Capstone Documentation

Diego Moylan, Brayden Miller

~ CSCI 468: Compilers ~

Section 1: Program

Please see the zipped source code & specification in source.zip

Section 2: Teamwork

Our team split work into implementation (team member 1) and testing/documentation (team member 2). Member 1 contributed most of the working code and member 2 contributed tests, found in `csci468.partner.PartnerTests`, as well as documentation. Time spent was roughly a 70/30 split between team member 1 and 2.

Partner Tests

```
@Test
void stringConcatenationEvaluatesProperly() {
    assertEquals("helloworld", evaluateExpression("\"hello\" + \"world\""));
}

@Test
void stringAndIntConcatenationEvaluatesProperly() {
    assertEquals("value:5", evaluateExpression("\"value:\" + 5"));
}

@Test
void nestedExpressionEvaluatesProperly() {
    assertEquals(14, evaluateExpression("2 + 3 * 4"));
}
```

Section 3: Design Pattern

In our project we implemented the Memoization Pattern in `csci468.parser.CatscriptType.getListType()`.

We did this by caching the instantiated `ListType` into a private class variable the first time `getListType()` is called. On all subsequent method calls, we check if the cached variable is already populated and return it instead of instantiating a new object.

This improves the compiler significantly in a few ways. First, it optimizes memory usage and avoids unnecessary garbage collection overhead by not rapidly creating duplicate type objects every time a list is parsed or validated. Second, it guarantees that only one instance of a specific list type ever exists (so only one `list<int>` object put into memory). This ensures strict referential integrity, meaning type-checking systems can safely and efficiently compare list types using reference equality (`==`) rather than expensive and deeper `.equals()` lookups.

Section 4: Technical Writing/Catscript Guide

1. Program Structure

A Catscript program consists of either:

- A single expression
- A sequence of statements and function declarations

Execution proceeds from top to bottom

---

2. Types

Catscript is statically typed, meaning types are checked at compile time.

Supported types:

- `int` — 32-bit integer
- `string` — text values
- `bool` — `true` or `false`
- `list<T>` — list of elements of type `T`
- `object` — general-purpose type (can hold any value)
- `null` — absence of a value

Example:

```
var x = 10
var name: string = "cat"
var flags: list<bool> = [true, false, true]
```

Type annotations are optional in most implementations but are part of the grammar and should be used when possible.

---

3. Variables

Variables are declared using `var`:

```
var x: int = 5
var y = 10
```

Variables can be reassigned:

```
x = x + 1
```

List indexing is supported:

```
var nums = [1,2,3]
nums[0] = 10
```

---

4. Expressions

Catscript supports standard expressions with clear and explicit precedence:

- Unary: `-x`, `not x`
- Multiplicative: `*`, `/`, `%`
- Additive: `+`, `-`
- Comparison: `<`, `>`, `<=`, `>=`
- Equality: `==`, `!=`

Example:

```
var result = (5 + 3) * 2
var check = result > 10
```

---

5. Control Flow

If Statements

```
if (x > 0) {
    print("positive")
} else {
    print("non-positive")
}
```

Supports `else if` chains:

```
if (x == 0) {
    print("zero")
} else if (x > 0) {
    print("positive")
} else {
    print("negative")
}
```

While loops

```
var i = 0
while (i < 5) {
    print(i)
    i = i + 1
}
```

For Loops

Used to iterate over lists:

```
var nums = [1,2,3]
for (n in nums) {
    print(n)
}
```

---

6. Functions

Functions are first-class language constructs.

Declaration

```
function add(a: int, b:int): int {
    return a + b
}
```

Calling Functions

```
var result = add(3,4)
print(result)
```

Void Return

```
function greet(name: string) {
    print("Hello " + name)
}
```

7.Lists

Lists are typed collections:

```
var numbers: list<int> = [1,2,3]
```

Access elements using indexing:

```
print(numbers(0))
```

Lists can be passed into functions and iterated over.

---

8. Printing

Output is produced using:

```
print(expression)
```

Example:

```
print("Hello World")
print(42)
```

---

9. Return Statements

Functions may return values:

```
return expression
```

Or return nothing:

```
return
```

---

10. Complete Example Program

This program demonstrates variables, functions, loops, conditionals, and lists.

```
function sumList(values: list<int>): int {
    var total: int = 0
    for (v in values) {
        total = total + v
    }
    return total
}

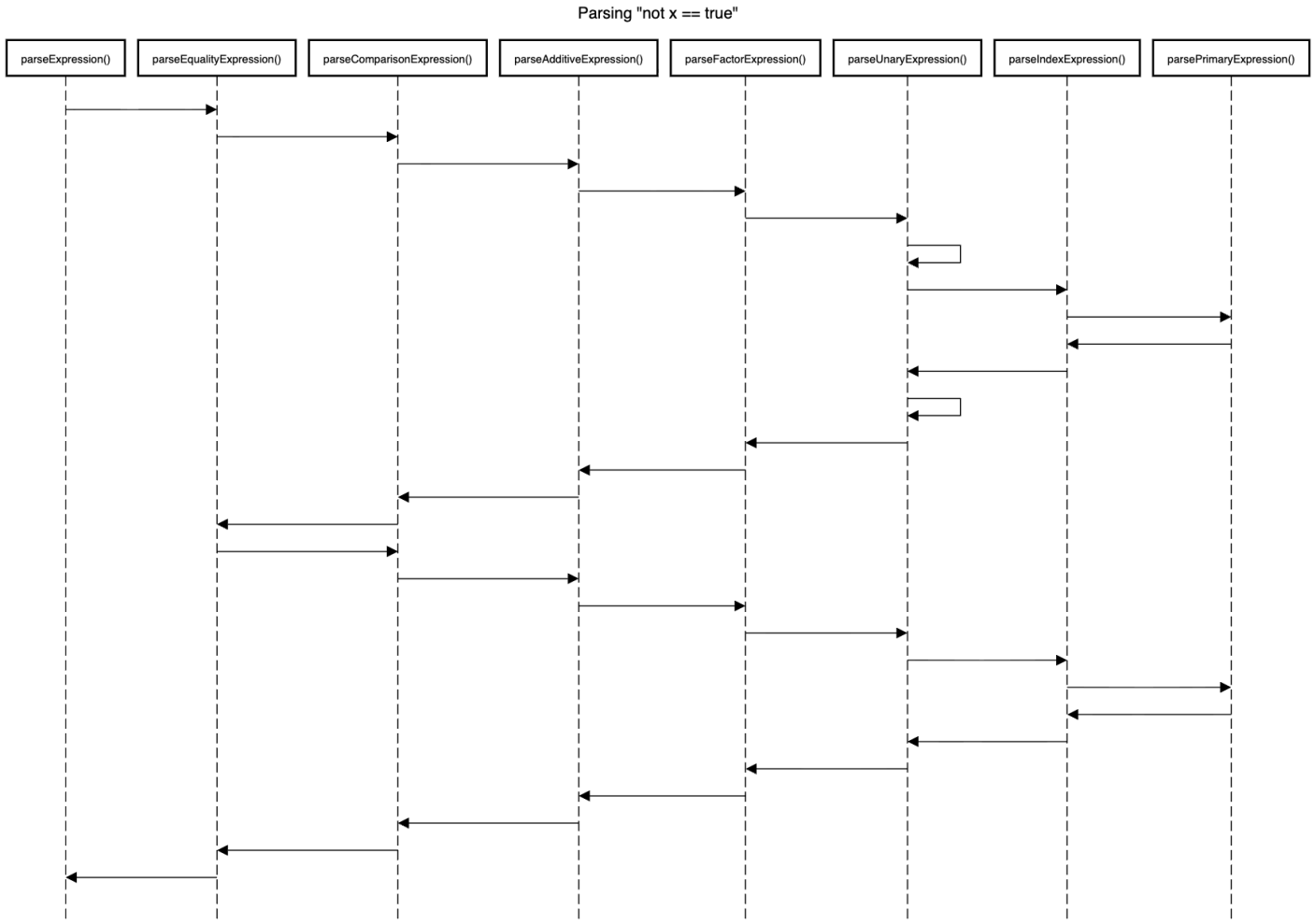
function main() {
    var numbers: list<int> = [1,2,3,4,5]
    var total = sumList(numbers)
    print("Total:")
    print(total)

    if (total > 10) {
        print("Large sum")
    } else {
        print("Small sum")
    }
}

main()
```

Section 5: UML

Here is an example diagram of the expression parsing logic:



And here is the code from https://sequencediagram.org.

```
parseExpression()->parseEqualityExpression():
parseEqualityExpression()->parseComparisonExpression():
parseComparisonExpression()->parseAdditiveExpression():
parseAdditiveExpression()->parseFactorExpression():
parseFactorExpression()->parseUnaryExpression():
parseUnaryExpression()->parseUnaryExpression():
parseUnaryExpression()->parseIndexExpression():
parseIndexExpression()->parsePrimaryExpression():

parseIndexExpression()<-parsePrimaryExpression():
parseUnaryExpression()<-parseIndexExpression():
parseUnaryExpression()<-parseUnaryExpression():
parseFactorExpression()<-parseUnaryExpression():
parseAdditiveExpression()<-parseFactorExpression():
parseComparisonExpression()<-parseAdditiveExpression():
parseEqualityExpression()<-parseComparisonExpression():

parseEqualityExpression()->parseComparisonExpression():
parseComparisonExpression()->parseAdditiveExpression():
```

```
parseAdditiveExpression()->parseFactorExpression():
parseFactorExpression()->parseUnaryExpression():
parseUnaryExpression()->parseIndexExpression():
parseIndexExpression()->parsePrimaryExpression():

parseIndexExpression()<-parsePrimaryExpression():
parseUnaryExpression()<-parseIndexExpression():
parseFactorExpression()<-parseUnaryExpression():
parseAdditiveExpression()<-parseFactorExpression():
parseComparisonExpression()<-parseAdditiveExpression():
parseEqualityExpression()<-parseComparisonExpression():

parseExpression()<-parseEqualityExpression():
```

Section 6: Design Trade-offs

We used a handwritten recursive descent parser in this class, which has certain tradeoffs compared to a parser generator. Generally, a parser generator is used when you want quicker development times and a parser that will definitely work if the grammar provided is correct. However, they have generally unhelpful error messages if you get something wrong, and debugging is kind of a nightmare. Handwritten parsers can have error messages crafted with inherent knowledge of the domain you are working in, and don't just depend on the grammar you put in. The code for the parser isn't generated, so it's generally much easier to read and debug than the alternative. However, it is much more work, and writing the code means there is room to introduce small bugs that are difficult to notice or only affect edge cases. Also, if you change the grammar you have to change the code again.

Section 7: Software Development Life Cycle Model

We used Test Driven Development as our life cycle model. Having tests defining our goals for the functionality of the system made it very easy to have a direction to move in. If you have enough tests with high enough quality, all you need to do is make sure all the tests pass, and you'll have a functioning system. A clear tradeoff with this is that it makes changing the intended logic of the system difficult, because it breaks a bunch of tests, which might need to be removed or rewritten to agree with the new design.