A Tutorial on the Adele Programming Language

Jen-Chieh Huang (jh3478) Xiuhan Hu (xh2234)
Zixuan Gong (zg2203) Jie-Gang Kuang (jk3735)
Yuan Lin (ly2324)

March 24, 2015

1 Introduction

ASCII-art description language, Adele, is a domain-specific programming language aimed at providing a simple and intuitive way to describe and generate ASCII artworks. It includes several special language constructs for ASCII artwork. Besides, common programming language features such as functions, control statements are also supported. Adele is also designed to be executed in a portable run-time environment. The final executable of an Adele program can be run in any web browser supporting JavaScript.

Before discussing the details of the language, we'd like to present the basic concepts of how Adele processes the output ASCII art. For every Adele program, a canvas is given to the main function. The canvas is the destination on which a user can put his artwork on. When a user puts something on the artwork, it generates a corresponding 'layer' on the canvas. This is exceptionally useful when the user would like to create reusable parts and background parts of the artwork. The user can change the position of the reusable parts to create interactive or animated effects. When all the layers are created, the user would need to inform the run-time environment, and the resulting canvas will be displayed on the output device.

The purpose of the document is to show the audience how to write and execute an Adele program. The document will first start by giving a simple example of an Adele program and how to run this program followed by more complicated examples and details of the special constructs. Finally, we will present an example by putting everything together.

Good luck and have fun in Adele.

2 The First Adele Program

As a tradition in introducing a programming language, we would like to present the Hello World as our first example using Adele. The following codes illustrate what the Hello World program looks like.

```
void main (canvas c)
    print_str (0, 0, "Hello, world");
end
```

In the program, the entry point of any Adele program is the main function. 'void' indicates that the return value of the function is ignored, and the parenthesis after the function name shows the parameters of the function. In this example, c, whose type is canvas, is the input parameter to the main function, which is the output device of the user's artwork. For user-defined function, empty parameters are also possible. Note that, in Adele, every function has to be ended with an 'end' marker.

In the 2nd line, the function, print_str, is a predefined function which takes 3 parameters. The first 2 parameters are the x and y coordinator in the default canvas, and the 3rd parameter is the string which the user would like to show. The semantics of the function is that the a ASCII-string will be displayed on the user window at the given location.

After compilation and execution, you should be able to see the string printed on the screen at the upper left corner. The details of the compilation and execution is listed in the following section.

3 Compiling and Running

4 More Complex Examples

5 Special Constructs

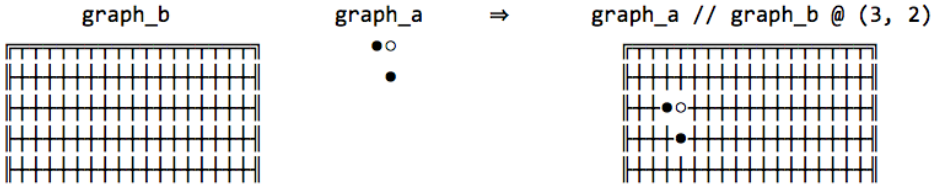
Function print(x, y, graph) is used to put a graph on the canvas at a position relative to it. But sometimes users may want to place a graph relative to another graph, rather than the canvas. The following special constructs provided by Adele are introduced for this purpose. More precisely speaking, these constructs offer different ways to combine two graphs into a new graph.

5.1 Overlay and locate

The overlay symbol // is used together with the locate symbol , and the two symbols form a ternary operator. The syntax of the construct is:

```
graph_a // graph_b @ (pos_x, pos_y);
```

The first two parameters are graphs, and the two parameters inside the parentheses are integers, which can be negative. The effect of this construct is that graph_a overlays graph_b, and the upper left corner of graph_a is placed at the position (pos_x, pos_y) relative to the upper left corner of graph_b. The return value of the construct is a graph. Consider the following example:



It's clear that the graph on the left side of the overlay symbol is in the upper layer; the one on the right side is in the lower layer. Notice that graph_a and graph_b stay unchanged after applying the operator. And as shown in the above example, the cells of the upper layer that are white spaces would not replace the corresponding cells of the lower layer.

5.2 Attach Horizontally/Vertically

To attach a graph to the border of another graph, users can use the overlay and locate construct, and specify the exact coordinate of a cell on that border. Adele provides two utility constructs to facilitate that operation.

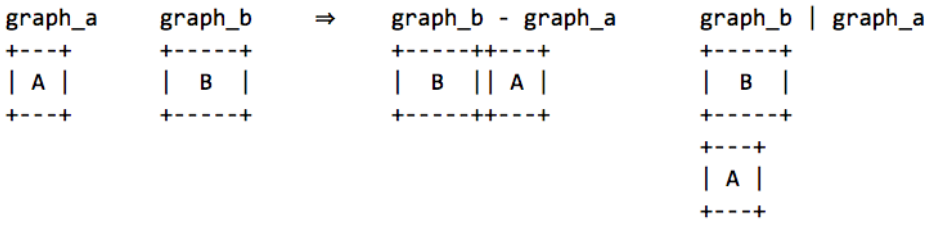
To attach graph_a to the right of graph_b (i.e. attach horizontally):

```
graph_b - graph_a
```

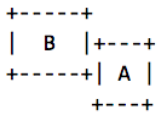
To attach graph_a to the bottom of graph_b (i.e. attach vertically):

```
graph_b | graph_a
```

The parameter on the left side of the operator is the "base" graph, and the one on the right side is the graph to attach to it. As the overlay and locate operator, applying the two operators ('-' or '—') does not change the value of the two graphs, and each construct returns a new graph. Consider the following example:



The example shows that, the "attach horizontally" operator sets the top-left corner of graph_a to the right of the top-right corner of graph_b; the "attach vertically" operator sets the top-left corner of graph_a to the bottom of the bottom-left corner of graph_b. Therefore, if users want to customize the position of graph_a, relative to the border of the graph_b, it's still suggested to use the overlay and locate construct. For example, to realize the following effect:



6 Predefined Functions

Besides the data types and flow control, some more utilities are provided either to enable display or to facilitate programming process.

6.1 draw

The draw function is the essential function to display the ASCII art arranged by the programmers. All the graph data defined or placed in the program is not displayed immediately. Adele will trigger the target language to render the output only when the draw function is invoked. The usage of draw is

```
draw ();
```

and it does not return any value. Here is a simple example to show the usage of the draw function.

```
void main (canvas c)
        graph g =    \(^o^)/    ;
        g@(0, 0);
        draw();
end
```

In the example, if the program ends without draw(), the output will actually not render the graph. Only with draw() will the Adele generate the rendering part of target code to show the graphs.

The purpose of this step is mainly to coordinate and synchronize the display timing of different graphs. When there are two or more graphs to be displayed at the same time, we can hardly achieve it without draw(). If there is no draw(), and we display the graph immediately when we put the graph on the canvas by g(0, 0). For more graphs operated with this mechanism, those graphs will be displayed one by one but not at the same time.

6.2 print

In addition to the ASCII art, sometimes the programmers would like to add some words in the picture for various reasons. Adele provides a simple function for such a situation. The usage of the string printing function is

```
print_str(<x>, <y>,     <string>    );
```

and it does not return values.

Since this is just a simple and quick implementation to display strings, there are two restrictions when using this function. One is that, although the displayed strings are treated as a graph, the lifetime of the graph is only to the next draw function. After the string is rendered by a draw(), it will not be displayed again in the next draw() unless the same statement is invoked again before the draw(). This is because there is no reference to the string(graph) so that we cannot remove it manually with statements.

The other is that the rendered string will always on the most bottom layer. The reason is similar to the previous one. Since we do not have a reference to the string, we cannot decide the layer of it, or place it over or under other graphs. As a result, in order not to affect the major part of the graphs, all the strings rendered by the string printing function will be under all other explicit graph types.

6.3 load

It is most of the case that the elemental ASCII graphs are preferably constructed from the existing files rather than locally with the character arrays in the programs. For these cases, a predefined function, load, can be used to import ASCII graphs from the files. The usage of load is

```
load("filename");
```

and it returns a graph-type reference which points to a ASCII character group.

For example, to show a ASCII graph of \ (ô)/ without the file importing, a simple program as in section 6.1 is

```
void main()
        graph g =     \(^o^)/    ;
        g@(0, 0);
        draw();
end
```

Listing 1: hail.adl

Alternatively, It can also be done by having a text file of \ (ô)/ and a program to import the graph from the file, such as

```
\(^o^)/
```

Listing 2: graph.txt

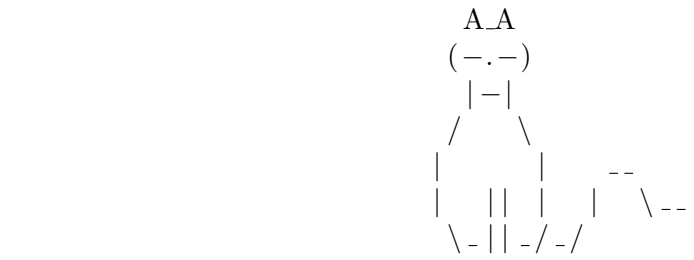
```
void main()
        graph g = load( graph.txt );
        draw();
end
```

Listing 3: hailv2.adl

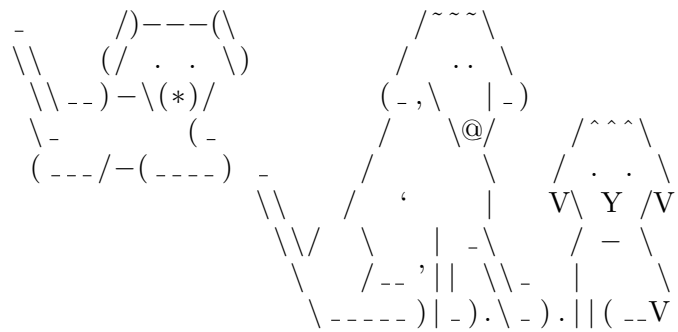
Although the above example may be too simple to show the convenience of using load to import the graphs from the files, the load function can be very useful when there are multiple rather bigger graphs to manipulate. The following program is an example to demonstrate the advantage of using the load function

```
      >>\.
     /-   ) `.
    /   -) `^) `.     -.---. -
  (-,'  \   `^-)""          `.\
         |                    | \
         \                    /  |
        /  \   /.---.'\    (\  (-
       <  ,"||       \  | `.  \`-'
        \\  ()        )|   )/
        |->|>       /-] //
          /-]          /-]
```

Listing 4: graph_horse.txt



Listing 5: graph_cat.txt



Listing 6: graph_dogs.txt

```
void main()
        graph gHorse = load( graph_horse.txt );
        graph gCat = load( graph_cat.txt );
        graph gDogs = load( graph_dogs.txt );
        print(0, 0, gHorse);
        print(25, 25, gCat);
        print(30, 0, gDogs);
        draw();
end
```

Listing 7: animals.adl

However, there are some restrictions or things to be aware of when using the load function. First of all, the source file should be ASCII-encoded text file. The result of opening a non-ASCII text file is unpredictable. Second, the content of a single text file will be grouped as one graph. There is no way for the load function to load partial graph in the source file.

6.4 sleep

Despite the basic flow control described in the previous chapter, a time control mechanism is necessary to generate the animation. Therefore, Adele

provides the sleep function to pause the procedure in the program to enable the graphs to be displayed step by step. The usage of the sleep function is

```
sleep(milliseconds);
```

and it does not return values.

An example program to show the usage of the sleep function is like

```
void main()
        graph g =   This is a scrolling text  ;
        int     <<waiting for checking while loop>>
end
```

Listing 8: marquee.adl

7 Put Everything Altogether