**Detailing an Interpreter for the SIMPLE Language**

Hemant Kunda

Honors Adv. Topics in CS: Compilers and Interpreters

**Introduction**

This document serves to outline the process of interpreting a language via a recursive descent approach to create intermediate abstract syntax trees. The design of such a project, from the initial provided grammar to a final product that interprets the code, will be explained in detail.

**Initial Grammar**

The initial grammar provided is listed here:

Program -> Statement P

P -> Program | e

Statement -> **display** Expression St1

| **assign id =** Expression

| **while** Expression **do** Program **end**

| **if** Expression **then** Program St2

St1 -> **read id** | e

St2 -> **end** | **else** Program **end**

Expression -> Expression **relop** AddExpr

| AddExpr

AddExpr -> AddExpr **+** MultExpr

| AddExpr – MultExpr

| MultExpr

MultExpr -> MultExpr \* NegExpr

| MultExpr / NegExpr

| NegExpr

NegExpr -> -Value

| Value

Value -> **id | number |** (Expression)

relop -> {<, >, >=, <=, <>, =}

One main issue exists currently with this grammar: the definitions of Expression, AddExpr, and MultExpr all cannot be broken down further. This is because, with the current definition, the very first line of a method to parse an Expression would immediately attempt to parse an Expression, sending the program into an infinite loop.

To solve this issue, the concept of whileterms shall be introduced. The new grammar becomes:

Program -> Statement Program | e

Statement -> **display** Expression St1

| **assign id =** Expression

| **while** Expression **do** Program **end**

| **if** Expression **then** Program St2

St1 -> **read id** | e

St2 -> **end** | **else** Program **end**

Expression -> AddExpr whileExpression

whileExpression -> **relop** AddExpr | e

AddExpr -> MultExpr whileAddExpr

whileAddExpr -> + MultExpr whileAddExpr

| – MultExpr whileAddExpr

| e

MultExpr -> NegExpr whileMultExpr

whileMultExpr -> \* NegExpr whileMultExpr

| / NegExpr whileMultExpr

| e

NegExpr -> -Value

| Value

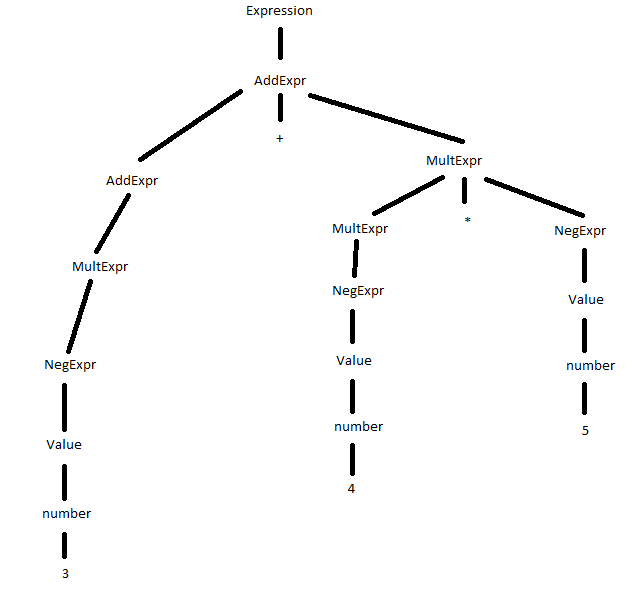
Value -> **id | number |** (Expression)

relop -> {<, >, >=, <=, <>, =}

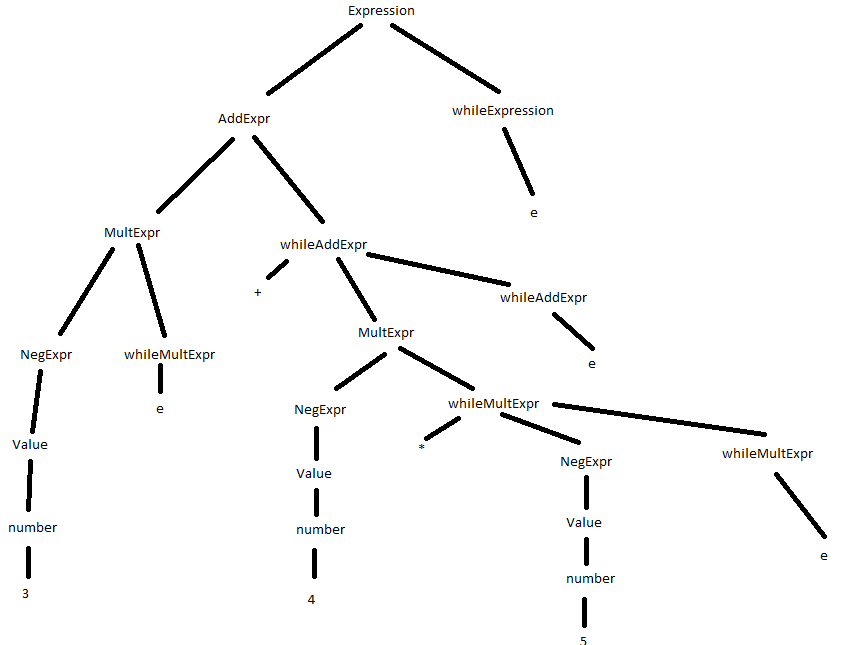
The Program and P definitions have been compressed into one for simplicity.

Two tree decompositions of the old and new grammar follow below to show that the grammar has not changed. The program being tested is a simple expression, 3 + 5 \* 4:

Old grammar:



New grammar:



Both trees will evaluate to the correct answer of 23.

**Project Decomposition**

The following classes will be defined as part of the decomposition:

* Parser
* Environment