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.

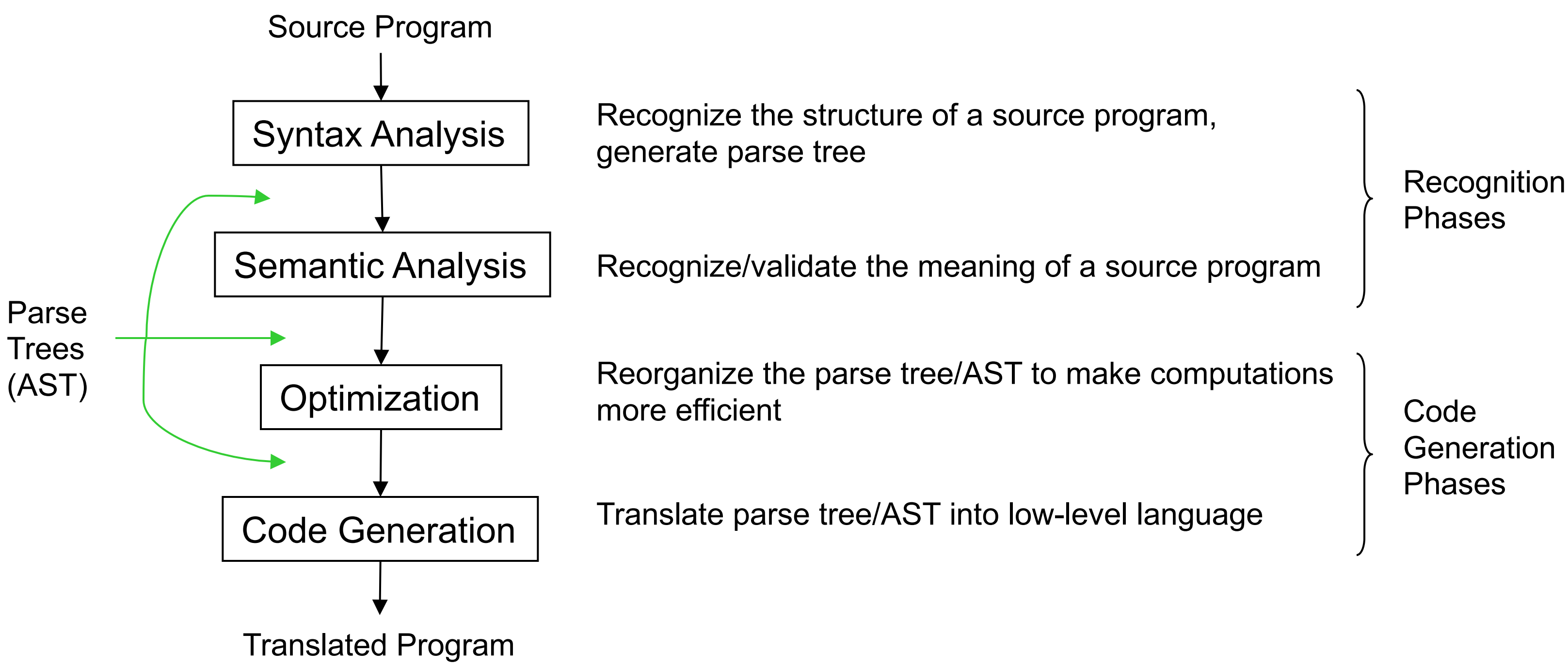
Compilers vs. Interpreters

- 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

consider: 3*2+5

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

Assembly Code:

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

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

Note: last argument to instructions is the target!

Interpreter Implementation

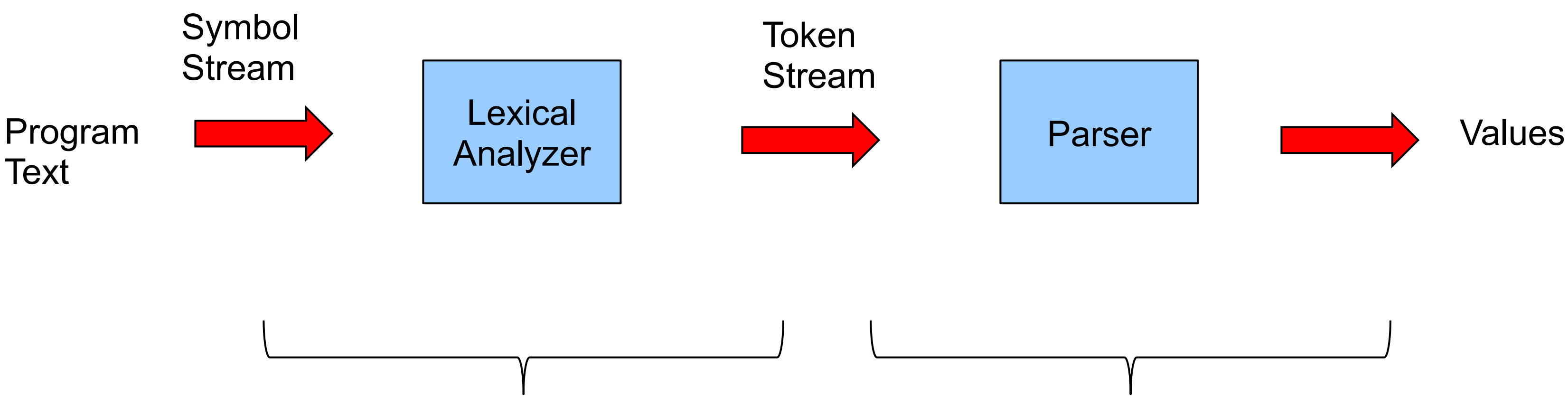
- A detailed look at an interpreter for a simple calculator language written in Rust.
- Here is the grammar for our language:

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

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

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

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



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.

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

```rust
#[derive(Debug, Clone)]
pub enum TokenType {
    LParen,
    RParen,
    Add,
    Sub,
    Mul,
    Div,
    Number(i64),
    NoToken,
}

type TokenStream = Vec<TokenType>;
type SymbolStream = String;
```

The Lexer

Tokenize the input stream:

```
fn tokenize(input_stream: SymbolStream) -> TokenStream {
    let mut output_stream: TokenStream = Vec::new();
    let mut it = input_stream.chars().peekable();

    while let Some(c) = it.next() {
        use TokenType::*;
        match c {
            '0'..='9' => output_stream.push(get_number_token(c, &mut it)),
            '+' => output_stream.push(Add),
            '-' => output_stream.push(Sub),
            '*' => output_stream.push(Mul),
            '/' => output_stream.push(Div),
            '('  => output_stream.push(LParen),
            ')'  => output_stream.push(RParen),
            ' ' | '\t' | '\n' => (),
            _ => panic!(format!("unexpected character {}", c)),
        }
    }
    output_stream
}
```

The Lexer

Tokenize the input stream:

```
fn get_number_token(c: char, iter: &mut Peekable<Chars>) -> TokenType {
    let mut number = c.to_string();
    while let Some(true) = iter.peek().map(|c| c.is_digit(10)) {
        if let Some(d) = iter.next() {
            number.push_str(&(d.to_string()));
        } else {
            panic!("error state");
        }
    }
    if let Ok(val) = number.parse::<i64>() {
        TokenType::Number(val)
    } else {
        panic!("bad i64 parse");
    }
}
```

The Lexer

The Lexer interface:

```
struct Lexer<'a> {
    token_iter: Peekable<Iter<'a, TokenType>>,
}

impl<'a> Lexer<'a> {

    fn next_token(&mut self) -> TokenType{
        match self.token_iter.next() {
            Some(t) => (*t).clone(),
            None => TokenType::NoToken,
        }
    }

    fn match_token(&mut self, item: TokenType)  {
        if discriminant(&self.peek_token()) == discriminant(&item) {
            self.next_token();
        } else {
            panic!(format!("Expected token {:?}",item));
        }
    }
```

...

The Lexer

The Lexer interface:

```
    fn peek_token(&mut self) -> TokenType {
        match self.token_iter.peek() {
            Some(t) => (**t).clone(),
            None => TokenType::NoToken,
        }
    }

    fn eof(&mut self) -> bool {
        match self.token_iter.peek() {
            Some(_) => false,
            None => true,
        }
    }
}
```

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:

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

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

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

```
fn expression(lexer: &mut Lexer) -> i64 {
    use TokenType::*;
    let mut val = mulexp(lexer);
    loop {
        match lexer.peek_token() {
            Add => {
                lexer.match_token(Add);
                val += mulexp(lexer);
            }
            Sub => {
                lexer.match_token(Sub);
                val -= mulexp(lexer);
            }
            _ => break,
        }
    }
    val
}
```

The Parser

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

```
fn mulexp(lexer: &mut Lexer) -> i64 {
    use TokenType::*;
    let mut val = rootexp(lexer);
    loop {
        match lexer.peek_token() {
            Mul => {
                lexer.match_token(Mul);
                val *= rootexp(lexer);
            }
            Div => {
                lexer.match_token(Div);
                val /= rootexp(lexer);
            }
            _ => break,
        }
    }
    val
}
```

The Parser

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

```
fn rootexp(lexer: &mut Lexer) -> i64 {
    use TokenType::*;
    let val;
    match lexer.peek_token() {
        Number(v) => {
            lexer.match_token(Number(v));
            val = v;
        }
        Sub => {
            lexer.match_token(Sub);
            val = - rootexp(lexer);
        }
        LParen => {
            lexer.match_token(LParen);
            val = expression(lexer);
            lexer.match_token(RParen);
        }
        t => {
            panic!(format!("syntax error at token {:?}",t));
        }
    }
    val
}
```

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

```
fn main() {
    // get input from stdio
    let mut input: SymbolStream = String::new();
    if let Err(_) = stdin().read_to_string(&mut input) {
        panic!("could not read from stdin")
    }
    // set up lexer
    let token_stream: TokenStream = tokenize(input);
    let mut lexer = Lexer {
        token_iter: token_stream.iter().peekable(),
    };
    // run the parser and print result to stdout (println)
    let val = expression(&mut lexer);
    if !lexer.eof() {
        panic!("syntax error, tokens still available on input");
    }
    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):