Floor Plan Language (FPL)
Language Reference Manual

Xinwei Zhang (xz2663) Manager
Chih-Hung Lu (cl3519) Language Guru
Dondong She (ds3619) System Architect
Yipeng Zhou (yz3169) Tester

1. Lexical Elements

1.1 Identifiers

Identifiers are strings used for naming different elements, such as variables, functions, and the words "if" in control flow, etc. The identifiers are consist of letters, digits, and underscore ‘_’, and the letters are case sensitive. Each identifier should always start with a letter. These rules are described by the definitions involving regular expressions below:

```
identifier := (letter) (letter | digit | underscore)*
        digit := '0'-'9'
       letter := ('A'-'Z') | ('a'-'z')
 underscore := '_'
```

1.2 Keywords

int	double	string	char	bool	if
else	for	main	void	null	INF
fun	struct	true	false	put	rotate
Wall	Bed	Desk	Door	while	break
continue	return	dirc	Rectangle	Circle	

1.3 Literals

\”	Insert a double in the string
\\	Insert a backslash in the string
\n	Insert a newline in the string
\t	Insert a tab in the string
+INF	Positive infinite number
-INF	Negative infinite number
true	Boolean true value
false	Boolean false value

1.4 Delimiters

Parentheses ()	Enclose arguments for a function
Braces []	Enclose indexes for an array
Brackets {}	Array initialization and assignment
Commas ,	Separate different function components
Semicolon ;	Terminate a sequence of code
Curly Braces{...}	Enclose function/struct definitions and code in if statements/ for loops
Periods .	Access fields of an object
Whitespace	Separate tokens. Include spaces, tabs, newlines

2. Data types

PLT maintains primitive data types like int, bool for general computation and other built-in data types like Wall, Bed, Desk to represent some basic element in a floor plan graphic. User can also define their own structure in PLT like C to satisfy their specific needs.

2.1 Primitive data types

int	a 32-bit unsigned integer
char	a single ASCII character
bool	a boolean variable (True and False)
string	a null-terminated sequence of characters
dirc	a non-negative integer value representing direction ranging from (0~359)

2.2 Built-in data types

Wall	a data structure to demonstrate a wall on floor plan, take two diagonal coordinates as parameters, e.g Wall(x1, y1, x2, y2)
Bed	a data structure to demonstrate a bed on floor plan, take two diagonal coordinates as parameters, e.g Bed(x1, y1, x2, y2)
Desk	a data structure to demonstrate a desk on floor plan, take two diagonal coordinates as parameters, e.g Desk(x1, y1, x2, y2)
Rectangle	a data structure to demonstrate a basic rectangle on floor plan, take two diagonal coordinates as parameters, e.g Rectangle(x1, y1, x2, y2)
Circle	a data structure to demonstrate a basic circle on floor plan, take one coordinate as center and a positive integer as radius.

2.3 User-defined data types

User can define their own data structure to generate special element in data floor such as a fancy sofa, a square toilet. Here is an example,

```
Struct SquareToilet{
        Rectangle seat = Rectanlge(0, 0, 3, 3)
        Rectangle cistern = Rectangle(0, 3, 3, 4)
}
```

3. Expression and Operators

3.1 Expressions

Expressions are made of at least one operand and zero or more operators. Innermost expressions are evaluated first and the priority of an expression is determined by parentheses. The direction of evaluation is from left to right.

3.2 Operators

Operator	Description	Associativity
!	Logical Not	Right to Left
=	Assignment	
* / %	Multiplication, Division, Remainder	Left to Right
+ -	Addition, Subtraction	
==	Equality	
!=	Not equal	
>	Greater than	
<	Less than	
>=	Greater than or equal to	
<=	Less than or equal to	
&&	Logical AND	
||	Logical OR	

Punctuation	Purpose
;	Used to end a statement
{ }	Used to enclose functions, while and for loops, and if statements. In other words, they are used to delineate the

	scope of blocks of code in the program.
( )	Used to specify and pass arguments for a function and the precedence of operators. Also used to enclose conditions in for and while loops and if statements.
,	Used to separate function arguments
“ ”	Used to declare a variable of string data type
/* */	Block comment

4. Control Flow

if (*expression*) {
statement
}
	The expression is evaluated and if it is non-zero, the statement is executed

if (*expression*) {
statement
} else {
statement
}
	Second substatement is executed if the expression is 0. As usual the “else” ambiguity is resolved by connecting an else with the last encountered elseless if.

while (*expression*) {
statement
}
	The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

for (*expression*; *expression*; *expression*) {
statement
}
	The first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

break;	Causes termination of the smallest enclosing while, do, for, or switch statement; control passes to the statement following the terminated statement.

continue;	Causes control to pass to the loop-continuation portion of the smallest enclosing while, do, or for statement; that is to the end of the loop.
return;	No value is returned.
return *expression*;	The value of the expression is returned to the caller of the function.

5. Functions

5.1 Function Definition

In FPL, the definition of a function consists of a keyword “fun”, a return type, a function identifier and some parameters and their types. Then, there is a block of code enclosed by curly braces. An example of a function definition is like this:

```
fun int multiply(int a, int b){return a*b;}
```

5.2 Function Call

A function call in FPL is the functiong’s identifier followed by its params enclosed by parentheses. An example of a function call is like this:

```
multiply(2,5);
```

6. Build-in Functions

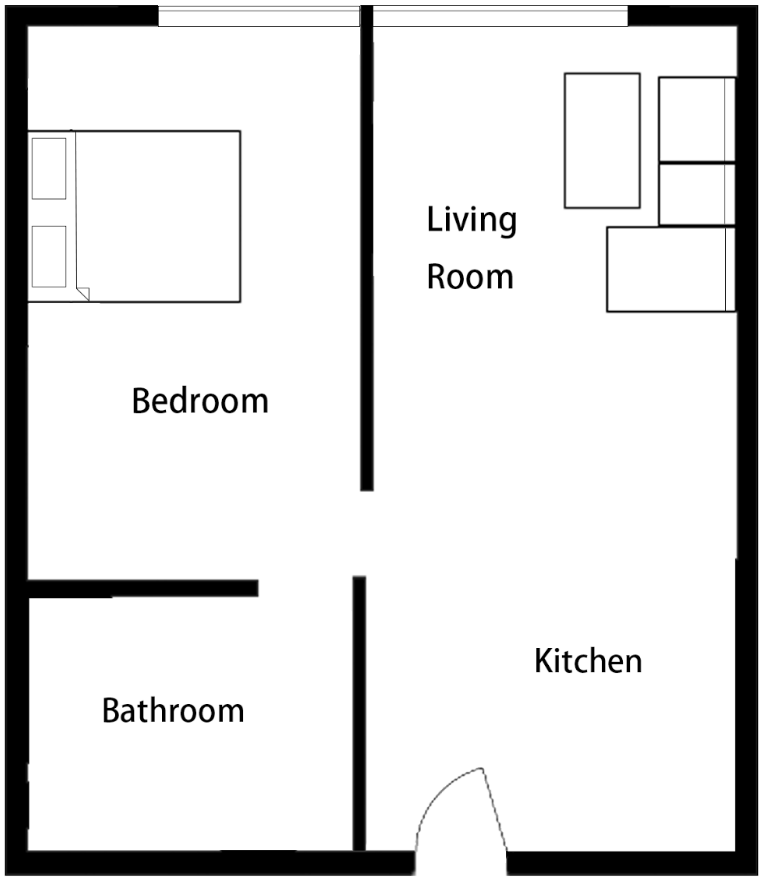
API		
Name	Function expression	Description
put	put(object, x, y)	Render the object to the specific position
rotate	rotate(object, direction)	Rotate the object with the specific direction

7. Example

7.1 Sample code

```
/* user defined struct */
Struct BiggerSofa{
        Rectangle c = Rectangle(0, 0, 3, 2);
        Circle c = Circle(1, 1, 1);
}

/* used defined function */
fun void livingRoomMake(){
        /* invoke user-defined struct */
        Struct BiggerSofa bs;
        rotate(bs, 180);
        put(bs, 8, 15);

        for(int i = 0; i < 4; ++i){
                Chair chair = Chair(0, 0, 2, 2);
                rotate(chair, 180);
                put(chair, 8, 17 + 2*i);
        }

        Desk desk = Desk(0, 0, 2, 6);
        put(desk, 6, 12);
}

int main(){
        Wall wall = Wall(0, 0, 10, 1);
        /* top wall */
        put(wall, 0, 19);

        /* bottom wall */
        put(wall, 0, 0);

        Wall sideWall = Wall(0, 0, 1, 20);
        /* left wall */
        put(sideWall, 0, 0);
        /* right wall */
        put(sideWall, 19, 0);
        Wall middleWall = Wall(0, 0, 1, 8);
        /* middle wall0 */
        put(middleWall, 5, 11);

        /* middle wall1 */
        put(middleWall, 5, 0);

        /* invoke user-defined function to build the living room */
        livingRoomMake();

        Door door = Door(0, 0, 3, 1);
        put(door, 12, 0);

        Bed bed = Bed(0, 0, 3, 3);
        put(bed, 0, 12);

        Window window = Window(0, 0, 3, 1);
        put(window, 2, 19);
        put(window, 6, 19);

        return 0;
}
```

7.2 Rendered floor plan


Living
Room
Bedroom
Kitchen
Bathroom