Creating a Grammar and Parser in ANTLR

Joe Faulls - 1447637

1 My Experience

From the first read through of the assignment, I predicted it to be 'maybe 2 day job'. What followed was 5 intense days of writing grammars, rewriting grammars, fighting with ANTLR and manically Googeling for any examples. Admittedly, it was very fun to be designing, writing code and seeing a working product at the end. I can confidently say I have learnt a lot through the assignment. This report will explain the grammar for my language 'Verbose2' and how I went about implementing it, along with challenges encountered along the way.

2 Designing the Grammar

As with designing any programming language, the first step is to design a suitable grammar that allows for all the functionality desired. The challenge here is to design the grammar in such a way that the language is as generic as possible to allow maximum extensibility, whilst still being constrained to correctly and uniquely identifying every expression. Therefore, having every expression being parsed as type '`expression`' will be very general but does not help us into parsing the grammar.

It is also imperative that the grammar must not be *ambiguous*. This means that there cannot be more than one unique parse tree for any given input string.

ANTLR has a sophisticated grammar development tool called *ANTLRWorks* to work with ANTLR version 3. The tool provides useful functionality such as syntax highlighting, grammar checks, parse tree generation and a debugger. This program proved to be very handy for designing grammars as it is quick to pick up if the grammar is in any way non-deterministic. Probably one of the most useful features is the debugger, which allows for step-by-step tree generation for any given input string. This enabled me to detect where and how the interpreter interpreted the input in an undesired way, and subsequently how to fix my grammar.

2.1 Lexer and Parsers

ANTLR is very useful in the fact it takes a grammar file as an input and produces two files (or classes): a lexer and parser. The lexer takes a character stream as

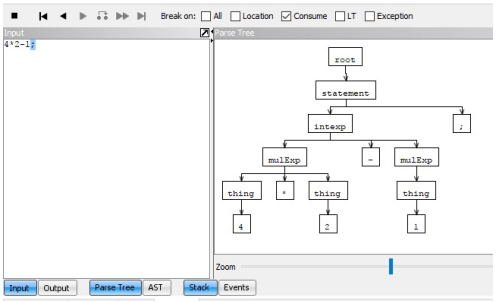

Figure 1: ANTLRWorks debug panel allowing for step-by-step evaluation

input and splits it up into tokens. The tokens are defined in the lexer rules, each rule being it's own token and identified with a regular expression. The parser then uses these tokens as building blocks, translating the token stream into a different structure.

Verbose2 has four tokens in total: numbers, characters, comments and whitespace. The only two tokens that are used in the grammar are numbers and characters. Comments and whitespace are defined so the parser can easily skip over and ignore them.

```
NUMBER  : '0'..'9'+ ;
CHARS   : ('a'..'z'| 'A'..'Z')+ ;
COMMENT :    '//' ~('\n'|'\r')* '\r'? '\n' -> skip ;
WS : [ \t\r\n]+ -> skip ;
```

Figure 2: All tokens in which the parser is built on

2.2 First Attempt

There is a reason the final grammar was named 'Verbose2': the first attempt was a trainwreck. It was very easy to dive into the assignment and start working on the grammar straight away. There were two main reasons the first attempt went badly: firstly, I was clueless as to how I would implement the parser, both how ANTLR allowed it and how it would work semantically. Therefore, when I came around to implementing the parser, it was extremely tricky given my messy grammar. Secondly, I took the approach of implementing features one at a time without regard for how future features would fit in (in other words, it was not very generic).

Implementing integer expressions is a a well established problem and often the first grammar rule people come to implement. Therefore, many online resources were available demonstrating how it was done. In my case (and many

other online examples), all integer expressions are evaluated as addition expressions, where the members are multiplication expressions plus or minus any number of additional multiplication expressions. These multiplication expressions are then evaluated as numbers times any number of numbers. This is best explained below:

```
intExp : mulExp ('+' mulExp | '-' mulExp)* ;
mulExp : number ('*' number )* ;
```

This not only allows for interpretation of long input string such as '`3*1+2-5*1`, but can ensure that it can be evaluated correctly according to the order of operation. Here, mulExp will always be evaluated before addition or subtraction. Furthermore, it allows for single numbers to be parsed without any errors.

When trying to improve the grammar so it was more generic, I found myself encountering the following error: 'The following sets of rules are mutually left-recursive'. This is a pit-fall for the LL parser. ANTLR3 cannot deal with left-recursion within the same rule or in the case where one rule references another and vice verser - the parser will throw this error. This was a big indicator that my grammar needed a serious reconsideration, and was the final straw to start again.

2.3 Second Attempt

Identifying what worked in the previous grammar and why it worked so well was a great starting point. Looking at the success of integer expressions, I recognised how it cascaded downwards, checking if it was an addition expression, then checking for multiplication - and if not that's fine. This cascading style was the foundation of how I designed the new grammar. All rules eventually cascaded down the smallest possible form, often called the atom in most grammars (named `primary` in mine), which was either a number or a word.

Before, I would attempt to fork off a rule to incorporate a new rule. This time, I would take an input string, check if satisfies one rule, then move on to the next in a chain. The disadvantage to this is how many rules it would have to check before it reached the base case.

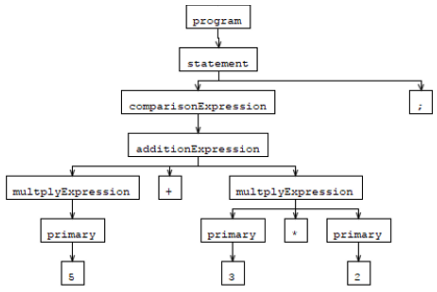

Figure 3: The interpretation of just a simple string '5+3*2;'

3 Implementing the Parser

With a grammar able to correctly parse different inputs, the next step is to bring the grammar to life by defining specific actions the parser should take when it identifies certain rules. In ANTLR3, this is done in line in Java (see figure 4).

```
intexp returns [int value]
        : t=mulExp {$value = $t.value;}(
          '+' t=mulExp {$value += $t.value;}
        | '-' t=mulExp {$value += $t.value;}
        )*
        ;
```

Figure 4: In-line specification of behaviour

After playing around with this for a while, I was able to successfully evaluate mathematical expressions. However, trouble struck when I tried to implement conditionals and functions.

3.1 Moving to ANTLR4

Upon research into how I would implement conditional statements and functions, I found it would be very tricky in ANTLR3. However, the newest version of ANTLR had it's development mentality switched from performance to ease-of-use. This came with an array of very useful features making conditionals far simpler. The two biggest changes was less fussy grammar (many cases of left recursion would be accepted and secretly changed), and the addition of *visitors*.

Before, defining behaviour depended on embedded code in the grammar. In ANTLR4, we can generate visitors, an object that responds to rule entry events triggered by a parse-tree walker as it traverses nodes. This means we can define behaviour upon visits to nodes. Most notably, we define what to traverse. By default, it would be all a node's children. Having a separate file for defining behaviour also allows us to write it in any supported language rather than the default Java.

In the main visitor class, you must specify the return type for all the rules. In the case of this exercise, it could be sufficient to just use the primitive type `int`, with 1 and 0 being used in the place of boolean values. However, for extensibility purposes, I made all rules to return the placeholder class `Value`. This class can store any object, and provides utilities to check and return the value as different types (for example booleans, strings or integers).

3.2 Simplification of the Grammar

The biggest difference between ANTLR3 and 4 is that ANTLR4 accepts grammars with direct left-recursion (i.e. in the same rule). This allowed for sim-

plification of my grammar so that it was no longer a single chain of cascading rules.

```
mathExpression  : left=mathExpression op=('*'|'/') right=mathExpression #opExpr
        | left=mathExpression op=('+'|'-') right=mathExpression         #opExpr
        | '(' expr=mathExpression ')'                                   #parenExpr
        | name=CHARS '(' exprList=expressionList? ')'                   #functionCall
        | primary                                                       #primaryExpr
        ;
```

Figure 5: With this simplified grammar, I could encapsulate multiple rules into one whilst maintaining correct order of application. (note the tags such as 'left=' allowed for easy access in the visitor class)

3.3 Assignment

Although it wasn't required to implement assignment in the exercise, the method of doing so was similar to what must implemented to allow functions to work. This is because words are used in function definitions as placeholders for the parameters, so my grammar must accept words in place of numbers to start with. All that must be implemented is the assignment grammar rules and a method of storing values in the visitor class. The latter of this was easily accomplished using a hash map.

Upon assignment, the key (name of the variable) was stored with the corresponding value. When the parser comes to evaluate `primary`, if it is a word, if looks up the value in the hash table.

3.4 Functions

The last and by far the most difficult feature to implement was functions. It involves more than one walk of the parse tree, saving function definitions and calling multiple visitors with different scopes.

3.4.1 Function Object

To store a function, I created a Function class. This class has three parameters and one method. The parameters stored are:

- `String name` - Name of the function.
- `List<TerminalNode> params` - List of parameters
- `ParseTree block` - Function code block

The only method this class has is `eval`. This takes a list of parameters, and two hashmaps of functions and memory for scope purposes. The method then evaluates the parameters and stores them in a new, updated memory. A

new main visitor is created and called to visit the Function block with the new memory. The method then returns the value that the visitor returns.

3.4.2 Function Visitor

My language allows for function definitions to be placed anywhere in the program, before or after the program return statement. I accomplished this by creating another visitor called `FunctionVisitor`. This visitor only has defined behaviour for function definitions. The main program visitor, on the other hand, is set to ignore all function definitions.

When the Function Visitor identifies a function declaration, it acquires the name of the function, all the parameters in a list form, and the block as a ParseTree object. It then creates a function object using these and places it in the functions hash table.

On the first walk-through by the Function Visitor, filling the functions hash table, the table is then passed through to the main visitor so it can call the relevant function when necessary.

3.4.3 Function Call

When the main visitor identifies a function call, it looks up the function name in the hash table. If it exists, the respective Function object's eval method is called with the parameters supplied.

4 Verbose2 Grammar

Here I will explain the relatively short grammar of the language, beginning from the start rules `program` and working down until the smallest possible rule `primary`.

```
program
: functionDef* retBlock functionDef*;
```

This is the start rule consisting of any number of function definitions with a retBlock somewhere

```
retBlock
: block returnStatement;
```

A retBlock is a block of statements ending with a return statement.

```
functionDef
: 'MY FUNCTION CALLED ' CHARS 'IT TAKES THESE PARAMETERS(' params? ')'
 '{' retBlock'}';
```

It may seem weird that retBlock is separated into it's own rule, this is because it makes it far easier from a parsing point of view to execute a visit on a retBlock when the function is called, rather than walking the block and then walking the returnStatement.

```
returnStatement
: 'RETURN ' comparisonExpression ';';
```

Simply the word 'RETURN' followed by an expression that has a value. The reason this is a comparisonExpression, is that that is the start of the 'cascade' for all expressions that evaluate to a single value.

```
block
: statement*;
```

A block is simply any number of statements.

```
params
: (CHARS',')* CHARS;
```

This is just a simple list of names separated by a comma.

```
statement
: comparisonExpression ';'
| 'IF IT HOLDS THAT' '(' comparisonExpression ')' '{' block '}'
  ('OTHERWISE {' block '}')?
| 'MY VARIABLE CALLED' CHARS 'WILL BIND TO THE FOLLOWING VALUE'
  mathExpression ';';
```

This is the bread and butter for every line. In assignment, the variable is bound to the result of the `mathExpression`. Note that this isn't a `comparisonExpression` because a `comparisonExpression` could result in a boolean value. Variables in this language are restricted to numbers. Similarly to what was discussed in `returnStatement`, `mathExpression` is the start of the cascade for all expressions that evaluate to a single number.

```
comparisonExpression
: '(' comparisonExpression ')'
| mathExpression ('<' | '>' | '>=' | '<=' | '==' | '!=') mathExpression
| comparisonExpression ('||' | '&&') comparisonExpression
| '!' comparisonExpression
| mathExpression;
```

ANTLR4 thankfully allows all of these statements to be caught in a single rule without any risk of left-recursion. The first four cases are boolean expressions, but if the input is none of these, it cascades down into a mathExpression.

```
mathExpression   :
 mathExpression ('*'|'/') mathExpression
| mathExpression ('+'|'-') mathExpression
| '(' mathExpression ')'
| CHARS '(' expressionList? ')'
| primary;
```

Similarly, this rule catches all mathematical expressions. The order in which the rules are listed ensures correct order of application of mathematical operations. The fourth rule is function application. Again, if the input is none of these, it cascades down to a primary rule.

```
expressionList
: comparisonExpression (',' comparisonExpression)*;
```

This is the same as `params`, but more versatile as we can have expressions. This is intuitive, we don't want a user to be able to put mathematical expressions in function definitions, as this makes no sense. However, it is useful to be able to put them in function calls.

```
primary
: ('TRUE' | 'FALSE')
| CHARS
| NUMBER;
```

This is the program's atom. All expressions eventually cascade down to being a boolean, characters or a number.

5 Examples

Here are a few examples of programs written in Verbose2 and their respective parse trees.

5.1 Fibonacci

```
MY FUNCTION CALLED fibonaci TAKES THESE PARAMETERS(x) {
        IF IT HOLDS THAT (x < 1) {
                MY VARIABLE CALLED y WILL BIND TO THE FOLLOWING VALUE 1;
        } OTHERWISE {
                MY VARIABLE CALLED y WILL BIND TO THE FOLLOWING VALUE x + fibonaci(x-1);
        }
        RETURN y;
}

RETURN fibonaci(5);
^Z
16
```

Figure 6: Return Value seen at the bottom (16)

5.2 Factorial

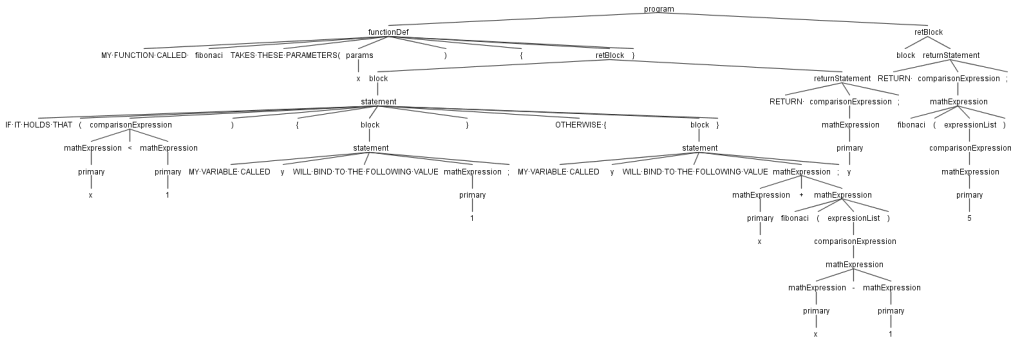

Figure 7: The parse tree for this input

```
RETURN factorial(10);

MY FUNCTION CALLED factorial TAKES THESE PARAMETERS(x){
        IF IT HOLDS THAT(x > 1) {
                MY VARIABLE CALLED y WILL BIND TO THE FOLLOWING VALUE x * factorial(x-1);
        } OTHERWISE {
                MY VARIABLE CALLED y WILL BIND TO THE FOLLOWING VALUE x;
        }
        RETURN y;
}
^Z
3628800
```

Figure 8: Note the return statement is above the function declaration

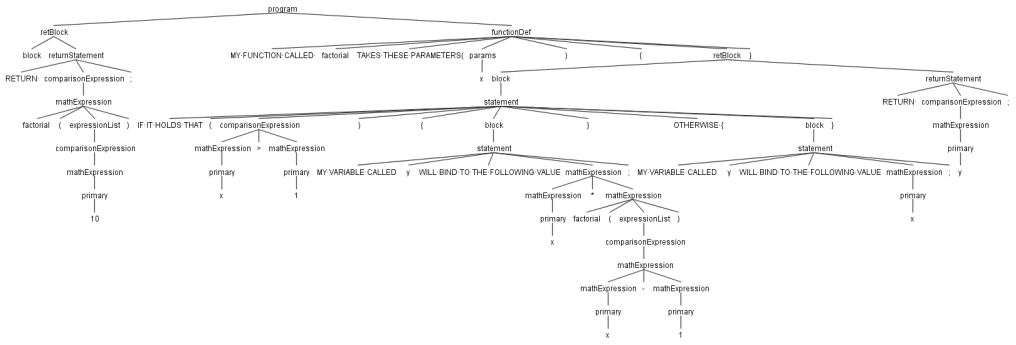

Figure 9: The parse tree for this input