

Nathaniel D. Priestley

Capstone Portfolio

Section 2: Teamwork:

Teams for this capstone project were formed with two total members. Team member one made contributions for team member two, and team member two made contributions for team member one. This section will be detailing the contributions made by team member two of this capstone project. Two major contributions were made by team member two. These contributions include three additional tests for the compiler test suite as well as a full documentation of the Catscript language used during the compilers course.

The tests provided by each team member were constructed with the goal of testing overlooked aspects which were absent from the entire test suite as a whole. In other words the tests needed to not be a repetition of other tests already present. The tests were also designed with simplicity in mind so the amount of effort required to get them passing wasn't too great. The tests provided by team member two targeted lists of lists, function line numbers, and nested unary operators. The list of lists test, checked to see if a list could contain multiple lists. Although there were list related tests, none of them verified whether or not a list of lists should work. The second test verified that line numbers were correct across a multi-line function. The only other line number tests within the project files were based around the tokenizer and not function checking. The final test provided aimed to verify that a nested unary operator works. An example of a nested unary operator would be: $-(-1)$, negative, negative one.

The second portion of contributions made by team member two was an in depth documentation on the catscript language. This documentation contains information regarding statements, expressions, and catscript programs. The structure of the language is clear when viewing the documentation and examples are also provided for the use of various sections.

Personal contributions towards the project were coding empty and missing sections of the compiler. The test suite was present to provide a general direction for where to start and what to do next. The first section to be completed was the tokenizer which was required for any of the other sections in the compiler to work properly. Afterwards, parsing, evaluation, and compiling were completed resulting in the finished compiler project.

Section 3: Design Patterns:

A design pattern used in this capstone project was **memoization** which is defined as a technique geared towards optimizing the speed of programs by caching results for later use instead of recreating that result. This design pattern was used in the CatscriptType.java class file in a method called “getListType”. Previously the method would create a new type every time it was called thus costing additional computing resources. After implementing memoization these types are now stored inside of a hashmap that can be easily accessed. Now whenever the method is called it will check the hashmap for the input type provided. If the type is present within the hashmap it will return it, otherwise a new type is created. Then if that newly created type were to be given as input for a new call of the method, it would be retrieved from the hashmap and returned.

Section 4: Technical Writing:

1

CatScript 2022 - Documentation

CatScript Language Written by Nathaniel Priestley

Documentation Written by Collin Wright

CatScript assignment and starter code provided by MSU’s CSCI 468 Compilers course

Introduction

This document outlines the grammar and usage of CatScript, a simplified scripting language used to demonstrate the fundamentals of programming language compilation. CatScript was created and coined by Carson Gross for the CSCI 468 - Compilers course at Montana State University. This rendition of CatScript was completed by Nathaniel Priestley. The language was created as a semester-long project to teach the fundamentals of programming languages and compiler systems. The following provides an overview of the basic features of CatScript and their uses.

Program Overview

As in most other languages, Catscript programs consist of a series of written expressions and statements to be evaluated and executed. In Catscript, expressions are identifying features which are to be evaluated to produce a valid and simplified resulting value. Statements, on the other hand, are meant to be executed to perform a directed action, often (but not always) within the bounds of a specified expression. The grammar rules relating to each instruction type are explained in the grammar below (Figure 1). This documentation provides an in-depth explanation of each of these features offered by CatScript.

Figure 1: Catscript Grammar Rules

```
catscript_program = { program_statement };

program_statement = statement |
                  function_declaration;

statement = for_statement |
            if_statement |
            print_statement |
            variable_statement |
            assignment_statement |
            function_call_statement;

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
               '{', { statement }, '};

if_statement = 'if', '(', expression, ')', '{',
               { statement },
               '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];

print_statement = 'print', '(', expression, ')'
```

(Cont.)

Figure1:CatscriptGrammarRules(Continued)

```
variable_statement = 'var', IDENTIFIER,  
    [ ':', type_expression, ] '=', expression;  
  
function_call_statement = function_call;  
  
assignment_statement = IDENTIFIER, '=', expression;  
  
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +  
    [ ':', type_expression ], '{', { function_body_statement }, '}'  
  
function_body_statement = statement |  
    return_statement;  
  
parameter_list = [ parameter, { ',', parameter } ];  
  
parameter = IDENTIFIER [ , ':', type_expression ];  
  
return_statement = 'return' [ , expression];  
  
expression = equality_expression;  
  
equality_expression = comparison_expression { ("!=" | "==") comparison_expression };  
  
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expression };  
  
additive_expression = factor_expression { ("+" | "-") factor_expression };  
  
factor_expression = unary_expression { ("/" | "*" ) unary_expression };  
  
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;  
  
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |  
    list_literal | function_call | "(" , expression , ")"  
  
list_literal = '[', expression, { ',', expression } ']'  
  
function_call = IDENTIFIER, '(', argument_list , ')'  
  
argument_list = [ expression , { ',', expression } ]  
  
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [ , '<' , type_expression , '>']
```

Statements

Program Statement:

```
program_statement = statement | function_declaration
```

A CatScript Program consists of a series of these Program Statements. A Program Statement is a high-level rule that encapsulates what a program instruction could be. This rule depicts that a Program Statement can be either a Statement (See **Statement**) or a Function Declaration (See **Function Declaration**).

Statement:

```
statement = for_statement | if_statement |  
           print_statement | variable_statement |  
           assignment_statement | function_call_statement
```

A Statement is any CatScript instruction to be carried out in a further specified manner. This Statement rule is an abstracted feature that could refer to one of any of the many statement types offered by CatScript including a For Statement, If Statement, Print Statement, Variable Statement, Assignment Statement, or a Function Call Statement (For more on each of these types of statements, see corresponding descriptions below).

For Statement:

```
for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',  
               '{', { statement }, '}' ;
```

A For Statement is a Statement type which allows one to iterate through a collection of data in a specified Expression, operating with a specified set of Statements for each iteration. In using a For Statement, the “for” keyword is followed by parentheses containing a given Identifier (which is used to access that iterative instance from the data in each respective iteration), the “in” keyword, and an Expression designating the data to iterate through. The for header is followed by a braced “body” which contains some sort of Statement (See **Statement**), to be called for each iteration of the loop.

If Statement:

```
if_statement = 'if', '(', expression, ')', '{', { statement },  
              '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];
```

An If Statement is a Statement type which allows the user to conditionally call a specified set of Statements given a specified condition. In using an If Statement, the “if” keyword is followed by parentheses containing an Expression that evaluates to a simple true/false boolean value. Following this condition is a braced set of Statements. These Statements will be called if the conditional Expression evaluates to true, and will be passed over if it evaluates to false. Finally, an optional “else” keyword can be added to either specify Statements to be called if the condition Expression is false, or to include more conditional If Statements if followed by another If Statement.

Print Statement:

```
print_statement = 'print', '(', expression, ')'
```

A Print Statement is a Statement type which allows the user to display a given expression to the output of the console. Using the print statement involves instructing the “print” keyword followed by parentheses containing the Expression that is to be outputted.

Variable Statement:

```
variable_statement = 'var', IDENTIFIER,  
                    [ ':', type_expression, ] '=', expression;
```

A Variable Statement is a Statement type which allows the user to store an evaluated value to a variable that can be referenced and manipulated elsewhere in the program. In using the Variable Statement, the “var” keyword is followed by an Identifier (given as a name by which this stored variable can be referenced). Following the Identifier, a colon followed by a Type Expression can be included to specify a specific type that the stored variable needs to be. Next, an equals sign indicates the assignment of the variable to whatever value is evaluated by the following Expression.

Function Call Statement:

```
function_call_statement = function_call;
```

A Function Call Statement is an abstract Statement type which indicates that the user can call a function as an option for a Statement. (See **Function Call**)

Assignment Statement:

```
assignment_statement = IDENTIFIER, '=', expression;
```

An Assignment Statement is a Statement type which allows the user to store an evaluated value into an identifier that can be referenced elsewhere in the program. In using an Assignment Statement, an Identifier is given as a name by which to reference this stored value, followed by an equals sign and a given Expression that is to be evaluated to provide the value to be stored by the Identifier. Until manipulated elsewhere, when this identifier is referenced following this assignment, the evaluated value stored here will be provided.

Function Declaration:

```
function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' + [
    ':' + type_expression ], '{', { function_body_statement }, '}';
```

A Function Declaration is a Statement type which allows the user to create a callable function of instructional statements. In declaring a function, the “function” keyword is used to indicate that the following is a function that can be referenced and called/executed from elsewhere in the code. This “Function” keyword is followed by an Identifier which acts as the given name of the function by which the function can be referenced/called from elsewhere in the program. The name/Identifier is then followed by parentheses. If any given inputs are required for the functions operations, then a Parameter List should be included within these parentheses. If a specific type of value is to be returned by this function, then a colon followed by a Type Expression should be provided, indicating that this function will (and must) return a value of the specified type. Finally, the function is given a Function Body Statement between braces, specifying the statements to be carried out upon the execution of this function.

Function Body Statement:

```
function_body_statement = statement | return_statement;
```

A Function Body Statement provides the contents of the body of a written function. A Function Body Statement consists of a series of Statements (See **Statement**) or Return Statement (See **Return Statement**).

Parameter List:

```
parameter_list = [ parameter, {',' parameter } ];
```

A Parameter list is a list of given inputs to a written function. A Parameter list is at least one Parameter (See **Parameter**), followed by any number of a comma followed by another Parameter.

Parameter:

```
parameter = IDENTIFIER [ , ':', type_expression ];
```

A Parameter is an Identifier-Type Expression pair that can be sent to a written function in a parameter list. Each Parameter indicates a value that will be sent to the function upon calling it from elsewhere. Here, a Parameter has an Identifier, which acts as a name that the body of the function can refer to the given value by. Optionally, each Parameter can include a colon followed by a Type Expression which specifies that the given value must be of that designated type.

7

Return Statement:

```
return_statement = 'return' [, expression];
```

A Return Statement is a special instruction exclusive to instructions within a Function Body Statement. Return Statements allow users to terminate the function they reside in and (if desired) to return a resulting value from the function. When a Return Statement is called the function it lies in will be done executing, and any specified values are returned as the result of the function. Using a Return Method requires the use of the “return” keyword” which halts

the function. An Expression may follow the keyword indicating that that Expression's evaluated value should be returned to the place in the code which called the function.

Expressions

Expression:

```
expression = equality_expression;
```

Expressions are instructions that lie within many of the Statements described above. Expressions encapsulate values and operations to manipulate those values, allowing for the simplification of more elaborate data via evaluation. All expressions can be evaluated down to a simplified value form to be assigned or stored elsewhere in a program. This Expression rule is an abstracted feature of more specific Expressions, meaning that any use of this generalized Expression can be replaced by some Equality Expression.

Equality Expression:

```
equality_expression = comparison_expression { ("!=" | "==")  
                                comparison_expression };
```

Equality Expressions are the highest level of non-abstract Expressions. All Equality Expressions consist of a Comparison Expression (See **Comparison Expression**) followed by an optional equality operator (“==” for equals and “!=” for not equals) and an additional Comparison Expression. If this optional side is included, then the Equality Expression will evaluate to a true/false boolean corresponding to whether or not the two sides are equal or not (respective to either the “equal” or “not-equal” operators).

8

Comparison Expression:

```
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=")  
                                additive_expression };
```

Comparison Expressions are contained under Equality Expressions, meaning they will be evaluated before Equality Expressions in evaluation order. All Comparison Expressions consist of an Additive Expression (See **Additive Expression**) followed by an optional

comparison expression (“>” for greater than, “>=” for greater than or equal to, “<” for less than, and “<=” for less than or equal to) and an additional Additive Expression. If this optional side is included, then the Comparison Expression will evaluate to a true/false boolean corresponding to how the two sides compare (relative to the comparison operator used).

Additive Expression:

```
additive_expression = factor_expression { ("+" | "-" ) factor_expression };
```

Additive Expressions are contained under Comparison Expressions, meaning they will be evaluated before Comparison Expressions in evaluation order. All Additive Expressions consist of a Factor Expression (See **Factor Expression**) followed by an optional additive operator (“+” for addition and “-” for subtraction) and an additional Factor Expression. If this optional side is included, the Additive Expression will provide the sum or difference (depending on the operator used) of the two Factor Expressions.

Factor Expression:

```
factor_expression = unary_expression { ("/" | "*" ) unary_expression };
```

Factor Expressions are contained under Additive Expressions, meaning they will be evaluated before Additive Expressions in evaluation order. All Factor Expressions consist of a Unary Expression (See **Unary Expression**) followed by an optional multiplicative operator (“*” for multiplication and “/” for division) and an additional Unary Expression. If this optional side is included, the Factor Expression will provide the product or quotient (depending on the operator used) of the two Unary Expressions.

9

Unary Expression:

```
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;
```

Unary Expressions are contained under Factor Expressions, meaning they will be evaluated before Factor Expressions in evaluation order. A unary Expression is either a unary operation (“not” for boolean values, and “-” for numerical values) followed by another Unary Expression, or concludes as a Primary Expression (See **Primary Expression**). The Unary Expression’s purpose is to allow the user to add unary operations to Primary Expressions. Using the “not”

operator will change a boolean's value from true if it is false, or to false if it is true. Using the “-” operator on a numerical value will evaluate as the negative value of the number.

Primary Expression:

```
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" |  
"null" | list_literal | function_call | "(", expression, ")"
```

A Primary Expression is the main basic form of values. A Primary Expression can simply be a literal value such as an IDENTIFIER(given name), STRING(a java-style string), INTEGER (32 bit integer), a boolean value “true” or “false” or the null type “null.” In these literal forms, no more extraction is needed and the value represents the raw value it should represent. Primary

Expressions can also encapsulate other forms of abstraction such as a List Literal (See **List Literal**) or a Function Call (See **Function Call**). Finally, to override the order of operations and create a new, prioritized order of operations, a Primary Expression can be stored as a set of parentheses containing another (generalized) Expression (See **Expression**). This parenthetical option indicates that the expression within the parentheses will be evaluated before any evaluates outside of the parentheses.

List Literal:

```
list_literal = '[' , expression , { ' , ' , expression } ' ] ' ;
```

A List Literal is a list containing any number of expressions which are each evaluated yet constitute a single Expression as a list. A List Literal is designated by a pair of brackets containing zero or more (generalized) Expressions (See **Expression**) each separated by commas.

10

Function Call:

```
function_call = IDENTIFIER , ' ( ' , argument_list , ' ) ' ;
```

A Function Call is a specified expression that instructs the execution of a written function (See **Function Declaration**). Upon evaluation, a Function Call (sending any necessary specified parameters) invokes the body of the designated function. A Function Call consists of an Identifier that corresponds to the identifier name of a previously declared function. This is the function that will be executed upon evaluation. The function name is always followed by a set of parentheses. If any Parameters were written into the Function Declaration, then a corresponding Argument List should be included within these parentheses to provide the function with its necessary values for reference or manipulation. The evaluation of a function call will execute any statements in the function body and will result in the value that is returned by the Return Statement written into the statements of the Function Declaration.

Argument List:

```
argument_list = [ expression , { ',' , expression } ]
```

An Argument List is a list of Expressions included in a Function Call (See **Function Call**) which corresponds to the Parameter List of a Function Declaration (See **Function Declaration**). In order for a function to be called in a function call, it must include an Argument (Expression within the Argument List) for each Parameter in the function's Parameter List. If types are specified for any Parameters in the Parameter List, then the corresponding given argument (corresponding meaning holding the same numerical placing in the lists) needs to be of the specified type. Once sent to the function, the values in the Argument List corresponding to the Parameters will be stored in the identifier names of each Parameter, and can be accessed or manipulated using said identifier. An Argument List consists of one or more (generalized) Expressions separated by commas.

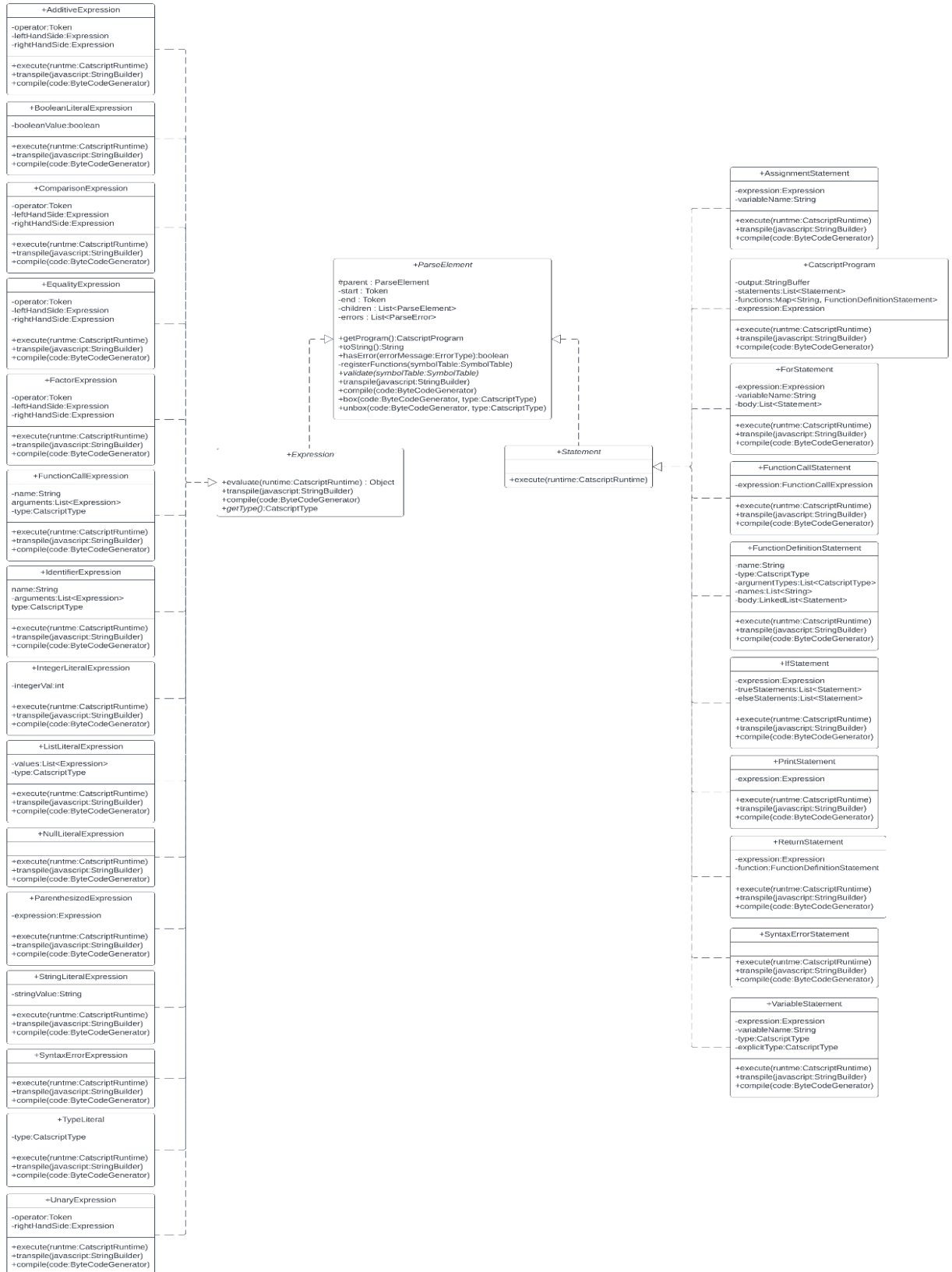
11

Type Expression:

```
type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' ,  
type_expression, '>']
```

Type Expression is an indicator expression used in Variable Statements, Function Declarations, and Parameters to specify that a value needs to be of a specific type. A Type Expression can refer to types such as “int” (indicating the value must be a 32 bit integer), “string” (indicating the value must be a java-style string), “bool” (indicating the value must be a boolean (true/false)), or an “object” (indicating the value can be of any type). Additionally, a Type Expression can specify that a value must be a list of a certain type. If a list is to be specified, the Type Expression takes the form of the “list” type followed by a set of angle brackets containing another Type Expression.

Section 5: UML



Section 6: Design Trade-offs:

There were two major design trade-offs made for this project. The first trade-off was the use of recursive descent instead of the more academic standard parser generator. This decision was made because recursive descent is a more simple alternative to the parser generator. Because of its simplicity, recursive descent can be better for students trying to learn the recursive nature of grammars. Recursive descent is not normal practice when it comes to a compilers course. The standard is a parser generator which was the other option next to recursive descent. When compared with recursive descent, parser generators are more complicated but faster. Significantly more hand written code is required for recursive descent. This brings into question the superiority of simplicity vs. speed. In the end simplicity is better from a learning and understanding perspective where speed is better from a progress perspective. Students want to learn and understanding is a key component of that. The second design trade-off is known as the separation of concerns. In this project there was no separation of concerns. What this means is that several major features which could have been distributed to other classes were all handled in the same class. In this project specifically there were three methods: execute, transpile, and compile. When practicing the separation of concerns these methods would not have been inside of the same class. However, it is the case that they are all used in the same classes in this project. The idea behind this is again, simplicity. By having everything present in when developing it is more clear how the code functions, and what needs to be implemented.

Section 7: Development Life Cycle Model:

The software development life cycle model used when developing this project is known as test driven development. Simply stated, test driven development is the use of numerous test cases which keep track of functionality and development progress. The goal is to have all the test cases pass. A passing test signifies the code works as intended. The total number of passing tests is what is used to track the development progress where the result of a test is used to see if the code does what the developer thinks it does. Test cases are designed in a way that both edge cases and more general cases are covered. Thorough tests lead to an improved and better functioning final product.