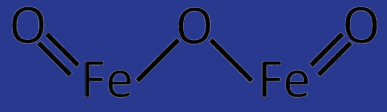
Iron Oxide



By: Gavin Wieckowski, Alexander Diaz, Mike Brueggman, Jacob Shepard

Grammar

Blue indicates terminal token

Black is a nonterminal

Grey is a meta operator

Program -> (func |let)+

func -> func identifier (paramList) block

paramList -> (param | param ,)*

param -> identifier : (type | identifier)

identifier -> ID(String)

type -> TYPE_INT32 | TYPE_FLT32 | TYPE_CHAR | TYPE_BOOL

let -> KW_let

identifier (: type (= identifier)) ;

block -> [(statement)*]

statement -> let | if | return | while | print | assign | funcCall

if -> KW_IF expression block KW_ELSE block

return -> KW_RETURN expression ;

while -> KW_WHILE expression block

print -> KW_PRINT expression ;

assign -> identifier = expression ;

expression -> (identifier | literal) expressionTail

literal -> LIT_INT32(i32) | LIT_FLT32(f32) | LIT_CHAR(char) | LIT_STRING(String) | LIT_BOOL(bool)

expressionTail -> (operator | identifier)*

operator -> OP_ADD | OP_NUL | OP_SUB | OP_LT | OP_GT | OP_EQ | OP_NGT | OP_NLT | OP_NEQ

funcCall -> identifier (arguments) ;

arguments -> (identifier (, identifier)*)

Lexer

- Adapted the Lexer from the previous Lexer Assignment.
- Divides Tokens into different groups for better readability and usability.
- Defines Lexer States for each group to handle characters which may represent multiple tokens e.g. >= vs. >

```
LexerState::Operator => {
    if c != ' ' && !(self.input_position == self.input_string.len()) {
        self.buffer_string.push(c);
    } else {
        if self.input_position == self.input_string.len() && c != ' ' {
            self.buffer_string.push(c);
        }
        let operator = self.buffer_string.clone();
        self.input_position -= 1;

        match operator.as_str() {
            "!=" => {
                self.current_token = Token::OP_NEQ;
                self.current_state = LexerState::Initial;
                return;
            }
            ">=" => {
                self.current_token = Token::OP_NLT;
                self.current_state = LexerState::Initial;
                return;
            }
            "<=" => {
                self.current_token = Token::OP_NGT;
                self.current_state = LexerState::Initial;
                return;
            }
            "<" => {
                self.current_token = Token::OP_LT;
                self.current_state = LexerState::Initial;
                return;
            }
            ">" => {
                self.current_token = Token::OP_GT;
                self.current_state = LexerState::Initial;
                return;
            }
            _ => {
                self.current_token = Token::UNDEFINED;
                self.current_state = LexerState::Initial;
                return;
            }
        };
    }
}
```

Parser

- Modified parser from previous assignment
  - Instead of using the same generic node, everything belongs to its own node type.
- Recursive descent parser
  - Continuously follow output stream from the lexer and ensure it conforms to our grammar.

```rust
pub fn analyze(&mut self) -> ProgramNode {
    self.indent = 0;
    self.advance(); // prime the lexer

    let mut program = ProgramNode::new();

    // parse lexer output until we reach the EOI token
    while !self.peek(Token::EOI) {
        match self.curr() {
            Token::KW_FUNC => {
                let func_node = self.parse_func();
                program.func_nodes.push(Rc::new(func_node));
            }
            Token::KW_LET => {
                let let_node = self.parse_let();
                program.let_nodes.push(Rc::new(let_node));
            }
            _ => {
                panic!("Unexpected token `{}` while parsing program.", self.curr())
            }
        }
    }

    self.expect(Token::EOI);

    program
}
```

Chained Expression Parsing

- Not currently using a Pratt parser
- Expressions are parsed as chained expressions
- Does not support parenthesis parsing

Example:

```
x = 5 + 4 + 2;
```

```
Assign(
    AssignNode {
        name: "x",
        expr: Add(
            Val(I32(5)),
            Add(
                Val(I32(4)),
                Val(I32(2)),
            ),
        ),
    },
),
```

```
fn parse_expr(&mut self) → ExprNode {
    self.indent_print( msg: "parse_expr()");
    self.indent_increment();
    let token : Token  = self.curr();

    let expr_node : ExprNode  = match token {...};

    let is_end_of_expr : bool  = match self.curr() {...};

    if is_end_of_expr {...}

    let expr_tail_node : ExprNode  = self.parse_expr_tail(expr_node);
    self.indent_decrement();
    expr_tail_node
}

fn parse_expr_tail(...) → ExprNode {
    self.indent_print( msg: "parse_expr_tail()");
    self.indent_increment();
    let token : Token  = self.curr();

    let expr_node : ExprNode  = match token {...};

    let is_end_of_expr : bool  = match self.curr() {...};

    if is_end_of_expr {...}

    let expr_tail_node : ExprNode  = self.parse_expr_tail(expr_node);
    self.indent_decrement();
    expr_tail_node
}
```

Analysis

- Check for variables used that are not initialized
- Check for unused variables

```
$ ./iron-oxide.exe declaration.tpl
[INFO] Starting Iron Oxide...
[INFO] Analyze.
thread 'main' panicked at src\analyzer.rs:115:25:
Variable 'z' used before declaration in function main!
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

```
$ ./iron-oxide.exe unused_var.tpl
[INFO] Starting Iron Oxide...
[INFO] Analyze.
[WARN] Warning: Variable 'a' declared but not used in function main!
[WARN] Warning: Variable 'z' declared but not used in function test!
[WARN] Warning: Variable 'input' declared but not used in function test!
[INFO] Execute.
[INFO] Execute Program.
[INFO] Program finished.
```

```
func main() [
    z = 5;
]
```

```
func main() [
    let a;
]

func test(input) [
    let z;
]
```

Evaluator

- Previously implemented support for Relational Operators
- Currently, we are using Dr. Ohl's recent Recursive Call Code.
- Operators are enumerated by type to be used with different evaluative functions.

```rust
#[derive(Debug, Clone)]
enum ArithmeticOp {
    Add,
    Sub,
    Mul,
    Div,
}

#[derive(Debug, Clone)]
enum RelationalOp {
    Equal,
    LessThan,
    GreaterThan,
    NotEqual,
    LessThanEqual,
    GreaterThanEqual,
}
```

Previous Implementation

```rust
ExprNode::LessThan(expr_a, expr_b) => {
    let value_a = Self::evaluate(expr_a.clone(), rc_frame.clone());
    let value_b = Self::evaluate(expr_b.clone(), rc_frame.clone());
    Value::Bool(value_a < value_b)
}
ExprNode::GreaterThan(expr_a, expr_b) => {
    let value_a = Self::evaluate(expr_a.clone(), rc_frame.clone());
    let value_b = Self::evaluate(expr_b.clone(), rc_frame.clone());
    Value::Bool(value_a > value_b)
}
ExprNode::EqualTo(expr_a, expr_b ) => {
    let value_a = Self::evaluate(expr_a.clone(), rc_frame.clone());
    let value_b = Self::evaluate(expr_b.clone(), rc_frame.clone());
    Value::Bool(value_a == value_b)
}
ExprNode::LessThanEq(expr_a, expr_b) => {
    let value_a = Self::evaluate(expr_a.clone(), rc_frame.clone());
    let value_b = Self::evaluate(expr_b.clone(), rc_frame.clone());
    Value::Bool(value_a <= value_b)
}
ExprNode::GreaterThanEq(expr_a, expr_b) => {
    let value_a = Self::evaluate(expr_a.clone(), rc_frame.clone());
    let value_b = Self::evaluate(expr_b.clone(), rc_frame.clone());
    Value::Bool(value_a >= value_b)
}
ExprNode::NotEqualTo(expr_a, expr_b) => {
    let value_a = Self::evaluate(expr_a.clone(), rc_frame.clone());
    let value_b = Self::evaluate(expr_b.clone(), rc_frame.clone());
    Value::Bool(value_a != value_b)
}
```

Executor

- Implemented Support for While Loops
- Implemented Support for If-Then-Else Statements
  - Else is optional with use of Rust's Option Type

```rust
StmtNode::IfElse(if_else_node) => {
    Logger::debug("executing if else statement");
    let condition = Evaluator::evaluate(if_else_node.condition.clone(), rc_locals.clone());
    if let Value::Bool(b) = condition {
        if b {
            Logger::debug("executing if body");
            return Self::execute_block_without_scope(if_else_node.ifBody.clone(), rc_locals.clone());
        }
        if !b && if_else_node.elseBody.is_some() {
            Logger::debug("executing else body");
            return Self::execute_block_without_scope(if_else_node.elseBody.clone().unwrap(), rc_locals.clone());
        }
        (Control::Next, Value::Nil)
    } else {
        panic!("If-then-else statement condition must be of type boolean!");
    }
}
```

```rust
#[derive(Debug, Clone)]
pub struct WhileNode {
    pub condition: Rc<ExprNode>,
    pub body: Rc<BlockNode>,
}

impl WhileNode {
    pub fn new(condition: ExprNode, body: BlockNode) -> WhileNode {
        WhileNode {
            condition: Rc::new(condition),
            body: Rc::new(body),
        }
    }
}

#[derive(Debug, Clone)]
pub struct IfElseNode {
    pub condition: Rc<ExprNode>,
    pub ifBody: Rc<BlockNode>,
    pub elseBody: Option<Rc<BlockNode>>,
}

impl IfElseNode {
    pub fn new(condition: ExprNode, ifBody: BlockNode, elseBody: Option<BlockNode>) -> IfElseNode {
        IfElseNode {
            condition: Rc::new(condition),
            ifBody: Rc::new(ifBody),
            elseBody: match elseBody {
                Some(block) => Some(Rc::new(block)),
                None => None,
            },
        }
    }
}
```

CLI

- Currently very basic in functionality
- Accepts a path to an input file
- Optionally a flag to determine the logging level

```
$ ./iron-oxide.exe --help
Iron Oxide Cli

Usage: iron-oxide.exe [OPTIONS] <FILE>

Arguments:
  <FILE>  File to process

Options:
  -l, --loglevel <LOGLEVEL>  Logging level [default: info] [possible values: info, debug, warn, none]
  -h, --help                 Print help
  -V, --version              Print version
```

Encountered Problems

- Usual code merge issues
- Broad scope

What’s Next?

- Better looking command line interfacing
  - Additional flags for more granular control over processes run and output shown.
- Data storage besides single variables (small arrays, enums, structs)
- Possible file writing/storage
- Supporting nested blocks
  - Also includes an issue with blocks of while statements not changing variable values from the previous block.