Compilers

CSCI 468

Spring 2024

Benjamin Huotari

Rory Schillo

Section 1: Program
Zip file of the final repository submitted.

Section 2: Teamwork
My teammate contributed 3 tests and the documentation. I was responsible for the implementation.
Tests are in ‘Demo\PartnerTest.java’
Documentation is in ‘Catscript.md’

Section 3: Design pattern
Here is how the memoization design pattern was implemented:

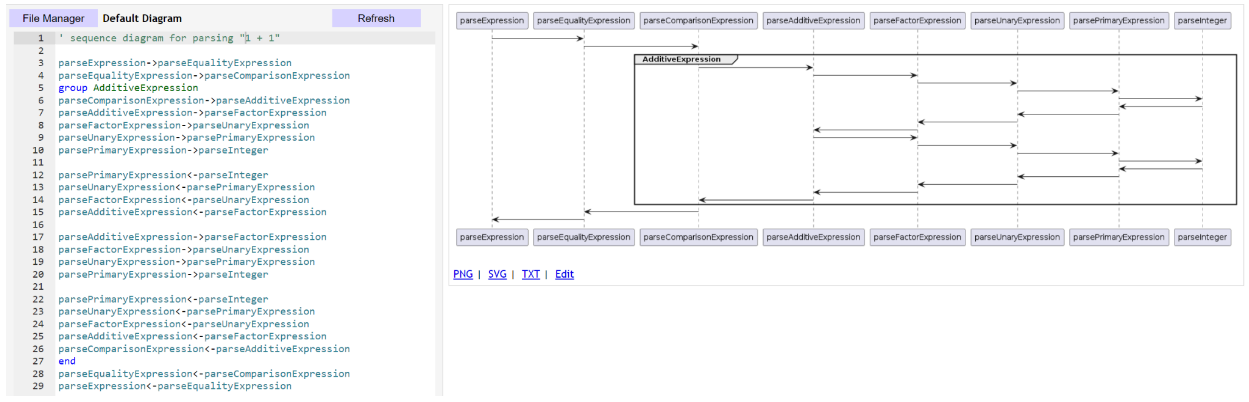
```
// memoize this call
static HashMap<CatscriptType, ListType> cache = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    // look in the map
    if (cache.containsKey(type)) {
        // if the thing is already there, return it
        return cache.get(type);
    }
    // else new something up
    ListType listType = new ListType(type);
    // add the new list type to the map
    cache.put(type, listType);
    return listType;
}
```

First we create a HashMap ‘cache’ that will be used to store each unique type we receive.
When we pass in a type we check if it is already in the map.
If we find the corresponding type already cached, we just return it.
Otherwise, we create a new ListType variable with this type and put it in the cache.
We then return the new ListType variable.

By using this design pattern, we only have to create a new ListType object when a new type is received while repeated types will already be cached and get returned much faster. This improves performance when multiple getListType calls are made with the same type by reducing the amount of unnecessary computations.

Section 4: Technical writing.
Completed in the teamwork section.

Section 5: UML.



```
' sequence diagram for parsing "1 + 1"

parseExpression->parseEqualityExpression
parseEqualityExpression->parseComparisonExpression
group AdditiveExpression
parseComparisonExpression->parseAdditiveExpression
parseAdditiveExpression->parseFactorExpression
parseFactorExpression->parseUnaryExpression
parseUnaryExpression->parsePrimaryExpression
parsePrimaryExpression->parseInteger

parsePrimaryExpression<-parseInteger
parseUnaryExpression<-parsePrimaryExpression
parseFactorExpression<-parseUnaryExpression
parseAdditiveExpression<-parseFactorExpression

parseAdditiveExpression->parseFactorExpression
parseFactorExpression->parseUnaryExpression
parseUnaryExpression->parsePrimaryExpression
parsePrimaryExpression->parseInteger

parsePrimaryExpression<-parseInteger
parseUnaryExpression<-parsePrimaryExpression
parseFactorExpression<-parseUnaryExpression
parseAdditiveExpression<-parseFactorExpression
parseComparisonExpression<-parseAdditiveExpression
end
parseEqualityExpression<-parseComparisonExpression
parseExpression<-parseEqualityExpression
```

Section 6: Design trade-offs

We used recursive descent parsing instead of implementing a parser generator. Although parser generators can be faster in some cases, because our grammar is not overly complex, our recursive descent parser is fast enough to be a practical choice. By using recursive descent, we can also mirror the recursive nature of our grammar in a much simpler and more intuitive way.

Section 7: Software development life cycle model

We used Test Driven Development (TDD) for this project. This meant we had clear requirements that needed to be met that were given to us in the form of tests. I found this to be a very useful strategy for multiple reasons. First, the tests make it extremely easy to gauge one's progress and to know which sections of code still need to be implemented correctly. It gives you clear feedback on which features are being fixed or broken when changes are made and is extremely useful for tracking down bugs.