SFPL

Simple Floor Plan Language
09/25/2005

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

Motivation and Introduction

Today, many people would like their homes custom-designed. They may want to make a floor plan for their newly bought properties or they may want to make changes on their living places such as to separate a big bedroom into two smaller ones. However, designing a house floor plan from scratch requires professional knowledge and can be costly and time-consuming. The language SFPL, which stands for Simple Floor Plan Language, provides an easy and efficient way for people to design their dream home. The language allows users to write a program that lays out basic dimensions and adds details such as doors, windows and other elements. The program will generate and output two-dimensional floor plan diagram(s). By changing variables in the program, users can obtain different diagrams to compare with different designs. In addition, there are flow-control mechanisms in the language to facilitate designing relatively large projects. In brief, the language is designed clear and easy to use, which will make designing a floor plan an easy job for ordinary people. The graph below shows a typical professional floor plan.

Overview of the Language

SFPL is basically a high-level object-oriented language. Object-oriented programming languages enable programmers to create modules that do not need to be changed when a new type of object is introduced. Although not as comprehensive as big object-oriented languages such as Java, SFPL provide built-in support for objects like walls, windows, staircases and doors. These built-in objects will make the language more intuitive and easy to use.

SFPL is designed to be a translated language, with its target language being Java. The user is able to design a floor plan in the form of a program in a text file. The translator will then output a Java source file that can be edited and compiled into Java byte-code. The SFPL translator is equipped with error detection so that SFPL syntactical errors are not passed on to the Java source code. This also makes error-checking and debugging become a relatively smooth process. Finally, the graphic output aspect of the language generally relies on the *Java.awt* package since it provides sufficient utilities to generate a static 2D diagram.

Goals:

The language is designed with the following goals in mind:

- **User Friendliness** - One of the primary goals of SFPL is user friendliness that is the syntax should be easy to learn, read, write, and manipulate. The syntax is meant to be simple and intuitive so that allows users who have little programming experience to learn it quickly. Code definitions in SFPL are structured to be modular that increases the readability of the language.
- **High Level** - Users do not have to worry about internal representation of data, and algorithm implementation. The modularity capabilities allow for extension to complex problem-solving routines.
- **Flexible** –To give users more flexibility and to support designing large scale building projects, SFPL allows users to define new objects through inheritance. In addition users are allowed to define Functions and Packages to reuse sectors of codes. Moreover, the language provides flow control mechanism such as conditional statements and iterative loops.
- **Portable** - Since SFPL is a translated language with its target language as Java and its translator implemented in Java, all SFPL code can be translated, executed, and evaluated on any machine that has a Java Runtime Environment with compiler. Java is a highly portable language, which therefore makes the translator and the SFPL code that it translates highly portable as well.

Some Language Features

- **Primitive Data Types:** There are two basic numerical data types, namely integer and floating-point. They are to be used for computing areas, locations, and sizes. In addition, the Boolean types are needed for control statements.

- **Basic objects:** These are objects like lines, rectangles, triangles and circles. They are the building blocks for the floor plan diagrams. Users can define other objects through inheritance.

- **Other built-in objects:**
  1. ***Wall, Window, Door, Stair…:*** The names of the objects speak for themselves. There will be methods associated with each object to determine their sizes, locations, shapes. They are subclasses of the basic objects.
  2. ***Name:*** This object can be used to mark the function of each room on the final floor diagram, e.g., *kitchen*, *bedroom*, or *bathroom*.
  3. ***Dimension:*** Dimensions are an essential elements of any floor-plan. This object can be used to specifically mark the dimensions of windows, doors, and walls on the floor-plan diagram.
  4. ***Design:*** This object is to be used to specify the properties of the final diagram, such as the background color, and the specific colors and/or symbols to designate the walls, doors, windows, and stairs on the floor plan.

- **Operators:** Operators are necessary for comparisons, loops, and size calculations. So SFPL will have all the essential mathematical operators, i.e., at least, the +, -, <, >, and = operators.

- **Control Statements**: To control the program flow, we are implementing the if/else, and while clauses for control statements. For instance, a while-loop can be used to distribute windows evenly along a particular wall of a room, as well as to check when the end (corner) of the room is reached. The if-statement may be used to choose the number of windows or doors (if any fit at all) to put at a particular location, for example, along a particular wall.

- **Array/Vector Built-In Data Structures** Since at the beginning of a design process, the quantity of some (or all) elements may be uncertain, it would be preferable to use dynamic allocation for storing those elements.

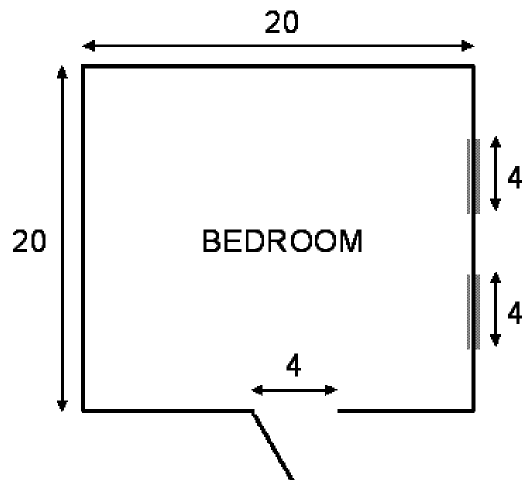
Sample Code

Sample code to construct the room shown below in SFPL:

```
wall WALLS[4]  // instantiate 4 walls
wall[0].put(10,10,30,10) // position wall[0]
wall[1].put(30,10,30,30)
wall[2].put(30,30,10,30)
wall[3].put(10,30,10,10)

window WIN[2]  // instantiate a window
WIN[0].put(30,14,30,18)
WIN[1].put(30,22,30,26)

door DOOR  // instantiate a door
DOOR.put(20,30,24,30)
```



The code to specify the name and dimensions of the room is not shown

Optional Features

It is possible to extend SFPL to provide methods for changing the materials out of which different elements of a design are made, and to calculate prices for different options. Such functionality would be very useful because it would allow the user to compare various possibilities for the choice of materials and choose the one that is most desirable financially. It would also be useful to introduce a method that could show the list of materials to be used for the maximum cost reduction.

The potential users of SFPL would be very grateful to its creators if the SFPL compiler could check for, and help them correct unpleasant design mistakes, e.g., windows overlapping doors. It would be nice if the compiler generated an error message and pointed out to the user what the problem is and how to fix it.

The above functionalities are desirable but may be difficult to implement. Thus whether to implement these optional functionalities will depend on the time constraints of the whole project.