Home Exam D7050E

Niklas Lundberg
inaule-6@student.ltu.se

January 17, 2021

1 Rubigo-lang Syntax

1.1 EBNF

```
(* Definition of Program *)
Program = Module ;


(* Definition of Module *)
Module = { Statement } ;


(* Definition of Statement *)
    Function = "fn", Identifier, "(", [ Identifier, ":",
        Type_Declaration, { ",", Identifier, ":",
        Type_Declaration } ], ")", "->", Type_Declaration,
        "{", { Statement }, "}" ;
    While = "while", Expression, "{", { Statement }, "}" ;
    If = "if", Expression, "{", { Statement }, "}", [ "else",
        "{", { Statement }, "}" ] ;
    Let = "let", [ Mutable ], Identifier, ":", Type_Declaration,
        "=", Expression, ";" ;
    Assignment = [ Dereference ], Identifier, "=",
        Expression, ";" ;
    Return = "return", Expression, ";" ;
    Function_Call = E_Function_Call,  ";" ;
    Statement = Function | While | If | Let | Assignment |
        Return | Function_Call ;


(* Definition of Expression (E stands for expression) *)
    E_Binary_Operation = Expression, Binary_Operator,
        Expression ;
    E_Unary_Operation = Unary_Operator, Expression ;
    E_Function_Call = Identifier, "(", [ Expression, { ",",
        Expression } ], ")" ;
```

```
    E_Variable = Identifier ;
    E_Borrowed = "&", Expression ;
    E_Dereferenced = "*", Expression ;
    E_Mutable = Mutable, Expression ;
    Expression = E_Binary_Operation | E_Unary_Operation |
        E_Function_Call | E_Variable | Literal | E_Borrowed |
        E_Dereferenced | E_Mutable ;


(* Definition of Type_Declaration *)
    Mutable = "mut" ;
    Borrow = "&" ;
    Dereference = "*" ;
    Type_Declaration = [ Borrow ], [ Mutable ], Type ;


(* Definition of Binary_Operator *)
    Add = "+" ;
    Sub = "-" ;
    Div = "/" ;
    Multi = "*" ;
    Mod = "%" ;
    And = "&&" ;
    Or = "||" ;
    Equal = "==" ;
    Not_Equal = "!=" ;
    Less_Then = "<" ;
    Larger_Then = ">" ;
    Less_Equal_Then = "<=" ;
    Larger_Equal_Then = ">=" ;
    Binary_Operator = Add | Sub ¦ Div ¦ Multi | Mod | And |
        Or | Not | Equal | Not_Equal | Less_Then |
        Larger_Then | Less_Equal_Then | Larger_Equal_Then ;


(* Definition of Unary Operator *)
    Sub = "+" ;
    Not = "!" ;
```

```
    Unary_Operator = Sub | Not ;


(* Definition of Literal (L stands for literal) *)
    L_I32 = Integer ;
    L_F32 = Integer, ".", Natural_Number ;
    L_Bool = True | False ;
    L_Char = "'", Character, "'" ;
    L_String = """, { Character }, """ ;
    Literal = L_I32 | L_F32 | L_Bool | L_Char | L_String;


(* Definition of Type (T stands for type) *)
    T_Int32 = "i32" ;
    T_Float32 = "f32" ;
    T_Bool = "bool" ;
    T_Char = "Char";
    T_String = "String" ;
    Type = T_Int32 | T_Float32 | T_Boolean | T_Char |
        T_String ;


(* General definitions *)
    Digit_Excluding_zero = r[1-9] ;
    Digit = "0" | Digit_Excluding_Zero ;
    Natural_Number = Digit_Excluding_Zero, { Digit } ;
    Integer = "0" | [ "-" ], Natural_Number ;
    Letter = r[ a-ö ] ;
    Symbol = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">"
        | "'" | '"' | "=" | "|" | "." | "," | ";" | "_" | "-" ;
    Character = Letter | Symbol | " " ;
    Identifier = ( Letter | "_" ), { Letter | "_" } ;
```

1.2 Example explination

Code example of a program that prints the 10 first prime numbers.

```
fn is_prime(num: &i32) -> bool {
    if *num < 2 {
        return false;
    }
    let half: i32 = *num/2;
    let mut count: i32 = 2;

    while count <= half {
        if (*num % count) == 0 {
            return false;
        }
        count = count + 1;
    }
    return true;
}


fn print_n_prime(n: &mut i32) -> () {
    let mut count: i32 = 1;
    while *n > 0 {
        if is_prime(&count) {
            print(count);
            *n = *n - 1;
        }
        count = count + 1;
    }
}


let mut number: i32 = 10;
print_n_prime(&mut number);
```

TODO

1.3 Solution compared to requirements

TODO

2 Rubigo-lang Semantics

2.1 SOS

General Defenitions:

$$
\begin{aligned}
i &= \text{Integer} \\
f &= \text{Float} \\
n &\in \{i, f\} \\
b &= \text{Boolean} \\
v &\in \{n, b\} \\
uop &= \text{Unary Operator} \\
bop &= \text{Binary Operator} \\
x &= \text{Variable} \\
p &= \text{Pointer} \\
e &= \text{Expression} \\
stmt &= \text{Statement} \\
\sigma &= \text{State/Memory} \\
fc &= \text{Function Call}
\end{aligned}
$$

Program:

$$\frac{\langle stmt_1, \sigma \rangle \Downarrow \langle void, \sigma' \rangle}{\langle stmt_1; stmt_2; \cdots ; stmt_n, \sigma \rangle \Downarrow \langle stmt_2; \cdots ; stmt_n, \sigma' \rangle}$$

Block:

$$\frac{\langle stmt_1, \sigma \rangle \Downarrow \langle void, \sigma' \rangle}{\langle stmt_1; stmt_2; \cdots ; stmt_n, \sigma \rangle \Downarrow \langle stmt_2; \cdots ; stmt_n, \sigma' \rangle}$$

$$\frac{\langle stmt_1, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle stmt_1; stmt_2; \cdots ; stmt_n, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}$$

Statement:

$$\frac{\langle stmt, \sigma\rangle \Downarrow \langle \mathbf{void}, \sigma'\rangle}{\langle stmt, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle stmt, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle stmt, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}$$

While:

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \mathbf{true}, \sigma'\rangle \quad \langle block, \sigma'\rangle \Downarrow \sigma''}{\langle \mathbf{while}\ e\ \mathbf{do}\ block, \sigma\rangle \Downarrow \langle \mathbf{while}\ e\ \mathbf{do}\ block, \sigma''\rangle}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \mathbf{false}, \sigma'\rangle}{\langle \mathbf{while}\ e\ \mathbf{do}\ block, \sigma\rangle \Downarrow \sigma'}$$

If:

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \mathbf{true}, \sigma'\rangle \quad \langle block_1, \sigma'\rangle \Downarrow \sigma''}{\langle \mathbf{if}\ e\ \mathbf{then}\ block_1\ \mathbf{else}\ block_2, \sigma\rangle \Downarrow \sigma''}$$

$$\frac{\langle e, \sigma\rangle \Downarrow \langle \mathbf{false}, \sigma'\rangle \quad \langle block_2, \sigma'\rangle \Downarrow \sigma''}{\langle \mathbf{if}\ e\ \mathbf{then}\ block_1\ \mathbf{else}\ block_2, \sigma\rangle \Downarrow \sigma''}$$

Return:

$$\frac{\langle e, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \mathbf{return}\ e, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}$$

Let/Assignment:

$$\frac{\langle e, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle x := e, \sigma\rangle \Downarrow \langle \sigma'[x := v]\rangle}$$

$$\frac{\langle e, \sigma \rangle \Downarrow \langle p, \sigma' \rangle}{\langle x := e, \sigma \rangle \Downarrow \langle \sigma'[x := p] \rangle}$$

Function Call:

$$\frac{\langle fc, \sigma \rangle \Downarrow \langle \mathbf{void}, \sigma' \rangle}{\langle fc, \sigma \rangle \Downarrow \sigma'}$$

$$\frac{\langle fc, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle fc, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

Expression:

$$\frac{\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

$$\frac{\langle e, \sigma \rangle \Downarrow \langle p, \sigma' \rangle}{\langle e, \sigma \rangle \Downarrow \langle p, \sigma' \rangle}$$

Binary Operations:

$$\frac{\langle e_1, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad \langle e_2, \sigma' \rangle \Downarrow \langle v_2, \sigma'' \rangle \quad \langle v_1 \textbf{ bop } v_2, \sigma'' \rangle \Downarrow \langle v_3, \sigma'' \rangle}{\langle e_1 \textbf{ bop } e_2, \sigma \rangle \Downarrow \langle v_3, \sigma'' \rangle}$$

Unary Operations:

$$\frac{\langle e, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \quad \langle uop\ v, \sigma' \rangle \Downarrow \langle v', \sigma' \rangle}{\langle uop\ e, \sigma \rangle \Downarrow \langle v', \sigma' \rangle}$$

Borrow Variable:

$$\frac{}{\langle \&x, \sigma \rangle \Downarrow \langle p, \sigma \rangle}$$

Dereference Pointer:

$$\frac{\langle p, \sigma \rangle \Downarrow \langle p', \sigma \rangle}{\langle p, \sigma \rangle \Downarrow \langle p', \sigma \rangle}$$

$$\frac{\langle p, \sigma \rangle \Downarrow \langle v, \sigma \rangle}{\langle p, \sigma \rangle \Downarrow \langle v, \sigma \rangle}$$

Variable:

$$\frac{}{\langle x, \sigma \rangle \Downarrow \langle v, \sigma \rangle}$$

Value:

$$\frac{}{\langle v, \sigma \rangle \Downarrow \langle v, \sigma \rangle}$$

2.2 Example explination

TODO

2.3 Solution compared to requirements

TODO

3 Rubigo-lang Type Checker

3.1 Type Checking Rules

General Defenitions:

$$\begin{aligned} i &= \text{32 bit Integer Type} \\ f &= \text{32 bit Float Type} \\ n &\in \{i, f\} \\ b &= \text{Boolean Type} \\ uop &= \text{Unary Operator} \\ bop &= \text{Binary Operator} \\ x &= \text{Variable} \\ e &= \text{Expression} \\ \sigma &= \text{State/Memory} \end{aligned}$$

While:

$$\frac{\langle e, \sigma \rangle \Downarrow \langle b, \sigma' \rangle}{\langle \textbf{while } e \textbf{ do } block, \sigma \rangle}$$

If:

$$\frac{\langle e, \sigma \rangle \Downarrow \langle b, \sigma' \rangle}{\langle \textbf{if } e \textbf{ then } block_1 \textbf{ else } block_2, \sigma \rangle}$$

Let/Assignment:

$$\frac{\langle x, \sigma \rangle \Downarrow \langle n, \sigma \rangle \quad \langle e, \sigma \rangle \Downarrow \langle n, \sigma' \rangle}{\langle x := e, \sigma \rangle \Downarrow \langle \sigma'[x := n] \rangle}$$

$$\frac{\langle x, \sigma \rangle \Downarrow \langle b, \sigma \rangle \quad \langle e, \sigma \rangle \Downarrow \langle b, \sigma' \rangle}{\langle x := e, \sigma \rangle \Downarrow \langle \sigma'[x := b] \rangle}$$

Arithmetic Binary Operation:

$$\frac{\langle e_1, \sigma\rangle \Downarrow \langle n, \sigma'\rangle \quad \langle e_2, \sigma'\rangle \Downarrow \langle n, \sigma''\rangle \quad \frac{\langle bop, \sigma\rangle \Downarrow \langle op \in \{+.-,*,/,\%\}, \sigma\rangle}{\langle n \ op \ n, \sigma''\rangle \Downarrow \langle n, \sigma'''\rangle}}{\langle e_1 \ bop \ e_2, \sigma\rangle \Downarrow \langle n, \sigma'''\rangle}$$

Boolean Binary Operation:

$$\frac{\langle e_1, \sigma\rangle \Downarrow \langle b, \sigma'\rangle \quad \langle e_2, \sigma'\rangle \Downarrow \langle b, \sigma''\rangle \quad \frac{\langle bop, \sigma\rangle \Downarrow \langle op \in \{\&\&, ||\}, \sigma\rangle}{\langle b \ op \ b, \sigma''\rangle \Downarrow \langle b, \sigma''\rangle}}{\langle e_1 \ bop \ e_2, \sigma\rangle \Downarrow \langle b, \sigma''\rangle}$$

Comparison Binary Operation:

$$\frac{\langle e_1, \sigma\rangle \Downarrow \langle n, \sigma'\rangle \quad \langle e_2, \sigma'\rangle \Downarrow \langle n, \sigma''\rangle \quad \frac{\langle bop, \sigma\rangle \Downarrow \langle op \in \{==, !=, <=, >=, <, >\}, \sigma\rangle}{\langle n \ bop \ n, \sigma''\rangle \Downarrow \langle b, \sigma''\rangle}}{\langle e_1 \ bop \ e_2, \sigma\rangle \Downarrow \langle b, \sigma''\rangle}$$

$$\frac{\langle e_1, \sigma\rangle \Downarrow \langle b, \sigma'\rangle \quad \langle e_2, \sigma'\rangle \Downarrow \langle b, \sigma''\rangle \quad \frac{\langle bop, \sigma\rangle \Downarrow \langle op \in \{==, !=\}, \sigma\rangle}{\langle b \ bop \ b, \sigma''\rangle \Downarrow \langle b, \sigma''\rangle}}{\langle e_1 \ bop \ e_2, \sigma\rangle \Downarrow \langle b, \sigma''\rangle}$$

Arithmetic Unary Operation:

$$\frac{\langle e, \sigma\rangle \Downarrow \langle n, \sigma'\rangle \quad \frac{\langle uop, \sigma\rangle \Downarrow \langle -, \sigma\rangle}{\langle uop \ n, \sigma'\rangle \Downarrow \langle n, \sigma'\rangle}}{\langle uop \ e, \sigma\rangle \Downarrow \langle n, \sigma'\rangle}$$

Boolean Unary Operation:

$$\frac{\langle e, \sigma\rangle \Downarrow \langle b, \sigma'\rangle \quad \frac{\langle uop, \sigma\rangle \Downarrow \langle !, \sigma\rangle}{\langle uop \ b, \sigma'\rangle \Downarrow \langle b, \sigma'\rangle}}{\langle uop \ e, \sigma\rangle \Downarrow \langle b, \sigma'\rangle}$$

3.2 Example explination

TODO

3.3 Solution compared to requirements

TODO

4 Rubigo-lang Borrow Checker

4.1 Borrow Checking Rules

TODO

4.2 Example explination

TODO

4.3 Solution compared to requirements

TODO

5 Rubigo-lang LLVM backend

Currently the LLVM backend has not been implemented for Rubigo-lang.

6 Overall course goals and learning outcomes

I have learnt a lot from implementing a parser and EBNF because I encountered a lot of problems when doing so. I learnt a lot about lexical analysis from those problems, like the order the tokens are parsed is very important.

And a good way to order them is by longest parser first. I didn't learn as much about syntax analysis because my compiler doesn't support context free grammar.

From implementing the type checker I feel i haven't learnt a lot, execpt how complicated it can be to organize the symbol table and keep track of all the identifiers. But I have definitely learnt a lot from making the borrow checker because it was very hard to implement and I have never used Rust before. So I learnt a lot about pointers and how Rust borrow rules work.

The interpreter also made me learn a lot because of how hard it can be to keep track of all the scopes, variables and functions. It made me think of how i could improve my parser, typechecker and borrowchecker so that it could interpret more complex features like pointers.

Overall I think I have learnt a lot about what not to do when designing a compiler and what the challenges are. But I fell that I haven't learned a lot about the theory and solutions to those hard and complex problems. I have also learnt a lot about Rust and how borrowing works.