PLT 4115 LRM: **JaTesté**

Andrew Grant
amg2215@columbia.edu

Jemma Losh
jal2285@columbia.edu

Jared Weiss
jbw2140@columbia.edu

Jake Weissman
jdw2159@columbia.edu

March 3, 2016

1 Introduction

The goal of JaTesté is to design a language that promotes good coding practices - mainly as it relates to testing. JaTesté will require the user to explicitly define test cases for any function that is written in order to compile and execute code. This will ensure that no code goes untested and will increase the overall quality of programmer code written in our language. The user will be required to provide some test cases for their code, and the language will also generate some important test cases for their code as well. JaTesté is mostly a functional language with a syntax quite similar to C. The details of our language usage is provided in the rest of the document.

2 Lexical Conventions

This chapter will describe how input code will be processed and how tokens will be generated.

2.1 Identifiers

Identifiers are used to name a variable, a function, or other types of data. An identifier can include all letters, digits, and the underscore character. An identifier must start with either a letter or an underscore - it cannot start with a digit. Capital letters will be treated differently from lower case letters.

2.2 Keywords

Keywords are a set of words that serve a specific purpose in our language and may not be used by the programmer for any other reason. The list of keywords the language recognizes and reserves is as follows:

```
int char float struct if else for while break continue with test using func return
```

2.3 Constants

2.3.1 Integer Constants

2.3.2 Character Constants

2.3.3 Real Number Constants

2.3.4 String Constants

2.4 Operators

Operators are special tokens such as multiply, equals, etc. that are applied to one or two operands. Their use will be explained further in chapter 4.

2.5 Separators

2.6 White Space

3 Data Types

The data types in JaTeste can be classified into three categories: primitive types, structures, and arrays.

3.1 Primitives

3.1.1 Integer Types

The integer data type is a 32 bit value that can hold whole numbers ranging from $-2,147,483,648 to 2,147,483,647$. Keyword 'int' is required to declare a variable with this type.

```
int a = 10;
int b = a + 10;
```

3.1.2 Character Type

The character type is an 8 bit value that is used to hold a single character. The keyword "char" is used to declare a variable with this type.

```
char c = 'h';
char b = 'e';
char d = 'l';
```

3.2 Structures

The structure data type is a collection of primitive types and other structures. The keyword "struct" followed by the name of the struct is used to define structures. Curly braces are then used to define what the structure is made of. As an example, consider the following:

3.2.1 Defining Structures

```

struct person = {
string name;
int age;
int height;
};


struct manager = {
struct person name;
int salary;
};
```

This syntax is very similar to C.

3.2.2 Initializing Structures

To create instances of structs:

```
struct manager yahoo_manager;
struct person sam;
```

Here, we create two variables yahoo_manager and sam. The former is of type "struct manager", and the latter is of type "struct person".

3.2.3 Accessing Structure Members

To access structs and modify its variables , a period following by the variable name is used:

```
 yahoo_manager.name = sam;
yahoo_manager.age = 45;
yahoo_manager = 65000;
```

Ultimately, all structures are backed by some collection of primitives. For example, the first struct, manager, is made up of another struct and an int. Since the struct person is made up of two ints, manager is really just made up of three ints.

3.3 Arrays

3.3.1 Defining Arrays

3.3.2 Initializing Arrays

3.3.3 Accessing Array Elements

3.3.4 Multidimensional Arrays

3.3.5 Arrays of Structures

Text

4 Expressions and Operators

4.1 Expressions

An expression is a collection of one or more operands and zero or more operators that can be evaluated to produce a value. A function that returns a value can be an operand as part of an expression. Additionally, parenthesis can be used to group smaller expressions together as part of a larger expression. A semicolon terminates an expression. Some examples of expressions include:

- `35 - 6;`
- `foo(42) * 10;`
- `8 - (9 / (2 + 1));`

4.2 Assignment Operators

Assignment can be used to assign the value of an expression on the right side to a named variable on the left hand side of the equals operator. The left hand side can either be a named variable that has already been declared or a named variable that is being declared and initialized in this assignment. Examples include:

- `int x = 5;`
- `float y;`
  `y = 9.9;`

Additionally, the following operators can also be used for variations of assignment:

- `+=` increments the left hand side by the result of the right hand side
- `-=` decrements the left hand side by the result of the right hand side

4.3 Incrementing and Decrementing

This can be done using the `++` operator to increment and the `--` operator to decrement a value. If the operator is placed before a value it will be incrememnted / decremented first, then it will be evaluated. If the operator is placed following a value, it will be evaluated with its original value and then incremented / decremented.

4.4 Arithmetic Operators

- `+` can be used for addition
- `-` can be used for subtraction (on two operands) and negation (on one operand)
- `*` can be used for multiplication
- `/` can be used for division
- $\wedge$ can be used for exponents
- % can be used for modular division

4.5 Comparison Operators

- `==` can be used to evaluate equality
- `!=` can be used to evaluate inequality
- `>` can be used to evaluate is the left greater than the right
- `>=` can be used to evaluate is the left greater than or equal to the right
- `<` can be used to evaluate is the left less than the right
- `<=` can be used to evaluate is the left less than or equal to the right

4.6 Logical Operators

- `!` can be used to evaluate the negation of one expression
- && can be used to evaluate logical and
- || can be used to evaluate logical or