

CatScript

Compilers CSCI468

Spring 2022

Jesse Russell

Philip Gales (Tester)

Section 2: Teamwork

We were all required to team up with a partner in the class. They wrote the documentation for CatScript and 3 tests for our parser. We did the same for them. This way, the work of writing the parser was done individually while other jobs were handled by a peer. This is similar to the interaction in the workplace where different jobs are done by different people.

Section 3: Design Pattern

We are using the Flyweight pattern in CatScript for the CatscriptType.ListType class (which in this case would be the Flyweight). The goal of the flywheel pattern is to save memory by caching object creation. In the case of the CatscriptType class, which represents a type like int or string, there only ever needs to be one instance of each type; however, we can't simply use an enum because there is an infinite number of list types. This is because the list type takes a type parameter which can also be a list with its own type parameter. This allows the list type to be arbitrarily recursive (`List<List<List<...>>>`).

In order to prevent duplicate list types, the Flyweight pattern can be used to cache list type creation. Caching is simple (cache invalidation is hard, but not needed here). A HashMap is used to store cached list types, mapped against the types given to they're type parameters.

```
private static final Map<CatscriptType, CatscriptType.ListType> listTypeCache
= new HashMap<>();
```

The creation of new ListType instances is delegated to a static method which checks the cache before creating a new instance and if the cache does not contain the type needed, a new one is instantiated, cached, and returned.

```
public static CatscriptType getListType(CatscriptType componentType) {
    // check cache
    final var fromCache = listTypeCache.get(componentType);
    if (fromCache != null) return fromCache;

    // cache miss...
    // create new list type and cache it.
    final var newListType = new ListType(componentType);
    listTypeCache.put(componentType, newListType);

    return newListType;
}
```

Section 4: Technical Writing

Catscript Guide

Catscript is a simple language without the nuance of a more advanced language used for making learning how to make a compiler simple

Features

Expressions

Returns an equality expression. The equality expression can then return a comparison expression which can return an additive expression and so on.

Equality Expression

Either returns a comparison expression or compares two comparison expressions to see if they are equal or not equal.

Comparison Expression

Either returns an additive expression or compares two additive expressions to see if they are

- Less than
- Less than or equal to
- Greater than
- Greater than or equal to

Additive Expression

Adds or subtracts two factor expressions. If either factor expression is a string then the result will be a string. Can also return a single factor expression.

Examples include

```
num = 3 + 5
```

Factor Expression

Can either multiply/divide two unary expressions or return a single unary expression.

Unary Expression

Will apply a not or negative to another unary expression or will return a primary expression

Primary Expression

Returns one of the following

- Identifier

- String
- Integer
- "true"
- "false"
- "null"
- List literal
- Function call
- (expression)

List Literal

Returns an array of primary expressions

Statements

One of the following

- For statement
- If statement
- Print statement
- Variable statement
- Assignment statement
- Function call statement

For Statement

A for in loop similar to one in javascript.

```
for(IDENTIFIER in expression) {
    statement
}
```

If Statement

Can be an if, if else, or else. The expression must evaluate as true or false and the body contain a statement.

```
if( expression ) {
    statement
} [ else( if_statement | { statement } ) ]
```

Print Statement

Prints the result of an expression

```
print(1)
```

Variable Statement

Assigns the value to the right of the equality to the variable on the left.

```
var color = "green"
```

Assignment Statement

Used for changing the value of a variable. The variable's value is updated to the value on the right of the equality.

```
color = "red"
```

Function Call Statement

Calls a function by its name and passes the required parameters.

```
Identifier([ expression, { expression } ])
```

Examples include

```
int num = sum(5, 7)  
foobar()
```

Function Declaration

Creates a new function with parameters and a body that contains a statement or return statement. Recursion is also supported.

```
function Identifier( Identifier [ : Type expression ] ) {  
    Statement | Return statement  
}
```

Return statement

Used within functions to return an expression.

```
return 10
```

Type Expression

Strongly worded and includes one of the following

- 'int': 32 bit signed whole number
- 'string': Collection of plain text characters
- 'bool': true or false
- 'object': Any type

- 'list': List of values
- 'void': No type

Comments

Catscript supports single line comments by using a double backslash. Comments start with the double backslash and go to the end of the line.

```
// This is a comment
```

Type Inference

Can implicitly determine the type. For example, it will determine the number 42 is an int.

```
var implicit = 7
```

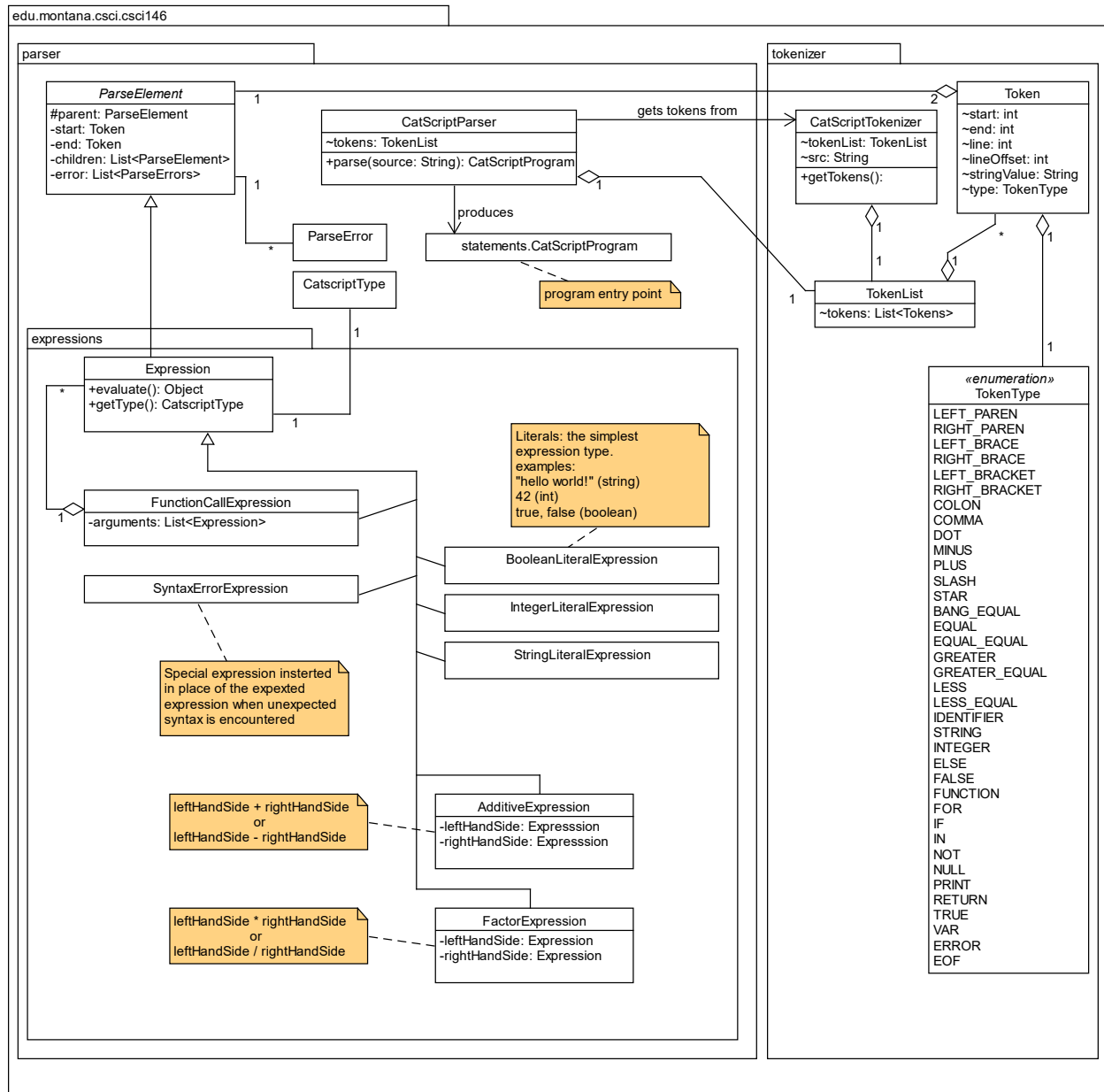
Explicit Types

You can tell Catscript what type your variable is

```
var implicit: int = 7  
  
function x(a : object, b : int, c : bool) {}
```

Section 5: UML

Here is a class diagram representing the parser, tokenizer, and some expressions.



Section 6: Design trade-offs

The visitor pattern is a way to extend a class's functionality externally instead of adding new methods or otherwise modifying the class's code directly. If we had used a parser generator, this pattern would have been the best way to add our evaluation and compilation logic. The visitor pattern would also have preserved the design principle of separation of concerns by breaking our logic into multiple files which could be better organized, and the Open-Close principle by making the parse tree open to extension but closed to modification. These principles of design lend themselves to better organization and maintainability which are both crucial properties for a large code base belonging to a long-term project.

CatScript forgoes the visitor pattern and opts for the simplicity of adding the evaluation and compilation logic directly to the parse tree. We were able to do this because we hand wrote a recursive decent parser instead of using a parser generator, which gave us greater control over the parser and parse tree's functionality and design.

Unfortunately, the large-scale organization and long-term maintainability of our code base may have suffered due to this choice; but CatScript is neither a large-scale, nor long-term project. CatScript is a learning project, and the primary goal of a learning project is to learn from the experience. The primary goal of writing CatScript was to learn about recursive decent parsing and compilation, and the added complexity from the visitor pattern could have distracted from that.

Section 7: Software development life cycle model

We used test driven development, which is a development model in which tests are written first before the code. This way, the tests provide clear and unambiguous requirements while also serving their original purpose of providing a simple way to test the code. This model works well for projects with clear requirements given up front with a reasonable timeframe; Unfortunately, this rarely applies perfectly in industry, where deadlines and requirements are flexible; although, one area where it could fit perfectly is education. In the classroom, requirements are usually fully known up front and the deadlines rarely change.

In our case, the requirements for Catscript were given to us in the form of unit tests, with checkpoints spread out across the semester that required a growing number of tests to pass. Most of the grade for the main project was determined by these checkpoints. This made grading simpler and reduced a significant source of stress for the students, since we could know ahead of time what grade we'd receive. Effort was still required to understand the material and write quality code that works. Additionally, while test driven development is rarely followed strictly, unit tests are still a popular way of ensuring code quality during the maintenance phase of development, so being exposed to unit tests and learning how to write them is a valuable experience for computer science students.

Of course, none of this is to imply that test driven development should not be used outside of the classroom. A development model may not always apply fully to every situation, but it doesn't always have to be fully used either. Some parts of a project might fit test driven development, and some might not. If the requirements of a class or method are clear and immutable, then writing tests first could be a good idea.