Cale Sassano: Capstone Portfolio

Section 1: Progam

Provided source.zip file

Section 2: Teamwork

For teamwork in this project I worked with Joshua Elmore. We each provided documentation and extra testing for each other and did the implementation of the project ourselves on our own. Joshua provided me with the documentation for my project as I provided him with the documentation for his project. We also each wrote 3 more tests to further test each other's compilers.

Section 3: Design Pattern

I implemented a memoization design pattern on this getListType function. Memoization is a design pattern used to improve the performance of computationally expensive functions by caching their results. It stores the results of expensive function calls and returns the cached result when the same inputs occur again, instead of re-computing them. Here the function checks if the type has been cached already, returns the cached result if it has or computes the ListType, caches it, and returns it if it hasn't been cached yet. This is useful to eliminate repetitive computations.

```
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
5 usages  Carson Gross *
public static CatscriptType getListType(CatscriptType type) {
    if (cache.containsKey(type)) {
        // If cached, return the cached result
        return cache.get(type);
    } else {
        // If not cached, compute the ListType
        ListType listType = new ListType(type);
        // Cache the result for future use
        cache.put(type, listType);
        return listType;
    }
}
```

Section 4: Technical Writing

Catscript: Programing Programming Language

Variables and Data Types

String
```
var x = "Hello World!"
var i : int = 1
var n = null
var x : list<int> = [1, 2, 3]
```
In CatScript, you can declare a string variable using the `var` keyword. Strings are enclosed in double quotes. Optionally you can use `var : {type}` to declare the type of a var.

Output

Print Statement
```
print(x)
print("Hello World!")
```
The `print` statement writes the object you pass it to the console like `x`.

If Statements

If Statement
```
if (x > 10) {
   print(x)
}
```
The If statement takes a boolean expression if it returns `true` the body of the statement is executed. If it returns `false` the body is skipped.

If-Else Statement
```
if (x > 10) {
   print(x)
}else{
   print(x+5)
}
```
The If-Else Statement is the same as If but if the expression returns `false` the else statements are executed.

Collections

Lists

```
var list = [1, 2, 3]
var 2Dlist = [[1,2],[3,4]]
```
Lists contain an ordered number of primitive data types they can also be nested.

Functions

Functions
```
function foo(x: int, y: String) {
   print(x + y)
}

foo(9, "Nine")
```
Output
```
9Nine
```
The `foo` function accepts an integer `x` and a string `y`. It then prints the value of `x` concatenated with `y`.

For Loops

For Loop
```
for(x in [1, 2, 3]) {
   print(x)
}
```
The For loop takes an Identifier and a List to iterate over

Expressions

Equality and Comparison Expression
```
5 == 7  #false
7 >= 7  #true
6 != 6  #false
```
These expressions take 2 numbers and perform the following operation and resolve to a boolean value
```
< , > , == , >= , <= , !=
```

```

### Additive and Factor Expression
```
5 * 7   #35
7 / 7   #1
6 + 6   #12
1 / 3   #0

"Cat" + "Dog"   #"CatDog"
```
These expressions take 2 numbers and perform arithmetic operations and resolve to a number. Since Catscript only supports Integers all these operations are integers arithmetic so you will run into strange things like `1/3 == 0`. The Additive can also be used as a string concatenation operator
```
+ , - , * , /
```
### Unary Expression
```
-(3+7)   #-10
not true   #false

```
These expresion take either a number or a boolean and resolve to the opposite. In the case of a number it multiplies it by -1 to reverse the sign. In the case of a boolean it inverts it aka true becomes false and false becomes true.

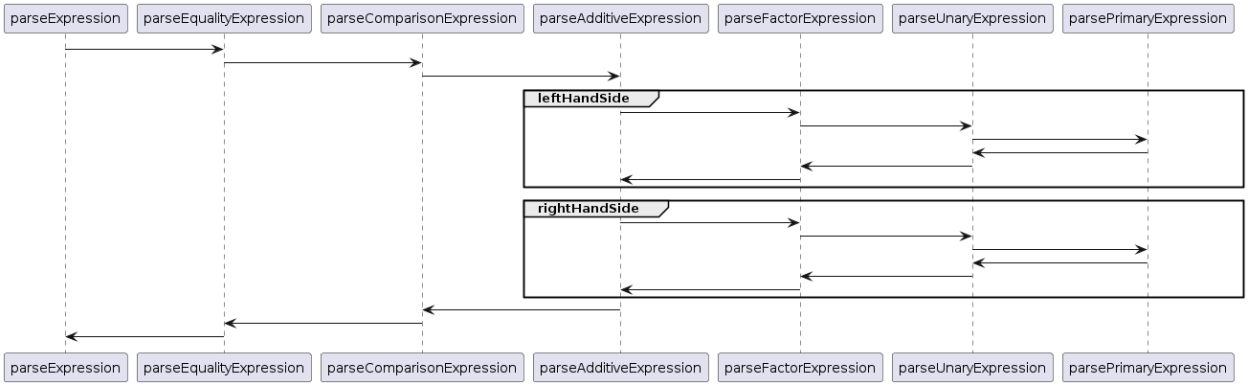
### Function Call Expression
```
addOne(1) + 6   #13
(not returnFalse()) == true   #true

```
Since function calls are implemented as expressions you can have on inline with other expressions

---
## Literal Types
```
true
1
null
[1,2,3]
[true,false,false]
"Hello World!"
```
```

```
You can also declare literal types of the following types with the above
notation these resolve to their natural types
```
Int, String, Bool, Null
```

Section 5: UML




Here is a sequence diagram for parsing an additive expression in Catscript, specifically the expression ‘1 + 1’. It starts with parseExpression and calls a few different functions until it gets to the parseAdditiveExpression function. This function makes a leftHandSide that calls parseFactorExpression which then goes through a few functions until it reaches parsePrimaryExpression where it can parse the integer. Then it makes a rightHandSide which follows the same path to parse its integer. This is a great visual representation of our recursive descent algorithm in action.

Section 6: Design Trade-offs

Looking at the design trade-offs between using recursive descent parsing compared to parser generators, there are a few different elements to look into. Recursive descent parsers give more control and flexibility to the parsing process as they are handcrafted. This makes them easier to maintain and debug because they directly follow the grammar rules for the language being parsed. However, they can be more work to develop as you get to more complex grammars. On the other hand, parser generators use automation for most of the parsing process which can allow for developers to specify grammars using high-level notations. This can make parser generators more effective for more complex grammars but this makes them harder to understand and to modify the parsing logic. Considering catscript uses a more simple grammar we implemented recursive descent parsing in our compiler.
```

Section 7: Software Development Life Cycle Model

We used test driven development in this compiler project. Test driven development uses tests that are written before writing the actual code. These failing tests test the desired behavior of a specific function or feature in the program, the developer then uses these tests to write the necessary code to implement these tests. I felt that this was a great way to learn real world application of the material we learn in this class. It also helped in learning how to debug when writing code. Overall I enjoyed TDD and look forward to seeing this method in my future career.