PolyGo! Language Reference Manual

Jin Zhou jz2792 | Jianpu Ma jm4437 | Pu Ke pk2532| Yanglu Piao yp2419

1. Introduction

PolyGo! is a mathematical language that allows users to conveniently manipulate polynomial calculation. It can be used for solving a bunch of engineering polynomial algebra problems including, but not limited to, evaluate stability of a linear system, circuit analysis and electromagnetic field calculations. These questions are mostly procedural and can finally simplify their forms as polynomial equations.

PolyGo! follows the basic C-like structure with its own unique polynomial data types and utilities, which can greatly facilitate polynomial manipulation thus greatly helping with engineering practice.

2. Lexical Conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators.In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens.At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator-pairs.If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

2.1. Comment

Comments are confined by `/*` and `*/`.
While single line comments starting with `//`.

The structure is like

```
//This a single-line comment.
/* This
is a
block
comment.*/
```

2.2. Identifiers

An identifier is a sequence of letters and digits used for naming variables and functions; We follow the manner of C to make the first character be alphabetic (underscore ‘‘`_`’’also considered as alphabetic). Identifiers in PolyGo! are case sensitive, which means upper and lower case letters are considered different.

2.3. Keywords

Keywords in PolyGo! are special identifiers to be used as part of the programming language itself. They may not be used or referenced in any other way; function definitions and variable naming cannot override keywords.The following identifiers are reserved for use as keywords:

int	float	cint	cfloat	char	bool
string	void	main	while	if	else
continue	break	true	false	return	const

2.4. Literal constants

There are several kinds of constants in PolyGo!, as follows:

2.4.1. Integer constants

An integer constant is a sequence of digits in decimal.

2.4.2. Character constants

A character constant is 1 or 2 characters enclosed in single quotes, `'` and `'`. To represent a single quote character as a character constant, it must be preceded by a backslash, e.g. `\'`. The backslash character is used as an escape for several other special character constants, as shown in the following:

Backslash	`\\`
Single quote	`\'`
Double quote	`\"`
New line	`\n`
End of string / null byte	`\0`

2.4.3. Floating constants

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. Every floating constant is taken to be double-precision.

2.4.4. Boolean constants

The Boolean constants are the keywords true and false.

2.4.5. String constants

A string is a set of characters surrounded by double quotes, `"`and `"`. A string is considered in the back end as an array of characters, which is held in memory as a contiguous block of data. To represent the double quote character within a string, it must be preceded with the escape character as specified for character constants, e.g. `\"`. In addition, the same escapes as described for character constants may be used.

2.5. Punctuation

Punctuator	Use	Example
`,`	List separator	`int sum(int a, int b)` `int a[3] = {0, 1, 2};`
`;`	Statement terminator	`int x = 3;`
`' '`	Character constant delimiter	`char c = 'a';`
`" "`	String constant delimiter	`string x = "hello";`
`[ ]`	Used in array declaration and as array subscript operator.	`int x[4];` `a = x[1];`

`{ }`	Statement delimiter and array/poly initialization list delimiter	`if (expr) { statements }`
		`int a[3] = {0, 1, 2};`
		`float p[[3]] = {2.5, -1.2, 0, 5};`

`( )`	Conditional parameter delimiter, expression precedence	`while( i > 2 )`
`< >`	Complex number delimiter	`cint a = <1, 2>;`
`[[ ]]`	Used in polynomial declaration and as polynomial coefficient operator	`float p[[3]] = {2.5, -1.2, 0, 5};` `p[[1]] = 1.5;`

2.6. Operators

Operator	Use	Associativity
`=`	Assignment	Right
`==`	Test equivalence	Non-associative
`!=`	Test inequality	Non-associative
`>`	Greater than	Non-associative
`<`	Smaller than	Non-associative
`>=`	Greater than or equal to	Non-associative
`<=`	Less than or equal to	Non-associative
`&&`	AND	Non-associative
`||`	OR	Non-associative
`.`	Access	Left
`*`	Multiplication	Left
`/`	Division	Left
`+`	Addition	Left
`-`	Subtraction	Left
`^`	Exponentiation	Left
`%`	Modulo	Left

3. Expressions

The following expressions categories are ordered by precedence with the highest precedence first.

3.1. Primary Expressions

Primary expressions are *identifiers, literals, (expressions), identifier(expression-list).* Primary expressions group left to right.

● *identifier:* An identifier that is not immediately followed by parentheses '()' is taken to be a variable identifier. The result of the expression is the value that the identifier 'points' to.

● *literal*: The value of a literal expression is simply the value corresponding to the literal constants.

● *(expression)*: Any expression can be wrapped in parenthesis '()' to create another expression.

● *identifier(expression-list)*: An identifier followed by parenthesis '()' is taken to be a function invocation. Each expression inside the expression list is evaluated and passed into the function that is identified by the identifier.

3.2. Unary operators

The operators in postfix expressions group left to right.

● *expression.expression* : call of a module function

● *expression* ++: Increment of number type

● *expression* - -: Decrement of number type

● |*expression*|: Modulus of complex type

● !*expression*: Logical 'not' of number type

3.2.1. Polynomial reference

● *expression[[expression]]*: A postfix expression followed by an expression in double square brackets is a postfix expression denoting a polynomial reference.

3.2.2. Array reference

● *expression[expression]*: A postfix expression followed by an expression in square brackets is a postfix expression denoting an array reference.

3.3. Multiplicative Operators

● *expression*expression*: Multiplication of number types, complex number types, and polynomial types

● *expression/expression*: Division of number types, complex number types, and polynomial types

● *expression%expression*: Modulus of number types

3.4. Additive Operators

● *expression+expression:* Addition of number types, complex number types, and polynomial types

● *expression-expression:* Subtraction of number types, complex types, and polynomial types

3.5. Relational Operators

All relational operators are non associative. They all yield 1 if the specified relation is true and 0 otherwise. Valid for number types.

- *expression<expression*
- *expression>expression*
- *expression<=expression*
- *expression>=expression*

3.6. Equality Operators

Valid for number types, complex types and polynomial types.

- *expression==expression*
- *expression!=expression*

3.7. Logical Operators

All logical operators have the same precedence, and they are all left associative.

- *expression* && *expression:* Return 1 if both expressions are non 0. Valid for number types
- expression || expression: Return 1 if either expressions are non 0. Valid for number types

3.8. Assignment Expressions

Assignment is right associative and returns the assigned value.

- *lvalue* = *expression*: Assign the result of expression to the identifier corresponding to the lvalue.
- *lvalue* must be modifiable: it must not be an array, and must not have an incomplete type, or be a function. Also, its type must not be qualified with constant; if it is a complex or polynomial, it must not have an ember or, recursively, sub member qualified with constant.

4. Statements

4.1. Expression statement

We have the following form for expression:

expression;

Usually this statement is assignment or function call.

4.2. Conditional statement

We have two form of conditional statements:

if (expression) statement

if (expression) statement else statement

4.3. while statement

The while statement has the form:

while (expression) statement

4.4. for statement

The for statement has the form:

for (expression1; expression2; expression3) statement

The first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is terminated when the expression becomes 0; the third expression typically specifies an increment which is performed after each iteration.

4.5. break statement

This statement

break;

causes termination of the smallest enclosing while, for statement; control passes to the statement following the terminated statement.

4.6. continue statement

The statement

continue;

causes control to pass to the loop-continuation portion of the smallest enclosing while or for statement; that is to the end of the loop. More precisely, in each of the statements.

4.7. return statement

A function returns to its caller by means of the return statement, which has the form:

return(expression);

The value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of the function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value.

5. Modules

Currently, we have 2 modules: `std` and `poly`. std module includes standard operations like `std.print(), std.re(), std.im()`. `poly` module includes operations dealing with polynomials, e.g. `poly.findroots()` and `poly.order()`.

6. Declaration

Both variables and function have to be declared before use.

6.1. Variable Declaration

Variables have to be declared as following primitive data types: int, float, bool, char and some advanced data types: cint, cfloat and string. They refer to integer, floating number, boolean, character, complex integer, complex floating number and string. Complex integer means both real part and imaginary part of a complex number are integers, complex floating number means both real part and complex part are floating numbers. What's more, we have array and polynomial data type. Array refers to array of primitive or advanced data types mentioned above, and polynomial refers to polynomial data type. Variables can be declared by expressions, and they can only be declared and used inside a function. In other words, a variable defined in a function cannot be used inside another function. Here are some examples:

- Declaration and initialization of proto type variables :
  type ID = *expr*;
- Declaration and initialization of array, *int const* indicates the size of the array:
  type ID[*int const*] = {*expr1*,*expr2*,...};
- Declaration and initialization of polynomial, *int const* indicates the highest order:
  type ID[[*int const*]] = {*expr1*,*expr2*,...};

6.2. Function Declaration

A function must have a return type, a function name, parameters if any, and the main body. For the return type, in addition to data types mentioned above, also can be declared as "void", which means the function won't return anything. Then functions have to be named in characters, numbers and _. And parameters are needed to be declared inside parentheses next to the function name. Multiple parameters are separated by commas. What's more, a function without parameters are allowed, when user just need the side effect of this function.

type ID (*type* ID, *type* ID, …) {
    return(*expr*); (optional)
}

7. Example Code

7.1. Hello world

This is a simple "Hello World" example code.

```
Open std;

void main(){
    std.print("Hello World!");
}

```

7.2. Application of polynomial

Polynomials are useful in solving some practical problems. For example, we can use PolyGo! to analyze stability of a linear system. Because the necessary and sufficient condition for a stable linear system is that all roots of its characteristic equation have negative real parts.

```
Open std;
Open poly;

bool stab(int p[[]]){                          //define function “stab”
  bool result = true;
  cfloat root[poly.order(p)]=poly.findroots(p);    //find roots of p
    for(int i = 0 ; i < poly.order(p) ; i++){
      if ( std.re(root[i]) >= 0 )    //if all roots have negative real parts,
      result = false;         //this system is stable.
    }
  return result;
}

void main(){
  int po[[3]] = {1,1,2,8} ;
  if(stab(po))
    std.print("This system is stable.\n");
  else
    std.print("This system is unstable.\n");
}
```