The Adele Programming Language Reference Manual

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

March 24, 2015

1 Introduction

Adele is a programming language designed specifically for simplifying the creation of ASCII artwork. The output space is defined as a canvas object on which the user is allowed to put several custom layers. By controlling the custom layers properly, the user will be able to create interactive ASCII artwork with reusable components. To further simplify these operations, various special operator and predefined functions are included in the language specification.

In addition to the extra constructs for ASCII artwork, Adele also supports common elements such as logic control and loop control facilities. One can not only use Adele for ASCII artwork, but other more general tasks.

In this reference manual, the fundamentals of the Adele programming language will be introduced, and details of the syntactic features are also covered.

2 Lexical Specifications

2.1 Comments

Character '#' introduces a comment, which terminates with a newline. Considering elegance and simplicity of our lexer, we do not provide any multi-line comment pattern. You can do multi-line comment via your text editor.

2.2 Identifiers

An identifier is a sequence of letters, digits and underscore. The first character of any identifier must be a letter or underscore. For example, 1a2_3d will not be considered as an identifier, but a2_3d or _1a2_3d is in our standard. Identifiers may have any length, but only the first 31 characters are guaranteed to be considered. So any identifier longer than 31 characters is not recommended. And any identifier is case-sensitive, e.x., 'ab' and 'Ab' is distinguishable by our compiler.

2.3 Keywords

The following identifiers is reserved as keywords, so you are not supposed to use them as an identifier, otherwise an error may occur:

if	end	while
return	group	int
float	void	false
true	draw	print_str
load	sleep	

2.4 Constants

2.4.1 Number

A number constant is either an integer or a float. A positive integer starts with a non-zero digit([1-9]) and can be followed by any digit, or it's zero itself. A negative integer starts with a minus '-' then followed by a positive integer. Note that 012 and -012 are not integers. A float starts with an integer, then a '.', then followed by a sequence of any digits, e.x., 0.1345.

2.4.2 String

A string literal is a string constant, which consists of a sequence of characters surrounded by double quotes as in "...". A string is actually an array of characters.

2.4.3 Boolean

A boolean constant is either the keyword true or false.

2.4.4 Character

A character constant is a sequence of one or more characters enclosed in single quotes as in 'x'. But you can also put it in a string like: "abcx", where 'x' is a character constant. A table of special character constant is shown below:

- newline: \n
- backslash: \\
- horizontal tab: \t
- single quote: \'
- backspace: \b
- double quote: \"

2.5 Punctuation and Separators

A comma ',' is used to separate parameters in function's parameter list. A semicolon ';' is used to separate two lines. Note that you needn't and shouldn't use semicolon ';' at the end of the line which starts with 'if', 'while', 'end' or the end of first line of a function declaration.

3 Types

In the Adele programming language, static type checking is performed. Type inconsistencies will be reported as errors in the compile time if the type conversion is not possible. The basic types of Adele includes integers, floating point numbers, boolean, and void.

- Integer type (int)
  Integer type is defined as a signed decimal integer. The range is defined from +9007199254740991 to -9007199254740991. When operating with floating number, the integer can be converted into a corresponding floating point number. When overflow or underflow happens, the behavior is not defined in the language specification. For integer constants, the lexical definition is [-]?[1-9]+[0-9]* — [0]

  Note that in the definition, we do not allow preceding 0s in front of the integer, and we also do not allow multiple 0s. That is, 00000 or 0001 are not legal integers. Besides, -0 is also an invalid integer.

- Floating point number type (float)
  Floating type is defined as a floating number conforming double-precision 64-bit format IEEE 754 values. Loss of precision can happen if a floating number is converted into an integer. For floating point constants, The lexical definition is [-]?[0-9]+ '.' [0-9]+. Note that we do allow 00.00 in this case since the meaning of 0 and 0.0 is somehow different in terms of precision.

- Boolean type (bool)
  Boolean type represents the boolean variables, true or false. Note that you can NOT convert boolean values to numeric values like integers.

- void type (void)
  As the name suggests, the void type will not check the type until the compiler or the computation has to. This is particularly useful when

you want to pass groups which may have several different types into a single function.

For derived types, Adele supports the following types: array, functions, references, groups.

- Arrays
  Array type in Adele is similar to the Java programming language. The user can put either a set of primitive-typed data or a set of user-defined type references. Note that all the data in the same array have to be the same type.

- Functions
  Functions in Adele are composed by the following parts.

  - Return type: indicates the type which will be returned by the function.
  - Function name: used to identify the function itself.
  - Parameter list: a finite set of 2 tuples (type and name), worked as input of the function.
  - Function body: description of the computation to be carried out in the function.
  - End marker: an end to indicate the end of the function

- Groups

  - user-defined: Groups in Adele is just like struct in C. It provides a mechanism for the users to group the logically related data into a type. However, other object-oriented features are not supported in Adele for simplification. Complicated operations in OOP can be scaring. To define a group, the user has to start with the keyword, group, followed by a group name, and an end marker in the end.
  - pre-defined: Pre-defined group includes canvas, graph.

4 Operators

The following table summarizes the precedence and associativity of all the operators supported by Adele. Operators on the same line have the same precedence, and the rows are in order of decreasing precedence.

Operators	Associativity
() [] .	left to right
* / %	left to right
+ - —	left to right
¡ ¡= ¿ ¿=	left to right
== !=	left to right
// @	left to right

The operator () refers to function call; [] refers to array subscripting; . is used to access fields of groups.

The operator - can serve as both the arithmetic subtraction operator and a graph operator. Other special graph operators include — and // @ (details can be found in the following sections).

5 Syntax

5.1 Literals

- Number literals: integers and floating-point numbers;
- Character literals: single ASCII characters surrounded by single quotes;
- String literals: a sequence of characters surrounded by double quotes;
- Boolean literals: true or false;

5.1.1 Identifiers

Identifiers for variables, functions, groups or arrays.

5.1.2 Multiplicative Expressions

Multiplicative expressions consists of expressions involving *, /, %. The operands of all the three operators must have arithmetic type; the operands of % must have integral type. As the convention, multiplicative operators have a higher precedence than additive operators.

5.1.3 Additive Expressions

The additive operators include +, -. Rigorously speaking, + is the arithmetic addition operator; - can serve as both the arithmetic subtraction operator

and a graph operator. The type of the operands determines whether the operator - works as subtraction or attaching horizontally.

The operands of the arithmetic additive operators must be numbers. The type of values of arithmetic additive expressions is number.

5.1.4 Graph Combining Expressions

The graph operators are -, —, and the ternary operator // @. These three operators does not have side effects on their operands.

The operands of the two binary graph operators must be graphs. When serving as a graph operator, - means "attaching horizontally." The operator |means "attaching vertically." As for the ternary operator, i.e. the overlay and locate operator, the first two operands must be graphs, and the third operand is a pair of integers in the form of "(x, y)", indicating the coordinates.

The type of value of graph combining expressions is graph. Notice that the part of a graph that exceeds the canvas will be truncated.

5.1.5 Boolean Expressions

Boolean expressions include expressions involving the relational operators: ¡, ¡=, ¿, ¿=; and the equality operators: ==, !=. The value of these expressions is true or false.

5.1.6 Braced Expressions

Expressions surrounded by parentheses are equivalent to the original expressions.

5.1.7 Function Calls

A function call expression consists of the function identifier, and a parameter list surrounded by a pair of parentheses. The value of a function call expression is the return value of the function called.

5.2 Statements

There are five kinds of statement in Adele programs: declarations, expressions, if statements, while statements and return statements. Simple statements like declarations, expressions and return statements end with a semicolon. Control flow statements, if statements and while statements, are ended by an end keyword.

5.2.1 Declarations

A declaration statement assign an identifier to a specific memory space according to its type. A primitive-typed variable declaration consists of a type, an id and an optional initializer:

```
type id;
type id = expression;
```

For example:

```
int x;       # declare an int variable x
void v = x; # declare an void variable v,
             # and v is referred to x;
```

A group structure variable is declared by given a defined group type identifier and an instance id:

```
group group-type id;
```

For example:

```
group point p1;
group circle c1;
```

5.2.2 Expressions

An expression along with a semicolon can be a valid statement. For example:

```
sum = sum + x;
flag = (a+b)<(a-b);
```

5.2.3 return statements

return statements will end a function call and returns control to the calling function with a return value:

```
return result;
return;
```

For a function whose return type is void, we can omit the return value as above and pass a NULL value back to the calling function.

5.2.4 if statements

The if statement allows you to control if a program enters a block of code based on whether a given condition is true.

```
if (condition_expression)
    statement1;
    statement2;
    ...
end
```

The statements between the if line and the end line will be executed if and only if condition_expression returns a true value (numeric non-zero, boolean true, assigned void variables etc).

5.2.5 while statements

The while statement loops through a block of code as long as a specified condition is true.

```
while (condition_expression)
    statement1;
    statement2;
    ...
end
```

The statements between the while line and the end line will be executed iteratively until condition_expression is not a true value.

6 Built-in Functions

Besides the grammar defined above, Adele also provides some built-in functions to accomplish the job of rendering combined or moving ASCII art. Because the syntax of the built-in functions are simple and fixed, the built-in functions will take all the acceptable arguments and do not return any values (except for the load function). When the types of the arguments for the functions are incorrect, the compiler should report them as errors.

6.1 sleep

The prototype of the draw function is simply

```
void draw ()
```

Since the draw function do not take any arguments and do not return any values, the statement involving draw() should be just like the description above. When the draw function is invoked, Adele will render the target codes of drawing all the currently placed graphs. The graphs which is not placed on the canvas or is removed from the canvas by the time when draw() is invoked will not be displayed at that time.

6.2 print

The one-time string printing function should be invoked according to the prototype

```
void print_str (int <x>, int <y>, string <string>)
```

where ¡x¿ and ¡y¿ are the position of first character of the string on the x-axis and y-axis respectively, and ¡string¿ is the target string to be displayed. The string printing function will place the desired string on the bottom layer over the canvas for the next draw(). After a draw() is invoked, all the strings rendered and placed by the string printing function will be removed from the canvas.

6.3 load

To construct the graphs more efficiently, a load function is provided by Adele to import prepared ASCII art from the existing files. The prototype of the load function is

```
<reference to graph> = load (string <filename>)
```

The file will only be opened in text/string mode but not in binary mode. That is, only plain-text files will be valid for the load function. If the source file is not plain-text, the non-text part will be read in and could induce unexpected results. The load function is also the only built-in function returning a value, a reference to a graph constructed from the source file.

6.4 sleep

To achieve better flow control, sleep, a function which pauses the program is provided in Adele to enable the programmer to control the flow more flexible. The prototype of the sleep function is

```
void sleep(int <millisecond>)
```

The argument for the sleep function is the duration in milliseconds and the sleep function does not return a value, either.

7 Appendix

```
grammar adele;

prog    :                                       /* empty
   programs       */
        |   (
                func                            /* functions
                                */
        |       type_declaration                 /* user
           defined types   */
        |       (declaration SEMICOLON)         /*
           declarations         */
            )*
        ;

type_declaration:
        GROUP ID
        (TYPE ID SEMICOLON)*
        END
        ;

func:   (TYPE | GROUP ID) ID LPAREN plist RPAREN
        stmts
        END ;

plist:
        |   ( (TYPE ID COMMA) | (GROUP ID ID) )* (TYPE ID
            | GROUP ID ID)
        ;

stmts:  stmt* ;
stmt:       SEMICOLON
        |   if_stmt
        |   while_stmt
        |   expr SEMICOLON
        |   declaration SEMICOLON
        |   RETURN expr SEMICOLON
        ;

if_stmt:        IF LPAREN expr RPAREN
                stmts
                END ;
```

```
while_stmt:     WHILE LPAREN expr RPAREN
                stmts
                END ;

declaration:   GROUP ID ID
            |   TYPE ID
            |   TYPE ID EQUAL expr
            ;

expr:       LPAREN  expr    RPAREN            /*
   parenthesis */
         |   ID LPAREN func_plist RPAREN       /* function
             call */
         |   expr    ADDITIVE_OP     expr        /*
            addition */
         |   expr    MULTIPLICATIVE_OP   expr        /*
            multiplication & division */
         |   expr    NE      expr        /* not equal */
         |   expr    GT      expr        /* less than */
         |   expr    LT      expr        /* less than */
         |   expr    GET     expr        /* less than */
         |   expr    LET     expr        /* less than */
         |   ID OVERLAY ID AT LPAREN NUM COMMA NUM RPAREN
            /* @lfred: to fix - lame overlay */
         |   ID      EQUAL   expr        /* assignment */
         |   ID
         |   NUM
         ;

func_plist:  | ( fpitem COMMA )* fpitem;
fpitem:     ID | NUM | STR ;

/* keywords */
IF:     'if'        ;
END:    'end'       ;
WHILE:  'while'     ;
RETURN: 'return'    ;
GROUP:  'group'     ;

/* symbols */
fragment ADD:        '+'  ;
fragment SUB:        '-'  ;
fragment MULTI:      '*'  ;
fragment DIV:        '/'  ;
LPAREN:     '('  ;
```

```
RPAREN:      ')'  ;
COMMA:       ','  ;
SEMICOLON:   ';'  ;
EQUAL:       '='  ;
OVERLAY:     '//' ;
AT:          '@'  ;
GT:          '>'  ;
LT:          '<'  ;
GET:         '>=' ;
LET:         '<=' ;
NE:          '!=' ;

ADDITIVE_OP:    ADD | SUB ;
MULTIPLICATIVE_OP:  MULTI | DIV ;

/* types */
K_INT:  'int';
K_VOID: 'void';
K_FLOAT: 'float'

/* identifiers */
ID:     [_a-zA-Z]+[_0-9a-zA-Z]* ;

/* primitive types */
fragment INT_NUM:    [-]?[1-9]+[0-9]* | [0] ;
fragment FLOAT_NUM: INT_NUM '.' [0-9]+;
fragment CHR:        [A-Za-z0-9_] ;
NUM:    INT_NUM ;
STR:    '"' CHR* '"' ;

WS:     [ \t\r\n]+ -> skip ;

COMMENT: '#' .* ;
```