PLS Language Reference Manual
COMS W4115

Bennett Rummel
bwr2116@columbia.edu

July 2, 2014

Contents

1 Introduction

PLS (PipeLine Script) is intended to be a small, imperative, dynamically typed, statically scoped, programming language designed to process streams of data. It allows users to construct pipelines for processing data

based on the concept of Kahn processes [1]. A pipeline is made up of a sequence of discrete processes called `kernels`. `kernels` communicate through single direction FIFO pipes. A program is made of a sequence of one or more `kernels`.

2 Lexical Conventions

2.1 Comments

Comments begin with the characters "\\" and continue until the end of the current line.

2.2 Identifiers

An identifier is a sequence of letters, digits, and underscores ("_") beginning with a letter or an underscore. Identifiers in PLS are case-sensitive and can be of any length greater than one character.

2.3 Reserved Identifiers

The following identifiers are reserved and may not be specified as identifiers otherwise:

```
if      elseif    else
for     while     foreach
true    false     null
return  kernal    main
print   len       append
read    write     in
out     function
```

2.4 Constants

2.4.1 Integers

An integer constant is represented by one or more decimal ($[0-9]$) digits.

2.4.2 Floating Point Numbers

A floating point number constant follows the same format as the C programming language in [2]:

> A floating constant consists of an integer part, a decimal point, a fraction part, an e , and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

2.4.3 Boolean Values

Boolean constants are either `true` or `false`.

2.4.4 Strings

Strings are represented by any sequence of characters surrounded by double quotes ("`"`"). A double-quote character in a string must be escaped by a preceding backslash ("\"). Strings may not contain new line characters - instead the character sequence "`\n`" should be used.

2.4.5 Lists

A list is declared using the follow syntax:

$$identifier = \{expressions_{opt}\}$$

where *elements* is an optional comma-separated sequence of expressions. The new list will be bound to $identifier$ and will contain copies of the values in *expressions* (if any).

3 Syntax Notation

In the following portion of this manual depicting syntax, non-terminal symbols are displayed in *italics* while terminal symbols are displayed in `fixed width`. Optional symbols are indicated by a following subscript "opt".

4 Data Types and Conversions

4.1 Integers

Integers are represented by the platform's signed 64-bit integer types. Integers may be converted to a floating point number without loss of precision.

4.2 Floating Point Numbers

Floating point values (hereafter "floats") are represented by the platform's double precision IEEE-754 type (e.g. `double`). Converting a floating point value to an integer will cause it be truncated towards 0. If a floating point value is outside the range of values able to be represented by an integer, the result of the conversion is undefined.

4.3 Strings

Strings are represented by a sequence of characters of an arbitrary length. Strings may not be converted to other data types.

4.4 Boolean Values

Simple true/false type. Integers and floats will be evaluated as `true` in a boolean context if they are non-zero, and `false` otherwise. All other types are always converted to boolean `false`.

4.5 Lists

A list is a composite type representing an ordered sequence of 0 or more elements of heterogenous types. Elements in a list are referred to by integer indices starting at zero. Items can be modified in place or added to a list but not removed. Lists can also be nested to allow for multi-dimensional lists.

4.6 `null`

`null` represents the abscence of a value.

5 Variables and Objects

Variables in PLS are dynamically typed and are declared by assigning them a value (see below). A variable's value can be changed (including the data type) by assigning a new value. Functions, kernels, and pipelines can not be assigned to a variable unless otherwise indicated.

6 Expressions

6.1 Primary Expressions

Primary expressions are the basic "building blocks" of other expressions. That is, only a primary expression can result in a primary expression.

6.1.1 *identifier*

An identifier can be used as an expression if it exists in a scope visible from the current scope (see below). The expression results in the type and value of *identifier*.

6.1.2 *constant*

A constant can be used as an expression and its type is the most appropriate given how the exact string specified in the source code is parsed (see above).

6.1.3 (*expression*)

Simply yields *expression*.

6.1.4 $primary - expression[expression]$

A primary expression followed by an integer expression surrounded by square brackets is used to refer to a specific element in a list. The result of *expression* is used to retrieve the n'element of $primary - expression$, counting from zero. The result of the expression is a reference to the element, meaning the list can be changed by assigning a new value to the result of the expression (see below). It is an error if *expression* is not an integer, is less than 0, or is greater than or equal to the length of the list.

6.1.5 $identifier(expression_{opt}(,expression_{opt})*)$

An identifier followed by a pair of parentheses containing an optional comma-separated list of expressions is used to call a function named *identifier* with the results of the specified *expression*s as arguments, assuming it has been previously declared. The result of the expression is the return value of the function or `null` if the function does not return a value. See below for more information.

6.2 Unary Operators

6.2.1 $-expression$

If *expression* is an integer or a float, the expression computes the negative of *expression*. It is an error for *expression* to be of any other type.

6.2.2 !*expression*

If *expression* is a boolean, the expression computes the logical negation of *expression*. It is an error for *expression* to be of any other type.

6.3 Binary Operators

In the following operations, an expression in which one operand is an integer and the other is a float will invoke an implicit conversion of the integer to a float (see above) before the operation is performed unless otherwise indicated. Additionally, if any expression would result in a value outside the range able to be represented by its type, the result in undefined. Finally, the following operators are left-associative unless otherwise indicated.

6.3.1 *expression* `+` *expression*

If both expressions are numeric types (i.e. integers or floats), the expression yields the sum of the two values. If both expressions are strings, the expression yields the concatenation of the first string with the second string.

6.3.2 *expression* `-` *expression*

Computes the difference of the two operands if both are numeric types. Otherwise, it is an error.

6.3.3 *expression* `*` *expression*

Computes the product of the two operands if both are numeric types. Otherwise, it is an error.

6.3.4 *expression* `/` *expression*

Computes the quotient of the two operands. If both operands are integers, the fractional part of the result may be truncated towards 0. It is an error if the types of any operand is not numeric.

6.3.5 *expression* `%` *expression*

Computes the remainder of dividing the first expression by the second (i.e. the value of the first expression modulo the value of the second). It is an error unless both operands are numeric types.

6.3.6 *identifier_or_list_subscript* `=` *expression*

Assign to the variable or list subscript as the left operand the result of the right operand *expression*. This operator is right-associative. If the left operand is an identifier that is not bound, a new one is allocated and set to the value of *expression*. If the identifier is bound, its value is replaced. The result of the expression is that of *expression*.

6.3.7 *expression* `==` *expression*

Comparison operator – yields `true` if both operands represent the same value (after necessary type conversions), and `false` otherwise.

6.3.8 *expression* `!=` *expression*

Yields `true` of the operands do not represent the same value, `false` otherwise.

6.3.9 *expression* `<` *expression*

Yields `true` if both operands are numeric types and the value of the first is strictly less than the value of the second. It is an error if either type is not numeric.

6.3.10 *expression* `>` *expression*

Yields `true` if both operands are numeric types and the value of the first is strictly greater than the value of the second. It is an error if either type is not numeric.

6.3.11 *expression* `<=` *expression*

Yields `true` if both operands are numeric types and the value of the first is less than or equal to the value of the second. It is an error if either type is not numeric.

6.3.12 *expression* `>=` *expression*

Yields `true` if both operands are numeric types and the value of the first is greater than or equal to the value of the second. It is an error if either type is not numeric.

6.3.13 *expression* `&&` *expression*

Computes the logical “and” of both expressions. It is an error if either expression is not a boolean.

6.3.14 *expression* || *expression*

Computes the logical "or" of both expressions. It is an error if either expression is not a boolean.

7 Statements

7.1 Expressions

A statement may have the form:

expression;

7.2 Statement Lists

Statements may be placed back-to-back and will be executed sequentially.

7.3 Blocks

Statements may also take the form:

$\{statement_{opt}\}$

Such a statement may be useful for improving code readability or limiting the scope of a variable (see below).

7.4 `if` Statement

A `if` statement has the form:

$\texttt{if}(expression)\{statement_{opt}\}\,(\texttt{elseif}(expression)\,\{statement_{opt}\})*\,(\texttt{else}\,\{statement_{opt}\})?$

In any case, if the first *expression* evaluates to `true` than the first *statement* is executed, otherwise it falls through to the next group in the statement. This continues for each (optional) `elseif`, *expression*, and *statement* group. If none of the *expressions* evaluate to `true`, then the *statement* following the (also optional) `else` is executed. It is important to note that only the *expression*s up to the first one that results in `true` will be evaluated.

7.5 `while` Loop Statement

The `while` statement has the form:

$\texttt{while}(expression)\{statement_{opt}\}$

If the *expression* evaluates to true, then *statement* is executed once. Then the cycle repeats until *expression* evaluates to false.

7.6 `for` Loop Statement

The `for` statement has the form:

$$\texttt{for}(expression_{opt};expression_{opt};expression_{opt})\{statement_{opt}\}$$

In this statement, the first *expression* is executed once before anything else. Any new variables are local to the statement (see below). The second *expression* is the termination test – if it evaluates to `true`, then the *statement* is executed once. The third *expression* specifies an operation to be performed after each execution of *statement*. After *statement* and the third *expression* are executed and evaluated once, the test is checked again and the cycle repeats. Any of the *expression*s may be omitted.

7.7 `foreach` Loop Statement

The `foreach` statement has the form:

$$\texttt{foreach}(identifier{:}expression)\{statement\}$$

In this statement, *expression* must produce a list. $identifier$ is bound to a copy of each element of the list starting with index 0 and proceeding in order to the final element. *statement* is executed once for element in the list.

7.8 `return` Statement

The `return` statement has the form:

$$\texttt{return}(expression);$$

It is valid only in a function definition (see below). When a `return` statement is executed, the value of *expression* is used as the result of the function call expression (see above) and control is returned to the caller.

8 Functions

8.1 Function Definitions

A function definition has the form:

$$\texttt{function}\ identifier(arglist_{opt})\{statement\}$$

where *arglist* is the optional comma-separated list of identifiers representing the parameters to the function. The function name $identifier$ is bound to the function. Functions cannot carry state from one invocation to the next. Functions can optionally contain one or more `return` statements to return a value to the caller (see above). If control reaches a `return` statement, then control is returned to the caller. If control reaches the end of the body of the function, `null` is returned (i.e. there is an implicit "`return(null);`" at the end of each function definition). Function arguments are passed by value, meaning a function can not directly affect the value of its parameters when called (except if the function is passed a global variable, see below).

Functions cannot be nested (they must be defined in global scope), preventing constructs such as closures. Additionally, they are not first-class types and can therefore not be assigned or passed to other functions.

8.2 Special Functions

There are several builtin functions defined.

8.2.1 `print(x)`

`print` is simply prints an ASCII representation of its single argument to `stdout` and returns `null`.

8.2.2 `len(l)`

`len` returns an integer containing the number of elements in `l` if `l` is a list, or `null` otherwise.

8.2.3 `append(l, x)`

Append `x` to the end of list `l`. Returns `l`.

8.2.4 `read(c)`

`read` is a special function used to pop a single token from a `kernel`'s input pipe `c`. If no tokens are available, the call will block until one is available. Calling `read` from anywhere but a `kernel` definition is an error.

8.2.5 `write(c, x)`

`write` is a special function used to push a single token `x` to a `kernel`'s output pipe `c`. `write` will not block. Calling `write` from any context but a `kernel` definition is an error.

9 Kernels

A `kernel` is a discrete processing unit. It communicates with other `kernel`s via single direction FIFO queues. `kernel`s appear similar but have different semantics than functions.

9.1 Kernel Definition

A `kernel` definition has the form:

`kernel` $identifier(arglist)\{statement\}$

Where $identifier$ is the name of the `kernel`. The $arglist$ is a comma separated list of input and output channels. An single argument in the $arglist$ has the form:

$direction\ identifier$

where $direction$ is either `in` for an input channel or `out` for an output channel. Currently a `kernel` must have exactly one of each. A `kernel` must also have a body, $statement$, that is executed once in parallel with the other `kernel`s in the program.

9.2 Kernel Input and Output

There are two special functions available for use inside of a `kernel` body: `read` and `write` (see above). It is important to note that `kernel` inputs and outputs need not be symmetric, e.g. a `kernel` could read two tokens for every one that it outputs. Additionally, the communication pipes are unbounded. Finally, there is no way for a `kernel` to check if a pipe has any tokens or is empty.

10 Scope

Variables are lexically scoped, meaning they are valid within the scope in which they are first declared, including "child" scopes. Variables can also be declared in the global scope and are visible in every subsequent scope. Function parameters have the scope of the function and will hide any other variable or function with the same name.

11 Pipelines

A pipeline is a chain of `kernel` instances where the output of one is connected to the input of the next, and so on. A pipeline is built from left to right using the "|" operator:

$$kernel \mid kernel$$

This means the output of the first *kernel* is fed in to tho input of the second *kernel*. This expression returns a pipeline.

Pipelines can be assigned to variables in the global scope for ease of construction. the "|" operator can be used to connects pipelines in the same way it connects `kernel`s.

Recursion is prohibited in pipelines, meaning no cycles are allowed in the flow of data.

12 Programs

A program is simply a pipeline or kernel assigned to the special identifier `main`. ASICC input from stdin is fed in to the pipeline, and the output is printed to stdout. Tokens from the input are parsed one at a time according to the same rules as in the "Constants" section above and pushed in to the pipe at the head of `main`.

Because PLS is designed around the concept of Kahn processes, programs are deterministic, meaning a program will always produce the same sequence of output values given the same input. However, the exact order of operations within a program is unspecified to allow for parallel computation.

References

[1] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74 : proceedings of IFIP Congress 74*. North-Holland American Elsevier, 1974.

[2] B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.