Implementation

- There are two main classes of programming language implementations:
  - Compilers
  - Interpreters
Compilers vs. Interpreters: What is the difference?

- **Compilers translate** high-level languages (Java, C, C++) into low-level languages (Java Byte Code, assembly language).

- **Interpreters execute** high-level languages directly (early versions of Lisp and Basic, Asteroid).

**Note:** Virtual machines can be considered interpreters for low-level languages; they directly execute a low-level language without first translating it.
Compilers vs. Interpreters

- Why choose compilation over interpretation?
  - Compilers can generate very **efficient** code and, consequently, the compiled programs run **faster** than interpreted programs.
Compilers vs. Interpreters

Why choose interpretation over compilation?

- Responsive programming system – no compile/link step
- Architecture independent – no code generation
- Partial evaluation of a program
  - REPL – ‘read, evaluate, print, loop’
  - E.g. Python’s ‘>>>’ interface.
A detailed look at an interpreter for our simple calculator language written in Asteroid.

Here is the grammar for our calc language:

\[
<expression>* ::= \langle expression \rangle + \langle mulexp \rangle \\
| \langle expression \rangle - \langle mulexp \rangle \\
| \langle mulexp \rangle \\
\]

\[
\langle mulexp \rangle ::= \langle mulexp \rangle * \langle rootexp \rangle \\
| \langle mulexp \rangle / \langle rootexp \rangle \\
| \langle rootexp \rangle \\
\]

\[
\langle rootexp \rangle ::= number | - \langle rootexp \rangle | ( \langle expression \rangle )
\]
Interpreter Implementation

Program Text → Symbol Stream → Lexical Analyzer → Token Stream → Parser/Interpreter → Values

Lexical Analysis: Convert symbol stream into a token stream.

Syntax Analysis/Interpretation: Make sure the sequence of tokens in the token stream conforms to the rules of the structure of the programming language and interpret the structures as soon as they are recognized.

Example/Demo: Our simple calc interpreter
Our implementation is based on something called *syntax-directed interpretation* – here interpretation of expressions happens as soon as they are recognized by the interpreter.

Other schemes exist where the interpreter first builds an intermediate representation of the program (similar to what we saw with the compiler) and then interprets this intermediate representation.

Our interpreter architecture consists of 2 parts:
- Lexer
- Parser/Interpreter
The Lexer

- Turns an input stream into a token stream
- Provides a convenient interface to the token stream

```plaintext
--- test the lexer

load system "io".
load "lexer".

let input = read().
let lexer = Lexer(input).

while not lexer @eof() do
  let t = lexer @get(). -- get a token
  println t.
end
```

```
[1] lutz$ asteroid test_lexer.ast
2*3+1
Token(number,2)
Token(mul,*)
Token(number,3)
Token(add,+)
Token(number,1)
lutz$
```
Here we use a parsing scheme called a “recursive descent parser”
We derive the parser directly from the grammar.
In this scheme we have one function for each non-terminal in the grammar.
These function implement all the rules for the respective non-terminals.
This gives rise to mutually recursive functions since most grammars are highly recursive.
In order to make this scheme work we need to rewrite our grammar slightly using an extended grammar notation called EBNF.

Our grammar:

\[
\begin{align*}
\text{<expression>*} & \ ::= \text{<expression>} + \text{<mulexp>} \\
& \quad | \quad \text{<expression>} - \text{<mulexp>} \\
& \quad | \quad \text{<mulexp>}
\end{align*}
\]

\[
\begin{align*}
\text{<mulexp>} & \ ::= \text{<mulexp>} * \text{<rootexp>} \\
& \quad | \quad \text{<mulexp>} / \text{<rootexp>} \\
& \quad | \quad \text{<rootexp>}
\end{align*}
\]

\[
\begin{align*}
\text{<rootexp>} & \ ::= \text{number} \mid - \text{<rootexp>} \mid ( \text{<expression>} )
\end{align*}
\]
The Parser

Becomes:

\[
\begin{align*}
<expression>^* & ::= <mulexp> \{ (+ <mulexp>) | (- <mulexp>) \} \\
<mulexp> & ::= <rootexp> \{ (\* <rootexp>) | (/ <rootexp>) \} \\
<rootexp> & ::= \text{number} | - <rootexp> | \text{(} <expression> \text{)} \text{)}
\end{align*}
\]

Notes: expressions written as \{\text{something}\} mean that \text{something} can appear zero or more times in the input.

Observation: we have replaced recursion in the grammar with the \{\ldots\} operators. You should convince yourself that we are still parsing the same language.
Building the parser is now straightforward:

- For each of the non-terminals we write a function that implements the rule(s)
- The functions interface to the lexer to ask for tokens from the token stream as needed.
- The functions also perform the interpretations of the operators as they are being recognized.
The Parser

<rootexp> ::= number | - <rootexp> | \( <expression> \) \)

```
function rootexp with lexer do
    --- <rootexp> ::= number | - <rootexp> | \( <expression> \) \\
    let token = lexer @peek().
    if not token do
        throw Error("syntax error: expected rootexp")
    elsif token @type == "number" do
        let val = lexer @token_match("number") @value.
        return val.
    elsif token @type == "sub" do
        lexer @token_match("sub").
        let val = rootexp(lexer).
        return - val.
    elsif token @type == "lparen" do
        lexer @token_match("lparen").
        let val = expression(lexer).
        lexer @token_match("rparen").
        return val.
    else do
        throw Error("syntax error at token "+token).  
    end
end
```
The Parser

```
<mulexp> ::= <rootexp> { (\* <rootexp>) | (\/ <rootexp>) }
```

```ruby
function mulexp with lexer do
  -- <mulexp> ::= <rootexp> { (\* <rootexp>) | (\/ <rootexp>) }
  let val = rootexp(lexer).
  loop do
    let token = lexer @peek().
    if not token do
      break.
    elsif token @type == "mul" do
      lexer @token_match("mul").
      let val = val * rootexp(lexer).
    elsif token @type == "div" do
      lexer @token_match("div").
      let val = val / rootexp(lexer)
    else do
      break.
    end
  end
  return val.
end
```
The Parser

\[
\text{<expression>\(^*\) ::= <mulexp> \{ (+ <mulexp>) | (- <mulexp>) \}}
\]

```ruby
function expression with lexer do
  -- <expression> ::= <mulexp> \{ (+ <mulexp>) | (- <mulexp>) \}
  let val = mulexp(lexer).
  loop do
    let token = lexer @peek().
    if not token do
      break.
    elsif token @type == "add" do
      lexer @token_match("add").
      let val = val + mulexp(lexer).
    elsif token @type == "sub" do
      lexer @token_match("sub").
      let val = val - mulexp(lexer)
    else do
      break.
    end
  end
  return val.
end
```
Putting it all together:

- Read the input stream from stdin
- Instantiate the lexer on the input stream
  - Tokenize
  - Provide nice interface to token stream
- Call parser functions – start with start symbol.
- Print out the computed value
The Interpreter

```plaintext
-- driver part of the script

let input = read().
let lexer = Lexer(input).

-- parse and interpret input
let val = expression(lexer).
if not (lexer @ eof()) do
    throw Error("tokens still in input stream")
end

-- print out the final value of the parsed and interpreted expression
println ("=> "+val).
```
The code for the interpreter is available on repl.it:

https://repl.it/@lutzhamel/Calc
The Interpreter

Running the interpreter on a Unix-like system (repl.it shell):
Compilers

Program Text → Symbol Stream → Lexical Analyzer → Token Stream → Parser/Code Gen → Target Program

Lexical Analysis:
Convert symbol stream into a token stream.

Syntax Analysis/Code Gen:
Make sure the sequence of tokens in the token stream conforms to the rules of the structure of the programming language and generate code for the structures as soon as they are recognized.
Compilers

- Notice that the architecture of compilers is very similar to interpreters except that compilers generate code.
- Our aim is to build a compiler from our expression language to a stack machine language.
**Problem**: Build a simple compiler from arithmetic expressions to a stack machine.

The translator accepts the same language as our calc language:

```
<expression>* ::= <mulexp> { (+ <mulexp>) | (- <mulexp>) }
<mulexp> ::= <rootexp> { (* <rootexp>) | (/ <rootexp>) }
<rootexp> ::= number | - <rootexp> | ( <expression> )
```

The compiler generates the following stack machine language:

```
<comlist>* ::= <command> <comlist> | <empty>
<command> ::= add | sub | mul | push <number> | pop | print
<number> ::= -- any valid integer --
```
In stack machines all computations happen using a stack, e.g.

- push 3  // push 3 onto the stack
- push 2  // push 2 onto the stack
- add     // pop 3 and 2, add, push result
- print   // pop top of stack and print

The result of this computation would be the value 5 printed to the screen.
Given the expression \((1+2)\times 3\) our compiler should produce:

- push 1
- push 2
- add
- push 3
- mul
- print

Given this it is easy to see that our compiler should implement the following translation scheme:

- number -> push <value>
- + -> add
- - -> minus
- * -> mul
- / -> div
- parentheses -> ignore, just focus on the expression at the top level -> insert a print statement
Note: it is assumed that the arithmetic commands pop the values off the stack that they use and push the result back onto the stack.

Base your compiler implementation on the calculator code given here: https://repl.it/@lutzhamel/Calc

You can test drive your generated code with the stack machine given here: https://repl.it/@lutzhamel/Machine

See Assignment #4 in BS
Another look at compilers.
Here we implemented a very simple compiler for arithmetic expressions.
Real compilers are more complex…
A Simple Compiler

Lexical Analysis: Convert symbol stream into a token stream.

Syntax Analysis/Code Gen: Make sure the sequence of tokens in the token stream conforms to the rules of the structure of the programming language and **generate code** for the structures as soon as they are recognized.
The Anatomy of a Compiler

Source Program

Syntax Analysis
- Recognize the structure of a source program, generate parse tree

Semantic Analysis
- Recognize/validate the meaning of a source program

Optimization
- Reorganize the parse tree/AST to make computations more efficient

Code Generation
- Translate parse tree/AST into low-level language

Translated Program

Observations:
- Language definitions have two parts: syntax and semantics
- Compilers have two phases which deal with each of these language definition components: syntax analysis, semantic analysis.