SFPL Reference Manual

By: Huang-Hsu Chen (hc2237)
Xiao Song Lu(xl2144)
Natasha Nezhdanova(nin2001)
Ling Zhu(lz2153)

1. Lexical Conventions

1.1 Tokens

There are six classes of tokes: identifiers, keywords, constants, string literals, operators, and other separators. Blanks, horizontal and vertical tabs, newlines, and comments as described below (collectively, "white space") are ignored except as they separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been separated into tokens up to a given character, the next token is the longest string of characters that could constitute a token.

1.2 Comments

The characters /* introduce a comment, which terminates with the characters */. The characters // introduce a single-line comment.

1.3 Identifiers

An identifier is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are treated differently. There is no limit on the length of identifiers.

1.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

```
boolean     line
color       point
ellipse     rectangle
else        return
false       string
function    true
if          void
int         while
label
```

1.5 Integer Constants

An integer constant is a sequence of digits. It is always taken to be decimal.

1.6 String Literals

A string literal is a sequence of characters surrounded by double quotes, as in "this is a string". String literals do not contain newline or double-quote characters.

1.7 Point Literals

A point literal is a sequence of digits separated by the @ symbol. It represents the coordinates of a point. I.e., 2@3 represents the coordinates of p(2,3), where the first integer is the x-coordinate, and the second integer is the y-coordinate.

1.8 Color Literals

A color literal is the # symbol followed by 6 characters to specify the color on the RGB scale or by one of the following words: `red`, `orange`, `yellow`, `green`, `blue`, `purple`, `brown`, `black`, `grey`, or `white`. For example, `#c6e2ff` is the slate-gray color, and `#blue` is the blue color.

2. Operators

2.1 Multiplicative Operators

The multiplicative operators * and / group left-to-right. The operands of * and / must be of type int.

The binary * operator denotes multiplication.
The binary / operator yields the quotient of the division of the first operand by the second; if the second operand is 0, the result is undefined.

2.2 Additive Operators

The additive operators + and - group left-to-right. The operands of + and - must be of type int.

The result of the + operator is the sum of the operands.
The results of the - operator is the difference of the operands.

2.3 Relational Operators

The relational operators group left-to-right. The operand of the <, >, <=, and >= operators must be of type int.

The operators < (less than), > (greater than), <= (less than or equal to), >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is int.

2.4 Equality Operators

The == (equal to) and the != (not equal to) operators are analogous to the relational operators except for their lower precedence. (Thus a<b == c<d is 1 whenever a<b and c<d have the same truth-value.)

2.5 Logical Operators

The && operator groups left-to-right. It returns `true` if both its operands are not equal to zero or `false`, `false` otherwise.

The || operator groups left-to-right. It returns `true` if either of its operands is not equal to zero or `false`, and `false` otherwise.

The operands of the && and || operators must be of type int or boolean.

2.6 Assignment Operator

Assignment operator = groups right-to-left.

The left operand must be of type `int`, `boolean`, `string`, `color`, or `point`.

The right operand must be of type `int`, `boolean`, or an `integer constant` if the left operand is of type `int` or `boolean`. It must be of type `string` or a `string literal`, if the left operand is of type `string`. It must be of type `color` or a `color literal`, if the left operand is of type `color`. Finally, it must be of type `point` or a `point literal`, if the left operand is of type `point`.

2.7 Comma Operator

A pair of expression separated by a comma is evaluated left-to-right.

2.8 Coordinate operator

Coordinate operator @ groups left-to-right.

Left and right operands must be arithmetic expressions or they must be of type `int`. Combinations are possible, e.g., `int@(a*b + c),` where a,b, and c are all of type `int`.

2.9 Operator Precedence and Associativity

Below, the operators are listed from the highest to the lowest precedence:

Operators	Associativity
( )	left to right
* /	left to right
+ -	left to right
@	left to right
< <= > >=	left to right
== !=	left to right
&&	left to right

||	left to right
=	right to left
,	left to right

3. Declarations

Declarations specify the interpretation given to each identifier; they do not reserve storage associated with the identifier. Declarations that reserve storage are called *definitions*.

3.1 Variable declarations:

variable-declaration:
    type-specifier : *declarator-list* ;

type-specifier:

```
void
int
boolean
color
point
string
```

declarator-list:
    declarator
    declarator-list, *declarator*
    declarator = *initializer*
    declarator-list, *declarator* = *initializer*

initializer:
    integer constant | *string literal* | *point literal* | *color literal*

3.2 Function definitions:

Functions must be defined when they are declared. A function definition must be an external declaration, i.e., it cannot be inside any other function. Function definitions must be made at the beginning of a translation unit.

function-definition:

```
function type-specifier identifier( parameter-list )
{
        variable-declaration-list
        statement-list
}
```

parameter-list:

type-specifier : *identifier*
parameter-list, *type-specifier* : *identifier*

variable-declaration-list:
variable-declaration
variable-declaration-list variable-declaration

statement-list:
statement
statement-list statement

4. Statements

Except as described, statements are executed in sequence. They fall into the following groups:

statement:
assignment-statement
compound-statement
if-statement
while-statement
function-call-statement
return statement
break-statement
continue-statement

4.1 Assignment statements

assignment-statement:
identifier = *initializer*;

4.2 Compound Statements (Blocks)

compound-statement:
{ *statement-list* }

statement-list:
statement
statement-list statement

4.3 If statements

if-statement:
`if (` *expression* `)` *statement* `else` *statement*

4.4 While Statements

while-statement:

`while` (*expression*) *statement*

4.5 Function Call Statement

function-call-statement:
identifier(expression-list) ;

expression-list:
expression
expression-list, expression

4.6 return Statements

return-statement:
`return ;`
`return` *expression* ;

4.7 Break Statement

break ;

4.8 Continue Statement

continue ;

5. Expressions

Expressions include arithmetic, relational, and logical expressions.

expression:
logical-expression
logical-expression || *logical-expression*

logical-expression:
relational-factor
relational-factor && *relational-factor*

relational-factor:
point-expression
point-expression relational-operator point-expression

relational-operator: one of
`>=    <=    >    <    ==    !=`

point-expression:
arithmetic-expression
arithmetic-expression @ arithmetic-expression

arithmetic-expression:
arithmetic-term
arithmetic-term +*arithmetic-term*
arithmetic-term - *arithmetic-term*

arithmetic-term:
atom
atom * *atom*
atom / *atom*

atom:
identifier
function-call-statement
int
string
point
boolean
(*expression*)

6. Scope

The scope of an identifier is the region of the program text within which the identifier's characteristics are understood.

The scope of a variable or function identifier in an external declaration begins at the end of its declarator and persists to the end of the translation unit in which it appears.

The scope of a parameter of a function definition begins at the start of the block defining the function, and persists through the function: the scope of a parameter in a function declaration ends at the end of the declarator.

The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block.

7. Built-in Functions

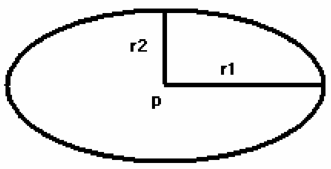
SFPL's built-in functions provide basic functionality for drawing simple shapes. They cannot be overridden by the user.

`line( point p1, point p2, color c)`
Draws a line of color `c` from `p1` to `p2`.

`rectangle( point p, int w, in l, color c1, color c2 )`
Draws a rectanle of width `w` and length `l` with upper-left corner at `p`. `c1` is the contour color, `c2` is the filling color.

`ellipse( point p, int r1, in r2, color c1, color c2 )`

Draws an ellipse centered at `p`, with radii `r1` and `r2`. `c1` is the contour color, `c2` is the filling color.



`label( point p, string l, color c )`

Puts label "l" of color `c` on the diagram, e.g. BATHROOM, KITCHEN, etc. `p` is the upper-left corner of the label.

8. Standard Library

SFPL's standard library is a part of the language. All standard library functions are compiled with every SFPL program by default. There is no need to include or import those functions in your program.

Standard Library Functions

`wall( point p1, point p2, color c )`

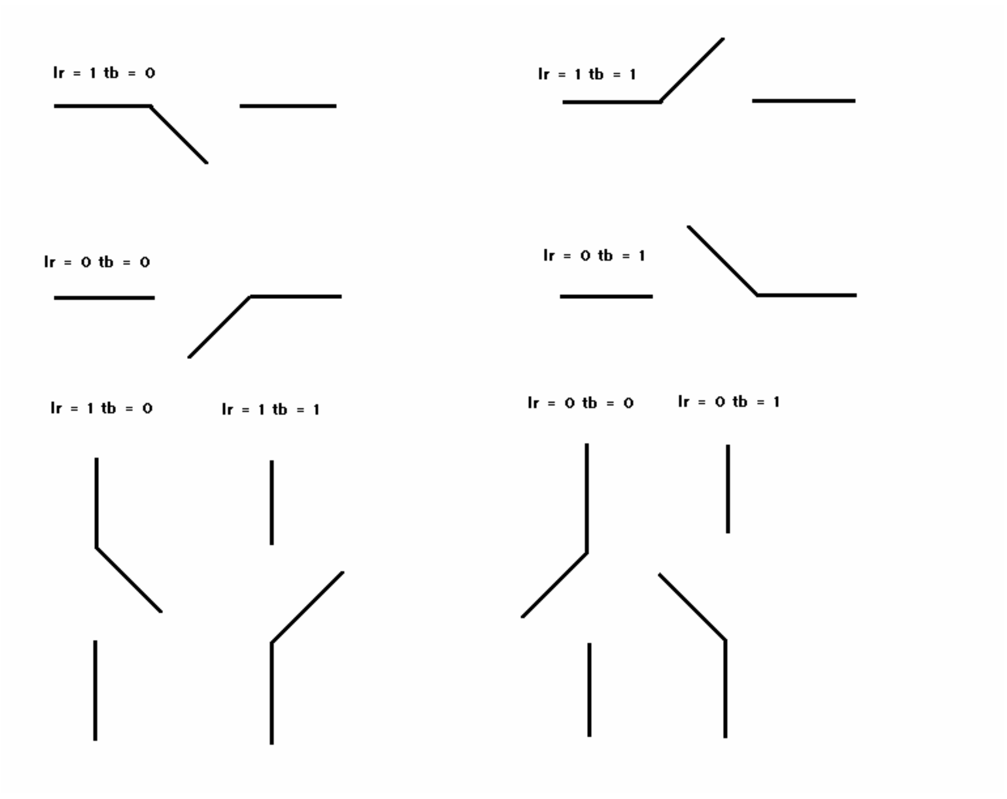
Draws a wall of color `c` from `p1` to `p2`.

`window( point p1, point p2, color c )`

Draws a window of color `c` from `p1` to `p2`.

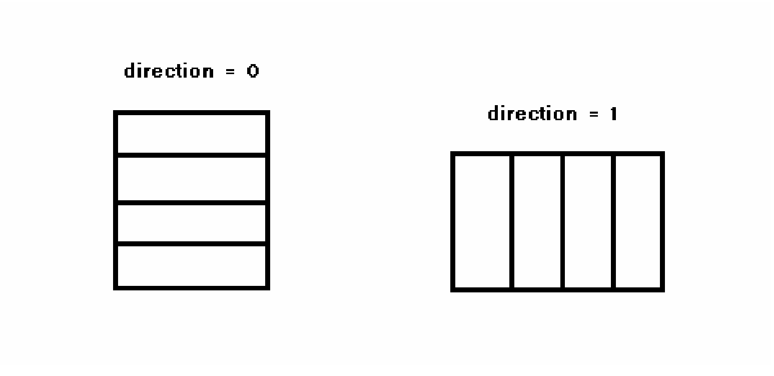
`door( point p1, point p2, color c, int lr, int bt )`

Draws a door of color `c` from `p1` to `p2`. The last two parameters of type int specify the direction the door opens and the position of the door's handle. `lr`=0 for left, `lr`=1 for right; `bt`=0 for bottom, `bt`=1 for top. The possible configurations are shown below:



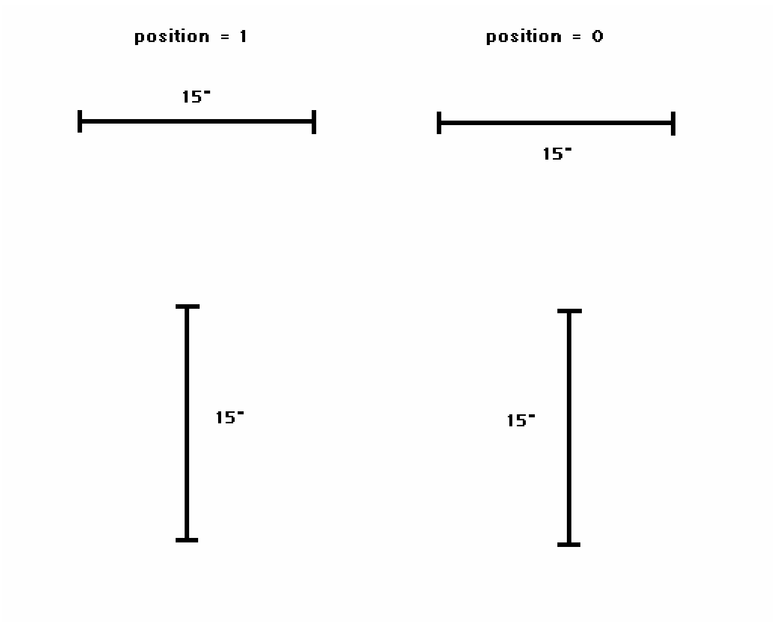
`stair( point p, int w, int l, color c1, color c2, int direction )`

Draws a staircase of width `w` and length `l` with upper-left corner at `p`. `c1` is the contour color, `c2` is the filling color; `direction`=0 for steps to be drawn horizontally, `direction`=1 for steps to be drawn vertically, as shown below.



`dimension( point p1, point p2, string s, int position, color c )`

Draws a labeled arrow of color `c` from `p1` to `p2` to show a dimension of an object. `s` is the label. If `position`=0, the label will be drawn to the left of the arrow if the arrow is vertical, or below the arrow if it is horizontal; if `position`=1, the label will be drawn to the right of a vertical arrow or above a horizontal arrow.



9. Sample SFPL program

```
//function definitions

function int f1(int:a, point:b) {…}
function void f2(color:c, point:d) {…}

//beginning of program

int: A = 5;
point: B = 2@3, D = 5@1;
color: C = #554466;

f1(A, B);
f2(C, D);
//end
```