MALBOGLE: Designing an Interpreter for an Imaginary Programming Language

Victor Gong

Harker School

ATCS: Compilers

Introduction

This paper explores the structure and implementation of an interpreter for the imaginary language, MALBOGLE, which contains highly limited grammar compared to other conventional languages. Firstly, MALBOGLE only has four types of statements: write, assignment, while loops, and if statements. Additionally, in the place of conditionals exists “Expressions” (not to be confused with AddExpr, which represents classic expressions). These Expressions consist of AddExpr’s chained with “relop” operators (<, >, >=, <=, <>, =) but can also take on the form of single AddExpr, effectively combining standard conditionals and standard expressions.

Grammar Transformation

Although most of the MALBOGLE grammar does not contain any left factoring issues, there still exists one definition that must be altered:

```
Expression -> Expression relop AddExpr | AddExpr
```

Because Expression is on the left side of the grammar, this definition will lead to infinite recursion as the leftmost Expression continues expanding infinitely. Thus, let’s redefine:

```
ChainConditional -> AddExpr Expression
Expression -> relop Expression | AddExpr | ε
```

Now, we can check for the “relop” character in order to parse the newly-defined Expression, with ChainConditional replacing the old Expression.

Parsing: Program

Compared to our previous PASCAL interpreter, a MALBOGLE interpreter will have a slightly different Parser. Firstly, a Program in MALBOGLE is defined as a list of statements (e.g. Statement1, Statement2, Statement3…etc) compared to the PASCAL definition of a single program statement. Therefore, the altered parseProgram method will consist of a while loop which parses statements until EOF

```
while (!EOF) {
     stmts.add(parseStatement());
}
```

Then, the execute method of Program should simply execute all of the statements:

```
for (stmt in stmts):
     stmt.execute();
```

Parsing: Statement

In contrast to Program, much of the parsing and classes associated with Statement do not need to change. For example, WRITELN remains exactly the same, evaluating the expression and printing the value. Similarly, the Assignment statement also does not need to be altered; it just maps the variable id to the Expression evaluated value using a Map data structure. WHILE barely changes, with Program simply replacing Statement in the while loop body.

However, MALBOGLE's definition of an if statement slightly differs from the PASCAL construction. In our PASCAL interpreter, If and IfElse resided within two separate classes, but in MALBOGLE, If statements are defined as:

```
if Expression then Program St2
St2 -> end | else Program end
```

IfElse statements essentially serve as an extended If statement, attaching to the back of the grammar as “St2.” So, in the If section of parseStatement, we must parse the chain condition (old Expression), parse the program in the then-body, and parseSt2 (which can be done quite easily, by following the exact grammar definition).

Thus, we can create a new class called St2, which represents the hanging “else” piece of the if statement (if not just “end”). St2’s execution method will either do nothing (“end”) or execute its associated program (“else Program end”), as follows:

```
if (elseProgram != null) elseProgram.execute()
```

To wrap up If and If-Else statements, the new If execution method will just evaluate its ChainConditiona; if true, it’ll execute the if-body program, and if false, it’ll execute the associated St2.

Parsing: Chain Conditionals

One of the most distinct and unique features of MOLBOGLE is its ability to chain conditionals. Typically, conditionals in mainstream programming languages are binary conditions, meaning that only two expressions can make up a conditional at once. In MOLBOGLE, on the other hand, a conditional such as `cond1 == cond2 < cond3 <= cond4` can be defined, evaluated in a left to right direction.

To parse these ChainConditionals, we can follow our left-recursion-eliminated definition. First, we parse addExpr and then we parse Expression (which essentially consists of a chain of relop followed by AddExpr). Parsing addExpr can be implemented in the exact same fashion as the PASCAL interpreter, because AddExpr perfectly parallels factors from the original interpreter, functioning the same way (left to right; parentheses, negative sign, * /, + -).

Now, we simply have to define a parseChainConditional method (no need for a parseExpression because we can do that inside parseChainConditional). First, we parse the first AddExpr, and then we use a while loop to continuously check if the next token is a relop and adding the operator and subsequent addExpr into respective Lists, as such:

```
List addExprs, relopOps;
addExprs.add(parseAddExpr());
while (nextToken is a relop) {
     relopOps.add(nextToken); advance_token();
     addExpr.add(parseAddExpr());
}
```

Finally, for the ChainConditional evaluate method, we simply instantiate two pointers, one on each of the two lists, and run through the lists with a while loop. We can assume that the MOLBOGLE definition of ChainConditional is that as soon as one set of (addExpr1, relop, addExpr2) evaluates to false, the whole conditional evaluates to false. Therefore, we check each conditional set with the pointers:

```
int i,j;
while (i, i+1, j still in bounds) {
     if (!check_set(exp[i], relop[j], exp[i+1])) {
          return false; //false immediately
     }
}
return true; //all sets passed
```

And, with that, our MOLBOGLE interpreter is complete.

Conclusion

Even though MOLBOGLE does differ in some areas compared to other programming languages and PASCAL, the fundamentals remain the same. Looking beyond the realm of the paper, this demonstration proved just how versatile interpreters can be, as simple changes can result in the processing of completely different programming languages.