Implementation

- There are two main classes of programming language implementations
  - Compilers
  - Interpreters

Compilers vs. 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

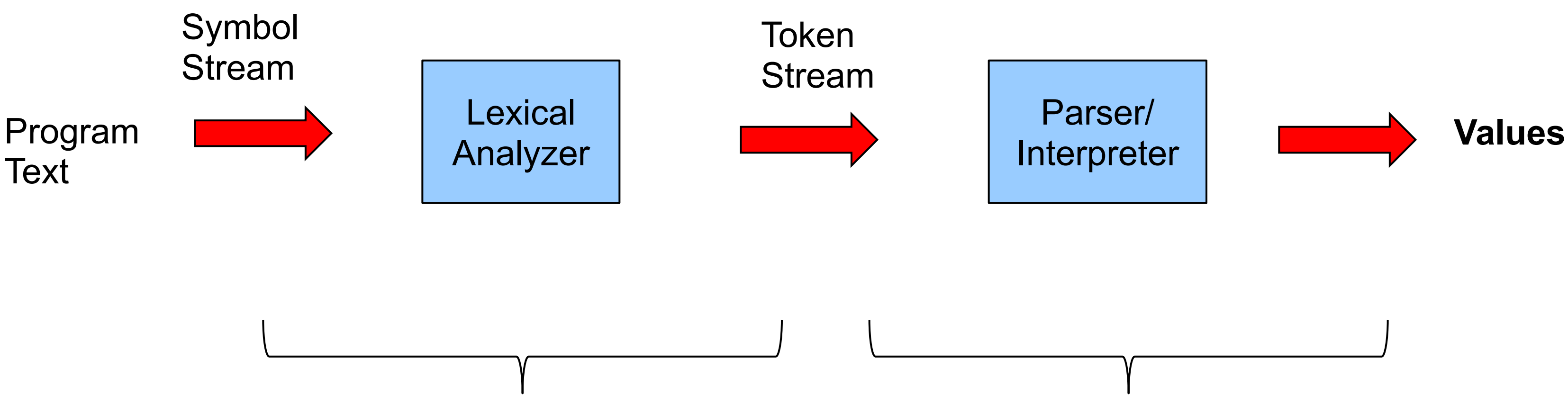
- Why choose interpretation over compilation?
  - Responsive programming system – no compile/link step
  - Architecture independent – no code generation
  - Partial evaluation of a program
    - REPL – ‘read, evaluate, print, loop’
    - E.g. Python’s ‘>>>’ interface.

Interpreter Implementation

- A detailed look at an interpreter for our simple calculator language written in Asteroid.
- Here is the grammar for our calc language:

```
<expression>* ::= <expression> + <mulexp>
               |   <expression> - <mulexp>
               |   <mulexp>

<mulexp> ::= <mulexp> * <rootexp>
          |   <mulexp> / <rootexp>
          |   <rootexp>

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

Interpreter Implementation



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

Interpreter Implementation

- 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
- Provide a convenient interface to the token stream

```
-- 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
```

ln012 — -zsh — 61×13

```
lutz$ asteroid test_lexer.ast
2*3+1
Token(number,2)
Token(mul,*)
Token(number,3)
Token(add,+)
Token(number,1)
lutz$
```

The Parser

- 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.

The Parser

- In order to make this scheme work we need to rewrite our grammar slightly using an extended grammar notation called EBNF.
- Our grammar:

```
<expression>* ::= <expression> + <mulexp>
               |   <expression> - <mulexp>
               |   <mulexp>

<mulexp> ::= <mulexp> * <rootexp>
          |   <mulexp> / <rootexp>
          |   <rootexp>

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

The Parser

- Becomes:

<expression>* ::= <mulexp> { (+ <mulexp>) | (- <mulexp>) }

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

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

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

Observation: we have replaced recursion in the grammar with the {…} operators. You should convince yourself that we are still parsing the same language.

The Parser

- Building the parser is now straight forward:
  - 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

<expression>* ::= <mulexp> { (+ <mulexp>) | (- <mulexp>) }

```
function expression
  with lexer:%Lexer do
    let val = mulexp(lexer).
    loop do
      let token = lexer @peek().
      if not token do
        break.
      elif token @type == "add" do
        lexer @token_match("add").
        let val = val + mulexp(lexer).
      elif token @type == "sub" do
        lexer @token_match("sub").
        let val = val - mulexp(lexer)
      else do
        break.
      end
    end
    return val.
  end
```

Note: our calculator computations are in the parser.

The Parser

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

```
function mulexp
  with lexer:%Lexer do
    let val = rootexp(lexer).
    if not lexer @peek() do
      return val.
    end
    loop do
      let token = lexer @peek().
      if not token do
        break.
      elif token @type == "mul" do
        lexer @token_match("mul").
        let val = val * rootexp(lexer).
      elif token @type == "div" do
        lexer @token_match("div").
        let val = val / rootexp(lexer)
      else do
        break.
      end
    end
    return val.
  end
```

The Parser

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

```
function rootexp
  with lexer:%Lexer do
    try
      let Token(type,val) = lexer @peek().
    catch _ do
      return none.
    end
    if type == "number" do
      lexer @token_match("number").
      return val.
    elif type == "sub" do
      lexer @token_match("sub").
      return - rootexp(lexer).
    elif 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 "+val).
    end
  end
```

The Interpreter

- 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

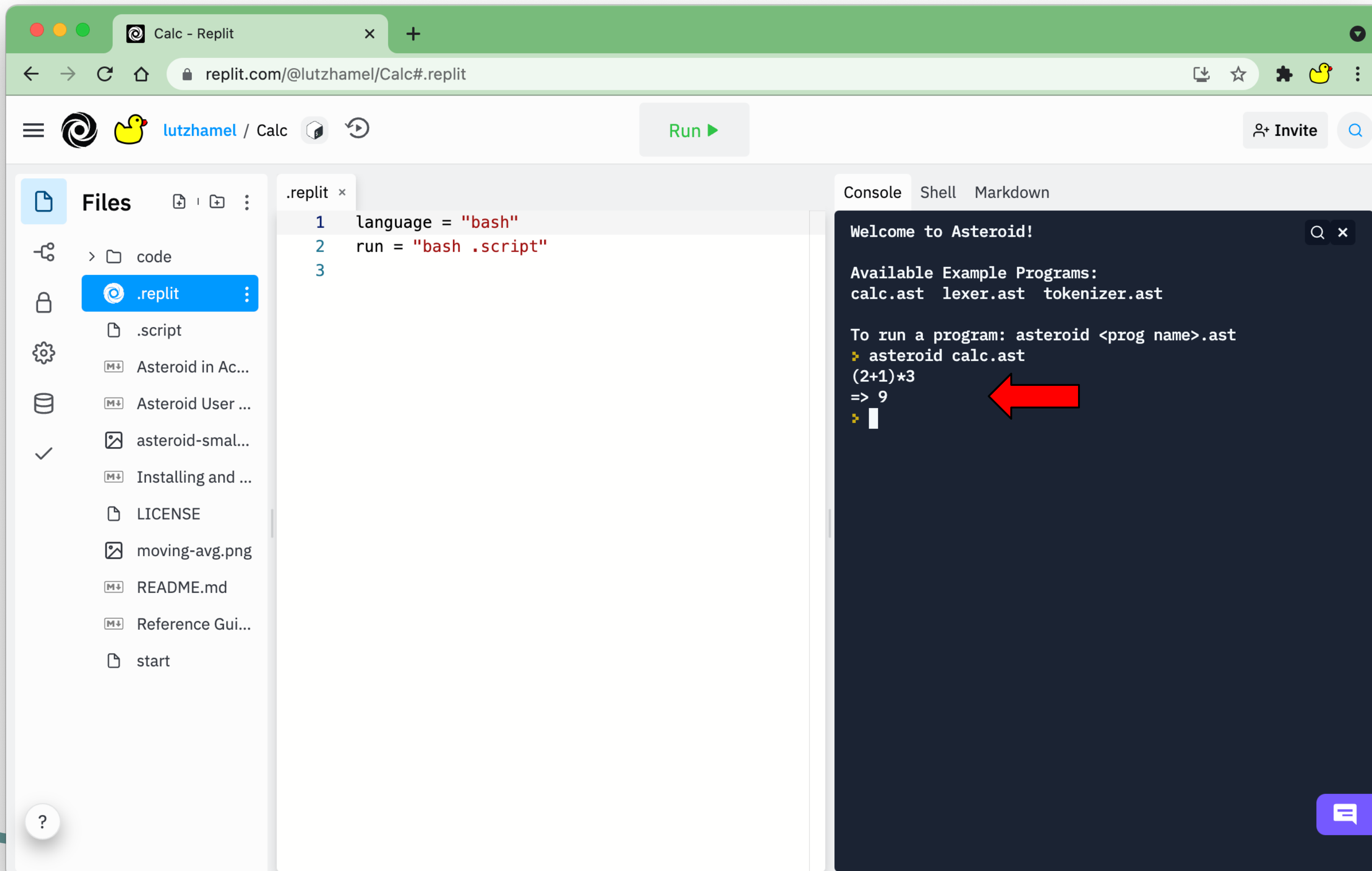
```
-- driver part of the script

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).
```

Interpreter Code

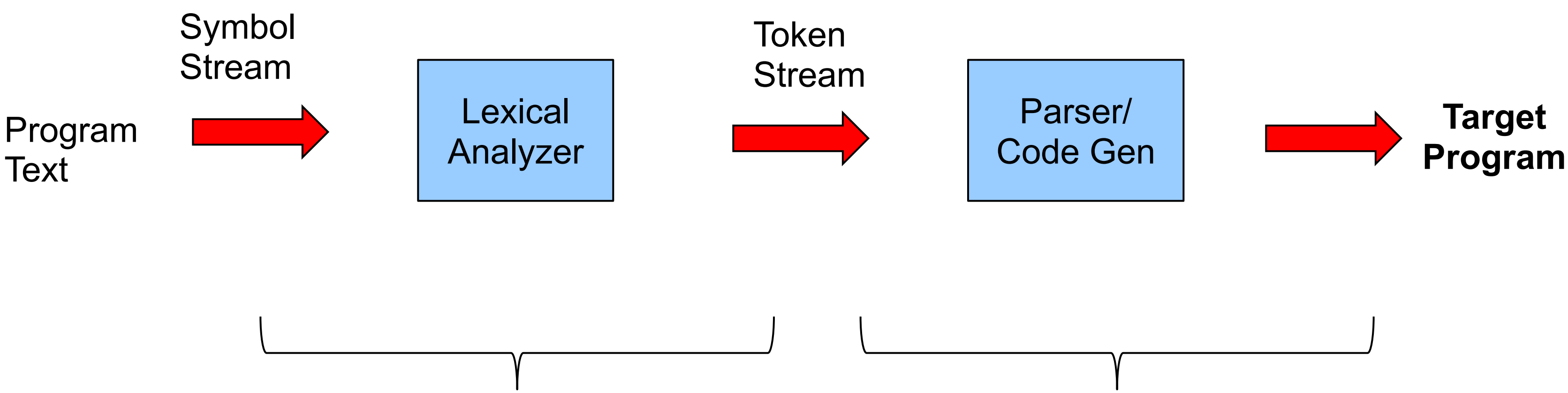
- The code for the interpreter is available on repl.it:
  - https://repl.it/@lutzhamel/Calc

The Interpreter

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



Compiler Implementation



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.

Compilation Example

Source code, a simple expression: 3*2+5

Assembly Language

`load` *<address>,reg*
`load` *<value>,reg*
`store` *reg,<address>*
`add` *reg,reg,reg*
`sub` *reg,reg,reg*
`mul` *reg,reg,reg*

Note: last argument to instructions is the target!

Our machine has three registers: *r1, r2, r3*

Assembly Code:

load 3,r1
load 2,r2
mul r1,r2,r1
load 5,r2
add r1,r2,r1

Assignment: Compiler

Problem: Build a simple compiler from arithmetic expressions to a stack machine.

The translator accepts the same language as our calc language:

```
<expression>* ::= <mulexp> + <expression>
               |  <mulexp> - <expression>
               |  <mulexp>

<mulexp> ::= <rootexp> * <mulexp>
          |  <rootexp> / <mulexp>
          |  <rootexp>

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

 <number>    ::= -- any valid integer --
```

The compiler generates the following stack machine language:

```
<comlist>*  ::= <command> <comlist> | <empty>
<command>   ::= add | sub | mul | push <number> | pop | print
<number>    ::= -- any valid integer --
```

Assignment: Compiler

- Given the expression (1+2)*3 your compiler should produce:

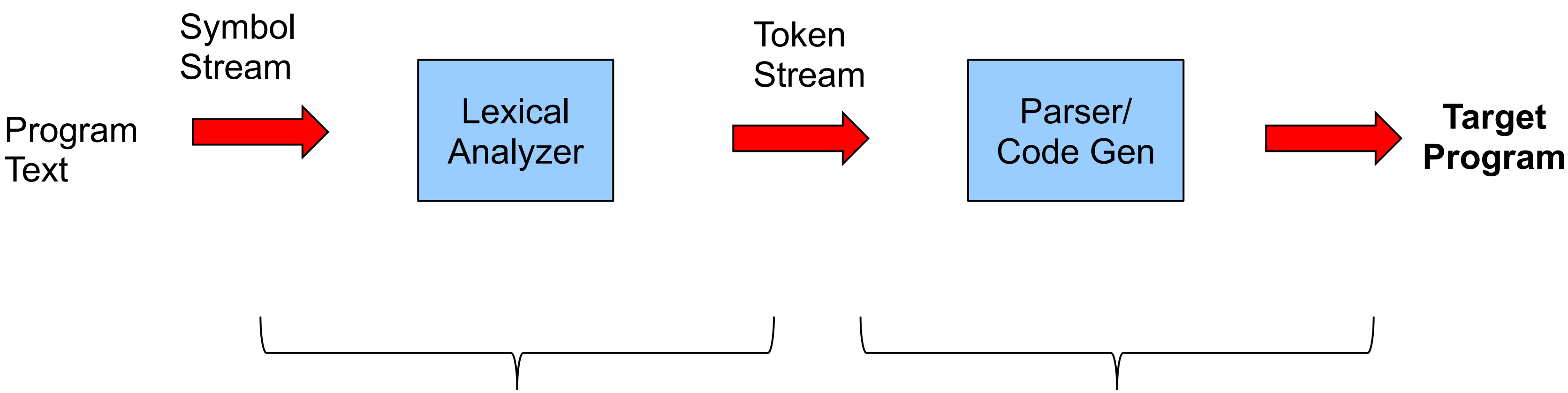
```
push 1
push 2
add
push 3
mul
print
```

- 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**

Compiler

- Another look at compilers.
- Here we implemented a very simple compiler for arithmetic expressions.
- Real compilers are more complex…

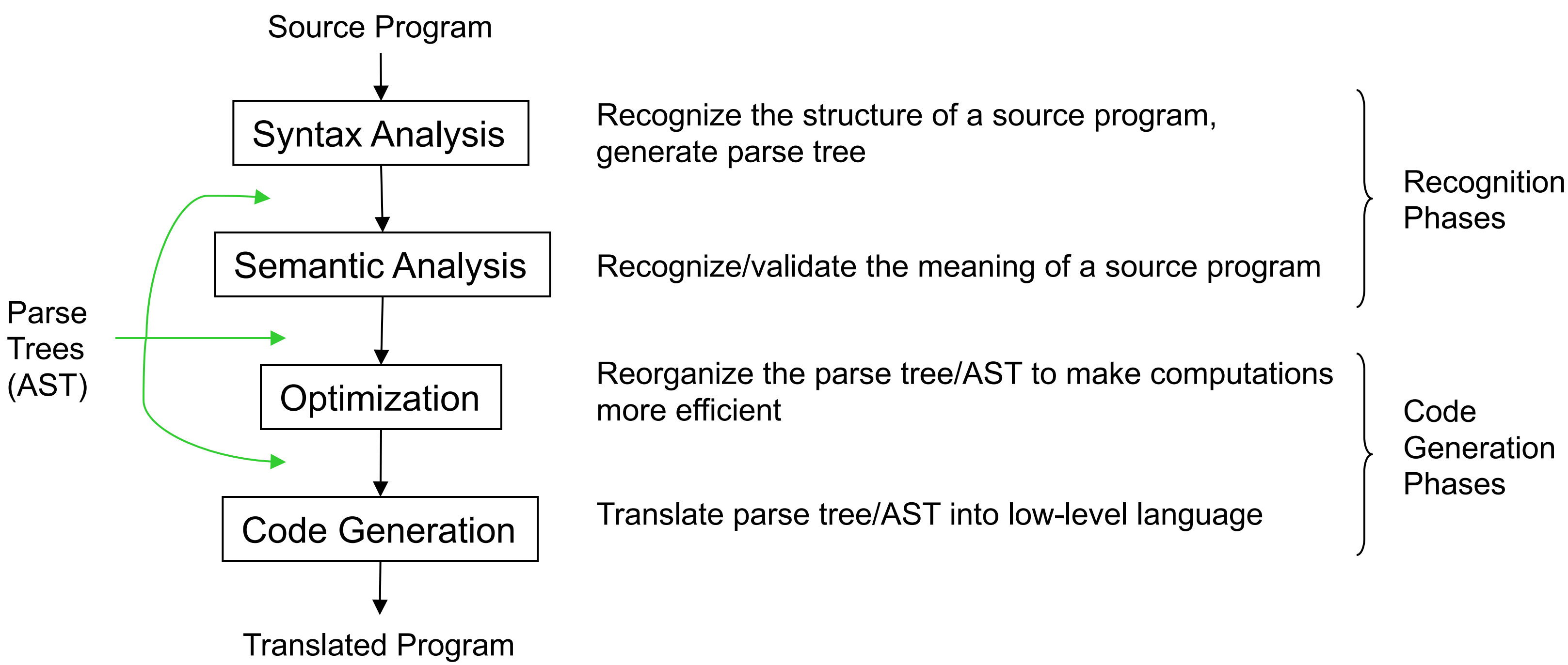
A Simple Compiler



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



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.