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 6, 2016

Contents

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.

```
ID = "(['a'-'z' 'A'-'Z'] | '_') (['a'-'z' 'A'-'Z'] | ['0'-'9'] | '_')*"
```

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, double, struct, if, else, for, while, with test, using, func, return, string`

2.3 Constants

Our language includes integer, character, real number, and string constants. They're defined in the following sections.

2.3.1 Integer Constants

Integer constants are a sequence of digits. An integer is taken to be decimal. The regular expression for an integer is as follows:

```
digit = ['0' - '9']
int = digit+
```

2.3.2 Double Constants

Real number constants represent a floating point number. They are composed of a sequence of digits, representing the whole number portion, followed by a decimal and another sequence of digits, representing the fractional part. Here are some examples. The whole part or the fractional part may be omitted, but not both. The regular expression for a double is as follows:

```
double = int | digit*['.']digit+ | digit+['.']digit*
```

2.3.3 Character Constants

Character constants hold a single character and are enclosed in single quotes. They are stored in a variable of type char. Character constants that are preceded with a backslash have special meaning. The regex for a character is as follows:

```
char = ['a' - 'z' 'A' - 'z']
```

2.3.4 String Constants

Strings are a sequence of characters enclosed by double quotes. A String is treated like a character array. The regex for a string is as follows:

```
string = char*
```

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 White Space

Whitespace is considered to be a space, tab, or newline. It is used for token delimitation, but has no meaning otherwise.

```
WHITESPACE = "[' '  '\t' '\r' '\n']"
```

2.6 Comments

A comment is a sequence of characters beginning with a forward slash followed by an asterisk. It continues until it is ended with an asterisk followed by a forward slash. Comments are treated as whitespace.

```
COMMENT = "/\* [^ \*/]* \*/ "
```

2.7 Separators

Separators are used to separate tokens. Separators are single character tokens, except for whitespace which is a separator, but not a token.

```
'('        { LPAREN }
')'        { RPAREN }
'{'        { LBRACE }
'}'        { RBRACE }
';'        { SEMI }
'['        { LBRACKET }
']'        { RBRACKET }
'.'        { DOT }
','        { COMMA }
```

3 Data Types

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

3.1 Primitives

The primitives our language recognizes are int, char, and string

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. A variable must be declared before it can be assigned a value, this cannot be done in one step.

```
int a;
a = 10;
a = 21 * 2;
```

The grammar that recognizes an integer deceleration is:

```
typ ID
```

The grammar that recognizes an integer initialization is:

```
ID ASSIGN expr
```

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. A variable must be declared before it can be assigned a value, this cannot be done in one step.

```
char a;
a = 'h';
```

The grammar that recognizes a char deceleration is:

```
typ ID SEMI
```

The grammar that recognizes a char initialization is:

```
typ ID ASSIGN expr SEMI
```

3.1.3 String Type

The string type is variable length and used to hold a string of chars. The keyword `string` is used to declare a variable with this type. A variable must be declared before it can be assigned a value, this cannot be done in one step.

```
string a;
a = "hello";
```

The grammar that recognizes a char deceleration is:

```
typ ID SEMI
```

The grammar that recognizes a char initialization is:

```
typ ID ASSIGN expr SEMI
```

3.2 Structures

The structure data type is a collection of primitive types and other structure data types. 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;
};
```

Here we have defined two structs, the first being of type `struct person` and the second of type `struct manager`. The grammar that recognizes defining a structure is as follows:

```
STRUCT STRING_LITERAL ASSIGN LBRACE vdecl_list RBRACE
```

3.2.2 Initializing Structures

To create a structure, the struct type is followed by a variable name.

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

Here, we create two variables yahoo_manager and sam. The first is of type "struct manager", and the second 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.salary = 65000;
```

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

3.3 Arrays

An array is a data structure that allows for the storage of one or more elements of the same data type consecutively in memory. Each element is stored at an index, and array indices begin at 0. This section will describe how to use Arrays.

3.3.1 Defining Arrays

An array is declared by specifying its data type, name, and size. The size must be positive. Here is an example of declaring an integer array of size 5:

```
arr = new int[5];
```

```
ID ASSIGN NEW prim_typ LBRACKET INT_LITERAL RBRACKET
```

3.3.2 Initializing Arrays

An array can be initialized by listing the element values separated by commas and surrounded by brackets. Here is an example:

```
arr = { 0, 1, 2, 3, 4 };
```

It is not required to initialize all of the elements. Elements that are not initialized will have a default value of zero.

3.3.3 Accessing Array Elements

To access an element in an array, use the array name followed by the element index surrounded by square brackets. Here is an example that assigns the value 1 to the first element (at index 0) in the array:

```
arr[0] = 1;
```

JaTeste does not test for index out of bounds, so the following code would compile although it is incorrect.

```
arr = new int[2];
arr[5] = 1;
```

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) );
```

The regex for an expression is as follows:

```
expr SEMI
```

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

The grammar for assignment is as follows:

```
ID ASSIGN expr
```

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 incremented / 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

  The grammar for the above operators, in order, is as follows:

```
expr PLUS     expr
expr MINUS    expr
expr TIMES    expr
expr DIVIDE   expr
```

- $\wedge$ 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 less than the right
- `<=` can be used to evaluate is the left less than or equal to the right
- `>` 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

  The grammar for the above operators, in order, is as follows:

```
expr EQ    expr
expr NEQ   expr
expr LT    expr
expr LEQ   expr
expr GT    expr
expr GEQ   expr
```

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

The grammar for the above operators, in order, is as follows:

```
NOT  expr
expr AND  expr
expr OR   expr
```

4.7 Operator Precedence

4.8 Order of Evaluation

5 Statements

Statements include: `if, while, for, return` as explained in the following sections

5.1 If Statement

The if, else if, else construct will work as expected in other languages.

```
if (x == 42) {
  print("Gotcha");
}
else if (x > 42) {
  print("Sorry, too big");
}
else {
  print("I\'ll allow it");
}
```

The grammar that recognizes an if statement is as follows:

```
IF LPAREN expr RPAREN stmt ELSE stmt
IF LPAREN expr RPAREN stmt %prec NOELSE
```

5.2 While Statement

The while statement will evaluate in a loop as long as the specified condition in the while statement is true.

```
/* Below code prints "Hey there" 10 times */
int x = 0;
while (x < 10) {
  print("Hey there");
  x++;
}
```

The grammar that recognizes a while statement is as follows:

```
WHILE LPAREN expr RPAREN stmt
```

5.3 For Statement

The for condition will also run in a loop so long as the condition specified in the for statement is true. The expectation for a for statement is as follows:

`for ( <initial state>; <test condition>; <step forward> )`

Examples are as follows:

```
/*  This will run as long as i is less than 100
  i will be incremented on each iteration of the loop */
for (int i = 0; i < 100; i++) {
  /* do something */
}

/* i can also be declared or initialized outside of the for loop */
int i;
for (i = 0; i < 100; i += 2) {
  /* code block */
}
```

The grammar that recognizes a for statement is as follows:

```
FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN
```

5.4 Code Blocks

Blocks are code that is contained within a pair of brackets, { `code` }, that gets executed within a statement. For example, any code blocks that follow an `if` statement will get executed if the `if` condition is evaluated as true:

```
int x = 42;
if (x == 42) {
  /* the following three lines are executed */
  print("Hey");
  x++;
  print("Bye");
}
```

The grammar that recognizes a block of code is as follows:

```
LBRACE stmt RBRACE
```

5.5 Return Statement

The `return` statement is used to exit out of a function and return a value. The return value must be the same type that is specified by the function deceleration. Return can be used as follows:

```
/* The function trivially returns the input int value */
func int someValue(int x) {
  return x;
}
```

The grammar that recognizes a return statement is as follows:

```
RETURN SEMI
RETURN expr SEMI
```

6 Functions

Functions allow you to group snippets of code together that can subsequently be called from other parts of your program, depending on scope. Functions are global, unless they are prepended with the keyword "private". While not necessary, it is encouraged that you declare functions before defining them. Functions are usually declared at the top of the file they're defined in. Functions that aren't declared can only be called after they have been defined.

6.1 Function Declarations

The keyword "func" is used to declare a function. A return type is also required using keyword "return"; if your function doesn't return anything then use keyword "void" instead. Functions are declared with or without parameters; if parameters are used, their types must be specified. A function can be defined with multiple, different parameters. Though a function can only have one return type, it can also be any data type.

```
1 func int add(int a, int b); /* this functions has two int parameters as input and
      returns an int */
2 func void say_hi(); /* this function doesn't return anything nor takes any
      parameters */
3 func int isSam(string name, int age); /* this functions has two input parameters,
      one of type string and one of type int */
```

6.2 Function Definitions

Function definitions contain the instructions to be performed when that function is called. The first part of the syntax is similar to how you declare functions; but curly brackets are used to define what the function actually does. For example,

```
1 func int add(int a, int b); /* declaration */
2
3 func int add(int x, int y) /* definition */
4 {
5 return x + y;
6 }
```

```
fdecl:
          FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list stmt_list RBRACE
```

This snippet of code first declares add, and then defines it. Declaring before defining is best practice. Importantly, functions can *not* reference global variables; that is, the only variables they can act on are formal parameters and local variables. For example:

```
1 func int add_to_a(int x); /* declaration */
2 int a = 10;
3 func int add_to_a(int x) /* definition */
4 {
5 return x + a; /* this is NOT allowed */
6 }
```

This code is no good because it relies on global variable "a". Functions can only reference formal parameters and/or local variables.

6.3 Calling Functions

A function is called using the name of the function along with any parameters it requires. You *must* supply a function with the parameters it expects. For example, the following will not work:

```
func int add(int a, int b); /* declaration */

func int add(int x, int y) /* definition */
{
return x + y;
}

add(); /* this is wrong and will not compile because add expects two ints as
    parameters */
```

Functions can only be called after they have been declared and/or defined. Functions are first class objects and so can be used anywhere a normal data type can be used. Of course, a function's return type must be compatible with the context it's being used in. For example, a function that returns a char cannot be used as an actual parameter to a function that expects an int. Consider the following:

```
func int add_int(int a, int b); /* declaration */

func int add_int(int x, int y) /* definition */
{
return x + y;
}

func float add_float(float x, float y)
{
  return x + y;
}

func int subtract(int x, int y)
{
  return x - y;
}

int answer = subtract(add(10,10), 10); /* this is ok */
int answer2 = subtract(add_float(10.0,10.0), 10); /* this is NOT ok because
    subtract expects its first parameter to be an int while add_float returns a
    float */
```

6.4 Function Parameters

Formal parameters can be any data type. Furthermore, they need not be of the same type. For example, the following is syntactically fine:

```
func void speak(int age, string name)
{
  print_string ("My name is" + name + " and I am "  + age);
}
```

```
formal_opts_list:
          /* nothing */
        | formal_opt

formal_opt:
```

```
            any_typ_not_void ID
          | formal_opt COMMA any_typ_not_void ID
```

While functions may be defined with multiple formal parameters, that number must be fixed. That is, functions cannot accept a variable number of arguments.
TODO: Are we passing by value or reference?

6.5 Recursive Functions

Functions can be used recursively. Each recursive call results in the creation of a new stack and new set of local variables. It is up to the programmer to prevent infinite loops.

6.6 Function Test Cases

Functions can be appended with test cases directly in the source code. Most importantly, the test cases will be compiled into a separate (executable) file. The keyword "with test" is used to define a test case as illustrated here:

```
func int add(int a, int b); /* declaration */

func int add(int x, int y) /* definition */
{
return x + y;
}
with test {
  add(1,2) == 3;
  add(-1, 1) == 0;
}
with test {
  add(0,0) <= 0;
  add(0,0) >= 0;
}
```

```
FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list stmt_list RBRACE testdecl

testdecl:
        WTEST LBRACE stmt_list RBRACE usingdecl
```

Test cases contain a set of boolean expressions. Multiple boolean expressions can be defined, they just must be separated with semi-colons. As shown above, you can define separate test cases one after another too.
Snippets of code can also be used to set up a given test case's enviornment using the "using" keyword. That is, "using" is used to define code that is executed right before the test case is run. Consider the following:

```
func void changeAge(struct person temp_person, int age)
{
temp_person.age = age;
}
with test {
  sam.age == 11;
}
using {
struct person sam;
sam.age = 10;
changeAge(sam, 11);
}
```

```
FUNC any_typ ID LPAREN formal_opts_list RPAREN LBRACE vdecl_list stmt_list RBRACE testdecl usingdecl

usingdecl:
        USING LBRACE vdecl_list stmt_list RBRACE
```

"using" is used to create a struct and then call function changeAge; it is setting up the enviornment for it's corresponding test. Variables defined in the "using" section of code can safely be referenced in its corresponding test case as shown. Basically, the code in the "using" section is executed right before the boolean expressions are evaluated and tested.
The "using" section is optional. As a result some test cases may contain "using" sections and others might not. As per convention, each "using" section will match up with its closest test case. For, example:

```

func int add(int x, int y) /* definition */
{
return x + y;
}
with test { /*  variables a, b defined below are NOT in this test case's scope*/
  add(1,2) == 3;
  add(-1, 1) == 0;
}
with test { /* variables a and b ARE in this test case's scope */
  add(a, b) == 20;
}
using {
int a = 10;
int b = 10;
}
```

As explained in the comments, the "using" section is matched up with the second test case.
Test cases are compiled into a separate program which can subsequently be run. The program will run all test cases and output appropriate information.

7 Scanner and Parser Code

7.1 scanner.mll

```


{ open Parser }

(* Regex shorthands *)
let digit = ['0' - '9']
let int = digit+
let double = int | digit*['.']digit+ | digit+['.']digit*
let char = '''['a' - 'z' 'A' - 'Z']'''
let string = char+

rule token = parse
     [' ' '\t' '\r' '\n' ] { token lexbuf } (* White space *)
  | "/*"       { comment lexbuf }
  | '('      { LPAREN }
  | ')'      { RPAREN }
  | '{'      { LBRACE}
  | '}'      { RBRACE}
```

```
  | ','      { COMMA }
  | ';'      { SEMI }

  (* Operators *)
  | "+"      { PLUS }
  | "-"      { MINUS }
  | "*"      { TIMES }
  | "/"      { DIVIDE }
  | "%"      { MODULO }
  | "^"      { EXPO }
  | "="      { ASSIGN }
  | "=="       { EQ }
  | "!="       { NEQ }
  | "!"      { NOT }
  | "&&"       { AND }
  | "||"       { OR }
  | "<"      { LT }
  | ">"      { GT }
  | "<="       { LEQ }
  | ">="       { GEQ }
  | "["      { LBRACKET }
  | "]"      { RBRACKET }
  | "."      { DOT }

  (* Control flow *)
  | "if"       { IF }
  | "else"     { ELSE }
  | "return"     { RETURN }
  | "while"      { WHILE }
  | "for"      { FOR }

  (* Datatypes *)
  | "void"     { VOID }
  | "struct"     { STRUCT }
  | "double"     { DOUBLE }
  | "int"      { INT }
  | "char"     { CHAR }
  | "string"     { STRING }
  | "func"     { FUNC }
  (* Pointers *)
  | "int*"     { INT_PT }
  | "double*"    { DOUBLE_PT }
  | "char*"    { CHAR_PT }
  | "struct*"    { STRUCT_PT }
  (* Arrays *)
  | "int[]"    { INT_ARRAY }
  | "char[]"     { CHAR_ARRAY }
  | "double[]"     { DOUBLE_ARRAY }

  | "new"      { NEW }

  (* Testing keywords *)
  | "with test"      { WTEST }
  | "using"    { USING }

  | ['a' - 'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm)}
  | int as lxm       { INT_LITERAL(int_of_string lxm)}
  | double as lxm    { DOUBLE_LITERAL(lxm) }
  | char as lxm      { CHAR_LITERAL(String.get lxm 1) }
```

```
  | string as lxm   { STRING_LITERAL(lxm) }

  | eof { EOF }
  | _ as char { raise (Failure ("illegal character " ^
      Char.escaped char))}


(* Whitespace*)
and comment = parse
  "*/" { token lexbuf }
  | _ { comment lexbuf }
```

7.2 parser.mly

```
{ open Parser }

(* Regex shorthands *)
let digit = ['0' - '9']
let int = digit+
let double = int | digit*['.']digit+ | digit+['.']digit*
let char = '''['a' - 'z' 'A' - 'Z']'''
let string = char+

rule token = parse
     [' ' '\t' '\r' '\n' ] { token lexbuf } (* White space *)
  | "/*"       { comment lexbuf }
  | '('      { LPAREN }
  | ')'      { RPAREN }
  | '{'      { LBRACE}
  | '}'      { RBRACE}
  | ','      { COMMA }
  | ';'      { SEMI }

  (* Operators *)
  | "+"      { PLUS }
  | "-"      { MINUS }
  | "*"      { TIMES }
  | "/"      { DIVIDE }
  | "%"      { MODULO }
  | "^"      { EXPO }
  | "="      { ASSIGN }
  | "=="       { EQ }
  | "!="       { NEQ }
  | "!"      { NOT }
  | "&&"       { AND }
  | "||"       { OR }
  | "<"      { LT }
  | ">"      { GT }
  | "<="       { LEQ }
  | ">="       { GEQ }
  | "["      { LBRACKET }
  | "]"      { RBRACKET }
  | "."      { DOT }

  (* Control flow *)
  | "if"       { IF }
  | "else"     { ELSE }
```

```
  | "return"    { RETURN }
  | "while"     { WHILE }
  | "for"     { FOR }

  (* Datatypes *)
  | "void"    { VOID }
  | "struct"    { STRUCT }
  | "double"    { DOUBLE }
  | "int"     { INT }
  | "char"    { CHAR }
  | "string"    { STRING }
  | "func"    { FUNC }
  (* Pointers *)
  | "int*"    { INT_PT }
  | "double*"   { DOUBLE_PT }
  | "char*"   { CHAR_PT }
  | "struct*"   { STRUCT_PT }
  (* Arrays *)
  | "int[]"   { INT_ARRAY }
  | "char[]"    { CHAR_ARRAY }
  | "double[]"    { DOUBLE_ARRAY }

  | "new"     { NEW }

  (* Testing keywords *)
  | "with test"     { WTEST }
  | "using"   { USING }

  | ['a' - 'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm)}
  | int as lxm      { INT_LITERAL(int_of_string lxm)}
  | double as lxm   { DOUBLE_LITERAL(lxm) }
  | char as lxm     { CHAR_LITERAL(String.get lxm 1) }
  | string as lxm   { STRING_LITERAL(lxm) }

  | eof { EOF }
  | _ as char { raise (Failure ("illegal character " ^
      Char.escaped char))}


(* Whitespace*)
and comment = parse
  "*/" { token lexbuf }
  | _ { comment lexbuf }
```