PROGRAMMING LANGUAGE

[लिपि]

PROGRAMMING LANGUAGE

Language Overview

The language needs to define the data types of variables being used.
The language supports various arithmetic operation and operators like + , - , / , * and ^ .These operators have their own precedence as in general mathematics.

The basic rules for identifiers are as follows:

1 .The first character must be alphabet.
2 .All other characters can be either numbers, alphabets or underscore (_).
3.LiPi is case sensitive.

The language has associativity l to r for operators of same precedence i.e. left to right.

1 .Basic Data Types:

LiPi supports various data types such as:

1. int for storing integer
2. char for storing characters
3. point for storing floating points

Which is similar to data types of the C language.

2 .Arithmetic Operations:

The language supports various arithmetic operation and operators like + , - , / , * and ^ .These operators have their own precedence as in general mathematics.

Operator	Operation	Example
+	Addition	a+b
-	Subtraction	x-y
*	Multiplication	a1*b1
%	Modulo(Remainder)	a_1*q_1
/	Divide	a/b
^	Power(exponent)	x^g

Operators with same precedence have left to right associativity.

The precedence and associativity of operators is given below.

Operator	Precedence	Associativity
+ , -	1	L → R
*, / , %	2	L→R
^	3	R →L

3.Keywords

There are many keywords used in LiPi and are used to perform various tasks and declarations.

List of a few keywords and their use:

Keyword	Purpose
fx	For declaration of a function
main	For definition of the main function(to be executed first)Which controls the flow of program
int	For declaration of a integer variable
point	For declaration of a floating or a rational number
char	For declaration of a character variable
let	For assignments
NULL	Representing void(void data type as in C)
show	to display to the standard console

Grammar:

BNF Versions of Expression Grammar

<assign>→<id>=<expr>
<expr> → <expr> + <item>
| <expr> - <item>
| <item>
<item> → <item> * <element>
| <item> / <element>
| <element>
<element> → <exp> ^ <element>
|<exp>
<exp> → (<expr>)
| <id>
<id> → <letter> |
<id> | <letter> | <digit > | <underscore>
<letter> → A | B | C | D |……..| Z | a | b | c | d |……..| z
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<underscore> → _

EBNF Versions of Expression Grammar

<expr> → <item> {(+ | -) <item>}
<item> → <element> {(* | /) <element>}
<element> → <exp> { ^ <exp>}
<exp> → (<expr>)
| <id>

<id>→<letter>{ <letter> | <digits>| <underscore> }
<letter> → A | B | C | D |……..| Z | a | b | c | d |……..| z
<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<underscore> → _

A sample program of the language :

```
fx main(NULL)
{
    int a,b,c,d;
    let a=b+c*d;
    show (a);
}
```

Sample statement for the Grammar:

Let us take an arithmetic expression: A = B + C * D

A = B + C * D

Derivation of statement from the Grammar:

<assign>→<id> = <expr>
A = <expr>
A = <expr> + <item>
A = <item> + <item>
A = <element> + <item>
A = <exp> + <item>
A = <id> + <item>
A = B + <item>
A = B + <item> * <element>

A = B + <element> * <element>
A = B + <exp> * <element>
A = B + <id> * <element>
A = B + C * <element>
A = B + C * <exp>
A = B + C * <id>
A = B + C * D