1 Code Generation

Consider the following grammar:

\[
\begin{align*}
  \text{start} & \rightarrow \text{BOF expr EOF} \\
  \text{expr} & \rightarrow \text{expr PLUS term} \\
  \text{expr} & \rightarrow \text{expr MINUS term} \\
  \text{expr} & \rightarrow \text{term} \\
  \text{term} & \rightarrow \text{term EXPT factor} \\
  \text{term} & \rightarrow \text{factor} \\
  \text{factor} & \rightarrow \text{LPAREN expr RPAREN} \\
  \text{factor} & \rightarrow \text{ID}
\end{align*}
\]

This grammar represents mathematical equations involving +, −, and ^ (exponentiation). Write a Scala, C++, or Racket program which reads in an unindented preorder traversal of a parse tree and produces MIPS code which, when executed, produces the result of evaluating the expression represented by the parse tree. Interpret all IDs as 1.

1.1 Helpful conventions

You may find the following conventions and starting points helpful:

- Set $4$ to always contain 4.
- Set $11$ to always contain 1.
- Write push and pop functions which return the instructions corresponding to pushing and popping on to and off of the stack.
- Write expt as an assembly function and call that function whenever you need to compute an exponent.
1.2 Example

Given the following input:

```
start BOF expr EOF
expr expr PLUS term
expr term
term factor
factor ID
term factor
term factor
factor ID
```

You might produce the following code (but will likely produce something significantly longer):

```
lis $1
.word 1
lis $2
.word 1
add $3, $1, $2
jr $31
```