PLATO Language Reference Manual

Daniel Perlmutter (dap2163) -- Project Leader
Yasser Abhoudkhil (ya2282) Joaquín Ruales (jar2262)

1) Introduction

This document describes the PLATO programming language. There are seven sections to this manual:

1) This introduction
2) Tokens -- a description of each class of tokens
3) Types -- a brief description of each primitive and derived data type, including syntax
4) Expressions -- parts of statements which need to be evaluated during execution
5) Statements -- components of a PLATO program which are executed
6) Blocks and Scope -- information regarding scoping of identifiers
7) PLATO Programs -- The structure of PLATO programs
8) PLATO CFG - The context free grammar describing the syntax of PLATO

2) Tokens

There are 4 classes of tokens: identifiers, reserved words, constants and separators. Comments are ignored by the compiler. Whitespace is ignored by the compiler except in serving to separate tokens from each other.

2.1) Comments

Comments begin with an opening "/*" and end with a closing "*/". They may span multiple lines and may be nested.

2.2) Constants

There are two types of constants, booleans and integers. Their syntax is described in section 3.

2.3) Identifiers

Identifiers are defined by a sequence of alphanumeric characters beginning with a letter.
For a mathematically precise and formal definition, refer to the PLATO CFG in section 8

2.4) Reserved Words

Keywords and operators are characters or sequences of characters that have special meaning in PLATO.

They cannot be used as identifiers.
For a mathematically precise and formal definition, refer to the PLATO CFG in section 8.

2.3.1) Keywords

Keywords can be typed in all upper or all lower case but not a mixture of cases.
The keywords in PLATO are: true, false, print, and, or, not, if, elseif, else, as, to, by, some, all, which, in, satisfies, copied

2.3.2) Types

Types can be typed in all upper or all lower case but not a mixture of cases.
The types in PLATO are: boolean, integer, number, tuple, matrix<t>, set<t>where t is recursively defined to be any type. The value t is called a type parameter.

2.3.3) Operators

The operators in PLATO are: “+”, “-”, “*”, “/”, “%”, “**”, “^”, “>”, “>=”, “<”, “<=”, “=”, “!=”

2.5) Separators

Separators are individual characters with special meaning in PLATO
The separators are “{“, “}”, “[“, “]”, “(“, “)”, “,”, “;”

3) Types

PLATO is strongly-typed with variable types determined at declaration time.
PLATO has 2 primitive and 4 derived types

3.1) Primitive Types

The two primitive types in PLATO are booleans and numbers.

3.1.1) Booleans

A boolean can take the value TRUE or FALSE and is specified by one of those keywords.

3.1.1) Integers

An integer is an optional ‘-’ sign followed by a sequence of one or more digits. The first digit of an integer may not be ‘0’ unless it is exactly the integer ‘0.’
For a mathematically precise and formal definition, refer to the PLATO CFG in section 8.

3.2) Derived Types

The four derived types in PLATO are numbers, matrices, sets and tuples.

3.2.1) Numbers

Numbers use the same syntax as Integers but they must be over some group, ring or field. If the group, ring or field is not specified by the user it is assumed to be the mathematical group of Integers.

3.2.2) Matrices

A matrix can be defined by an open "[" and a close "]" surrounding a ";" separated list of vectors. Each vector must have the same number of elements.

A vector is specified by a comma separated list expressions which evaluates to numbers, representing the elements of the vector, or the syntax *start* "TO" *end* optionally followed by "BY" *increment* where *start*, *end* and *increment* are expressions which evaluate to integers. The *start* "TO" *end* "BY" *increment* syntax is equivalent to a comma separated list of numbers containing exactly those values in the infinite sequence startVal, startVal + incrementVal, start + incrementVal + incrementVal that are less than or equal to endVal where startVal*,* endVal and incrementVal are the value that *start*, *end*, and *increment* evaluate to, respectively.

A matrix can also be defined with the syntax [*number* COPIED *dim_1*,*dim_2*,*…*,*dim_n*] where *number* is any number and *dim_1*...*dim_n* are expressions that evaluate to positive integers representing the dimensions of the matrix. This matrix will have the specified dimensions and all the values in the matrix will be equal to the specified number.

The coordinates of the elements in a matrix are, first, the index of the vector from which the element came and, second, the index of that element in the vector from which it came. Matrix elements are ordered by their coordinates with the leftmost coordinate and the rightmost through second to leftmost coordinates being the 2nd most to least significant in that order.

3.2.3) Sets

A set consists of zero or more numbers all of which must be over the same group, ring or field. A set is defined by an opening "{" and closing "}" surrounding a comma separated list of expressions that evaluate numbers.

3.2.4) Tuples

A tuple consists of one or more elements. The elements may be of any type and do not have to be the same type as each other. A tuple is defined by an opening "(" and closing ")" surrounding a comma separated list of elements.

4) Expressions

Expressions in PLATO can be evaluated to produce a value which is a primitive or derived type.

4.1) Values

A value is any primitive or derived type. A value evaluates to itself.

4.2) Unary operators

PLATO contains the the unary operators “NOT”, which takes a Boolean value and evaluates to its logical negation, and “-” which takes an integer and returns its arithmetic inverse.

4.3) Binary operators

PLATO has several classes of binary operators including, arithmetic combiners, boolean combiners, comparators and matrix combiners. All binary operators are written in infix notation such as expr1 binOp expr2. The result of evaluating a binary operator is the result of evaluating the expression on the left hand side of the operators followed by the expression on the right hand side of the operators and then combining the two values with the given operator.

4.3.1) Arithmetic combiners

The arithmetic combiners take two integers and return an integer. They are “+”, “-”, “*”, “^”, ‘/’ and ’%’. The first 3 represent the standard addition, subtraction, multiplication and exponentiation operators on integers and the last 2 represent the quotient and remainder of integer division respectively.

4.3.2) Boolean combiners

The two boolean combiners are “AND” and “OR”. They take two booleans and return a boolean. They represent logical conjunction and disjunction respectively.

4.3.3) Comparators

Comparators take two integers and return a boolean. They are “>”, “>=”, “<”, “<=”, “=” and “!=”. They represent greater than, greater than or equal to, less than , less than or equal to, equal to, not equal to respectively.

4.4) Quantifier expressions

A quantifier expression is of the form *quant id* “IN” *vectorExpr* **SATISFIES** *booleanExpr* where *quant* is either “WHICH”, “SOME” or “ALL”, *id* is an identifier *setExpr* is an expression that evaluates to a vector and *booleanExpr* is an expression that evaluates to a boolean. The identifier is called the quantifier identifier and the boolean expression is called the quantifier body. First *vectorExpr* is evaluated and then *booleanExpr* is evaluated with *id* bound to tbe elements of the result of evaluating *vectorExpr* in turn. If the *quant* is “SOME” and any of the evaluations of *booleanExpression* returns true or *quant* is “ALL” and all of the evaluations of *booleanExpression* return true then the quantifier expression evaluates to true, otherwise it evaluates to false. In the case of “SOME” computation is terminated as soon some value satisifies *booleanExpression* and in the case of “ALL” computation is terminate as soon as some value does not satisfy *booleanExpression*. If the quantifier is WHICH the expression evaluates to the set of all values in *setExpr* which cause the *booleanExpr* to evaluate to true.

4.5) Operating on groups and extended groups

Numbers share some operators with integers

4.5.1) Operating on groups

Any number over a group can use the unary “-” operator as well as the binary “+” and “-” operators. These three operators represent arithmetic inverse, addition and subtraction over the mathematical group.

4.5.2) Operating on extended groups

In addition to the operators defined on groups, any number over a ring or field can use the binary “*” and operator which is defined a multiplication over the mathematical ring or field. Any number over a field can also use the “/” operator which represents division over the mathematical field.

4.6) Operating on matrices

A matrix is indexed by the syntax *matrixIdentifer*[*indexDefinition*] where *matrixIdentifier* is the identifier of a matrix and *indexDefinition* is either a comma separated list of sets or a matrix of booleans.

4.6.1) Numerical indexing

If the *indexDefinition* during matrix indexing is a comma separated list of expressions each of which evaluate to a set of integers, then the result of the indexing is the submatrix consisting of elements whose coordinated are dictated by the cartesian product of the sets. The elements of the submatrix are in the same order as they were in the original matrix. If any of the expressions evaluate to a an integer it will be treated as if it evaluated to a set containing that integer. If any element in the cartesian product of the sets is not a valid coordinate of an element in the matrix it will cause an IndexOutOfBoundsException.

4.6.2) Logical indexing

If the *indexDefinition* is an expression which evaluates to a matrix of booleans of the same dimensions as the indexed matrix then the result is a 1 dimensional matrix whose elements are those elements of the indexed matrix for which the element at the corresponding coordinates in the indexing matrix has the value TRUE, in the same order as in the indexed matrix. If the dimensions of the boolean matrix are not the same as those of the indexed matrix it will cause a DimensionMismatchException.

4.6.3) Unary operators on matrices

Any unary operator can be applied to a matrix as long as the elements are of a type to which the unary operator can be applied. The result is a matrix formed by applying the operator to the matrix, element-wise.

4.6.4) Binary operators on pairs of matrices

Binary operators may be applied to any pair of matrices with the same dimensions and types and long as the elements are of a type to which the binary operator can be applied. The result is a matrix formed by applying the operator to the pairs of elements from the two matrices.

4.6.5) Vectorization

Binary operators may be applied to a matrix and primitive as long as the types of the elements are the same as the type of the primitive and the primitive is of a type to which the binary operator can be applied. The primitive will be treated as a matrix of dimensions equal to the given matrix whose elements are all identical and equal to the value of the primitive.

4.7) Operating on sets

The “+”, “^”, “\” and “*” operators may be used on sets and represent set union, set intersection, set difference and set cartesian product respectively.

4.8) Operating on tuples

An element may be retrieved from a tuple by the syntax *tupleIndeifier*(*tupleIndex*) where *tupleIdentifier* is the name of a tuple and *tupleIndex* is an expression that evaluates to an integer representing the index of the desired element. The index of the first element of a tuple is 1 and the index of every subsequent element increases by 1. If *tupleIndex* is not the index of a valid element it will cause an IndexOutOfBoundException.

A subtuple may be formed from a tuple by the syntax *tupleIndeifier*(*tupleIndices*) which is the same as the above syntax except that *tupleIndices* is an expression that evaluates to a 1 dimensional matrix of integers. The result if the tuple formed by concatenating the value that would be returned by *tupleIndeifier*(tupleIndex) for each value of t*upleIndex* in *tupleIndices*, in order, into a single tuple.

4.9) Casting

The syntax *expr* AS *newType* where *expr* is any expression and *newType* is any type evaluates to the value of *expr* cast to the type *newType* in the manner specified by the list below. Any cast between types not on the follow list will cause a ClassCastException.

Number (Integer) -> Matrix:	Returns a matrix containing a single element which is the value of the given number which is over the same group or extended group as the given number.
Number (Integer) -> Set:	Returns a set containing a single element which is the value of the given number.
Matrix-> Set:	Returns a set containing all the elements of the matrix with duplicates removed.
Set -> Matrix:	Returns a 1 dimensional matrix containing all the elements

of the set.

5) Statements

PLATO has 4 types of statements, a print statement, an assignment statement, an if statement and a bare statement.

5.1) Print statements

A print statement is of the form “PRINT” *expr*“;” where *expr* is any expression. This first evaluates *expr* and then print the resulting value to the console.

5.2) Return statements

A return statement is of the form “RETURN” *expr*“;” where *expr* is any expression. This evaluates *expr* and then terminate the most recent function call. A function must end with a return statement if and only if it specifies a return type. If the value of the return statement cannot be cast to the return type of the function it causes a ClassCastException. Otherwise the value is cast to the return type of the function and used as the return value.

5.3) Assignment statements

An assignment statement is of the form *optionalType id* := *expr*; where *optionalType* is either a type or empty, *id* is an identifier and *expr* is any expression. If the type is not empty this is called a declaration and the type given must match the type of the value returned when the expression is evaluated. If the type is empty the identifier must already be declared and the type of the identifier when it was most recently declared must match the type of the value returned when the expression is evaluated.

In an assignment statement *expr* is evaluated and the resulting value is bound to the specified identifier. The identifier will be bound to this value until it is used in another assignment statement or until the end of its scope.

5.3.1) Number assignments

A number can also be assigned by the syntax NUMBER OVER *gId* n*Id* := *expr*; where *gId* is the identifier of a group of extended group, *nId* is any identifier and *expr* is an expression that evaluates to an Integer that is contained in the elements set of *gId*. If a number is not assigned in this fashion it is assumed to be over mathematical the group of integers.

5.3.2) Matrix assignments

A matrix can be updated by the syntax *matrixIdentifer*[*indexDefinition*] := *expr*;. The left hand side of the assignment must be the same form as described in section 4.6.1 and *expr* must be an expression which evaluates to a matrix with elements of the same type as, and of the same dimensions as, the submatrix of the indexed matrix . This replaces every element in the submatrix defined by left hand side of the assignment with the corresponding element in the result of evaluating expression. If the result of evaluating *expr* does not have the correct dimensions it will cause a DimensionMismatchException.

Values assigned to a matrix must be able to be cast to the type specified by the matrix's type parameter. If they cannot it causes a ClassCastException. Otherwise the values are cast to the type of the set's type parameter.

5.3.3) Set assignments

Values are assigned to a set with normal syntax but they must be able to be cast to the type specified by the set's type parameter. If they cannot it causes a ClassCastException. Otherwise the values are cast to the type of the set's type parameter.

5.3.4) Tuple assignments

A tuple can be updated by the syntax t*upleIndeifier*(*tupleIndex*) := *expr*; or t*upleIndeifier*(*tupleIndices*) := *expr*;. The left hand side of the assignment must be of the same form as described in section 4.8 and *expr* must evaluate to a value, in the case of *tupleIndex*, or tuple of the same length as *tupleIndices*. In the case of *tupleIndex* we treat the expression as if it evaluated to a tuple containing a single element which is the value to which it actually evaluates.

This first evaluates *tupleIndex* or *tupleIndices* and then evaluates *expr*. This will cause a DimensionMismatchException if *expr* does not evaluate to a tuple of the same length as *tupleIndices*, or a value in the case of *tupleIndex*. It then updates the value or values of the tuple specified by *tupleIdentifier* such that, for every i between 1 and the length of *tupleIndices*, the element whose index is specified by the i*th* element of *tupleIndices* is set to i*th* element of the result of evaluating *expr*. In the case of *tupleIndex* this just sets the value at the specified index to the result of evaluating *expr*.

5.4) If statements

An if statement start with the "if" keyword followed by an opening "(" and closing ")" surrounding an expression called the if condition and then a block of statements called the if body. This may, in turn, be followed by zero or more else if blocks each of which starts with "elseif" followed by an opening "(" and closing ")" surrounding an expression called the else if condition followed by a block of statements called the else if body. Finally there may be an else block which is an "else" followed block of statements called the else body.

The if condition is evaluated first. If it evaluates to TRUE then the the if body is executed. Otherwise, each else if expression is evaluated in order until one evaluates to TRUE. The else if body corresponding to the else if condition that evaluated to true is then executed. If none of the if or else if conditions hold true and there is an else block then the else body is executed. After this process control flow proceeds to the first line following the all the if, else if and else blocks.

6) Blocks and Scope

6.1) Blocks

A block in PLATO is a sequence of text surrounded by matching “{“ and “}”.
Blocks may be nested, in which case a closing bracket “}” is matched with the most recent preceding opening bracket “{“.

6.2) Scope

Group and function identifiers are global and can be referenced anywhere within the file.
Function parameters are local to the function block and cannot be referenced outside that function. Function parameters will mask any group or function identifier with the same name within the scope of that function.
All other identifiers except quantifier identifiers are local to a block. The scope of an identifier is from the line on which it was declared to the end of the smallest enclosing block. Local identifiers with the same name as a function parameter or group or function identifier will mask that parameter or identifier within their local scope.
Quantifier identifiers are the identifiers that appear in a quantifier expression and are visible only within the quantifier body. They mask any other identifiers.

7) PLATO Programs

A PLATO program consists of one or more sections. Each section consists of a header and a block. A PLATO program must have a main section and control flow begins at the first line of the block associated with that section.

7.1) Functions

7.1.1) Function Declaration

A function is a section that consists of a function header followed by a block called the function body. The function header is composed of an optional type, called the return type, followed by an identifier, called function name, and finally a comma separated list of parameter declarations, where a parameter declaration is a type followed by an identifier, in parentheses, called the function parameters. If a return type is specified then the last line of the function must be a return statement. In this case the function evaluates to this value and can be used in an assignment or print statement. Otherwise, the function does not evaluate to a value and cannot be used in an assignment or print statement.

7.1.2) Function calls

A function is called with the syntax functionIdentifer(expression_1...expression_n). The section before the parentheses must be a function name and the function whose name it is will be referred to as the specified function. The section in parentheses consists of zero or more comma separated expressions, which must match the number of function parameters of the specified function. The expressions will be evaluated in left to right order after which point control will jump to the first line of the function block of the specified function. After the last line of the function control flow will return to immediately after the location where the function was called.

7.1.3) Main

A main section is the as a function except that its header must be exactly “main()”.

7.2) Groups and Extended Groups

PLATO supports groups as well as 2 types of extended groups, rings and fields.

7.1.1) Groups

A group starts with a header that simply says “GROUP” followed by an identifier that is called the group identifier. This is followed by a block called the group body. The group body consists of an assignment statement to a special identifier called “elements” which must be a set. It also contains a function of two variables which must be called “add” which specifies the behavior of the binary operator ‘+’ on any two values in the “elements” set. The set of elements must form a mathematical group under the ‘+’ operator.

7.1.1) Extended Groups

An extended group has identical syntax to a group except that its header must start with “RING” or “FIELD” instead of “GROUP” and it must also contain a function called “multiply” which specifies the behavior of the binary operator ‘*’ on any two values in the “elements” set. The set of elements must form a ring or field, respectively, under the ‘+’ and ‘*’ operators.

8) PLATO CFG

The following CFG characterizes the syntax of PLATO. **BOLDED** characters represent literal characters that can appear in a PLATO program. Any literal typed in all upper case is also allowed in all lower case in an actual PLATO program. Any space indicates 1 or more required whitespace characters. Regular expression syntax is used to simplify the grammar.

boolean = **TRUE** | **FALSE**
nonZeroDigit = [‘**1**’-’**9**’]
digit = **0** | nonZeroDigit
number = **0** | (**-**)?nonZeroDigit(digit)*
letter = [‘**a**’-‘**z**’] | [‘**A**’-’**Z**’]
alphaNumeric = digit | letter

identifier = letter (alphaNumberic)*
setLiteral = **{}** | **{**expr(**,**exp)*__}__
vectorBody = expr **TO** expr (**BY** expr)? | expr(**,**exp)*
matrixBody = vectorBody(**;**vectorBody)* | expr **COPIED** expr(,expr)*
matrixLiteral = **[**matrixBody**]**
singleLiteral = boolean | number | identifier | setLiteral | matrixLiteral
tuple = **(**expr(**,**expr)*__)__
value = singleLiteral | tuple
binop = **+** | **-** | * | **/** | **%** | ** | **^** | \ | **>** | **>=** | **<** | **<=** | **=** | **!=** | **AND** | **OR**
type = **BOOLEAN | INTEGER | NUMBER** (**OVER** id)? | **SET<**type**>** | **MATRIX<**type**> |**
TUPLE
quantifier = **SOME** | **ALL** | **WHICH**
indexer = **(**expr**) | [**expr**]**
expr = value | **NOT** expr | **(**expr**)** | id**(**(expr(,expr)*)?**)** | id(indexer) | expr **AS** type |
expr binop expr | quantifier id **IN** expr **SATISFIES** expr
elseIfBlock = **elseif (**expr**)**statementBlock
elseBlock = **else** statementBlock
statement =**if (**expr**)** statementBlock (elseIfBlock)* (elseBlock)? | **RETURN** expr**;** | **PRINT** expr**;**
| (type)? id **:=** expr**;** | id(indexer) **:=** expr**;**
statementBlock = **{**statement*__}__
parameter = id type
parameterList = (param(**,**param)*)?
functionHeader = (type)? id**(**parameterList**)**
functionBlock = functionHeader statementBlock
groupBody = **SET elements :=** expr**;** type **add(**type id**,** type id**)** statementBlock
groupHeader = **GROUP** id
groupBlock = groupHeader**{**groupBody**}**
extendedGroupBody = groupBody type **multiply(**type id, type id) statementBlock
extendedGroupHeader= **RING** id | **FIELD** id
extendedGroupBlock = extendedGroupHeader**{**extendedGroupBody**}**
bodyBlock = functionBlock | groupBlock | extendedGroupBlock
mainBlock = **main()**statementBlock
program = mainBlock (bodyBlock)*