

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).

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

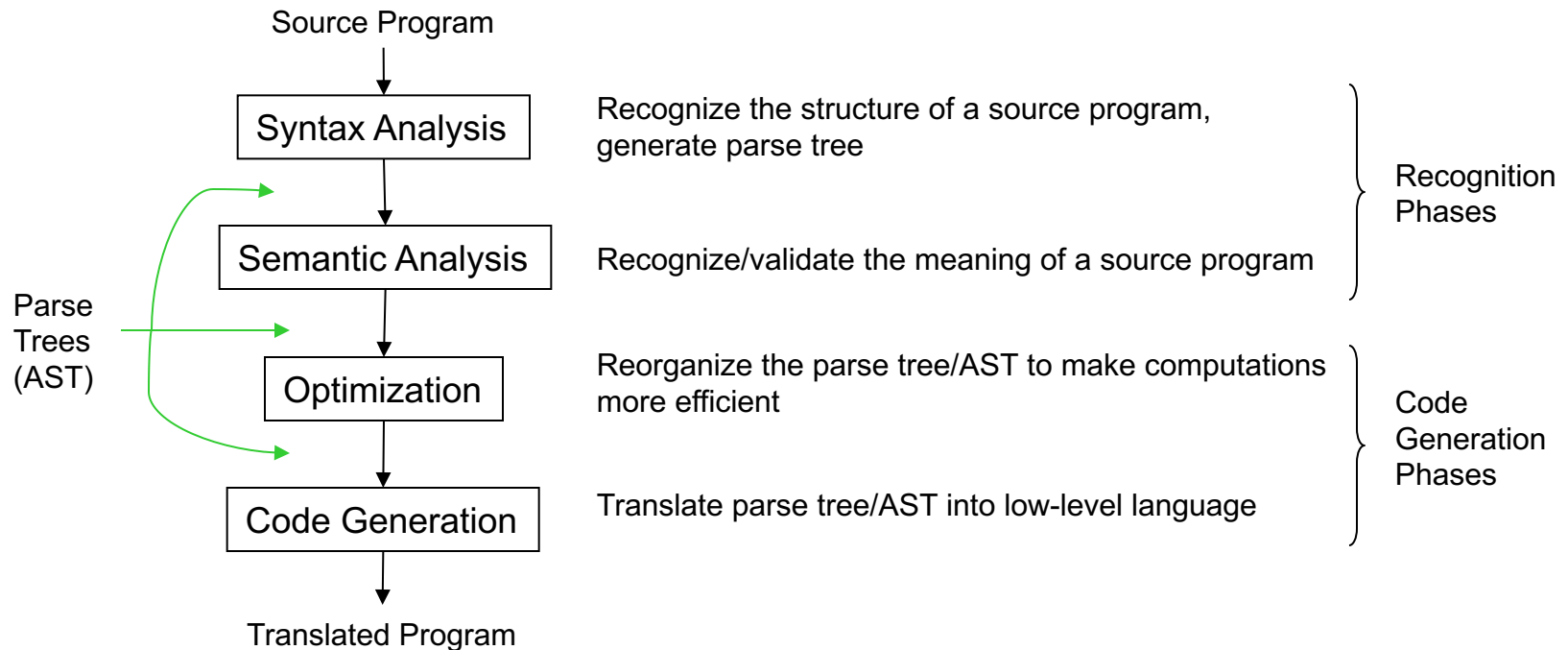
Compilers vs. Interpreters

- 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.

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.

Compilation Example

Assembly Language

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

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

consider: $3*2+5$

Assembly Code:

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

Note: last argument to instructions is the target!

Interpreter Implementation

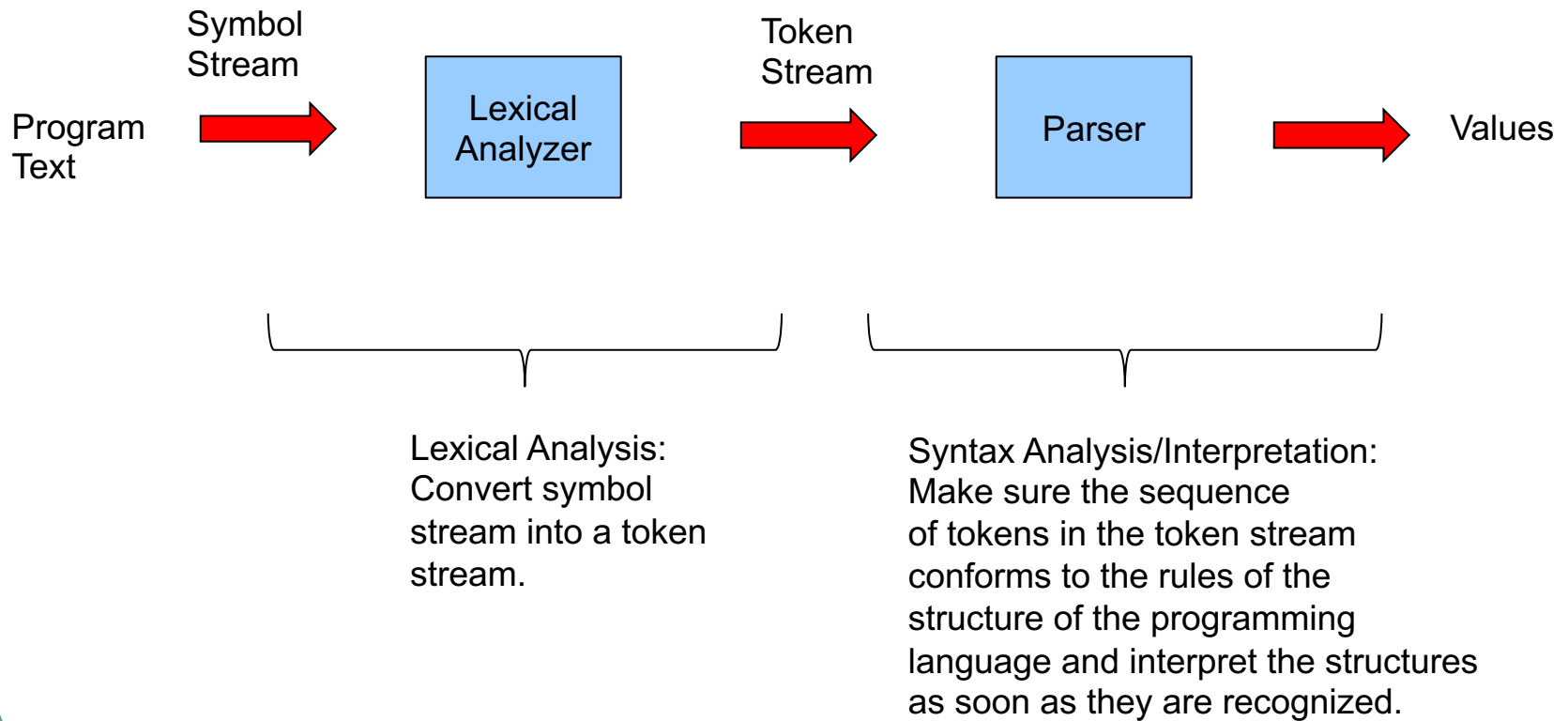
- A detailed look at an interpreter for a simple calculator language written in Rust.
- Here is the grammar for our language:

$$\begin{aligned} \langle \text{expression} \rangle^* &::= \langle \text{mulexp} \rangle + \langle \text{expression} \rangle \\ &\quad | \langle \text{mulexp} \rangle - \langle \text{expression} \rangle \\ &\quad | \langle \text{mulexp} \rangle \end{aligned}$$
$$\begin{aligned} \langle \text{mulexp} \rangle &::= \langle \text{rootexp} \rangle * \langle \text{mulexp} \rangle \\ &\quad | \langle \text{rootexp} \rangle / \langle \text{mulexp} \rangle \\ &\quad | \langle \text{rootexp} \rangle \end{aligned}$$
$$\langle \text{rootexp} \rangle ::= \text{number} \mid - \langle \text{rootexp} \rangle \mid (\langle \text{expression} \rangle)$$

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 Implementation



Interpreter Implementation

- Instead of using tools for the lexical analysis and the parser such as 'lex' and 'yacc' we will hand code these.
- Hand coding at least at this stage has the advantage in that we see exactly what's going on.

The Lexer

- We need to define what kind of tokens we want to encode
- Define that symbol and token stream types

```
7  #[derive(Debug, Clone)]
8  pub enum TokenType {
9      LParen,
10     RParen,
11     Add,
12     Sub,
13     Mul,
14     Div,
15     Number(i64),
16     NoToken,
17 }
18
19 type TokenStream = Vec<TokenType>;
20 type SymbolStream = String;
21
```

The Lexer

Tokenize the input stream:

```
42 fn tokenize(input_stream: SymbolStream) -> TokenStream {
43     let mut output_stream: TokenStream = Vec::new();
44     let mut it = input_stream.chars().peekable();
45
46     while let Some(c) = it.next() {
47         use TokenType::*;
48         match c {
49             '0'..='9' => output_stream.push(get_number_token(c, &mut it)),
50             '+' => output_stream.push(Add),
51             '-' => output_stream.push(Sub),
52             '*' => output_stream.push(Mul),
53             '/' => output_stream.push(Div),
54             '(' => output_stream.push(LParen),
55             ')' => output_stream.push(RParen),
56             ' ' | '\t' | '\n' => (),
57             _ => panic!(format!("unexpected character {}", c)),
58         }
59     }
60     output_stream
61 }
```

The Lexer

Tokenize the input stream:

```
26 fn get_number_token(c: char, iter: &mut Peekable<Chars>) -> TokenType {
27     let mut number = c.to_string();
28     while let Some(true) = iter.peek().map(|c| c.is_digit(10)) {
29         if let Some(d) = iter.next() {
30             number.push_str(&(d.to_string()));
31         } else {
32             panic!("error state");
33         }
34     }
35     if let Ok(val) = number.parse::<i64>() {
36         TokenType::Number(val)
37     } else {
38         panic!("bad i64 parse");
39     }
40 }
```

The Lexer

The Lexer interface:

```
67 struct Lexer<'a> {
68     token_iter: Peekable<Iter<'a, TokenType>>,
69 }
70
71 impl<'a> Lexer<'a> {
72
73     fn next_token(&mut self) -> TokenType{
74         match self.token_iter.next() {
75             Some(t) => (*t).clone(),
76             None => TokenType::NoToken,
77         }
78     }
79
80     fn match_token(&mut self, item: TokenType) {
81         if discriminant(&self.peek_token()) == discriminant(&item) {
82             self.next_token();
83         } else {
84             panic!(format!("Expected token {:?}",item));
85         }
86     }
87 }
```

...

The Lexer

The Lexer interface:

```
88     fn peek_token(&mut self) -> TokenType {
89         match self.token_iter.peek() {
90             Some(t) => (**t).clone(),
91             None => TokenType::NoToken,
92         }
93     }
94
95     fn eof(&mut self) -> bool {
96         match self.token_iter.peek() {
97             Some(_) => false,
98             None => true,
99         }
100     }
101 }
```

The Parser

- Here we use a parsing scheme called a “recursive descent parser”
- 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.
- Our grammar:

$$\begin{aligned} \langle \text{expression} \rangle^* &::= \langle \text{mulexp} \rangle + \langle \text{expression} \rangle \\ &\quad | \langle \text{mulexp} \rangle - \langle \text{expression} \rangle \\ &\quad | \langle \text{mulexp} \rangle \end{aligned}$$
$$\begin{aligned} \langle \text{mulexp} \rangle &::= \langle \text{rootexp} \rangle * \langle \text{mulexp} \rangle \\ &\quad | \langle \text{rootexp} \rangle / \langle \text{mulexp} \rangle \\ &\quad | \langle \text{rootexp} \rangle \end{aligned}$$
$$\langle \text{rootexp} \rangle ::= \text{number} \mid - \langle \text{rootexp} \rangle \mid (\langle \text{expression} \rangle)$$

The Parser

- Becomes:

$$\langle \text{expression} \rangle^* ::= \langle \text{mulexp} \rangle \{ ('+' \langle \text{mulexp} \rangle) | ('-' \langle \text{mulexp} \rangle) \}$$
$$\langle \text{mulexp} \rangle ::= \langle \text{rootexp} \rangle \{ ('*' \langle \text{rootexp} \rangle) | ('/' \langle \text{rootexp} \rangle) \}$$
$$\langle \text{rootexp} \rangle ::= \text{number} | '-' \langle \text{rootexp} \rangle | '(' \langle \text{expression} \rangle ')'$$

Notes: expressions written as **{something}** mean that **something** can **appear zero or more times** in the input. Also, we have put operator tokens in quotes in order to distinguish them from operators in the grammar language itself, i.e. parentheses.

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

$\langle \text{expression} \rangle^* ::= \langle \text{mulexp} \rangle \{ ('+' \langle \text{mulexp} \rangle) | ('-' \langle \text{mulexp} \rangle) \}$

```
114 fn expression(lexer: &mut Lexer) -> i64 {  
115     use TokenType::*;  
116     let mut val = mulexp(lexer);  
117     loop {  
118         match lexer.peek_token() {  
119             Add => {  
120                 lexer.match_token(Add);  
121                 val += mulexp(lexer);  
122             }  
123             Sub => {  
124                 lexer.match_token(Sub);  
125                 val -= mulexp(lexer);  
126             }  
127             _ => break,  
128         }  
129     }  
130     val  
131 }
```

The Parser

$\langle \text{mulexp} \rangle ::= \langle \text{rootexp} \rangle \{ ('*' \langle \text{rootexp} \rangle) | ('/' \langle \text{rootexp} \rangle) \}$

```
133 fn mulexp(lexer: &mut Lexer) -> i64 {
134     use TokenType::*;
135     let mut val = rootexp(lexer);
136     loop {
137         match lexer.peek_token() {
138             Mul => {
139                 lexer.match_token(Mul);
140                 val *= rootexp(lexer);
141             }
142             Div => {
143                 lexer.match_token(Div);
144                 val /= rootexp(lexer);
145             }
146             _ => break,
147         }
148     }
149     val
150 }
```

The Parser

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

```
152 fn rootexp(lexer: &mut Lexer) -> i64 {
153     use TokenType::*;
154     let val;
155     match lexer.peek_token() {
156         Number(v) => {
157             lexer.match_token(Number(v));
158             val = v;
159         }
160         Sub => {
161             lexer.match_token(Sub);
162             val = - rootexp(lexer);
163         }
164         LParen => {
165             lexer.match_token(LParen);
166             val = expression(lexer);
167             lexer.match_token(RParen);
168         }
169         t => {
170             panic!(format!("syntax error at token {:?}",t));
171         }
172     }
173     val
174 }
```

The Interpreter

- Putting it all together:
 - Read the input stream from stdin
 - Tokenize the input stream
 - Instantiate the lexer with an iterator on the token stream
 - Call parser functions – start with start symbol.
 - Print out the computed value

The Interpreter

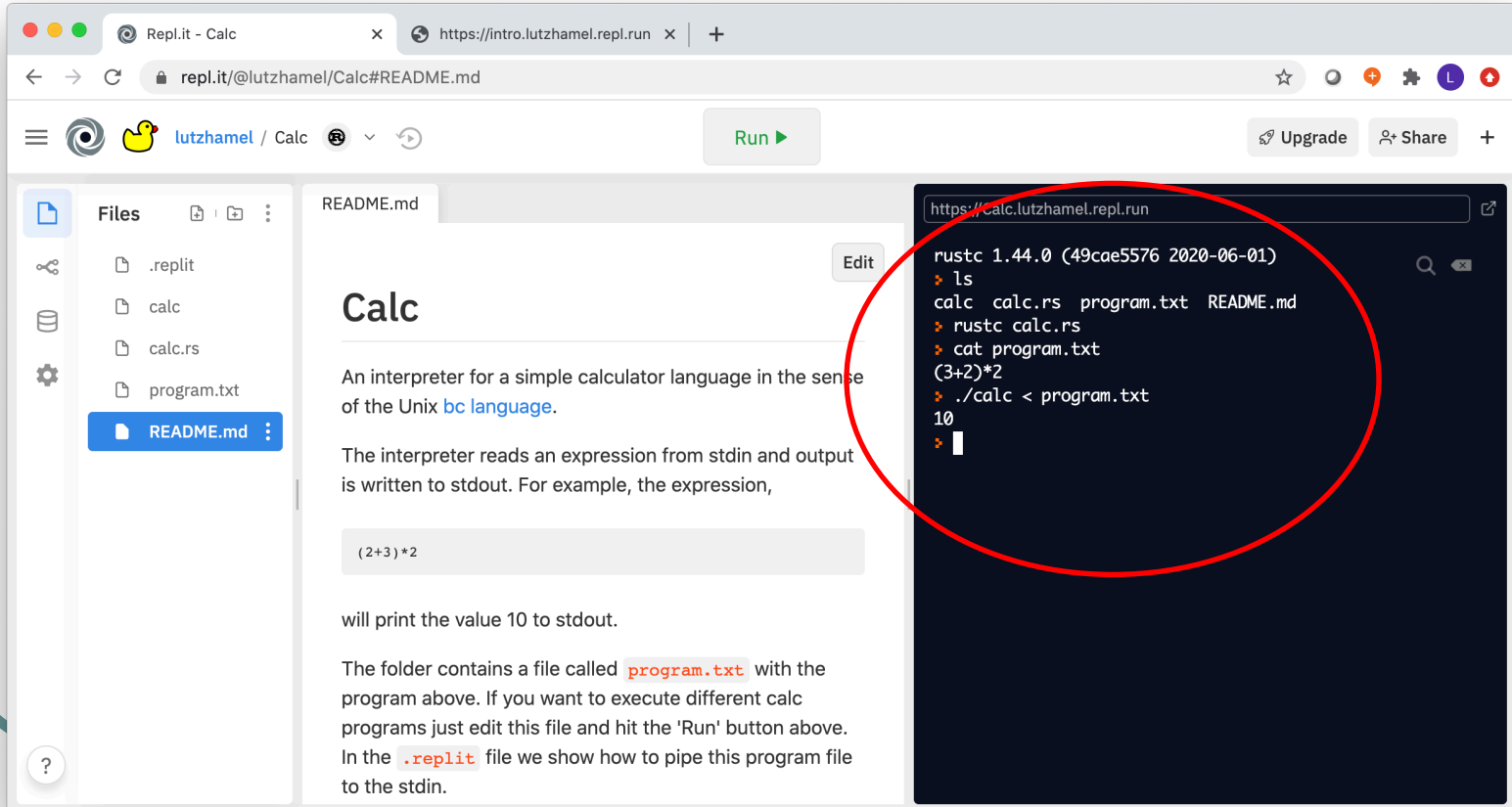
```
179 fn main() {
180     // get input from stdio
181     let mut input: SymbolStream = String::new();
182     if let Err(_) = stdin().read_to_string(&mut input) {
183         panic!("could not read from stdin")
184     }
185     // set up lexer
186     let token_stream: TokenStream = tokenize(input);
187     let mut lexer = Lexer {
188         token_iter: token_stream.iter().peekable(),
189     };
190     // run the parser and print result to stdout (println)
191     let val = expression(&mut lexer);
192     if !lexer.eof() {
193         panic!("syntax error, tokens still available on input");
194     }
195     println!("{}", val);
196 }
197
```


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):



The screenshot shows the Repl.it web interface for a project named 'Calc' by 'lutzhamel'. The left sidebar shows a file explorer with files: `.replit`, `calc`, `calc.rs`, `program.txt`, and `README.md`. The main area displays the `README.md` file, which describes the 'Calc' interpreter. A red circle highlights the terminal output, which shows the execution of a Rust program that runs the 'calc' interpreter on `program.txt`.

```
rustc 1.44.0 (49cae5576 2020-06-01)
> ls
calc  calc.rs  program.txt  README.md
> rustc calc.rs
> cat program.txt
(3+2)*2
> ./calc < program.txt
10
> 
```

Assignment: Translator

Problem: Build a simple translator 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> )
```

The translator generates the following stack machine language:

```
<comlist>* ::= <command> <comlist> | <empty>
<command> ::= <arithmetic> | <stack>
<arithmetic> ::= add | sub | mul | div
<stack> ::= push number | pop | print
```

Assignment: Translator

- Given the expression $(1+2)*3$ your translator 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 translator implementation on the calculator code given here: <https://repl.it/@lutzhamel/Calc>
- You can test drive you generated code with the stack machine given here: <https://repl.it/@lutzhamel/Machine>
- See Assignment #6 in BS