg* is copied to directory `<your install directory>/bin`, and the POET libraries is 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 transformation routine invocations
-md:    allow code template syntax to be multiply defined (overwritten)

```

Chapter 2

Tutorials and Examples

This chapter goes over some of the example POET programs in the `POET/examples` directory.

2.1 The Identity Translator

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

```
<parameter inputFile type=STRING default="" message="input file name" />
<parameter outputFile type=STRING default="" message="output file name" />
<parameter inputLang type=STRING default="" message="where to read syntax definitions"/>

<input cond=(inputLang=="") from=inputFile annot=0 to=inputCode/>
<input cond=(inputLang!="") from=inputFile syntax=inputLang to=inputCode/>

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

Three different commands, *parameter*, *input*, and *output*, are used in the above code. The following explains each of them.

2.1.1 The *parameter* Declaration

The first three lines are *parameter* declarations, where each *parameter* keyword declares a global variable whose value can be modified through command-line options. For example,

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

declares a global variable named *inputFile*, the variable should contain values of type *STRING*, the default value of the variable is empty string (“”), and the meaning of the variable is to store “input file name”. Similarly, the *outputFile* is a *STRING* variable that contains the output file name, and the *inputLang* variable contains a *STRING* value that indicates the name of the file that contains language syntax definitions of the input code. The command-line to invoke the Identity translator therefore is the following (assuming the current directory is *POET/examples*).

```
> ../src/pcg [-pinputFile=<myInputFile>] [-poutputFile=<myOutputFile>]
           [-pinputLang=<myInputLang>] IdentityTranslator.pt
```

Here each command-line option `-p <var>=<value>` specifies a new value for a global variable used in the *IdentityTranslator* File. The capability allows the *IdentityTranslator.pt* program to read input from an arbitrary file and write output to another arbitrary file. All the command-line options are optional — when no value is given for a global parameter, the parameter contains its default value (declared using the *default* keyword within the *parameter* declaration).

2.1.2 The *input* command

Following the three *parameter* declarations are two *input* commands. Each *input* command has a *cond* specification, which enforces that the first *input* command is evaluated only when *inputLang* is empty (no language syntax is defined for the input code), and the second *input* command is evaluated only when *inputLang* is not empty (a language syntax file is specified). In either case, the *input* commands opens the input file (name specified by the *inputFile* variable), read in the input code (the content of the input file), and save the input code as value of the global variable *inputCode*. If *inputLang* == "", the input code will be read in as a sequence of integer/string tokens, and all annotations in the code will be ignored (enforced by the *annot* = 0 specification); otherwise, the input code is parsed based on the grammar definition contained in file *inputLang* (enforced by the *syntax* = *inputLang* specification), and if the input code contains POET annotations, the annotations will be used to help the parsing process.

2.1.3 The *output* command

After reading the input code, the last two *output* commands write the input code (contained in global variable *inputCode*) to an external file whose name specified by the *outputFile* variable. Each *output* command similarly has a *cond* specification which enforces that the first *output* command is evaluated only when *inputLang* is empty (no language syntax is defined for the input code), and the second *output* command is evaluated only when *inputLang* is not empty (a language syntax file is specified). In either case, the *output* commands opens the output file (name specified by the *outputFile* variable) and then write the content of the global variable *inputCode* into the external file. If *inputLang* == "", the input code will be output as a sequence of integer/string tokens; otherwise, the input code is unparsed based on the grammar definition contained in file *inputLang* (enforced by the *syntax* = *inputLang* specification).

NOTE1: when the *inputFile* variable contains an empty string, the input code will be read from the standard input (the user will be prompted to type in the input code); when the *outputFile* variable contains an empty string, the input code will be written to the standard output (the screen).

NOTE2: as illustrated by the *input* code variable, global variables in POET can be used without being declared. While the global *parameter* variables can have their types specified (declared), the specification is optional (not required).

2.2 The String Translator

In general, after reading in the input code, we would like to apply some transformations and then output the transformed code. The following POET code (POET/examples/StringTranslator.pt)

tries to substitute a pre-defined set of strings in the input code with other strings.

```
< parameter inputFile type=STRING default="" message="input file name" />
< parameter outputFile type=STRING default="" message="output file name" />
< parameter inputString type=(STRING...)|STRING default=""
    message="string to replace" />
< parameter outputString type=(STRING...)|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);}
return
/>

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

The first four lines are again *parameter* declarations. Here in addition to the *inputFile* and *outputFile* variables, two more variables (*inputString* and *outputString*) are declared as POET *parameters*. Note that the *inputString* and *outputString* variables have a different type — their types are defined as $(STRING...)|STRING$, which specifies that they could contain a single string as value (specified by the *STRING* type specifier), or contain a list of strings as value (specified by the $(STRING...)$ specifier). The `|` operator is a type union operator, which connects multiple types and allows a value to belong to any one of many alternative types.

Following the *parameter* declarations, the *StringTranslator* program reads in the input code from *inputFile* and write the output (contained in the *return* variable) to *outputFile* in the same way as in the identity translator described in Section 2.1. The only difference is that instead of writing the *inputCode* as result, the *StringTranslator* writes the value of *return* as output. Further, an *eval* command is used to evaluate the value of the *return* variable.

2.2.1 The *eval* command

The *eval* command is used to evaluate expressions and statements at the global scope in order to compute the final output. **All POET expressions must be embedded within an eval command to be evaluated at the global scope.**

In the *StringTranslator*, embedded in the *eval* command are two statements followed by an expression. The first statement is an assignment statement

```
return = inputCode;
```

Here, a new global variable named *return* is created and assigned with the value of *inputCode*. The assignment expression *return = inputCode* returns the value of *inputCode* as result. However, by appending a “;” (semicolon) at the end, the expression is converted to a statement and the value of expression is ignored.

The second statement in the *eval* command is a *for* loop, which has identical syntax as the *for* loop in C/C++. This 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. After each iteration of the *for* loop, both *p_input* and *p_output* are modified with the *TAIL* of their original values.

The last expression in the *eval* command is simply the variable *return*, which indicates that the value of the *return* variable should be the final value (result) of the *eval* command.

NOTE: The *HEAD* and *TAIL* keywords are operators that supports the extraction of values from a list (see section 6.5). Specifically, *HEAD* returns the first element in the list, *TAIL* returns the rest 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 an empty string (“”).

2.3 The Translator Driver

The following POET code (POET/examples/TranslatorDriver.pt) illustrates the structure of a more general driver for building code transformation tools.

```
include utils.incl

<parameter inputFile message="input file name"/>
<parameter outputFile message="output file name"/>
<parameter xformFile message="xform specification file name"/>

<xform TransformCode />

<input from=(inputFile) syntax=(xformFile) annot=0 to=inputCode/>

<eval resultCode=TransformCode(inputCode); />

<output to=(outputFile) syntax=(xformFile) from=resultCode/>
```

The first line is an include command, which reads the *utils.incl* file (located in POET/lib directory) before reading any instructions in the current file. The following three lines are again *parameter* declarations. Here in addition to the *inputFile* and *outputFile* variables, a variable *xformFile* is used to store the name of an external file that defines the transformations to apply to the input code.

Following the *parameter* declarations is a *xform* declaration that declares *TransformCode* as the name of a transformation routine. The declaration ensures that *TransformCode* will be treated as a routine name instead of a regular global variable. The actual routine needs to be defined in the *xformFile*.

Following the declaration of *TransformCode*, an input commands makes sure that the *xformFile* is read and processed before the *eval* command, which invokes the *TransformCode* routine on the *inputCode* and saves the result of invocation to *resultCode*. Finally, the content of *resultCode* is written to *outputFile*.

NOTE1: The include commands must be at the start of a POET program to be recognized. If appearing in mid of a program, they will be ignored (treated as source code of an input language).

NOTE2: All the commands in a POET program are first read in and saved in an internal representation before being evaluated. Except for the *include* commands at the start of the program, the *input* commands in the mid of a program are not evaluated until all the other commands and declarations have been read. Because of this, although the name *TransformCode* is defined in *xformFile*, it needs to be declared in *TranslatorDriver* as a transformation routine name to avoid being treated as a regular global variable.

2.4 The Language Translator

The following POET code (POET/examples/TranslatorDriver.pt) illustrates the structure of a general driver for building translators between different languages.

```
< parameter inputFile type=STRING default="" message="input file name" />
< parameter inputLang type=STRING default=""
      message="file name for input language syntax" />
< parameter outputFile type=STRING default="" message="output file name" />
< parameter outputLang type=STRING default=""
      message="file name for output language syntax" />
< parameter xformFile type=STRING default="" message="translation file name" />
< parameter extraFile type=STRING default=""
      message="file name to specify additional information" />

<code inputType/>
<code extraType/>
<xform TranslateCode extra=""/>

<input syntax=inputLang parse=POET/>
<input syntax=outputLang parse=POET/>
<input from=xformFile cond=(xformFile!="") parse=POET />
<input from=inputFile to=inputCode cond=(xformFile!="")
      syntax=(inputLang xformFile) parse=inputType/>
<input from=inputFile to=inputCode cond=(xformFile=="") />
<input from=extraFile cond=(extraFile!="") to=extraCode
      syntax=(outputLang xformFile) parse=extraType/>

<eval resultCode=((xformFile=="")? inputCode
      : (TranslateCode[extra=extraCode](inputCode))); />

<output cond=(xformFile!="") to=outputFile from=resultCode syntax=outputLang/>
<output cond=(xformFile=="") to=outputFile from=resultCode/>
```

The first six lines are *parameter* declarations that define the values of *inputFile*, *outputFile*, *inputLang* (file name that contains the syntax definitions for the input language), *outputLang* (file name that contains the syntax definitions for the output language), *xformFile* (file name that contains transformation routines that translation from the input language to the output language), and *extraFile* (file name that contains any additional information required for the translation).

Following the *parameter* declarations are two code template declarations that declare *inputType* and *extraType* as code template names. These code template names are defined in the *inputLang* and *extraLang* files respectively. The *TranslateCode* is declared as the name of a transformation routine (defined in file *xformFile*). The declarations ensure that the given variables will be parsed properly (not as regular global variables).

Following the *code* and *xform* declarations, six input commands are used to read in and process the specified files. Then, an *eval* command invokes the *TranslateCode* routine (which has an *extra* parameter set to be the code template *extraCode*). The result of language translation is saved in *resultCode*. Finally, the content of *resultCode* is written to *outputFile*.

NOTE1: Code template names are type names and can be used to both specify the concrete syntax of different languages and to specify the data structure that should be used to store the input code. For more details, see Section 5.2.4.

NOTE2: Transformation routines can define optional parameters whose values are re-configured inside a pair of [] when the routine is being invoked. For more details, see Section 5.3.

Chapter 3

Language Overview

The POET language is designed for building parameterized source-to-source transformations to programs written in arbitrary programming languages. Each POET translator typically contains the following components.

1. Language specialization. In order to process various kinds of input code, the syntax structure of the input languages must be defined in POET using a collection of *code templates*, which are essentially type declarations for the dynamic data structures used to internally store the input code. Each code template defines the concrete syntax of a language concept, e.g., a loop nest or a function definition. They are used both in the parsing process to recognize the structure of the input code and in the code generation process to produce output code. Note that you only need to define code templates for the language components that you are interested in applying transformations to; the other syntax definitions can be ignored and will be treated simply as strings.
2. Transformation routines. Each transformation routine in POET is a function which takes a sequence of input parameters (e.g., the input code fragment) and returns a result (e.g., the transformed code). POET transformation routines can make recursive function calls to each other, use loops to iterate over a body of computation, and use the *foreach* operator to apply transformations based on pattern matching. In summary, POET provides complete programming support to define arbitrary code transformations in these routines.
3. Driver scripts. POET transformations are not based on pattern-matching by default. You must explicitly define what is your input code and what transformations you want to apply to which pieces of code. The reason of this design is to enable the finest control over how to apply different transformations to different code.

The POET release includes an extensive library of transformation routines and language specializations which specialize the library for different programming languages such as C, FORTRAN, or Assembly. Both transformation libraries (named as “.pt” files) and the language specializations (named as “.code” files) are in POET/lib directory. The POET language interpreter takes as input one or more input files and a collection of parameter values from command-line. It invokes the language specializations to help parse input codes, and invokes transformation routines to produce desired output codes.

```

<* Base Expressions *>
<define ExpBase  INT|FunctionCall|ArrayRef|Name />
<define BopType ((("&&"|"||") ("=="|">="|"<="|!="|">"|"<")
                  ("+"|"-" ) ("*"|"/" ) "." ) />

<* A wrapper for all expressions *>
<code Exp pars=(content)
      parse=ParseExp[itemType=ExpBase; bopType=BopType]
      rebuild=IgnoreCode >
@content@
</code>

<* for loop *>
<code Loop pars=(i:Name, start:Exp, stop:Exp, step:Exp, reverse:(0,1)) >
@if (reverse == 0) {@for (@i@=@start@; @i@<@stop@; @i@+=@step@)@}
else {@ for (@i@=@start@; @i@>@stop@; @i@-=@step@)@}@
</code>

<* a loop nest *>
<code Nest pars=(loop: Loop, body : StmtList)>
@loop@ {
    @body@
}
</code>

```

Figure 3.1: Example code template definitions for the C language

3.1 Comments

In POET, comments can have two different formats.

```

<* my comments *>
or
<<* my comments

```

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

3.2 Language Specialization

In POET, both the parsing of input code and the generation of output code is based on a collection of “code templates” that defines the structure of the input or output language. Figure 3.1 shows some examples of the code template definitions for a subset of the C language. In the following, we use “*source language*” to denote the input/output language that we are trying to specialize POET for.

In Figure 3.1, the *Exp* declaration defines a code template named *Exp*. The code template has a single parameter named *content*. It is parsed by invoking a parsing function named *ParseExp* (the *itemType* and *bopType* parameters of the *ParseExp* function are reconfigured to suit the C language), and it can be re-constructed by calling a transformation routine *IgnoreCode*, which

ignores the expression wrapper and returns the content directly. The body of the code template defines the concrete syntax of the code template in the *source language* (e.g., C/C++/Java). In this case, the syntax of an expression is merely the value of its *content* parameter. Similarly the *Nest* declaration defines a code template named *Nest*. The code template has two parameters, *loop* and *body* respectively, and the concrete syntax of the code template is a loop nest formed by concatenating the values of *loop* and *body* in C syntax. Inside the code template body, the parameter names are wrapped inside pairs of `<` symbols to signal that they are part of the POET language instead of strings in the source language. The concrete syntax of the parameters *loop* and *body* are defined by code templates *Loop* and *StmtList* respectively.

In summary, each code template definition starts with its name followed with a number of attribute definitions. Each code template may have a number of special attributes. For example, the *pars* attribute defines the parameters of the code template; the *rebuild* attribute defines how to reconstruct the code template when it is used as argument to a *REBUILD* operation; and the *parse* attribute defines a parsing function for recognizing the code template from an input code. Each code template can have a body, which is wrapped inside the pair of `<code ...>` and `</code>` tags. The body of each code template defines a concrete syntax in the source language in terms of how to combine the template parameters. The entire code template body is treated as source strings in the source language (e.g., C, C++, or Java) unless they are wrapped inside a pair of symbols, which signals a POET expression embedded inside the source language.

Code templates may be declared without a body. These templates merely serve as definitions of special-purpose data structures and cannot be used in parsing/code generation unless a body is subsequently defined. Each code template may additionally declare optional attributes that represent additional semantic properties of the template objects. Code template as shown in Figure 3.1 can be used to define both the concrete and abstract syntax of the input source language. Specifically, they can be used to parse the input source code, to build an abstract syntax tree (AST) of the input code, to allow type safe transformations to the AST, and to emit the final transformed code. Each code template conveys a special meaning that could be expressed in different concrete syntax, for example, a loop could be expressed as “for (int i = 0; i < 100; ++i)” in C syntax or expressed as “DO I = 0,100” in Fortran syntax.

3.3 Transformation Routines

A POET transformation routine is essentially a function that typically takes internal representations of input codes and returns transformed code as result. Note that a POET transformation routine operates on the code template (known in the compiler world as Abstract-Syntax Tree) representation of the input code without knowing what source language the code templates are built from or what source language they will be output to. These transformation routines are therefore generic in the sense that they can be invoked to operate on code parsed from arbitrary source languages. To switch to another source language, the user often only needs to switch syntax description files.

Figure 3.2 shows a few example transformation routines from the POET/lib/utills.incl file. As shown in the figure, POET uses the keyword *xform* to define a transformation routine. Each *xform* routine uses the *pars* attribute to define a sequence of input parameters, uses additional attributes to define optional parameters of the transformation (each optional parameter has a default value), and uses the *output* attribute to define a tuple of return values. More than one results may be returned when the *output* attribute is defined.

```

<* Build a list, skipping the components that are empty strings *>
<xform BuildList pars=(first,rest)>
  (first == "")? rest
: (rest == "")? first
: first::rest
</xform>

<* Parse from $input$ a sublist of tokens. Stop if encountering a token that
  matches the $stop$ pattern or that does not match the $continue$ pattern *>
<xform ParseList pars=(input) stop="" continue="" output=(result, leftOver) >
if ((first = car(input)) != "")
{
  if (first : stop) { ("", input) }
  else if (continue == "" || first : continue)
  {
    resOfRest = ParseList(cdr(input));
    (BuildList(first,resOfRest[0]), resOfRest[1])
  }
  else { ("",input) }
}
else if (input : stop) { ("",input) }
else if (continue == "" || first : continue) { (input, "") }
else { ("",input)}
</xform>

```

Figure 3.2: Example transformation routines

Note that in POET, unless a *RETURN* statement is encountered, the last expression evaluated within a transformation routine is returned as the result of the routine. In Figure 3.2, the *BuildList* routine takes two parameters, *first* and *rest*, and composes them into a *list* (a built-in data structure) as result. The routine uses the “*cond? true_branch : false_branch*” expression, which has the same syntax and semantics as the corresponding C conditional expression, to check whether *first* or *rest* is empty. If either one of the input parameters is empty (equals to the empty string), the non-empty component is returned as result; otherwise, a list that contains both components is returned.

The second *ParseList* routine in Figure 3.2 is a *xform* routine that can be invoked to parse leading list of tokens from an input. This routine takes a single parameter, the *input* code, and optionally takes two additional parameters (both are patterns): *stop* which defines at what point to stop parsing, and *continue*, which defines whether to continue parsing. The routine returns two values, the result from parsing the leading tokens in *input* and the rest of the tokens that were not considered part of the list. This routine uses if-conditionals to perform different operations on different forms the input code.

3.4 Driver Scripts

Both the code template and the transformation routine definitions are provided to support the construction of general-purpose transformations that can operate on code in arbitrary languages. In order to apply the code templates and transformation routines to an input computation, we need

```

<input to=gemm syntax="Cfront.code" parse=FunctionDefn>
void ATL_USERMM(const int M, const int N, const int K,
    const double alpha, const double *A, const int lda,
    const double *B, const int ldb, const double beta,
    double *C, const int ldc)
{
    int i,j,l; //@=>[gemmDecl=Stmt]
    for (j = 0; j < N; j += 1) //@ =>[loopJ=Loop] BEGIN(gemmBody) BEGIN(nest3)
    {
        for (i = 0; i < M; i += 1) //@=>[loopI=Loop] BEGIN(nest2)
        {
            C[j*ldc+i] = beta * C[j*ldc+i]; //@ =>Stmt
            for (l = 0; l < K; l +=1) //@=>[loopL=Loop] BEGIN(nest1)
            {
                C[j*ldc+i] += alpha * A[i*lda+l]*B[j*ldb+l]; //@ =>Stmt
            } //@END(nest1:Nest)
        } //@END(nest2:Nest)
    } //@END(nest3:Nest) END(gemmBody:Nest)
}
</input>

```

Figure 3.3: Example input specification

a driver script which invokes the desired code templates and transformation routines accordingly.

Each POET driver script typically includes a sequence of global instructions, including

1. input instructions, which specify where to find the input computation and how to parse it;
2. evaluation instructions, which specify how to apply various parameterized transformations to the input code; and
3. output instructions, which define where to output resulting codes.

As example, Figure 3.3 shows an example input specification for *dgemm*, a matrix multiplication kernel. Figure 3.4 shows some of the evaluation and output instructions for the kernel.

3.4.1 Input Specification

A POET input specification, as illustrated in Figure 3.3, is used to define the input computation to transform. The input code defined in the specification will be parsed and translated into an internal code template representation based on the included language specialization. As shown in Figure 3.3, each input specification is wrapped inside a pair of *input* tags, and the parsed internal representation is stored in the global variable defined as the value of the *to* attribute (in Figure 3.3, the parsed code is stored in a global variable named *gemm*), the language specialization is defined using the *syntax* attribute, and the code template type of *gemm* is defined using the *type* attribute. The source of the input code could then follow (the input code could alternatively be read from a separate file, see Section 8.5). In Figure 3.3, the source of the input code is annotated with information to help parse the matrix computation and to define values of global trace variables (e.g., *gemmDecl*, *loopJ*, *nest1*, *nest2*, *nest3*). Many of the annotations could be optionally omitted. In general, the more annotations are included in the source input, the shorter time it will take the POET interpreter to parse the input code. The POET input parser uses the recursive descent

algorithm to decompose the input code into code templates — it is designed to be highly flexible for parsing arbitrary languages but is not designed to be particularly efficient.

As shown in Figure 3.3, each POET annotation either starts with “//@” and lasts until the line break, or starts with “/*@” and ends with “@*/”. The special syntax allows programmers to embed these annotations as comments in C/C++ code, so that the source input is readily accessible for other uses.

POET supports both single-line and nested annotations. A single-line annotation starts from the start of the current line and ends with an annotation in the format “=> [x = T]” or simply “=> T”, where x is the name of a global variable that will be used to store the result of parsing the code fragment, and T is the name of the code template that should be used to parse the annotated code. For example, in Figure 3.3, the annotation “int i, j, l; //@=>[gemmDecl=Stmt]” indicates that “int i, j, l;” is a statement that should be parsed using the *Stmt* template, and the result should be stored in the global variable *gemmDecl*. The definitions for both *Exp* and *Stmt* can be found in Figure 3.1.

In contrast to single-line annotations, nested annotations in POET are used to help parse nested language constructs such as functions and loop nests, which include other code fragments as components. Each nested POET annotation starts with “BEGIN(x)”, where x is the variable that should be used to store the compound code template, and ends with “END(x:T)”, where T is the name of the code template that should be used to parse the annotated code. In Figure 3.3, the annotation “for (l = 0; l < K; l += 1) //@ =>loopL:Loop BEGIN(nest1) ... END(nest1)” is a nested annotation which starts with the *for* loop (a singly annotated fragment stored in *loopL*) and ends after parsing the loop body *stmt1*. Other nested annotations in Figure 3.3 include code fragments stored in *gemmBody*, *nest3*, *nest2*, etc.

The input specification as illustrated in Figure 3.3 is necessary so that the POET can parse the input computation correctly without being source language specific. Because each code template used in parsing the input code can alternatively be defined using a different programming language, POET can be easily specialized to optimize code written in different source languages such as C or FORTRAN without requiring a parser for each language. We have designed the annotation syntax to minimize intrusion to the source code, so that if written in C, POET annotations can be treated merely as comments, and the source code can be compiled with a regular C compiler without requiring any additional bookkeeping.

3.4.2 Evaluation and Output specifications

After the input specification is processed by a POET language interpreter and translated into an internal representation, the evaluation specification can then invoke POET transformation routines to operate on the input code, and the transformed code can be output using output specifications. Figure 3.4 illustrates applying the transformations for optimizing the *dgemm* kernel in Figure 3.3. These specifications include a few POET declarations: *parameter*, *trace*, *eval*, and *output*,

In POET, each keyword *parameter* declares a number of global variables that can be re-configured through command-line options. For example, the first declaration declares a reconfigurable parameter named *NB* which must have a value greater than 1 (1... specifies a range from 1 to an arbitrary number), the default values is 62, and the meaning of the parameter is to define “the blocking size of the N dimension”. Similarly, the declaration of *nu* declares that the *nu* parameter must have a value that satisfy $1 \leq nu \leq NB$; it has 1 as its default value; and its semantics are to define the “Unroll and Jam factor for the N loop”.

Similar to the *parameter* declarations, each keyword *trace* declares global variables that can

```

<parameter NB type=1.._ default=62 message="Blocking size for the N dimension of the matrices" />
.....
<parameter nu type=1..NB default=1 message="Unroll and Jam factor for the N loop"/>

<trace gemm,gemmDecl,gemmBody,nest3,loopJ,nest2,loopI,nest1,loopL/>

<eval codegen_gemm_kernel = DELAY {
  < * APPLY A_ScalarRepl; * >
    dim = ( ArrayDim#("i","i",loopI[Loop.step]) ArrayDim#("l","l",loopL[Loop.step]));
    TRACE(Arepl,
      ScalarRepl[init_loc=nest1[Nest.body]; elem_type=fstype;
        trace_repl=Arepl; trace_decl=gemmDecl; trace=nest1]
      ("a_buf",alphaA, "i" * lda + "l", dim, nest1[Nest.body]));
    .....
    gemm
  }/>

<eval INSERT(gemm,gemm); APPLY(codegen_gemm_kernel)/>
<output to="dgemm_kernel.c" syntax="Cfront.code" from=gemm />

```

Figure 3.4: Applying transformations for kernel *dgemm*

serve as tracing handles, which means they can be embedded inside a computation to keep track of selected fragments as they go through a sequence of transformations. In Figure 3.4(b), the INSERT operation inserts all the trace handles (declared in the *trace* declaration), *gemm*, *gemmDecl*, *gemmBody*, *nest3*, *nest2*, and *nest1*, into *gemm*, to be embedded within the internal representation of the input code. As various transformations are applied to the input code, the values of these trace handles are replaced with equivalent new code fragments. For example, the input code in Figure 3.4 is optimized by applying 11 different transformations, each transformation can operate on the *trace* variables without worrying about what transformations have already been applied. The tracing capability therefore makes the ordering of different code transformations extremely flexible, and one can easily adjust transformation orders as desired.

Each keyword *eval* in POET serves to specify operations to evaluate. Each *eval* instruction is evaluated in order of their appearance, where at each assignment, the target code fragment is first evaluated and the result is then assigned as the new value of the variable. If the value of a global variable is a code transformation, the evaluation of the transformation can be delayed using the DELAY operation, which packages the code fragment until an APPLY command is invoked, which forces the evaluation of delayed transformations. Figure 3.4 illustrates the definition of a code transformations, *A_ScalarRepl*, which replaces references to matrix *A* with scalar variables. Here the transformation routine, *ScalarRepl*, is defined in the POET transformation library. The invocation to the routine redefines its optional parameters based on the reconfigurable parameters of the driver script.

Finally, the *output* specification in POET defines what code should be output to external files. The *output* declaration in Figure 3.4 specifies that the optimized code from applying *codegen_gemm_kernel* should be output to the file *dgemm_kernel*, and the output syntax is defined in file *Cfront.code*. A number of *output* declarations can be used to output multiple code fragments to different files.

3.5 Summary

In summary, in order to use POET for code transformation/generation, the POET language interpreter requires

- Language specialization, which includes a collection code templates that defines both the concrete and abstract syntax of the source languages that POET needs to parse or generate code.
- Transformation routine definition, which includes a collection of function routines that produce new code based on the input computations.
- Driver definition, which specifies where to find and how to parse an input code (the input specification) as well as how to generate new code as output (the evaluation and output specifications).

The POET release includes a library of predefined language specialization and transformation routine definitions. So in order to use POET, a user in most cases only needs to produce the *driver scripts*.

Chapter 4

Notations and Concepts

The following chapters explain the syntax and semantics of each POET concepts, including expressions, statements, control-flow, global declarations, and global commands. When explaining the language syntax, the following notations will be used.

- `<concept>`, which specifies a concept equivalent to a *non-terminal* in the BNF form of context-free grammars. 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 `{< identifier > [=<exp>]}` indicates that additional variable initializations (separated by “,”) can appear arbitrary times.

Here because characters `[,], {, }` have been given special meanings, they are quoted with “” when these special symbols are themselves part of the language syntax.

The following concepts are used consistently to specify syntax of the POET language.

- `<identifier>`: all variable names.
- `<exp>`: all POET expressions, defined in Chapter 5.
- `<type>`: all POET type specifications, defined in Section 5.4.
- `<parse_spec>`: all POET parse specifications, defined in Section 6.11.
- `<pattern>`: all POET pattern specifications, defined in Section 6.10.

Chapter 5

POET Expressions

While POET expressions must be embedded in global specifications as illustrated in Figures 3.1, 3.2, 3.3, and 3.4, expressions are the building blocks of all evaluations. This chapter explains the syntax and semantics for building POET expressions.

A POET expression 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, code templates, and mapping tables.
- A *xform* routine handle, which is essentially a function pointer in C.
- Type expressions, which specify types of expressions to support the dynamic pattern matching and type conversion of values.
- Variables, which are place holders for expression values. POET variables can be separated into three categories: local, global and dynamic variables, where global variables can additionally be declared as parameters and trace handles (see Section 8.4, 8.4.2, and 8.4.1).
- 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.

POET is a value oriented language. The only expressions that can be modified are variables and mapping tables. Transformations are performed by constructing new values to replace the old ones.

5.1 Atomic Values

POET supports two kinds of atomic values (literals), integers and strings. It does not support floating point values with the assumption that code transformation and analysis do not need floating point evaluations. An extension may be made 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.

5.1.1 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.

5.1.2 Strings

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

POET treats strings as atomic values and does not allow modifications to the contents of strings. It provides a binary operator \wedge to support string concatenation, e.g., “abc” \wedge 3 \wedge “def” returns “abc3def” (note that integer operands are automatically converted to strings before used in the concatenation). 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 n th character of the string. For example (*SPLIT*(1, “abc”) = (“a” “bc”).

5.2 Compound Data Structures

POET supports four kinds of compound data structures: lists, tuples, mapping tables, and code templates.

5.2.1 Lists

A POET list is simply a singly linked list of elements. Due to its flexibility, lists are extensively used in practice to conveniently store information that only needs to be accessed sequentially (one element after another).

A POET list can be composed by simply concatenating elements together. For example, (a “<=” b) produces a list with three elements, a, “<=”, and b. An operator *::* is provided to dynamically extend an existing list. For example, if b is a list, $a :: b$ inserts the content of a into b as the new first element. Because the size of b may be unknown, lists are dynamic data structures that may contain arbitrary numbers of elements. Unless a unparsing format is defined explicitly (see Chapter 6.11), when being unparsed to an external file, the elements in a list are output one after the other without any space, e.g., a list (“a” “+” 3) will be unparsed as “a+3”.

Elements in a list ℓ are accessed through two operations: *HEAD*(ℓ) (or *car*(ℓ)), which returns the first element of ℓ (if ℓ is not a list, it simply returns ℓ); and *TAIL*(ℓ) (or *cdr*(ℓ)), which returns the tail of the list (if ℓ is not a list, it returns empty string). For example, if $\ell = (a \text{ “<=” } d)$

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, $LEN(123) = 3$.

5.2.2 Tuples

Compared to lists, POET tuples can provide random indexed access to their elements. However, all elements within a tuple must be explicitly specified when constructing the tuple, so tuples cannot be built dynamically (e.g., using a loop). Further, no element in a tuple can be changed after the tuple is constructed. Because of the static nature about tuples, they are used to define values with a known structure, such as the tuple of parameters for code templates and *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 by invoking $t[i]$, where i is the index of the element being accessed (like C, the index starts from 0). For example, if $t = (i, 0, “m”, 1)$, then $t[0]$ returns “i”, $t[1]$ returns 0, $t[2]$ returns “m”, and $t[3]$ returns 1.

POET provides the LEN operator to obtain the size of an unknown tuple. For example, $LEN(1, 3, 4, 5) = 4$.

5.2.3 Mapping Tables

POET uses mapping tables as the dynamic data structure to provide efficient random accesses to organized information. A mapping table is internally implemented using C++ STL maps and can be used to efficiently associate two arbitrary types of values. In POET, mapping tables are the only data structure whose internal content can be modified.

The following illustrates how to create and operate on mapping tables.

```

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

```

To create an empty mapping table, use the MAP operator, which takes two parameters: the types of the first and second elements in the table respectively (the $_$ token indicates an arbitrary type). The elements within the table can be accessed using the “[]” operator and modified using

assignments. If an element e is inside the table $amap$, then $amap[e]$ returns the mapping result; 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 be enumerated using the built-in *foreach* statement, where each entry within the table is placed into a pair of undefined variables. For more details about the *foreach* statement and the *PRINT* operation, see Section 7.3.2 and 6.1.1.

5.2.4 Code Templates

Each POET code template is like a struct or class in C/C++ — each code template defines a unique user-defined compound data structure, where the template parameters are data fields within the structure. Compared to C++/Java classes, each POET code template additionally has a syntax definition that is used to build code templates from parsing input strings and to unparse code templates to external files.

The syntax for manually building a code template is

```
<code_template_name> # (arg1, arg2, ..., argn)
```

For example, $Loop\#("i", 0, "N", 1)$ and $Exp\#("abc/2")$ build objects of the code templates *Loop* and *Exp* respectively (the definitions of both code templates are shown in Figure 3.1).

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

```
<code_template_exp>[ <code_template_name> . <data_field_name>]
```

For example, $aLoop[Loop.i]$ returns the value of the i data field (see Figure 3.1) in $aLoop$. Similarly, $aLoop[Loop.step]$ returns the value of the $step$ field in $aLoop$. If $aLoop$ contains value $Loop\#("ivar", 5, 100, 1)$, then $aLoop[Loop.i]$ returns value "ivar", and $aLoop[Loop.step]$ returns value 1.

5.3 *xform* routine handles

Each *xform* routine handle is defined by the name of an existing *xform* routine. It can be used as function pointers and invoked at a later point when the *xform* parameters are given their actual values. In addition to just the *xform* routine name, a *xform* handle can additionally specify values for the optional parameters of the routine in the following syntax.

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

The syntax is identical to the *xform* routine invocation syntax except that no required arguments are given. For example, $ParseList[stop = ","]$ defines a *xform* handle for the *ParseList* routine in Figure 3.2, and when *ParseList* is invoked as the result of invoking the *xform* handle, the $stop$ parameter for *ParseList* will be set to have value ",". By allowing the values of optional parameters to be pre-determined in the *xform* handle, POET allows its routines to be dynamically re-configurable even before the routine is invoked.

5.4 Type Specifications

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, see Section 6.10)

and allows different types of values to be converted (see the type conversion operation in Section 6.11.3). Note that type specifiers can be combined with each other and with other expressions to form type expressions. For example, $(INT\ INT)$ specifies a list formed by two integers, and $INT|(INT\dots)$ specifies a type that is either a single integer or a list of integers.

5.4.1 The *INT* specifier

The keyword *INT* denotes the atomic integer type supported by POET. When used in pattern matching, the *INT* specifier can be successfully matched to all expressions that have integer values.

5.4.2 The *STRING* specifier

The keyword *STRING* denotes the atomic string type supported by POET. When used in pattern matching, the *STRING* specifier can be successfully matched to all expressions that have string values.

5.4.3 The *VAR* specifier

The keyword *VAR* denotes local or global variables supported by POET. When used in pattern matching, the *VAR* specifier can be successfully matched to all local or global variables (including trace handles).

5.4.4 The *CODE* Specifier

The keyword *CODE* denotes the code template type supported by POET. When used in pattern matching, it can be successfully matched to all code template objects. To match to a specific code template object, use syntax $name\#exp$, where *name* is the name of the code template, and *exp* specifies the type of the code template parameters.

5.4.5 The *XFORM* Specifier

The keyword *XFORM* denotes the *xform* routine handle type supported by POET. When used in pattern matching, it can be successfully matched to all *xform* routine handles.

5.4.6 The *TUPLE* Specifier

The keyword *TUPLE* denotes the tuple type supported by POET. When used in pattern matching, it can be successfully matched to all tuple expressions. To match to a specific tuple type, explicitly specify the types for all tuple components and connect them using “;”. For example, (INT,INT,INT) specifies a tuple type that contains three integers.

5.4.7 The ... Specifier

The special syntax $elemType\dots$ specifies a *list* type, where *elemType* indicates the type of elements within the list. When $elemType\dots$ is used in a pattern pattern operation, it can be successfully matched to all list expressions that contain only *elemType* elements. In particular, (\dots) can match to all list expressions that may contain arbitrary elements.

5.4.8 The .. Specifier

The special syntax *lb..ub* specifies a *range* type, i.e., a collection of integers $\geq lb$ and $\leq ub$. When used in a pattern pattern operation, it can be successfully matched to all integer expressions that have values between *lb* and *ub*. In particular, `_` can be used as values for either *lb* or *ub* to indicate an infinity bound(i.e, no lower or upper bound).

5.4.9 The MAP Specifier

The *MAP* keyword denotes a mapping table type,which takes two parameters, *fromType* and *toType*, to indicate the type of element pairs within the table. When *MAP(fromType,toType)* is used in a pattern pattern operation, it can be successfully matched to all tables that map values of *fromType* to values of *toType*. In particular, *MAP(,-)* can match to all mapping tables that may contain arbitrary types of elements.

5.4.10 The Special ANY (-) Specifier

POET uses a single underscore `_`, to denote a special type specifier that includes all types of values. The `_` specifier is pronounced *ANY*, and when used in pattern matching, it can be matched to arbitrary types of expression values.

5.4.11 Code Template Names

Each code template is essentially a user-defined data structure, and the name of the code template itself is the specifier for that particular type.

5.5 Variables

Variables are used as place holders to store results of previous evaluations. Variables in POET do not need to be declared before used. POET variables are allowed to hold any type of values dynamically, and their types are checked at runtime to ensure correctness of evaluation.

POET supports three kinds of variables: local, global, and dynamic variables. Global variables may be additionally categorized as parameters and trace handles.

5.5.1 Local Variables

The scope of each local variable is restricted within a single code template or transformation routine definition (see Section 3.2 and 3.3). Local variables are introduced by declaring them as parameters (attributes) of a code template or transformation routine, or by simply using them in the body of a code template or transformation routine. For example, in Figure 3.1, *content* is a local variable for code template *Exp*, and *start* and *stop* are local variables of code template *Loop*; in Figure 3.2, *first* and *rest* are local variables of routine *BuildList*, and *input*, *stop*, *first*, *rest* are all local variables of routine *ParseList*.

Note that code template or transformation 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 transformation routine should be considered local variables whether or not there happens to be global variables declared with identical names.

5.5.2 Global Variables

In contrast to local variables, the scope of global variables includes the entire POET program except the code template and *xform* routine definitions. Global variables are introduced by global definitions such as *parameter*, *define*, *eval*, *input*, and *trace* in the driver scripts such as Figure 3.4, where *NB*, *nu*, *gemm*, *gemmDecl*, *gemmBody* are all examples of global variables.

5.5.3 Dynamic Variables

Dynamic variables are variables that are created dynamically through type conversion; that is, these are variables created by converting an arbitrary expression to a *VAR* type (see Section 6.11). These variables are created in a symbol table separate from both local and global variables. Dynamic variables are provided mainly to support dynamic type 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 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).

5.5.4 Parameters

Parameters are global variables that act as configuration interfaces to POET programs. Specifically, the values of POET parameters can be set using the command line that invokes a POET program. For example, in Figure 3.4, *NB* and *nu* are both global parameters. To set the value of *NB* to be 50, the command line that invokes the POET program needs to include option “-pNB=50”. POET parameters are declared using the *parameter* declaration. For more details, see Section 8.4.2.

5.5.5 Trace Handles

Global variables may additionally be declared as *trace* handles, which can be embedded within a POET expression to trace transformations to its content. These *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 global variables that can be accessed (modified) inside a transformation routine. As different transformations are applied to an input expression, the transformation routines can modify the values of trace handles by replacing them with new values. Trace handles are declared using *trace* declarations as illustrated in Figure 3.4. For more details, see Section 8.4.1.

5.6 Invoking Transformation Routines

POET *xform* routines are general-purpose functions that operate on arbitrary expressions passed as input parameters. The syntax for invoking a *xform* routine is

```
<identifier> [ "["<identifier>=<exp> { ;.<identifier>=<exp>} "]" ] (<exp> {, <exp>} )
```

where the first *<identifier>* is the routine name, the list of expressions inside the () are values for the required parameters of the routine, and each *<identifier>=<exp>* defines a new value for an

optional parameter (named defined by <identifier>) of the routine. For example, the following invocation

```
ScalarRepl[init_loc=nest1[Nest.body]; elem_type=ftype;
trace_repl=Arepl; trace_decl=gemmDecl; trace=nest1]
("a_buf",alphaA, "i" * lda + "1", dim, nest1[Nest.body])
```

in Figure 3.4 invokes a transformation routine *ScalarRepl* with actual parameters “*a_buf*”, *alphaA*, ... the optional parameter *init_loc* is set to *nest1[Nest.body]*, which defines the location to insert initialization of scalar variables. The optional parameter *trace* is set to be *nest1*, which defines the trace handle to hold the resulting code (the trace handle will be updated within the routine to keep track of the transformation result).

Instead of statically specifying the name of the *xform* routine to invoke, a *xform* routine handle (contained in a variable) can be invoked similarly. The invocation succeeds, however, only if the *variable* indeed contains a *xform* routine handle; 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).

5.7 Invoking Built-in Operations

POET expressions can contain invocations to built-in operations in POET. The syntax and semantics of these operations are described in Chapter 6.

5.8 Delaying the evaluation of POET Expressions

POET provides a special *DELAY* operator, which can be used to save an expression for evaluation later. The *DELAY* operator can be used to group operations into parameterless functions. These delayed operations are just like *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 may operate on global variables if defined in the global scope. (In contrast, POET *xform* routines are not allowed to access global variables). The *DELAY* operator can also be used to define patterns that contain undefined variables (an error will be reported if the evaluation is not delayed until the pattern matching operation).

The syntax for defining delayed operations is

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

Specifically, the *DELAY* operator takes a POET expression 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, say *foo*. To evaluate the expression saved in *foo*, use operator *APPLY*. Specifically, *APPLY foo* will evaluate the operations stored in *foo* and return the evaluation result. The *foo* variable therefore can act like a function that operates on global variables, and the *DELAY* and *APPLY* operators can serve to define and invoke global functions respectively. Figure 3.4 contains the use of *DELAY* and *APPLY* operator to define and apply global functions.

Chapter 6

Built-in POET operations

POET provides built-in operations to operate on all types of expressions, to support dynamic checking of expression types and conversion between some types, and to support pattern matching based code transformations. Each operation 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.

6.1 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.

6.1.1 The PRINT operator

The syntax for the PRINT operation 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 expressions for debugging purposes.

6.1.2 The DEBUG operator

The syntax for the DEBUG operation 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 each *xform* invocation within *exp* will be stepped into once. Following shows some examples of the *DEBUG* operator.

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

6.1.3 The ERROR Operator

The syntax for the ERROR operation is

```
ERROR <exp>
```

The ERROR operator takes an arbitrary expression `<exp>`. It 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 instruction 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.

6.2 Generic comparison of values

6.2.1 The == operator

which takes two operands, *opd1*, and *opd2*, and returns a boolean (integer) value indicating whether the two operands are equal.

6.2.2 The != operator

which is the opposite of the == operator.

6.2.3 Integer and String comparison

including the binary `<`, `<=`, `>`, `>=`, `==`, `!=` operators, which have the same meaning as those in C except that they compare two integer or twostring values.

6.3 Integer Operations

6.3.1 Integer arithmetics

including the binary `+`, `-`, `*`, `/`, `%`, `+=`, `-=`, `*=`, `%=` operators and the unary `-` operator. These operations have the same meaning as those in C.

6.3.2 Boolean operations

including the binary `&&`, `||` operators and the unary `!` operator, all of which have the same meanings as those in C.

6.4 String operations

6.4.1 The string concatenation `^` 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" ^ 3 ^ "def"` returns `"abc3def"`. Note that integer operands are automatically converted to strings when used in the concatenation. The `^` operator can also be applied to a list of strings, e.g. `("abc" "def") ^ 3 = "abcdef3"`.

6.4.2 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. The operation returns a list of substrings obtained from splitting <exp2>. For example,

```
SPLIT(1,"abc") ==>"a" "bc"
SPLIT(", ",input) ==>"bc" ", " "ade" ", " "lkd" NULL
SPLIT(", ",#(MyCodeTemplate,input)) ==>MyCodeTemplate#("bc" ", " "ade" ", " "lkd" NULL)
```

6.5 List operations

6.5.1 List construction

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

6.5.2 The $::$ 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, l contains all elements in b ; otherwise, l contains b). Because the type of b may be unknown, the result may contain arbitrary numbers of elements.

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

Elements in a list ℓ are accessed through two operations: *car*(ℓ), which returns the first element of ℓ (if ℓ is not a list, it simply returns ℓ); and *cdr*(ℓ), which returns the rest of the list (if ℓ is not a list, it returns empty string). The *car* and *cdr* operators can alternatively be spelled as *HEAD* and *TAIL* respectively. The following are some examples illustrating these operations.

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

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

6.6 Tuple operations

6.6.1 Tuple Construction (the “,” operator)

A tuple 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).

6.6.2 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, “m”, 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, (“ < ”3), 4) = 3.

6.7 Table operations

6.7.1 Table Construction (the MAP Operator)

POET uses mapping tables as the dynamic data structure to efficiently associate two arbitrary types of values. To build a mapping table from two types of values, *type1* and *type2*, invoke the *MAP* operator using the following syntax.

```
MAP (type1, type2)
```

where *type1* and *type2* are type specifiers as defined in Section 5.4. Each invocation of *MAP*(*type1*, *type2*) returns a new empty mapping table that associates values of *type1* to values of *type2*.

6.7.2 Table Access (the [], LEN operators)

The elements within the table can be accessed using the “[]” operator and modified using assignments. The following illustrates how to create and operate on mapping tables.

```
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 be enumerated using the built-in *foreach* statement, where each entry within the table is placed into a pair of undefined variables. For more details about the *foreach* statement, see Section 7.3.2.

6.8 Code Template Operations

6.8.1 Code Template Object Construction (the # operator)

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

```
code_template_name "#" (arg1, arg2, ..., argn)
```

For example, $Loop\#("i", 0, "N", 1)$ and $Exp\#("abc/2")$ build objects of the code templates *Loop* and *Exp* respectively (the definitions of both code templates are shown in Figure 3.1).

6.8.2 Code Template Access (the [] operator)

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

```
<code_template_exp>[ <code_template_name> . <data_field_name>]
```

For example, $aLoop[Loop.i]$ returns the value of the i data field in $aLoop$, which is a variable that contains a *Loop* code template object. Similarly, $aLoop[Loop.step]$ returns the value of the *step* field in $aLoop$. If $aLoop$ contains value $Loop\#("ivar", 5, 100, 1)$, then $aLoop[Loop.i]$ returns value "ivar", and $aLoop[Loop.step]$ returns value 1.

6.9 Type operations (the | operator)

POET type expressions can be combined using the binary | (alternative) operator.

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

Here the resulting type is alternatively either $\langle type1 \rangle$ or $\langle type2 \rangle$. The type expressions are used in pattern matching to dynamically determine the types of regular expressions.

6.10 Pattern Matching And Assignment

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. Further, information (component expressions) often need to be saved during the pattern matching process to facilitate subsequent operations. This is supported by pattern matching combined with variable assignments. In particular, variables can have their values initialized/assigned during a pattern matching operation.

6.10.1 The Pattern Matching Operator (the “:” operator)

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 type expression which combines type specifiers (as defined in Section 5.4) with constant values and other regular expressions. The operation returns TRUE (integer 1) if <exp> matches the pattern specified by <pattern> (i.e., <exp> can be successfully matched against the specified type), 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 5.4. Here the pattern matching succeeds only if <exp> has the specified type, e.g., is a string (the *STRING* specifier), is an object of a particular code template (the code template name specifier).
- An undefined variable. Here the pattern matching always succeeds by assigning the variable with the value of <exp>.
- An expression with no undefined variables. The pattern matching succeeds if <exp> has the same value as <pattern>, 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. For example, both the *stop* and *continue* parameters in the *ParseList* routine in Figure 3.2 can take *xform* routine handles as values, so that the caller of *ParseList* can use a function to determine when to continue/stop parsing.
- An assignment operator in the format of *var = pat*, where *var* is a single variable and *pat* is a pattern specification. Here the pattern matching succeeds if <exp> can be successfully matched against the *pat* sub-pattern; and if succeeds, the variable *var* is assigned with the value of <exp>.
- Two sub-patterns connected by the | operator. The pattern matching succeeds if <exp> could be matched to either of sub-patterns. For example, <exp>: *INT|STRING* succeeds if <exp> is either an integer or a string value.

The following illustrates how to apply pattern matching to check the types of various expressions.

```
"3" : STRING ==>1
3 : STRING ==>0
```

```

MyCodeTemplate#"123" : STRING ==>0
MyCodeTemplate#123 : MyCodeTemplate ==> 1
3 : MyCodeTemplate ==> 0
MyCodeTemplate#123 : MyCodeTemplate#INT ==> 1
("abc" "." "ext") : STRING ==>0
("3" "4" "5") : (INT ...) ==>1
3 : (INT ...) ==>0
(3 4 5) : (INT ...) ==>1
(3 4 5 "abc") : (INT ...) ==>0
(3 4 5 "abc") : (_ ...) ==>1
3 : (0 .. 2) ==>0
3 : (0 .. 5) ==>1
"a" : (0 .. 5) ==>0
"a" : CODE ==>0
MyCodeTemplate : CODE ==>1
MyCodeTemplate#123 : CODE ==>1
("abc" "." "ext") : CODE ==>0
MyCodeTemplate#123 : XFORM ==>0
("abc" "." "ext") : XFORM ==>0
foo : XFORM ==>1 < * here foo is a xform routine name * >
"abc" : TUPLE ==>0
("abc",2) : TUPLE ==>1
MyCodeTemplate#123 : TUPLE ==>0

```

In summary, the <pattern> operand may contain arbitrary expressions as components. It may additionally contain undefined variables. For example,

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

Here during the evaluation process, the undefined variables (*second* and *third*) are assigned with the necessary values in order to make the pattern matching succeed; and when an assignment is seen inside the pattern specification, the left-hand side of the assignment is given the value of the matched expression.

Note that when uninitialized variables appear in the pattern matching operation, these variables are treated as place holders which can be matched to arbitrary expressions. If the matching is successful, all the uninitialized variables would have been assigned a valid value. Therefore the pattern matching operation can be used for dynamic type checking, for initializing variables, and for assigning values to variables (initialized or uninitialized) in general.

6.10.2 Un-initializing Variables (The CLEAR operator)

The *CLEAR* operator takes a single variable *v* as parameter and clears the value contained in *v* so that *v* becomes undefined after the operation. The operator is provided to support assignments to previously defined variables in the “:” pattern matching operator. For example, *input : Stmt#(CLEAR content)* will treat the variable *content* as an undefined variable and assigns a new value to it the pattern matching succeeds. The *CLEAR* operator needs to be applied

to all pattern variables if the pattern matching operation is inside a loop — without the CLEAR operator, all pattern variables inside a loop would be undefined only in the first iteration. For more examples that illustrate how to use the CLEAR operator in pattern matching loops, see Section 7.3.2.

6.10.3 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) of <lhs>. 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. Compared with the “:” pattern matching operator, all variables in <lhs> are treated as undefined 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 assignment statements,

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

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

6.11 Parsing and Type Conversion

Since POET is designed to process arbitrary programming languages, it provides built-in support for the dynamic parsing of arbitrary lists of tokens (each token is either a string or an integer) into compound data structures. This parsing support can also be used to effectively convert arbitrary types of expressions between their string and structured (code template) representations. The parsing process uses recursive descent parsing techniques and dynamically interpret the syntax of a language until it has exhausted all options. The parsing is invoked by a number of built-in operators.

6.11.1 The => and ==> Operators

POET provides two binary operators, => and ==>, to convert an input computation (normally represented as a nested list of strings, integers, and code templates) into structured representations (in particular, code templates, lists, and tuples). The parsing is based on the syntax definitions of the code templates (see Section 8.2). The syntax for applying the operators are

```

<input> => <parse_ppec>
<input> ==> <parse_spec>

```

Here `<input>` is an arbitrary expression (in most cases, a list of strings, integers, and code templates), and `<parse_spec>` is a parsing specification that defines how to parse the `<input>` computation. Both the `=>` and `==>` operators have similar semantics in that they both take the `<input>` parameter, parse `<input>` 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.

The `<parse_spec>` operand is similar to the `<pattern>` operand in Section 6.10, where the `==>` and `=>` operator would be similar to invoking the pattern matching `“:”` and `=` operators respectively. However, the `<parse_spec>` operator have different semantics (parsing instead of pattern matching) when the input is a list of strings instead of structured data structures. Specifically, the `<parse_spec>` could take any of the following forms.

- A single string or an integer (e.g., “abc”, 35, -23). The current input token (the first token `<input>`) will be compared with the string or integer, and the parsing succeeds only if the token equals to the string or integer value.
- The *STRING*, *INT*, or *VAR* type specifier, where the current input token will be converted to a string, an integer, or a local variable as the parsing result. Note that all tokens can be converted to a string (therefore matching all tokens to *STRING* or *VAR* — which adds a “_” prefix to the string and then uses the result as variable name — will succeed), but only integers or strings with an integer value (e.g., “134”) may be converted to integers. The parsing fails if the token cannot be converted to an integer.

The *VAR* type specifier allows local variables to be created with name made dynamically from an arbitrary string value. For example, `5 => VAR` returns a new dynamic variable (trace handle) that contains value 5. The *VAR* operator therefore allows place-holder variables to be dynamically created for convenient pattern matching and transformation tracing need.

- The name of a code template. In this case, the leading tokens in `<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>`; otherwise, the parsing fails.
- A *xform* routine handle which takes a single parameter (the `<input>` expression 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 will be saved, and the parsing continues with the *left_over* value returned by the routine invocation. In figure 3.2, the *xform* routine *ParseList* illustrates such a parsing routine.
- A compound data structure in any of the following forms.

```

<code_template_name> # <parse_spec>

```

```

(<parse_spec1> <parse_spec2> ..... <parse_spec_n>)
(<parse_spec1> , <parse_spec2>, ..... (<parse_spec_n>))

```

Note that here `<parse_spec>` is recursively defined. In these case, the parsing process not only tries to match the leading strings in the input with the particular 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.

```
[ <var> = <parse_spec1>]
```

Here the `<input>` is parsed against the given `<parse_spec1>`, and the parsing result is saved in the given variable `<var>`. For example, given the code template definitions in Figure 3.1, the following operation

```
SPLIT ("", "for(i=0;i<100;i+=2)") => [myLoop = Loop]
```

will convert the list of tokens from splitting the *for* loop to a *Loop* code template object and then save the object into variable *myLoop*.

- A *TUPLE* specification in the following format.

```
TUPLE ( <string> <type> { <string> <type> } <string> )
```

Here each `<string>` operand is a single string constant, and each `<type>` is a type specification. In this case, the parsing process tries to convert leading strings of `<input>` into a tuple of n elements, where the i th `<type>` specifies how to parse the i th element. The tuple syntax starts with the first `<string>` and ends with the $n + 1$ th (the last) `<string>`, and the i th and $i + 1$ th elements must be separated using the $i + 1$ th string. 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 strings 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>, <string>)
```

Here the parsing process will try to construct a list of elements from the leading strings in `<input>`, where `<parse_spec>` defines how to parse each element within the list, the `<string>` operand (must be a single string) defines what must be 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 used to connect a number of `<parse_spec>`s. Here the recursive descent parser tries to parse the `<input>` using each of the specifications in order — if one specification fails, the following one will be tried — until all options have been exhausted. For example, `exp => (INT, INT)|INT` will try to parse the input `exp` either as a single integer or a pair of integers. Note that once an option succeeds, none of the following operands will be tried 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 — because all integers can be treated as strings, the `INT` options will never be tried in the parsing process.

6.11.2 Parsing Annotations (The `=>`, `BEGIN` and `END` operators)

Base on parsing specifications embedded within code template definitions (see Section 8.2), POET can be made to automatically parse most input languages entirely. However, recursive descent parsing is not regarded as an efficient technique. To speed up (and to ease) the parsing process, the source of the input code often can be annotated to help it being parsed into a code template representation. In general, the more annotations included in the source input, the shorter time it will take the POET interpreter to parse the input code.

As shown in Figure 3.3, each POET input annotation either starts with `“//@”` and lasts until the line break, or starts with `“/*@”` and ends with `“@*/”`. The special syntax allows programmers to embed these annotations as comments in C/C++ code, so that the source input is readily accessible for other uses. POET supports two kinds of input annotations (both illustrated in Section 3.4.1.

- **Single-line Annotations** A single-line annotation applies to a single line of program source (e.g. the `//` comment in C++). It has the format `“=> T”`, where `T` is a parse specification as defined in Section 6.11.1. For example, the annotation `“int i, j, l; //@=>[gemmDecl=Stmt]”` indicates that `“int i, j, l;”` is a statement that should be parsed using the `Stmt` template, and the result should be stored in the global variable `gemmDecl`. The definitions for both `Exp` and `Stmt` can be found in Figure 3.1.
- **Nested Annotations** Nested annotations are used to help parse compound language constructs such as functions and loop nests, which include other code fragments as components. Each nested input annotation starts with `“BEGIN(x)”`, where `x` is the variable that should be used to store the compound code template, and ends with `“END(x:T)”`, where `T` is a parse specification for the annotated code. In Figure 3.3, the annotation `“for (l = 0; l < K; l += 1) //@ =>[loopL=Loop] BEGIN(nest1) ... END(nest1)”` is a nested annotation which starts with the `for` loop (a singly annotated fragment stored in `loopL`) and ends after parsing the loop body `stmt1`.

6.11.3 Type Conversion Between Expressions

Sometimes it is necessary to convert an expression value from one type to another. POET uses the parsing operators (`=>` and `==>` operators) to support the conversion between string/integer and code template expressions for most expressions. For example, `exp => [var = INT]` converts an expression `exp` to an integer and saves the integer to variable `var`. Note that here the 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 `exp : STRING`). For example,

```
3 => STRING =====> "3"
MyCodeTemplate#123 => STRING =====> "MyCodeTemplate#123"
```

6.12 Delaying Evaluation of Instructions (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 6.10). 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).

6.12.1 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 delayed operations is

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

Specifically, the *DELAY* operator takes a POET 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.

6.12.2 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. Figure 3.4 includes examples of using the *DELAY* and *APPLY* operator to define and apply global functions.

6.13 Trace Operations

POET supports a special kind of global variables called *trace* handles. These handles can be embedded within a POET expression to trace transformations to their content. Once embedded inside an input expression, *trace* handles become integral components of the expression value. Transformations to an input computation therefore can be implemented by simply modifying the values of trace handles within the computation. Note that when inside a *xform* function, trace handles must be modified through code transformation operations (e.g., *REPLACE*) defined in Section 6.14. The following POET operations can be used to set up trace handles.

6.13.1 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 Figure 3.4,

```
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.

6.13.2 INSERT (x, exp)

This operation inserts all tracing handles rooted at x (tracing handles that follow x in a global trace declaration as defined in Section 8.4.1) to be embedded within expression exp if possible so that x may be used to trace transformations within exp . For example, in Figure 3.4, the $TRACE(gemm, gemm)$ operation inserts all the trace handles (declared in the $trace$ declaration), $gemm$, $gemmDecl$, $gemmBody$, $nest3$, $nest2$, and $nest1$, into the expression contained in $gemm$, so that all the trace handles become embedded within the internal representation of the input code.

6.13.3 ERASE(x, exp)

Here x is a single or a list of trace handles. This operation removes all the occurrences of trace handle in x from 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) = Stmt#3
```

As a special case, the invocation $ERASE(x, x)$, return the value contains 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) = "abc".
```

6.13.4 COPY(exp)

While $ERASE(x, exp)$ removes the occurrences of a single trace handle x in exp , the operation $COPY(exp)$ replicates exp with a copy that has no trace handles. For example, if $input = Assign\#(x, y)$, where both x and y are trace handles with values “var” and 4 respectively, then

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

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

Here $v1, v2, \dots, vm$ is a tuple of trace handles. This operation saves the current value of each trace handle $v1, v2, \dots, vm$ 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.

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

Here v_1, v_2, \dots, v_m is a tuple of trace handles. This operation restores the last value saved for the trace handles v_1, \dots, v_m . 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.

6.14 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* variables 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 besides modifying trace handles, all built-in operations return their transformation results without any direct modification to input computation.

6.14.1 DUPLICATE(c1,c2,input)

Here c_1 is a single expression, c_2 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 c_1 in *input* by a different component in the list c_2 . 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
```

6.14.2 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 list *input* based on *config* which defines a position for each element in *input*. For example,

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

6.14.3 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* function defined for each code template object (see Figure 3.1 and Section 3.2) to eliminate redundancies (e.g., empty strings) in *exp*.

6.14.4 REPLACE(c1,c2,input)

Here c_1 and c_2 are POET expressions, and *input* is the input computation to transform. This operation replaces all occurrences of the code fragment c_1 in *input* with c_2 . For example,

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

6.14.5 REPLACE(config, input)

Here *input* is the input computation to transform, and *config* is a list of pairs in the format of (*orig*, *repl*). 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 the evaluation process. 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))
```

6.15 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 7

POET 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, \dots, s_m is composed into a sequence, the evaluation results of the previous s_1, s_2, \dots, 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, \dots, e_m is composed into a sequence, the evaluation result is a list that contains the result of all expressions e_1, e_2, \dots, e_m as components (in POET, when expressions are simply listed together, they are considered operands in a list construction operation. See Section 5.2.1.

POET statements serve to provide support for debugging (side effects) and control flows such as sequencing of evaluation, conditional evaluation, loops, and early returns from a *xform* routine calls.

7.1 Single Statements

7.1.1 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.

7.1.2 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.

7.1.3 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*.

7.2 Conditionals

7.2.1 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.

7.2.2 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 6.10, and *<stmts1>*, *<stmts2>*, ..., *<stmtsm>* and *<default_stmts>* are sequences of statements/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 cover the value of *<cond>*, the *<default_stmts>* is evaluated and returned as result. If no pattern matching succeeds and there is no default statements specified (the default branch is optional), an error message is issued.

Note that the POET switch statement is different from that in C in that only one pattern will be successfully matched through out the evaluation. Once a pattern matching succeeds, the corresponding statements are evaluated and the result is returned immediately (no statements in the following label will be evaluated even if there is no additional control flow statement). If

two patterns need to be combined, they should be combined using the `|` operator, as shown in Section 6.10.

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> } ]
```

7.3 Loops

7.3.1 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). The *for* loop has the same syntax and semantics as those of the C language. First, the `<init>` 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));
}
```

7.3.2 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 6.10, `<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 match 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 6.14) after a `foreach` loop has finished, as the transformation operations may disrupt the traversal by the `foreach` loop.

7.3.3 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 meaning 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 7.3.2 saves all the loop nests in the reverse order of their appearances in *input*.

7.3.4 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.

Chapter 8

Global Declarations and Commands

A POET program is composed of a sequence of global declarations and commands of the following kinds.

- Global macro declarations, which define names that are used as macros and modify the default behavior configurations of the POET interpreter.
- Code template declarations, which define global code template data types and their syntax in various source languages.
- Xform routine declarations, which define generic functions that can be invoked to operate on arbitrary input code.
- Tuning parameter declarations, which define parameters (a special kind of global variables) whose values can be modified via command line options when invoking a POET program.
- Trace handle declarations, which define a set of global variables that can be used to track transformations applied to input code.
- Input commands, which define what input computations to parse and process.
- Evaluation commands, which specify what expressions to evaluate at the global scope.
- Output commands, which define what results to output to external files.

The global declarations, e.g., macro, code template, xform routine, tuning parameter, and trace handle declarations, serve to specify attributes of global names (e.g., global macro, global variables, global code template names and xform routine names). 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 Global Macro Definitions and Reconfiguring POET

In POET, a macro variable is simply a name associated with a value. Macro variables can be defined only once and cannot be changed throughout the evaluation of a POET program. The syntax for a macro definition is:

```
< define <identifier> <exp> />
```

Here `<identifier>` is an arbitrary name (undefined so far), and `<exp>` is a POET expression involving only variables defined so far. Each macro definition defines the value for a single macro variable. The following shows some example 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. These macros include:

8.1.1 The TOKEN Macro

This macro adds a post-processor to POET's lexer (tokenizer) when reading program input. For example, the following definition appears in the `Cfront.code` (the C language specialization) file in the `POET/lib` directory.

```
<define TOKEN RecognizeTokens[tokens=((("==" ), "==")
                                         (<"<" ), "<=")
                                         (>">" ), ">=")
                                         (!"!" ), "!=")
                                         ("&"&" ), "&&"
                                         ("|" |" ), "| |" )] />
```

This definition instructs the POET interpreter to invoke the *RecognizeTokens* routine (defined in the `POET/lib/utils.incl` file) after the POET lexer has finished reading a program input (defined using the *input* command in Section 8.5) or a syntax definition for a code template (see code template definitions in Section 8.2). The *RecognizeTokens* routine is defined in the `POET/lib/utils.incl` file. When invoked with a list of tokens as input, this `TOKEN` routine can convert the list by replacing every pair of “<” “=” into a single “<=” token, every pair of “>” “=” into “>=”, and so forth. An alternative `TOKEN` routine can recognize all floating point numbers and convert them into single tokens. The *RecognizeTokens* routine provided in the `POET/lib` directory tries to be language neutral and makes several passes over the program input to recognize each token. A more efficient language-specific tokenizer could read each string/integer from the input one by one and build a new token stream based on standard scanning techniques in compiler construction.

8.1.2 The PARSE Macro

This macro adds a pre-processor to POET's recursive descent parser for program input (each program input is defined using the global *input* command in Section 8.5). For example, the following definition appears in the `Ffront.code` (the Fortran language specialization) file in the `POET/lib` directory.

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

This definition instructs the POET interpreter to invoke the *ParseLine* routine (defined in the `POET/lib/utils.incl` file) before the recursive descent parser is invoked to parse the program input. The *ParseLine* routine will be invoked with a (possibly nested) list of tokens returned by the POET lexer. The *ParseLine* routine will filter the token stream 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 column-based languages such as Fortran and Cobol.

8.1.3 The UNPARSE macro

This macro adds a post-processor to POET's unparser when outputting transformation results (see the global output command in Section 8.7). 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).

8.1.4 The PARSE_BOP Macro

This macro allows using the POET built-in parser (not the interpretative recursive descent parser for program input) for expressions to build code template representations of expressions. Specifically, if this macro is not defined, when seeing an expression `"abc" + 3`, the POET interpreter will report a runtime error (because only integers can be added in POET). However, if the following definition exists,

```
<define PARSE_BOP Bop />
```

the expression `Bop#("abc", 3)` will be returned as the result of `"abc" + 3`. This feature allows expressions in the source language to be more conveniently built and increases their readability. Note that the `PARSE_BOP` macro affects only the evaluation of regular POET expressions and does not affect how program input (defined by the global input command in Section 8.5) is parsed.

8.2 Code Template and Parsing/Unparsing Definitions

In POET, code templates are user-defined compound data structures that are used to represent the input computation in a structured fashion, to define how to parse an input computation, and to define how to unparse internal representations of computations to external files. The syntax for defining a code template is

```
< code <identifier> pars = ( <identifier> [: <type> ] {, <identifier> [: <type> ] })
    [ rebuild = <xform_handle> ]
    [ cond = <exp> ]
    [ parse = <parse_spec> ]
    [ output = <xform_handle> ]
    { <identifer> : <type> } "/>"
```

which declares the structure of a code template, or

```

< code <identifier>  pars = ( <identifier> [: <type> ] {, <identifier> [: <type> ] })
    [ rebuild = <xform_handle> ]
    [ cond = <exp> ]
    [ parse = <parse_spec>]
    [ output = <xform_handle> ] >
<exp>
</code>

```

which additionally defines the syntax for construct code template objects from parsing input code and for unparsing code template objects to external files. Here, the first `<identifier>` token specifies the name of the code template, and `pars=...` specifies parameters of the code template. Each parameter is either a single name or a `<identifier>:<type>` pair which specifies both the parameter name and its type constraints. For example, a parameter declaration could be either `pars = (a : INT, b : STRING)` or simply `pars = (a, b)`. Each `<type>` expression in the above specification could be a constant (an integer or a string literal), a type specifier as defined in Section 5.4, or a compound data structure (a list, a tuple, or a code template) with other types as components. In particular, the `<type>` specification for each parameter is similar to the `<pattern>` specification in Section 6.10 except that neither local nor global variables can be included as components.

The required components for a code template declaration include only the name of the code template and its parameters. Optional declarations include a sequence of attribute definitions in the format of `<identifier>:<type>`, where `<identifier>` is the attribute name, and `<type>` is the type constraint for the particular attribute. The values for these attributes can be specified together with the values for code template parameters when constructing code template objects using the `#` operator. For example, in the following code template declaration,

```
<code Loop pars=(i,start,stop,step,reverse:(0,1))  maxiternum:INT/>
```

the code template `Loop` has an additional attribute `maxiternum`, which remembers the maximal number of iterations that a loop may take. A `Loop` code template object such as the following `Loop#("i",0,m,1)#100` can be constructed which specifies (after the second `#` sign) that the maximal iteration number of the loop is 100.

While users can define arbitrary attribute names for a code template, several keywords (`rebuild`, `parse`, `output` and `cond`) are reserved and have special meanings to POET. These special-purpose attributes are used to specify how to parse/unparse/simplify/validate objects of the code template and are defined in the following.

- The `parse` attribute specifies that instead of parsing the code template based on its syntax definition (the body of the code template definition), use the given `<parse_spec>` to parse the input. The format of each `<parse_spec>` is defined in Section 6.11. The only difference here is that no local or global variables are allowed in the specification. The most common specification for the `parse` attribute is a `xform` routine handle (for more details on `xform` handles, see Section 5.3), which specifies a `xform` routine that should take the parsing responsibility for the code template; that is, the `xform` routine should be used to convert a list of tokens to an code template object. Such `xform` routines are called *parsing functions*. As illustrated by the `ParseList` routine in Figure 3.2, each parsing function must take a single parameter, the `input` token stream to parse, and returns a pair of values (`result`, `leftOver`), where `result` is the result of parsing the leading strings in `input` as a code template object, and `leftOver` is the rest of the input token stream to continue parsing.

The following example (taken from *POET/lib/utills.incl*) illustrates how to invoke the `ParseList` routine to parse a code template.

```
< *A marker for names composed of sub-components *>
<code Name pars=(content) parse=ParseList[continue=is_name]
      cond=(content!= "")
      rebuild=IgnoreCodeIfSingle[ignore=STRING;code=Name]>
@content@
</code>
```

Here *is_name* is another *xform* routine defined in the same file. Other than explicitly defining a parsing function for the code template, alternative parsing specifications can invoke the *LIST* and *TUPLE* operators which define how to parse a list or tuple of tokens. The following example (taken from the same file) illustrates how to invoke the *LIST* operator to build code templates via parsing.

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

- The *rebuild* attribute specifies a *xform* routine that should be invoked when rebuilding code template object using the *REBUILD* operator (see Section 6.14). In the above examples, both *IgnoreCode* and *IgnoreCodeIsSingle* are *xform* routines defined in *POET/lib/utills.incl*. Each *rebuild* routine must take the same parameters as those of the code template it tries to rebuild. When a POET expression *exp* is invoked as argument to the *REBUILD* operator, the *rebuild* function of each code template object within *exp* will be invoked with parameters of the code template object, and the returned value by the *rebuild* invocation is used to replace the original code template object in *exp*. Note that in addition to being invoked by the *REBUILD* operator, the *rebuild* routines are also invoked by the POET recursive-descent parsing when constructing code templates from parsing a token stream. This allows certain code template wrappers to be ignored after parsing when their sole duty is to enable proper parsing. For example, the *IgnoreCode* routine simply returns the content of the code template so that a *NameList* code template object is never constructed.
- The *output* attribute specifies a *xform* routine handle that should be invoked when a code template object needs to be unparsed to an external file. By default, the syntax definition (the body) of a code template definition 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 for *Bop* (taken from *POET/lib/Cfront.code*) invokes a *UnparseExp* routine (defined in *POET/lib/ExpStmt.incl*) to unparse all *Bop* objects.

```
<*binary operators*>
<code Bop pars=(op, opd1, opd2) rebuild=RebuildExp
      output=UnparseExp[opType=OpPrec]>
@opd1@ @op@ @opd2@
</code>
```

Here the *Bop* code templates needs a special unparse function because without context information, it cannot determine whether or not to put a pair of “()” around the expression. Each unparse function takes a single parameter, the code template to unparse, and produces a list of tokens to output to an external file.

- The *cond* attribute imposes additional constraints constructing code templates during parsing; that is, the *cond* attribute expression must evaluate to *true* (a non-zero integer) when given the values for the code template parameters. The *cond* expression is evaluated when building code template objects during parsing. For example, in the following code template definition, the *cond* expression ensures that the *Name* : code template won't be built with an empty string as content.

```
< *A marker for names composed of sub-components *>
<code Name pars=(content) parse=ParseList[continue=is_name]
      cond=(content!= "")
      rebuild=IgnoreCodeIfSingle[ignore=STRING;code=Name]>
@content@
</code>
```

The body of each code template is a POET expression that defines the concrete syntax both for parsing program input and for unparsing transformation results. The body itself is a list of strings in the input/output source language (e.g., C, C++, or Java). If evaluation of POET expressions are necessary within a code template body, the POET expressions need to be wrapped inside pairs of symbols. Examples of such code template definitions can be found in Figure 3.1.

By default, POET uses the body of code templates as the basis both for constructing code template representations of programs from parsing and for unparsing code template representations to external files. When trying to parse the leading 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, *_*, will be used). The substituted template body is then matched against the leading strings of the input. The parsing based matching is in many ways similar to the pattern matching process in Section 6.10, except when a code template type is encountered and the current leading input string is not a code template. In this case, the parser would try to recursively try to match the current input against the new code template, and if the new recursive parsing is successful (that is, a matching code template object has been successfully built from the leading input strings), the original continues with the remaining input.

It is convenient to utilize the built-in code-template based recursive descent parsing, but sometimes the *parse* attribute must be defined to specify a *xform* routine for parsing certain code templates. The theoretical basis of this requirement is the fundamental limitation of recursive descent parsers when parsing left-recursive productions (each code template body is equivalent to a production in the BNF description of language syntax). An example is the parsing of expressions, the syntax of which is left recursive in most languages (e.g., in expression *a+b*, the first operand *a* could itself be another expression). The *POET/lib/utiles.incl* files provides a reconfigurable *ParseExp* routine that can be used to parse expressions in most languages.

Another common use of the *parse* attributes for code templates is to utilize the built-in parsing capabilities for lists of elements. 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,",") rebuild=IgnoreCode/>
```

Since the *parse* attribute can be given any *<parse_spec>* value as defined in Section 6.11 (except variable assignment), other operators such as *TUPLE* can be used similarly. When the *LIST* or *TUPLE* specifier is used as value of the *parse* attribute, the unparsing is done accordingly as well;

that is, using *LIST* or *TUPLE* as values for the *parse* attribute can change how the code template is unparsed as well. For example, when a *NameList* object needs to be unparsed to an external file, each component will be separated with a “,”.

8.3 Xform Routine Declarations

POET *xform* routines are generic functions that apply transformations to input computations or analyze existing code to extract useful properties. Each routine takes some input parameters and return some value as result. The syntax for defining a *xform* routine is

```
< xform <identifier> pars=( <param> {, <param>} )
      [output=( <identifier> {, <identifier>} ) ]
      { <identifier> = <constant> } />
```

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

```
< xform <identifier> pars=( <param> {, <param>} )
      [output=( <identifier> {, <identifier>} ) ]
      { <identifier> = <constant> } >
<exp>
"</xform>"
```

which additionally defines the implementation of the routine. Here, the first *<identifier>* 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, each can be accessed using the corresponding *<identifier>* in the *output* tuple specification. An arbitrary number of *<identifier>=<constant>* specifies additional optional input parameters (parameter names defined by *<identifier>*) of the *xform* routine and their default values (defined by *<constant>*). 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 syntax for invoking *xform* routines is described in Section 5.6. 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. The names in the output specification also serve to document the meaning of each value returned by the routine.

Note that while the syntax (see Section 5.6) of an *xform* routine invocation is the following,

```
<identifier> [ "["<identifier>=<exp> { ;.<identifier>=<exp>} "]" ] (<exp> { , <exp>} )
```

a *xform* handle (see Section 5.3) can be defined as

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

So the optional parameters can be given values before the *xform* routine is actually invoked; that is, the values for the optional parameters can become an integral part of a function pointer, which can be used in all places where an expression is expected and can be invoked at an arbitrary time in the future. So all the parameters used to re-configure the behavior of an *xform* routine should be defined as optional parameters, and only pure input parameters (parameters that define the input of transformation) should be declared as required parameters (included within *pars = (...)*).

8.4 Global Variable Declarations

In POET, all variables used outside of code template and *xform* routine definitions are global variables. These variables can be simply used without declaration in expressions in the global scope. Global variables can be additionally declared as *trace handles* or *POET parameters* using the following declarations.

8.4.1 Trace Handle Declarations

Trace handles are a special kind of global variables which can be embedded within POET expression to trace transformations to the variables' contents. As various transformations are applied to an input expression, the trace handles can be replaced with different but equivalent values. Each transformation can therefore operate on the trace handles without being concerned with whether or how many other transformations have already been applied.

The syntax to declare trace handles is as the following.

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

As example, Figure 3.4 includes the following trace handle declaration.

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

Here a number of global variables are declared as trace handles. These variables may additionally be assigned with values within the *trace* declaration. The use of trace handles is illustrated somewhat in Section 3.4, and the operations that support trace handles are described in section 6.13.

Global variables need to be explicitly declared as trace handles in order to be embedded into POET expressions using the *INSERT* operation. NOTE that when a sequence of trace handles are declared in a single global 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 when they are embedded in POET expressions. Subsequently, POET allows these trace handles to be inserted into a POET expression using a single *INSERT* operation. It is therefore recommended to collect only related trace handles in a single declaration, and to declare unrelated trace handles in separate declarations to avoid confusion.

In order for the *INSERT* operation to be successful, the trace handles must have the correct values; that is, the values of the trace handles must indeed be part of the input code to *INSERT*. A special case of invoking the *INSERT* operation is *INSERT(tophandle, tophandle)*, where *tophandle* is the outermost tracing handle that includes a collection of all the other handles declared together. Note again for this to work, the tracing handle declaration must have listed all the handles according to pre-order traversal.

8.4.2 Parameter Declarations

POET is a language designed for dynamic code generation and allows different variations in the generated code through a collection global parameters whose values may be determined by command-line options that invoke a POET program. To declare a POET parameter, use the following syntax.

```
< parameter <identifier> type=<type>
      parse=<parse_spec>
      default=<exp>
      message=<string_literal> />
```

Here `<identifier>` specifies the name of the parameter variable; `<type>` specifies the type of the parameter value and must be a type specifier as defined in Section 5.4; the `<parse_spec>` specifies how to parse the parameter from command-line and must be a parsing specifier as defined in Section 6.11; the `<exp>` specifies the default value of the parameter when the command-line option does not specify an alternative value; and `<string_literal>` is a string literal that documents the meaning of the parameter in the declaration. The following shows several examples of POET 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" />
<parameter A type=((INT,INT),INT) default=((1,2), 3) message="affinity relations"/>
<parameter B type=( (INT|(INT...))...) default=((1 2) 3)
    parse=PLIST[elem=(INT|PLIST[elem=INT])] message="affinity relations"/>
```

The above examples illustrate the use of four common kinds of parameter declarations.

- 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 *A* parameter is nested tuple of integers. So here the type of *A* uses the `TUPLE` type specifier. No parsing is specified, so by default a list of integers will be taken with no additional syntax involved (e.g., 132), and the list of integers will be automatically converted to the given tuple data structure. Alternatively, a *TUPLE* operator could be used to specify how to parse the input from command line.
- The *B* parameter is a possibly nested list of integers. Here the parameter type uses the list (...) type specifier; i.e., in the format of (*elemType*...), where *elemType* specifies the allowed type of each element within the list. The parse attribute is defined which invokes a *xform* handle named *PLIST* (defined in POET/lib/utils.incl) to parse the corresponding input from command line.

The command line option to define parameter values is `-p<var>=<val>`, where *var* is the name of the POET 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 type, before the final result being assigned as value of the parameter. For example, `-pNB=50` defines the value of the *NB* parameter to be 50, and `-pA="345"` defines the value of parameter *A* to be a nested tuple ((3,4),5). Note that the value of *A* is specified as a string, which will be parsed internally to obtain a tuple (against the declared type of parameter *A*) before the tuple is as the value of *A*. The given values for all POET parameters will be checked against their type specifications to ensure the POET program is correctly invoked.

8.5 Input Commands

Each POET input command specifies an input computation to be read in and processed. The input computation could be embedded within the command or could be contained in an external file. 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 the input computation is contained in an external file, 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 command. Here each `<exp>` represents a POET expression; `<int>` represents a integer literal; and `<input_parse_spec>` is either a parsing specification as defined in Section 6.11 or a special keyword *POET* (which indicates that the input computation should be read in as POET programs). All the attribute specifications (*cond*, *syntax*, etc.) are optional and can be arbitrarily ordered. In particular,

- *cond* = `<exp>` specifies any pre-condition that must be satisfied before reading the input (if the *cond* expression evaluates to false, no input will be read);
- *DEBUG* = `<int>` specifies whether the parsing of input needs to be debugged and at what level (defined by the constant integer) to debug the parsing process (the higher the level is, the more debugging info 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 computation. 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 computation.
- *annot* = `<exp>` specifies whether or not to allow annotations in the input file (by default, the *annot* has a true value, and the parsing process will utilize annotations within the input; if *annot* is explicitly defined to be false, then annotations (if there is any) will be treated as part of the input computation). For details of parsing annotations, see Section 6.11.

- *parse* = <input_parse_spec> specifies how to parse the input computation, where <input_parse_spec> is either a parsing specification as defined in Section 6.11 or a special keyword *POET*, which indicates that the input computation should be read in as POET programs. In most cases, <input_parse_spec> is the name of a code template that defines the top-level structure of the input computation.

An example input command with the input computation as part of command is shown in Figure 3.3. The following shows some examples where the input computation is in a separate file.

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

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

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

Here the file name that contains the input computation is defined as the value of *inputFile*, a POET parameter variable. 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 *xformFile* should be read in as a POET program, while the *inputFile* should be read in as an object of *inputType* (a code template name). The input code defined in *inputFile* will be parsed and translated into an internal code template representation based on the included language specializations.

8.6 Evaluation Commands

The *eval* command is used to specify a collection of expressions and statements to evaluate in order to compute the final output. It's syntax is

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

Here <exp> is any POET expression or statement defined in chapter 5. **All POET expressions must be embedded within an eval command to be evaluated at the global scope.**

8.7 Output Commands

POET *output* commands are used to write transformation or analysis results to external files or standard output (in contrast, the output of the *PRINT* and *ERROR* operators are directed to standard error). The syntax for output commands 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 any 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 (AST) 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 examples of 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 specialization files. When an empty string is specified as the output name (or when no name is specified

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 | ID "=" expUnit | traceVars "," traceVars
inputAttrs : inputAttr { $$ . ptr = $0 . ptr; } inputAttrs | ;
inputAttr : "debug" "=" expUnit | "annot" "=" expUnit | "cond" "=" expUnit
  | "syntax" "=" expUnit | "parse" "=" "POET" | "parse" "=" parseSpec
  | "from" "=" expUnit | "to" "=" ID
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" "=" expUnit
codeRHS : ">" exp "</code>" | ">"
xformAttrs : xformAttrs xformAttr | ;
xformAttr : "pars" "=" "(" codePars ")" | "output" "=" "(" codePars ")"
  | ID "=" expUnit
xformRHS : ">" exp "</xform>" | ">"
codePars : ID | ID ":" typeSpec | codePars "," codePars

typeSpec : "_" | INT | STRING | ID
  | "STRING" | "INT" | "VAR" | "CODE" | "XFORM" | "TUPLE"
  | "MAP" "(" typeSpec "," typeSpec ")" | typeSpec "..."
  | typeSpec ".." typeSpec | ID "#" typeSpec
  | "(" typeList ")" | "(" typeTuple ")" | typeSpec "|" typeSpec
typeList : typeSpec | typeSpec typeList
typeTuple : typeSpec "," typeSpec | typeTuple "," typeSpec

```

```

parseSpec : "_" | INT | STRING | ID
           | "TUPLE" "(" parseSpecList ")" | "LIST" "(" parseSpec "," parseSpec ")"
           | "(" 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 | exp ";" | "{" exp "}"
     | "car" expUnit | "cdr" expUnit | "HEAD" expUnit | "TAIL" expUnit | "LEN" expUnit
     | "if" "(" exp ")" exp | "if" "(" exp ")" exp "else" exp
     | "for" "(" exp ";" exp ";" exp ")" exp
     | "switch" "(" exp ")" "{" cases "}"
     | "foreach" "(" exp ":" exp ":" exp ")" exp
     | "foreach_r" "(" exp ":" exp ":" exp ")" exp
     | "CONTINUE" | "BREAK" | "RETURN" 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 ":" exp | "-" 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 | "MAP" "(" typeSpec "," typeSpec ")"
     | exp "[" exp "]" | exp "#" expUnit

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

cases : cases "case" exp ":" exp
       | cases "default" ":" exp
       | "case" exp ":" exp

```