Physicalc:
A Language for (simple) Scientific Computation

Brian Foo, bwf2101@columia.edu
Changlong Jiang, cj2214@columbia.edu
Ici Li, il2117@columbia.edu
Stuart Sierra, ss2806@columbia.edu

Language Reference Manual – October 18, 2007

Contents

1 Introduction

Physicalc is a simple programming language for scientific computation, designed for students studying beginning and intermediate-level physics, chemistry, or other sciences. It has a plain-English syntax inspired by BASIC, and a mathematical model that can perform operations using scientific units. It is intended as an educational tool.

2 Prior Art

- Units[8], a command-line program included in early UNIX systems and GNU/Linux, provides conversion factors between various units.
- Calchemy[1] is software for the Palm OS that combines a scientific calculator with unit conversion and dimensional analysis. It is based on an earlier Windows program called UNICALC.
- The Google Calculator[2] performs arithmetic on numbers including unit conversions.
- JScience[4] is a Java library for scientific computation including units.

3 Conventions in This Document

In this document, text in `monospace type` indicates a keyword or literal syntax. Text in *italic type* indicates a placeholder for some other piece of source code.

4 Program Sources

4.1 Encoding

Physicalc accepts source files encoded in plain 7-bit ASCII. High bit characters are not allowed. Line breaks may be encoded in DOS `\r\n`, UNIX `\n`, or Macintosh `\r` style. ASCII characters with decimal value less than 9, i.e. the control characters, are not allowed.

4.2 Comments

Comments begin with a hash character (`#`) and continue to the end of the line. Comments may be placed on the same line as source code. The line break is not considered part of the comment text.

4.3 Whitespace

Whitespace characters—spaces, tabs, and form feed characters—may used to separate tokens in the input but are discarded during parsing.

4.4 Line Breaks and Semicolons

Line breaks are significant in Physicalc syntax. Line breaks serve as statement terminators. In the syntax rules that follow, all line breaks shown are mandatory.

To put multiple statements on a single line of source code, semicolons may be used in place of line breaks. Semicolons may be used anywhere a line break would normally be used, including between the parts of compound statements such as `if/elsif/else`.

Any number of consecutive line breaks and/or semicolons is read as a single terminator.

4.5 Identifiers

All identifiers begin with a letter or underscore, followed by zero or more letters, digits, and underscores. Identifiers are case-sensitive.

4.6 Reserved Words

The following words are reserved as keywords and may not be used as identifiers: `alias and break constant do done else elsif false for from function if in loop next not or quantity return set step then to true unit while`

Additionally, Physicalc defines several built-in functions (Section 9) which may not be redefined.

5 Fundamental Types

5.1 Numbers

All numbers in Physicalc are treated as arbitrary-precision decimals. There is no distinction among integers, rationals, and reals. All arithmetical calculations are decimal-accurate to a reasonable degree of precision. Floating-point arithmetic is never used.

Literal numbers may be written in source code as integers or as decimals using C-style floating point number syntax. A number has three parts:

1. An integer part composed of digits;
2. A decimal part composed of a `.` (period) character followed by digits;
3. An exponent beginning with the letter `e` (or `E`), followed by an optional `+` or `-` sign, followed by digits.

All parts are optional, but either the integer or decimal part must be present.

5.2 Strings

Strings are sequences of ASCII characters. Literal strings may be written in source code by placing them between double-quotation (`"`) characters. A literal `"` character is written in a string as two consecutive `"` characters, so the string

The “big” bus

could be written as

```
"The ""big"" bus"
```

C-style character escapes (`\n`, `\t`, etc.) are not supported. Line breaks are permitted inside strings.

5.3 Lists

Lists are one-dimensional, variable-length, zero-indexed arrays of objects. Lists are heterogeneous—they may contain any combination of different object types. Lists are automatically resized.

A list literal may be written in source code by enclosing the entire list in square brackets (`[` and `]`) and separating individual elements with commas. Whitespace, but not line breaks, is permitted between list elements; it is ignored. Nested lists are allowed. The elements in a list literal may be expressions; those expressions are evaluated and their results are stored in the list.

5.4 Vectors

Vectors are two-dimensional x, y component pairs. The components of a vector must be numbers, optionally including units (see Section 6.3). If the components of a vector include units, both components must have units of the same quantity (see Section 6.2).

Literal vectors may be written in source code by enclosing the entire vector in curly brackets (`{` and `}`) and separating the components with commas. Whitespace is permitted between vector components. The components in a vector literal may be expressions; those expressions are evaluated and their results are stored in the vector.

Nested vectors, vectors with more than two dimensions, and matrices are not supported.

5.5 Booleans

Boolean values are the literal identifiers `true` and `false`. These are global built-in constants and may not be redefined. In statements that use boolean expressions, any value that is not exactly equal to `false` is considered true. For example, the empty list and the number zero are both "true" in a boolean context.

6 Definitions

A typical Physicalc program consists primarily of definitions. There are five types of definitions: quantities, units, constants, functions, and aliases.

A definition permanently associates some identifier with an object. All definitions must occur at the top level of program scope; they may not be nested. Defintions are present in the running program from the point at which they are defined until the program terminates. Defined identifiers have global scope, and may not be overridden by local variables with the same name. An identifier, once defined, may be redefined, but this will cause the interpreter to print a warning message.

6.1 Note on Definitions Using Expressions

Some of the definition types below take the form *identifier*`=`*expression*. In these cases, the `=` is not an operator; it is part of the syntax. The *expression* that follows the `=` symbol is evaluated once, at the time the definition is read, and its result is stored in the *identifier*.

6.2 Quantities

Quantities are part of the Physicalc system for handling calculations involving scientific units. A quantity represents a physical property which can be measured or calculated, independent of the system of units used to measure it. Examples of quantities are mass, length, time, velocity, and force.

6.2.1 Base Quantities

Base quantities are fundamental quantities that are not mathematically derived from other quantities. The SI unit system has seven base quantities: length, time, mass, electric current, temperature, amount of substance, and luminous intensity.[3] Other unit systems might have different base quantities.

Base quantities are defined with the keyword `quantity`, followed by an identifier for the quantity.

Syntax: `quantity` *identifier*

6.2.2 Derived Quantities

Derived quantities are defined by mathematical relationships to existing quantities. An example of a derived quantity in the SI system is velocity, defined as distance divided by time.

Syntax: `quantity` *identifier* `=` *expression*

where *expression* is a restricted expression on previously defined quantities. See Section 7.8 for the meaning of "restricted expression."

An identifier that has been defined as a base quantity may not be subsequently redefined as a derived quantity; to do so is an error.

6.3 Units

A unit is a concrete standard of measurement for some physical quantity.

6.3.1 Base Units

Base units are units for base quantities. Examples of base units in the SI system are meters for length, seconds for time, and kilograms for mass.[6] Base units are defined by associating them with a quantity.

Syntax: `unit` $identifier_1$ `for` $identifier_2$

where $identifier_2$ is a previously-defined quantity and $identifier_1$ is the name of the new unit.

6.3.2 Derived Units

Derived units are defined in terms of mathematical relationships to other units. An example of a derived unit in the SI system is the Coulomb, defined as seconds multiplied by Amperes.[7] Derived units are defined in Physicalc with expressions involving other units.

Syntax: `unit` *identifier* `=` *expression*

where *expression* is a restricted expression on previously defined units. See Section 7.8 for the meaning of "restricted expression."

A derived unit may also be defined as a conversion factor from another unit for the same quantity. Typically, this type of derived unit will have a definition *expression* consisting of a base unit multipied by some constant number. In this way, conversion factors between different systems of measurement may be defined. Those conversion factors may be used for automatic unit conversion with the `in` operator (Section 7.7).

An identifier that has been defined as a base unit may not be subsequently redefined as a derived unit; to do so is an error.

6.4 Constants

Constants are static identifiers that hold any type of value. They have global scope and may not be changed by assignment. Constants may be changed by redefining them, but this produces a warning.

Syntax: `constant` *identifier* `=` *expression*

where *expression* is any expression.

6.5 Functions

Functions are named subroutines which receive input and return output. All function parameters are passed by value; i.e. a copy of the parameter is made and the function operates on the copy. Functions cannot modify any objects in their calling environment, nor can they define new global objects.

The first line of a function definition consists of the keyword `function`, an identifier, and a parameter list enclosed in parentheses. The body of the function, a sequence of statements, follows. The function definition ends with the keyword `done`.

Syntax: `function` *identifier* `(` *parameter list* `)`
statements
`done`

The indentation of *statements* is for easier reading and has no significance in the syntax. The parameter list consists of zero or more identifiers, separated by commas. The parentheses around the parameter list are mandatory, even if the parameter list is empty. Whitespace, but not line breaks, is allowed in the parameter list. Functions taking a variable number of parameters are not supported, but this can be achieved in practice by passing a list as one of the parameters.

Within the body of a function, a `return` statement (Section 8.3) terminates the function and returns its argument as the value of the function. The body of a function may refer to identifiers not yet defined, but those identifiers must be defined before the function is called or an error will result. See Section 7.4 for the syntax of function calls.

6.6 Aliases

To allow for multiple names for the same object, aliases may be defined. An alias is an identifier that may be used in place of another identifier. Aliases are alternate names for an object rather than true references. Aliases are subject to the same redefinition constraints as other definitions.

Syntax: `alias` *identifier*$_1$ `for` *identifier*$_2$

defines *identifier*$_1$ as a new alias for *identifier*$_2$, a previously-defined identifier. Subsequent uses of *identifier*$_1$ will be read as if they were *identifier*$_2$. It is an error if *identifier*$_2$ is not already defined.

Operator	Description	Associativity
`( )`	Grouping	N/A
identifier `[ ]`	List subscript	left
identifier `( )`	Function call	left
`{ }`	Vector Literal	N/A
`[ ]`	List Literal	N/A
`-`	Unary minus	right
`^`	Exponentiation	right
`* /`	Multiplication/Division	left
`+ -`	Addition/Subtraction	left
`> < >= <=`	Relational Comparison	left
`= !=`	Equality Comparison	left
`not`	Logical NOT	right
`and`	Logical AND	left
`or`	Logical OR	left
`in`	Unit Conversion	left
`,`	Comma separating expressions	left

Highest precedence is on the top line of the table.

Figure 1: Operator Precedence

7 Expressions

Expressions consist of operators, function calls, literals, and identifiers. An expression, when evaluated, returns a value. Operator precedence is summarized in Figure 1. The types of expressions are described below. Expressions may contain whitespace, which is ignored, but they may not contain line breaks.

7.1 Arithmetical Expressions

Arithmetical expressions consist of the unary negation operator (`-`), parentheses `()`, and binary operators for addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`^`)

The unary negation operator takes one argument on the right, and returns its opposite. There is no unary plus operator, because it serves no purpose that the authors can imagine.

Binary operators take one argument on the left and one argument on the right. The caret operator (`^`) performs exponentiation, raising its left argument to the value of its right argument. The operator `^` has highest precedence, followed by unary `-`, followed by binary `*` and `/`, followed by binary `+` and `-`.

Parentheses are used for grouping expressions and specifying precedence explicitly. An expression inside parentheses is always evaluated before other expressions. Parentheses may be nested to any level (within the bounds of computer memory) and the inner-most parenthetical expression will be evaluated first.

Arithmetic may be performed on numbers, vectors, and units. All arithmetical operators are supported when the operands are both numbers. All operations except exponentiation are supported between two vectors. A vector may be multiplied or divided by a number, and a number may be

		Right Operand				
		Number	Vector	Unit	String	List
Left Operand	Number	+ - * / ^	*	* /		
	Vector	* /	+ - * /	* /		
	Unit	* / ^		* /		
	String				+	
	List					+

Figure 2: Permitted Binary Operations by Operand Type

multiplied by a vector, but a number may not be divided by a vector. Units may be multiplied and divided with other units, numbers, and vectors, but may not be added or subtracted (see Section 7.2). Units may be raised to a numerical exponent. Finally, concatenation is supported for strings and lists using the binary `+` operator. Figure 2 summarizes the supported binary operations.

7.2 Combining Numbers and Units

Mathematically, numbers with units are said to be "multiplied" by a symbol representing the unit. In Physicalc, this is taken literally. Units are identifiers, and a number with units is simply an expression of the form "*number*`*`*identifier*" where *identifier* has been defined as a unit. Units are preserved in calculations and results. Limited handling of units as algebraic expressions is supported, so an expression such as "three meters per second multiplied by ten seconds" could be written

```
3 * meter / second * 10 * second
```

and would return the correct result of thirty meters as `30*meter`. Calculations requiring unit conversions might not always return the desired units in the result; the `in` operator (Section 7.7) forces conversion to the correct units.

7.3 List Member Access

Once a list has been stored in a variable, its elements may be accessed using bracketed indexes.

Syntax: *identifier* `[` *expression* `]`

where *expression* must evaluate to an integer, which is used as an index into the list stored in *identifier*. An attempt to access an index beyond the end of the list produces an error.

Bracketed indexes are only permitted on identifiers, not on literal lists nor on expressions that return a list. Items in nested lists may be accessed with multiple consecutive bracket expressions, so if the variable `x` contained the list

```
[ a, b, [ c, d ], e ]
```

the element `d` could be referenced as `x[2][1]`.

7.4 Function Calls

Built-in or user-defined functions are called with the name of the function, an opening parenthesis, an argument list, and a closing parenthesis. No space is permitted between the function name and the opening parenthesis. The parentheses are mandatory even if the argument list is empty.

Syntax: *identifier* `(` *argument list* `)`

The argument list is a sequence of expressions, separated by commas. The number of arguments in the argument list of the function call must match the number of arguments in the function definition.

When a function is called, the expressions in the argument list are evaluated. A new local scope is created, and the values of the arguments are bound to the named parameters from the function definition.

7.5 Relational Operators

Numbers, and only numbers, may be compared with the standard relational operators `>`, `<`, `>=`, and `<=`. These operators all return a boolean value.

Any two objects may be compared with the equality operator, `=`, which returns true if its left operand is of the same type and has the same value as its right operand, and false otherwise. Two vectors are equal if both of their components are equal. Two units are equal if they are the same unit.

The not-equals operator, `!=`, returns true if its operands are not equal under the definition of equality used for `=`, and false otherwise.

7.6 Logical Operators

Logical operators work on boolean values and expresions.

Syntax: `not` *expression*

returns true if *expression* is false and returns false if *expression* is true.

Syntax: $expression_1$ `and` $expression_2$

returns true if both $expression_1$ and $expression_2$ are true. This operator "short-circuits"—if $expression_1$ is false, it returns false without evaluating $expression_2$.

Syntax: $expression_1$ `or` $expression_2$

returns true if $expression_1$ is true, $expression_2$ is true, or both are true. This operator "short-circuits"—if $expression_1$ is true, it returns true without evaluating $expression_2$.

7.7 Unit Conversion

The special binary `in` operator is used to convert values from one set of units to another.

Syntax: *expression*$_1$ `in` *expression*$_2$

where *expression*$_1$ is an expression that evaluates to a number or vector with units, and *expression*$_2$ evaluates to units. The `in` operator searches through the set of defined relationships among units and quantities to find the correct conversion factor, applies that conversion, and returns the result number or vector in the requested units. It is an error if *expression*$_1$ and *expression*$_2$ do not have the same quantity, or if a valid conversion factor between the units of *expression*$_1$ and the units of *expression*$_2$ cannot be found.

7.8 Restricted Expressions

Some definitions (Section 6) refer to "restricted expressions." These are expressions involving only numbers, identifiers, parentheses, and the arithmetical operators `+`, `-`, `*`, `/`, and `^`. Some restricted expressions are described as "on" a type, such as units or quantities. In those cases, the restricted expression is futher limited to using identifiers only of that type.

8 Statements

Statements are source code constructs which do not return a value. An expression may be used as a statement by itself; its return value is discarded.

8.1 Loading Source Files

The special `load` statement reads in additional source files.

Syntax: `load` *string*

The *string* is interpreted as a path on the local filesystem, relative to the current working directory, to a file containing Physicalc source code. That file is read and its contents are executed as if they had been included in the current program at the point of the `load` statement. Any definitions in the loaded file will become part of the running program.

If the file cannot be found or cannot be read, an error results. `load` is only allowed at the top-level of a program source file; it may not appear inside functions or other statements.

8.2 Assignment

Syntax: `set` *identifier* `=` *expression*

An assignment statement evaluates *expression*, then binds its value to the local variable named *identifier*. If *identifier* is currently undefined, a new local variable is created with scope corresponding to the current function body. It is an error if *identifier* is already defined as a global object, i.e. a measure, unit, function, or constant.

Assignment statements may be used outside of a function body, but doing so does not create a global variable. Global variables are not supported, only global constants.

8.3 Return

Syntax: `return` *expression*

A `return` statement may only appear inside the body of a function; a `return` statement found outside of a function body is an error. When a `return` statement is executed, *expression* is evaluated and its value is returned as the value of the function.

8.4 If/Then/Else

An if/then/else block begins with the keyword `if`, followed by an boolean expression, followed by the keyword `then` and a terminator (line break or semicolon). After `then` comes a sequence of one or more statements, then any number of `elsif` blocks, then an optional `else` block, then finally the keyword `done`.

Syntax:
```
if expression1 then
      statements1
elsif expression2 then
      statements2
... additional elsif blocks ...
else
      statements3
done
```

The indentation of the statement blocks is for easier reading and is not significant in the syntax. First, $expression_1$ is evaluated. If it returns true, $statements_1$ are executed. After completing $statements_1$, control passes to the statement following the `done` keyword.

If $expression_1$ returns false, $expression_2$ is evaluated. If $expression_2$ returns true, $statements_2$ are executed, then control passes to the statement following the `done` keyword. Additional `elsif` blocks may specify additional test expressions and statements to execute. If all of the test expressions return false, and if the optional final `else` block is present, $statements_3$ are executed.

An if/then/else block might not execute any statements at all if there is no `else` block. An if/then/else block never executes more than one group of statements. Once the first test expression returns true, its associated statement block is executed and all other test expressions and statement blocks are skipped.

8.5 While Loops

Syntax:
```
while expression do
      statements
done
```

While loops evaluate a group of statements as long as a given conditional expression remains true. The conditional expression is evaluated before the statement body on every iteration of the loop. If it returns true, the statements are executed. The first time it returns false, control passes to the statement following the `done` keyword.

8.6 For Loops

Syntax: `for` *identifier* `from` *expression*$_1$ `to` *expression*$_2$ `step` *expression*$_3$ `do`
statements
`done`

At the beginning of a for loop, *expression*$_1$, *expression*$_2$, and *expression*$_3$ are all evaluated immediately. All three must evaluate to positive numbers, optionally including units; it is an error if they do not. The result of *expression*$_1$ is assigned to the local variable *identifier*. If *identifier* is undefined, a new local variable is created with scope corresponding to the current function body. It is an error if *identifier* is already defined as a global. The *statements* are executed, after which the value of *expression*$_3$ is added to the value in *identifier*, and that new value is stored in *identifier*. Then the value of *identifier* is compared to the value of *expression*$_2$. If *identifier* is greater than the value of *expression*$_2$, control passes to the statement following the `done` keyword. If *identifier* is less than or equal to the value of *expression*$_2$, *statements* are executed again. This process repeats until *identifier* is greater than the value of *expression*$_2$ or a `break` or `return` statement is executed.

8.7 Control Statements Within Loops

Within any loop structure there are two special statements which affect the execution of the loop. The `break` statement will immediately terminate the execution of the loop and transfer control to the statement following the loop's `done` keyword.

The `next` statement will immediately return control to the top of the loop. In the case of `while` loops, the loop test is applied as if the loop had reached the end of its statement block. In the case of `for` loops, the counter variable is incremented and the test is applied as if the loop had reached the end of its statement block.

Additionally, a loop used inside of a function body may contain a `return` statement, which will immediately break out of the loop and return from the function.

9 Built-In Functions

Physicalc provides a some built-in functions, which have the same syntax as normal function calls but carry out operations which could not be implemented with normal Physicalc code.

9.1 print()

The `print()` function takes a single argument, which may be any object, and prints it to the output stream, followed by a line break. Printing a string prints its contents without the enclosing `"` characters. Lists, vectors, units, and numbers are automatically converted to strings with the `toString()` function before being printed. Printing multiple objects with a single call to `print()` requires converting them to strings and concatenating those strings.

9.2 nprint()

The `nprint()` function acts like `print()` but does not print a line break.

9.3 toString()

The `toString()` function converts its argument, which may be any object, to a string. If the argument is already a string, it is simply returned. Other types of arguments are converted to a string that matches their literal syntax.

9.4 stripUnits()

The `stripUnits()` function takes an argument of a number with units and removes all units, leaving just the bare number. If a bare unit without any numbers is passed to the function, it returns the number one.

9.5 toPrecision()

The `toPrecision()` function takes two arguments, a number n and an integer i, and rounds n to have i significant digits. Rounding is performed using the unbiased "round-to-even" method.[5] If the number includes units, those units are preserved in the result.

9.6 toInt()

The `toInt()` rounds its argument to the nearest integer, using the same "round-to-even" method as `toPrecision()`. If the number includes units, those units are preserved in the result.

9.7 read()

The `read()` function takes no arguments. It reads a single value from the input stream, ignoring all preceeding whitespace and continuing up to the first whitespace character. The value is parsed as an expression of numbers and units and returned to the caller.

9.8 readline()

The `readline()` function, which takes no arguments, reads a line from the input stream up to the first linebreak character and returns it as a string.

9.9 exit()

The `exit()` function, which takes no arguments, immediately terminates the Physicalc program.

10 References

[1] Calchemy, http://www.calchemy.com/

[2] Google Calculator, http://www.google.com/help/calculator.html

[3] "International System of Units." Wikipedia, http://en.wikipedia.org/wiki/Si_units

[4] JScience, http://jscience.org/

[5] "Rounding numbers." Wikipedia, http://en.wikipedia.org/wiki/Rounding_numbers

[6] "SI base unit." Wikipedia, http://en.wikipedia.org/wiki/SI_base_unit

[7] "SI derived unit." Wikipedia, http://en.wikipedia.org/wiki/SI_derived_unit

[8] Units, GNU Project, http://www.gnu.org/software/units/units.html

11 Grammar

```
/* ************************************************************************
 * grammar.g : the lexer and the parser, in ANTLR grammar for Physicalc
 *
 * ANTLR Parser Generator Version 2.7.7 (2006-11-01)
 *
 * @author Changlong Jiang, cj2214@columbia.edu
 * @author Stuart Sierra, ss2806@colmbia.edu
 *
 * @version 1.0
 * ************************************************************************/


header {
    package physicalc;
}

/* ************************************************************************
 * LEXER *
 * ************************************************************************ */
class PhysiLexer extends Lexer;

options {
    charVocabulary = '\11'..'\177';  // Plain 7-bit ASCII
    testLiterals = false;
    k = 2;  // for >= or <= operators
}

protected DIGIT : '0'..'9';
protected LETTER : 'a'..'z' | 'A'..'Z';


/** Identifiers must begin with a letter or underscore, which may be
 * followed by any combination of letters, digits, and underscores. */
ID  options { testLiterals = true; }
    : ( LETTER | '_' ) ( LETTER | DIGIT | '_' )*;

/** Whitespace is ignored. */
WHITESPACE : (' ' | '\t' | '\f')+ { $setType(Token.SKIP); };

/** Line breaks are significant as statement separators, but are not
 * tokens on their own. */
protected NEWLINE : ('\n' | ('\r' '\n') => '\r' '\n' | '\r')
    { newline(); } ;
```

```
/** Comments begin with '#' and go to the end of the line.  Since line
 * breaks are used as statement separators, the comment text does NOT
 * include the newline. */
COMMENT : '#' ( ~( '\n' | '\r' ) )*
    { $setType(Token.SKIP); } ;

/** Statements are terminated by (any number of) newlines or
 * semicolons. */
TERMINATOR : (NEWLINE | ';')+;


/** There is no syntactic distinction among integers, decimal numbers,
 * and numbers with exponents.  They're all just numbers. */
NUMBER : ( (DIGIT)+ ( '.' (DIGIT)* )?
         | '.' (DIGIT)+
         )
         ( ('e'|'E') ('+'|'-')? (DIGIT)+ )?
         ;

/** Strings are surrounded by double quotation marks. A double
 *  quotation character may be inside a string by using two double
 *  quotation marks in a row. */
STRING  : '"'!
              (  ~('"')
               | ('"'!'"')
              )*
          '"'!
        ;

PLUS   : '+';
MINUS  : '-';
TIMES  : '*';
DIVIDE : '/';
CARET  : '^';
LPAREN : '(';
RPAREN : ')';
LBRACKET : '[';
RBRACKET : ']';
LBRACE : '{';
RBRACE : '}';
COMMA  : ',';
EQ     : '=';  // we use 'set' for assignment
NEQ     : "!=";
RELOP  : '>' | '<' | ">=" | "<=" ;
```

```
/* ****************************************************************
 * PARSER *
 * **************************************************************** */
class PhysiParser extends Parser;

options {
    k = 2;
    buildAST = true;
}

tokens { /* used in the abstract syntax tree */
    BLOCK;
    FUNCALL;
    IF;
    LIST;
    PARAMS;
    SUBSCRIPT;
    UMINUS;
    VECTOR;
}

program : (load | def | stmt)+;

/** Load statement: load the file at the path given. */
load : "load"^ STRING TERMINATOR!;

/* ************************************************
 * Definitions
 * ************************************************ */

/** Definitions. */
def : (quantity_def | unit_def | constant_def | alias_def | function_def)
      TERMINATOR;

quantity_def : "quantity"^ ID (EQ! expr)?;

unit_def : "unit"^ ID ("for"! ID | EQ! expr);

constant_def : "constant"^ ID EQ! expr;

alias_def : "alias"^ ID "for"! ID;

function_def : "function"^ ID LPAREN! params RPAREN! TERMINATOR!
    block "done";

/** Parameter list for function definitions. */
```

```
params : (ID)? (COMMA! ID)*
    {#params = #([PARAMS, "PARAMS"], params); } ;

/* ************************************************
 * Statements
 * ************************************************ */

/** Statements. */
stmt : simple_stmt | compound_stmt;

/** A block of one or more statements. */
block : (stmt)+
    {#block = #([BLOCK, "BLOCK"], block); } ;

/** Simple statement: A single-line statement that must end with a
 * TERMINATOR.  An expression by itself can be a statement.*/
simple_stmt :
    ( expr
    | "return"^ expr
    | "next"^
    | "break"^
    | "set"^ ID EQ! expr
    )
    TERMINATOR! ;

/** Compound statement: a multi-part statement like if/then/else or
 * while.  Compound statements always end with "done". */
compound_stmt : (if_stmt | while_stmt | for_stmt) "done"! TERMINATOR!;

/** If/then/else.  These rules transfrom an if/elsif/else sequence
 * into nested IF trees.  Each IF subtree has 3 arguments: (1) a test
 * expression, (2) a "then" block, and (3) an optional "else"
 * block. */
if_stmt :
    "if"! expr "then"! TERMINATOR! block
    elsif_stmt
    {#if_stmt = #([IF,"IF"], if_stmt); } ;

elsif_stmt : "elsif"! expr "then"! TERMINATOR! block (elsif_stmt)
    {#elsif_stmt = #([IF,"IF"], elsif_stmt); }
    | else_stmt ;

else_stmt : "else"! TERMINATOR! block
    | /* nothing */ ;
```

```
/** "while" loops. */
while_stmt : "while"^ expr "do"! TERMINATOR! block;

/** "for" loops. */
for_stmt : "for"^ ID "from"! expr "to"! expr "step"! expr "do"! TERMINATOR!
           block ;


/* ************************************************
 * Expressions
 * ************************************************ */

/** A list of expressions, separated by commas.  Used in literal lists
 * and function calls.  */
expr_list : expr (COMMA! expr)* ;


/** Expressions */
expr: in_expr;

/* Every binary operator is can repeat infinitely with a '*' closure.
 *
 * This parses expressions like "a < b < c" as "(< (< a b) c)", which
 * makes no sense.
 *
 * However, changing the '*' to a '?' means that anything after the
 * first operator gets ignored, which is clearly wrong.  Better let
 * the back-end decide if "(< (< a b) c)" is reasonable. */

in_expr : or_expr ( "in"^ or_expr )*;

or_expr : and_expr ( "or"^ and_expr )*;

and_expr : not_expr ( "and"^ not_expr )*;

not_expr : ("not"^)? eq_expr;  /* 'not' expressions cannot be chained */

eq_expr : neq_expr (EQ^ neq_expr)*;

neq_expr : rel_expr (NEQ^ rel_expr)*;

rel_expr : add_expr (RELOP^ add_expr)*;

add_expr : mul_expr ( (PLUS^ | MINUS^) mul_expr )*;

mul_expr : exp_expr ( (TIMES^ | DIVIDE^) exp_expr )*;
```

```
/** Exponentiation: tail-recursion makes it right-associative. */
exp_expr : uminus_expr (CARET^ exp_expr)?;

/** Unary negation operator. Unary plus ("+") is not included because
 * it's meaningless. */
uminus_expr :
    MINUS! atom
    {#uminus_expr = #([UMINUS, "UMINUS"], uminus_expr); }
    | atom;


/** atomic expressions (highest precedence) */
atom
    : ID
    | NUMBER
    | STRING
    | list_literal
    | vector_literal
    | subscript_expr
    | funcall_expr
    | LPAREN! expr RPAREN!
    ;

/** Literal list (in square brackets) */
list_literal : LBRACKET! expr_list RBRACKET!
    {#list_literal = #([LIST,"LIST"], list_literal); };

/** Literal vector (in curly brackets, must have exactly 2 elements. */
vector_literal : LBRACE! expr COMMA! expr RBRACE!
    {#vector_literal = #([VECTOR,"VECTOR"], vector_literal); };

/** Array/list subscripts like "a[b]".  Back-end is responsible for
 * checking that the subscript evaluates to an integer.  Chained
 * subscript expressions like a[b][c] are allowed, but the first token
 * (the 'a') must be an identifier. */
subscript_expr : ID (LBRACKET! expr RBRACKET!)+
    {#subscript_expr = #([SUBSCRIPT, "SUBSCRIPT"], subscript_expr); };

/** Function calls */
funcall_expr : ID LPAREN! expr_list RPAREN!
    {#funcall_expr = #([FUNCALL, "FUNCALL"], funcall_expr); };
```