FRAC: Language Reference Manual

- **Justin Chiang** `jc4127`
- **Kunal Kamath** `kak2211`
- **Calvin Li** `ctl2124`
- **Anne Zhang** `az2350`

1. Introduction

FRAC is a domain-specific programming language that enables the programmer to generate fractals in the Graphics Interchange Format, commonly abbreviated as GIF. Specifically, the user will be able to program a fractal using a formal grammar and call from a collection of available system functions to generate the actual GIF. FRAC is well-suited for those interested in the mathematical manipulation of fractal-like objects. We designed this language to be easy to learn, easy to use, and a delight to behold.

2. Data Types & Data Structures

The language supports two categories of data types: primitive types and complex types.

2.1 Primitive types

A primitive type requires a small and fixed amount of memory to store a single logical value. Typical primitive types include `int`, `double`, and `bool`. One exception is `string`, which requires a variable amount of memory storage.

Explicit and implicit type conversion between `int` and `double` is allowed, but no other type conversions are supported, with the exception of “reflexive conversions” (e.g. converting `double` to `double`). Explicit type conversion can be performed with the following syntax:

```
double x = 3.5;
x = (double) x;  // reflexive conversions are permitted
int y = (int) x;
```

```
double z = (double) y;
```

Alternatively, omitting the explicit parentheses-enclosed target type produces an identical outcome.

```
double x = 3.5;
x = x;
int y = x;
double z = y;
```

Note that explicit type conversion can operate on both variables and constants. Therefore, the following is permitted:

```
int a = (int) 3.5;
```

Converting a double to an integer results in truncation of the fractional part, while converting an integer to a double results in the addition of a fractional part with a value of zero.

When performing an arithmetic operation between an integer and a double, the compiler automatically promotes the integer into a double. Thus, the following three statements are operationally equivalent:

```
double c = 3.5 + 3;  // c is 6.5

double c = 3.5 + (double) 3;  // c is 6.5
double c = 3.5 + 3.0;  // c is 6.5
```

2.2 Complex types

A complex type contains multiple named fields and requires a larger and variable amount of memory to store a structured collection of values. A complex type is similar to the familiar object type in so far as it contains fields, but a complex type does not contain any methods. In fact, all instantiated complex types are immutable; operating on them requires the use of functions.

Two complex types are supported in the language: `gram` and `rule`.

A `gram` represents a formal grammar that is used to specify a fractal that can be drawn. A `rule` represents a production rule that is a part of the formal grammar. Later sections expand on how such grammars and rules can be declared in code.

Just like in Java, // for single line and /* */ for multi line comments.

3. Lexical Conventions

3.1 Identifiers

An identifier is a sequence of alphanumerics and underscores. An identifier may begin with neither a digit nor an underscore. Both uppercase and lowercase letters are permitted. The following are valid identifiers: `kunal_43`, `hello_ANNIE`, and `do_this____justin`. The following are invalid: `helloworld&`, `_dothis`, and `4calvin`.

3.2 Keywords

The following are a list of reserved keywords in the language:

```
rule
gram
func
if
else
while
return
true
false
```

as well as the literal types:

```
int
double
bool
string
```

No keyword may be used as an identifier.

3.3 Literals

A literal is a notation that represents the value itself as written. Literals can only be of one of the primitive types, which are discussed below. A literal may not be used as an identifier.

3.3.1 Integer constants

An integer consists of a sequence of digits not containing a decimal point.

```
int x = 10;
```

3.3.2 Floating point constants

A floating point constant consists of two sequences of digits, where one may be the empty sequence, separated by a decimal point.

```
double y = 4.55;
```

3.3.3 Boolean constants

There exist only two boolean constants:

```
bool is_there = true;
bool is_there = false;
```

3.3.4 String constants

A string constant consists of a sequence of characters enclosed by single quotes.

```
string name = 'Anne Zhang’;
```

3.4 Comments

Just like in Java, `//` are used for single line comments and `/* */` for nested or multi-

line comments.

```
// This is a single line comment
/* This is
   a multi-line
   comment
 */
```

In a single line, all characters after `//` are ignored by the compiler.

With multi-line comments, the compiler will ignore everything from `/*` to `*/`. Note, however, that multi-line comments cannot be nested within one another like so:

```
/* Multi-line comments
   /* cannot be nested */
   like in this example!
 */
```

This will result in a syntax error, as the compiler will treat the first `*/` as the end of the comment.

3.5 Operators

Operators specify logical or mathematical operations to be performed.

Arithmetic operators:
`+` addition
`-` subtraction
`*` multiplication
`/` division
`%` modulo

`=` assignment

Logical operators:
`!` negation
`==` equivalence
`!=` non-equivalence

`<` less than
`>` greater than
`&&` AND
`||` OR

The arrow `->` is a special operator used in rule definitions in grammars. In a rule, the string to the left of the arrow can be replaced by the string or system function to the right of the arrow. For example:

```
'F' -> 'F l F r r F l F',
'r' -> turn(60)
```

are both valid rules. The arrow has no meaning outside of rule definitions, and an error will be thrown if it is used outside of this context.

3.6 Punctuators

`;`

- terminate statements

`,`

- separate function parameters, separate key-value pairs in grammar definitions

`'`

- string literal declaration

`{}`

- grammar definitions
- scope

`()`

- function arguments
- expression precedence
- type casting
- conditional parameters

4. Syntax

4.1 Program Structure

FRAC programs should be written in a single file. A FRAC program consists of grammar definitions, function definitions, and a `main()` function. Functions and grammars are defined first and subsequently used in the `main()` function, although they cannot be defined within the `main()` function itself.

The `main()` function is the entry point for the program. It may contain variable and literal declarations, expressions, and statements. It may also use any previously defined functions and grammars. In addition, the `main()` function must use one, and only one, of the following system functions: `draw()`, `grow()`. This function specifies the type of image output that the program will create.

The following is an example of a valid FRAC program:

```
gram my_grammar = {
    init: 'F r r F r r F',
    rules: {
        'F' -> 'F l F r r F l F',
        'r' -> turn(60),
        'l' -> turn(-60),
        'F' -> move(1)
    }
}

main() {
    grow(my_function(my_grammar), 2);
}
```

In this example, the program will construct a grammar given in the declaration of my_grammar. Then, it will output a GIF showing the growth of the fractal generated by that grammar (the fractal will have undergone 2 iterations, as specified by the second parameter to the `grow()` function).

4.2 Expressions

4.2.1 Variable Declarations

Variables can be declared and assigned to a value simultaneously, or declared without assignment and assigned to a value later on. Declarations take the form:

```
// declaration without assignment
var_type var_name;
var_name = value;

// declaration with assignment
var_type var_name = value;
```

where `var_type` is any of the four literal type keywords (`int` , `double` , `bool` , `string`), `var_name` is any valid identifier as defined in 3.1, and `value` is either a literal of type `var_type` or an expression that evaluates to a literal of that type.

4.2.2 Function Definitions

Functions are declared and defined simultaneously - unlike variables, they cannot be declared without definition and defined later. All functions must return a value, although the return type is not be specified in the function declaration. Any function except for the `main()` function is defined as follows:

```
func my_name(params) {
    // function body
}
```

while the `main()` function is defined without the keyword `func` :

```
main() {
    // main function body
}
```

The `main()` function should not contain any return statements.

4.2.3 Function Calls

All functions except for the `main()` function must be called explicitly, with the correct

number of arguments as specified in the function definition. The `main()` function is called implicitly at the start of every program run, and calling `main()` explicitly in the program will throw an error.

```
// valid function call
my_func(args);

// this will throw an error
main();
```

Function calls may be placed on the right-hand side of an assignment expression, in which case the identifier on the left-hand side will be assigned the return value of the function call.

```
// n is assigned the return value of my_func(args)
int n = my_func(args);
```

Function calls may also be nested. They can be passed as arguments into other functions, in which case the return value of the inner function call will be passed as an argument to the outer function call. The return value of the inner function call must match the argument type specified in the outer function's definition. A type mismatch will throw an error.

```
/* my_func must return an object of type gram, otherwise this
   expression will throw an error */
draw(my_func(args), 2);
```

4.2.4 Grammar Definitions

Grammar definitions are similar to function definitions, but the grammars themselves are more similar to objects. Grammars are defined as follows:

```
gram my_gram {
    init: // init string here,
    rules: {
      // start string -> end string
    }
}
```

Every grammar must contain at least one recursive (string-to-string) rule - it wouldn't generate a fractal otherwise! Every character in the init string must have at least one and at most two corresponding rules. If a character has only one rule, that rule must be non-recursive and evaluate to a terminal. If a character has two rules, one rule must be recursive and the other must be non-recursive. Any other combination of rules is ambiguous and will throw an error.

Grammars are evaluated when they are passed into a drawing system function (`draw()` or `grow()`). Grammar evaluations start with the `init` string, which is then evaluated recursively for the number of times specified in the second argument to the drawing function call. For every recursive evaluation, the compiler will look for a recursive rule for each character, and will only use a non-recursive rule for a character if there is no recursive rule for that character. When the recursive evaluations have been completed, the compiler uses non-recursive rules to generate a final string of terminals, which are used to draw the fractal.

4.2.5 Arithmetic Expressions

Arithmetic expressions are expressions that contain an arithmetic operator, and evaluate to a literal value. They can be placed on the right-hand side of variable assignments, or passed as arguments to function calls.

```
int x = 3;
int y = 8;
int z = x + y; // z = 11
my_func(x + y); // 11 is passed into my_func
```

4.2.6 Boolean Expressions

Boolean expressions are expressions that contain logical operators, and evaluate to a boolean value `true` or `false`. They are used to evaluate conditional and loop statements.

```
bool isTrue = true;
bool isFalse = false;
if(isTrue || isFalse) {
    print(“truth”);
```

```
}
```

4.3 Statements

A statement is a complete instruction that can be interpreted by the computer. Statements are executed sequentially within a function.

4.3.1 Expression Statements

Expression statements are the most common type of statement, and can include any of the previously covered expressions. In FRAC, all statements are terminated with a semicolon `;`.

4.3.2 Conditional Statements

Conditional statements first check the truth condition of a boolean expression, and then execute a set of statements depending on the result. Here is an example `if` / `else` conditional statement:

```
if (expression) {
    statement
}
else if (expression) {
    statement
}
else {
    statement
}
```

Only the `if` clause of the conditional statement is required. The `else` statement is executed only if none of the previous conditions return true.

4.3.3 Loop Statements

Loop statements are constructed using the `while` keyword, which allows you to iterate over blocks of code.

```
while (expression) {
```

```
    statement
    ...
}
```

In the case of `while` loops, the truth condition of the boolean expression is checked before every execution of the body of the `while` loop, which is executed only if expression returns true.

4.3.4 Return Statements

Ends the execution of a function with the use of the keyword `return`. If a function does not have a `return` statement at the end, it is assumed to be a void function without a return type.

5. System Functions

1. `move()`

This is one of two possible terminals in a FRAC grammar:

```
move(int distance)
```

The function draws a line of length `distance`.

2. `turn()`

The other possible terminal in a FRAC grammar:

```
turn(int angle)
```

The function indicates to the grammar that the current line being drawn should be re-oriented by `angle` degrees, which can be in the positive or negative direction (abiding by the right hand rule).

3. `draw()`

This is one of two functions in a FRAC program that generates a fractal image:

```
draw(gram g, int n)
```

The function creates a static image of the fractal described by the grammar `g` over `n` number of iterations.

4. `grow()`

```
grow(gram g, int n)
```

The function resembles `draw()`, except instead of creating a static image, it creates a dynamically “growing” GIF (merely a collection of static images) of the fractal described by the given grammar `g` over `n` iterations.

5. `print()`

```
print(string s)
```

The function prints out the string `s` to the standard output. The same escape sequences as Java would be interpreted correspondingly (i.e. `\n` for newline).