Home Exam D7050E

Niklas Lundberg
inaule-6@student.ltu.se

January 18, 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 Code

Code example of a rubigo-lang 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);
```

1.3   Solution compared to requirements

My implementation of a lexer and parser is implemented using only the standard rust library and has some advance features like having location information in the error messages. It also has a very basic error recovery system which enables the parser to keep going and find more error in the code. My parser implementation also fulfills all the requirements like parenthesized sub expressions. Thus compared to the requirements my parser is a little bit more advance.

2   Rubigo-lang Semantics

2.1   SOS

General Definitions:

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

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:

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

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

$$\frac{\langle e, \sigma \rangle \Downarrow \langle \mathbf{false}, \sigma' \rangle}{\langle \mathbf{if}\ e\ \mathbf{then}\ block, \sigma \rangle \Downarrow \sigma'}$$

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:

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

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 explanation

Running the rubigo-lang interpreter on the example code above will result in the following:

First the two functions statements *is_prime* and *print_n_prime* will stored in memory, so that they can be called and interpreted later. Then the mutable variable *number* will be assigned a i32 value of 10. This relates to the following SOS rule.

$$\frac{\langle e, \sigma \rangle \Downarrow \langle 10, \sigma' \rangle}{\langle number := e, \sigma \rangle \Downarrow \langle \sigma'[number := 10] \rangle}$$

Next the function *print_n_prime* is called with the pointer to the variable *number*.

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

Next the interpreter jumps in to the function *print_n_prime* and stores the argument n in the memory. After that the mutable variable count is set to 1.

Next the interpreter starts by evaluating the condition in the while statement, which evaluates to *true*. This makes the interpreter interpret the code in the while loop.

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

Next the condition in the if statement is interpreted and it is evaluated to false. The condition expression is a function call to the function *is_prime*. Thus the interpreter will jump into that function, but there is nothing interesting to explain in there. Thus I will skip explaining it.

$$\frac{\langle e, \sigma \rangle \Downarrow \langle \textbf{false}, \sigma' \rangle}{\langle \textbf{if } e \textbf{ then } block, \sigma \rangle \Downarrow \sigma'}$$

After the if statement the variable *count* is increased by one. Then the condition in the while loop is evaluated again with the updated variables. The condition holds true again and the condition in the if statement is evaluated. This time the if condition will be evaluated to true and the interpreter will jump into the if statement code.

$$\frac{\langle e, \sigma \rangle \Downarrow \langle \textbf{true}, \sigma' \rangle \quad \langle block, \sigma' \rangle \Downarrow \sigma''}{\langle \textbf{if } e \textbf{ then } block, \sigma \rangle \Downarrow \sigma''}$$

Next the *print* function is called, the *print* function is hard coded into the interpreter and will print the argument to the console. After that the variable

number is decreased by one. This is done by dereferencing the pointer n and adding one.

$$\frac{\langle p, \sigma \rangle \Downarrow \langle 10, \sigma \rangle}{\langle p, \sigma \rangle \Downarrow \langle 10, \sigma \rangle}$$

Then the program will continue as it has done until 10 prime numbers are printed and thus the condition in the while loop will be false.

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

Next the interpreter will jump back out of the function *print_n_prime* and terminate because there is no more statements to interpret.

2.3 Solution compared to requirements

I have implemented the whole interpreter from scratch and follows all the SOS rules above. If the interpreter encounters a problem it will panic with a appropriate message.

3 Rubigo-lang Type Checker

3.1 Type Checking Rules

General Definitions:

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

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 explanation

Here is an example showing the type checking rule for while loops.

```
let mut i: i32 = 1;
while i < 10 {
    i = i + 1;
}
```

The rule says that the condition expression of the while loop must evaluate into a boolean. In this example the expression $i < 10$ will evaluate into a boolean.

The type checking rule for if statements are the same as for while loops in that the condition expression needs to evaluate into a boolean.

```
if 1 + 10 {
    \\ Code Block
}
```

In this example the condition expression $1 + 10$ in the if statement will evaluate into a i32 of value 11. Thus it does not satisfy the type rule and is not valid if statement.

For let and assignment statements the type rule is a little bit more advanced. The rule for these statements is that the variable and the value stored must be of the same type, but they can be of any type or even a pointer.

```
let var: bool = false;
var = 1;
```

In this example the let statement has a variable and value of type bool, thus it satisfies the rule. But the assignment statement has a variable of type bool and a value of type i32, thus it dose not satisfy the rule. A valid assignment in this case can look like this $var = true$;.

The rules for arithmetic binary operation is that both the expressions evaluate to the same type and that type needs to be a i32 or f32. The result of the operation will also be the same type as the two expressions.

```
let var: f32 = 10.2 / 2;
```

This example shows a valid arithmetic binary operation.

For boolean binary operation the rules are very similar in that both the expressions needs to be of the type boolean. The resulting value will also be a boolean.

```
let var: bool = true || false;
```

This example shows a valid boolean binary operation.

Comparison binary operation rule also results in a boolean value and the two expressions needs to be of the same type. There is also a special case here were if the expressions are of boolean type then only two of the operators are valid.

```
let var: bool = 10 != 2;
```

This example shows a valid comparison binary operation.

The rule for arithmetic unary operation is that the expression evaluates into a i32 or f32 type. Then the result of the operation will be the same type as the expressions type.

```
let var: f32 = -10.2;
```

This example shows a valid arithmetic unary operation.

Lastly the rule for boolean unary operation is that the expression must evaluate to a boolean type and then the result will also evaluate into a boolean type.

```
let var: bool = !true;
```

This example shows a valid boolean unary operation.

3.3   Solution compared to requirements

My type checker rejects all programs that doesn't follow the type rules above. It also has some advanced features like having location information in the error messages and multiple error messages, but it does not have type inference. Thus my type checker is a bit more advanced then the required one, but it does not have all the suggested advanced features.

I have implemented the whole type checker by my self.

4   Rubigo-lang Borrow Checker

4.1   Borrow Checking Rules

My borrow chcker only checks the number of borrow of a variable and the number of mutable borrows. The rule is that a variable can be borrowed any number of times as long as it is not borrowed as mutable at the same time. It is similar with mutable borrows where a variable can be borrowed as mutable as long as no one else is borrowing the variable. Here is a example of whell formed borrows.

```
let mut a: i32 = 10;
let b: &mut i32 = &mut a;
*b = 5;
a = 7;

let c: f32 = 10.2;
let d: &f32 = &c;
```

```
let e: &f32 = &c;
```

My borrow checker doesn't keep track of who is owning the value right now. An example of a error that can occur in my borrow checker is the following.

```
let mut a: i32 = 10;
let b: &mut i32 = &mut a;
a = 7;
*b = 5;
```

In this example the assigning of $a = 7$ should cause an error because the variable b owns the value.

4.2   Example of ill formed borrows that rubigo-lang catches

One of the ill formed borrows my borrow checker catches is when a variable is borrowed and borrowed as mutable.

```
let mut a: i32 = 10;
let b: &mut i32 = &mut a;
let c: &i32 = &a; \\ Error
```

Another case it catches ill formed borrows is when there are multiple mutable borrows of the same variable.

```
let mut a: i32 = 10;
let b: &mut i32 = &mut a;
let c: &mut i32 = &mut a; \\ Error
```

The last type of ill formed borrows that it catches is when borrowing a variable that lives shorter then the variable storing the borrow.

```
let a: i32 = 10;
let mut b: & i32 = & a;
if true {
    let c: i32 = 20;
    b = &c; \\ Error
}
```

4.3   Solution compared to requirements

My borrow checker is very basic and lacks some of the core functionality of a borrow checker, thus it dose not fulfill all the requirements. I have implemented the whole borrow checker my self.

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 about lexical analysis from defining a EBNF and implementing my own lexer. It also made me understand how important it is to tokenize the input before doing the syntax analysis. Implementing the parser also thought me a lot about how important it is to have a well defined abstract syntax tree and that the result can be different depending on the order the parsers are tried.

From implementing the type checker I feel i haven't learnt a lot, except 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 trying to make a borrow checker because it was very hard to do 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, type checker and borrow checker 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. I have also learnt a lot about Rust and how borrowing works.