Home Exam D7050E

Niklas Lundberg
inaule-6@student.ltu.se

February 24, 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:

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

Program:

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

Block:

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

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

While:

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

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

If:

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

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

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

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

Return:

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

Let:

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

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

Assignment:

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

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

Function Call:

$$\frac{\langle e_1, \sigma^0 \rangle \Downarrow \langle v_1, \sigma^1 \rangle \cdots \langle e_n, \sigma^{n-1} \rangle \Downarrow \langle v_n, \sigma^n \rangle, \sigma^{n+1}[a_1 := v_1, \cdots, a_n := v_n], \langle f_c, \sigma^{n+1} \rangle \Downarrow \langle r, \sigma^{n+2} \rangle}{\langle f(e_1, e_2, \cdots, e_n), \sigma^0 \rangle \Downarrow \langle r, \sigma^{n+2} \rangle}$$

Binary Operations:

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

Unary Operations:

$$\frac{\langle e, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle}{\langle \textbf{uop } e, \sigma \rangle \Downarrow \langle v_2, \sigma' \rangle}$$

Variable:

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

$$\frac{\sigma(x) = p}{\langle x, \sigma \rangle \Downarrow p}$$

Borrow:

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

$$\frac{\sigma(p') = p}{\langle \&p, \sigma \rangle \Downarrow p'}$$

Dereference Pointer:

$$\frac{\sigma(p) = v}{\langle *p, \sigma \rangle \Downarrow v}$$

$$\frac{\sigma(p) = p'}{\langle *p, \sigma \rangle \Downarrow p'}$$

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.

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

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

$$\frac{\langle \&number, \sigma \rangle \Downarrow p}{\langle f(\&number), \sigma \rangle \Downarrow \sigma'}$$

Next the interpreter jumps in to the function *print_n_prime* and stores the argument 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.

$$\frac{\langle *num > 0, \sigma\rangle \Downarrow \textbf{true} \quad \langle block, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{while} \ *num > 0 \ \textbf{do} \ block, \sigma\rangle \Downarrow \langle \textbf{while} \ *num > 0 \ \textbf{do} \ block, \sigma'\rangle}$$

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.

$$\frac{\langle is\_prime(\&count), \sigma\rangle \Downarrow \langle \textbf{false}, \sigma'\rangle}{\langle \textbf{if} \ is\_prime(\&count) \ \textbf{then} \ block, \sigma\rangle \Downarrow \sigma'}$$

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.

$$\frac{\langle is\_prime(\&count), \sigma\rangle \Downarrow \langle \textbf{true}, \sigma'\rangle \quad \langle block, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{if} \ is\_prime(\&count) \ \textbf{then} \ block, \sigma\rangle \Downarrow \sigma''}$$

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.

$$\frac{\langle *n, \sigma\rangle \Downarrow 10}{\langle *n = *n - 1, \sigma\rangle \Downarrow \sigma[number = 9]}$$

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.

$$\frac{\langle *n > 0, \sigma\rangle \Downarrow \langle \textbf{false}, \sigma'\rangle}{\langle \textbf{while} \ *n > 0 \ \textbf{do} \ block, \sigma\rangle \Downarrow \sigma'}$$

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:

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

While:

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

If:

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

Let:

$$\frac{\langle e, \sigma \rangle \Downarrow t}{\langle \textbf{let } x : t = e, \sigma \rangle \Downarrow \sigma[x := t]}$$

Assignment:

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

Function Call:

$$\frac{\langle e_1, \sigma \rangle \Downarrow t_1 \cdots \langle e_n, \sigma \rangle \Downarrow t_n, \langle a_1, \sigma \rangle \Downarrow t_1 \cdots \langle a_n, \sigma \rangle \Downarrow t_n}{\langle f(e_1, e_2, \cdots, e_n), \sigma \rangle \Downarrow t}$$

Binary Operation:

$$\frac{\langle e_1, \sigma \rangle \Downarrow t_1 \quad \langle e_2, \sigma \rangle \Downarrow t_1}{\langle e_1 \; bop \; e_2, \sigma \rangle \Downarrow t_2}$$

Unary Operation:

$$\frac{\langle e, \sigma \rangle \Downarrow t_1}{\langle uop \; e, \sigma \rangle \Downarrow t_2}$$

Variable:

$$\frac{\sigma(x) = t}{\langle x, \sigma \rangle \Downarrow t}$$

Borrow:

$$\frac{\sigma(x) = t}{\langle \&x, \sigma \rangle \Downarrow \&t}$$

Dereference Pointer:

$$\frac{\sigma(p) = t}{\langle *p, \sigma \rangle \Downarrow t}$$

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 And Examples of ill formed borrows

I have implemented a stack borrow checker that works by having a stack of borrows for each variable. The stack contains two types of borrows, mutable borrows and normal borrow. If one of borrows are used the borrow checker then checks that the borrow is still present in the borrow stack, if it is not present then the borrow checker will give a error. Changing the value of a variable using a mutable borrow then all the above borrows in the borrow stack will be removed. Here is an example of when this rule will give an error.

```
let mut local: i32 = 42;
let x: &mut i32 = &mut local;
let shared1: &i32 = &local;
let shared2: &i32 = &local;
let mut val: i32 = *x;
val = *shared1;
val = *shared2;
*x = 17 + *x;
val = *shared1;  // Error
```

The stack borrow doesn't cover all the cases thus I added some more rules. One of them is that a variable can't borrow another variable that has a shorter life span. This makes sure that a borrow value doesn't disappear before the pointer, here is an example of where this rule finds an error.

4.2 Example of ill formed borrows that rubigo-lang catches

```
let a: i32 = 3;
let mut b: & i32 = &a;
{
    let c: i32 = 4;
    b = &c;  // Error
}
print(*b);
```

The last rule is that multiple pointers that point to the same variable can't be used in function calls. The reason for this is that it makes it much more complicated to borrow check and it doesn't really limit what you can do with it. Here is an example of were this rule finds an error.

```
fn example(x: &mut i32, y: &mut i32) -> mut i32 {
    *x = 42;
    *y = 14;
    return *x;
}

let mut local: i32 = 5;
let pointer: &mut i32 = &mut local;
let result: i32 = example(pointer, pointer);  // Error
```

4.3 Solution compared to requirements

My solution fulfills the requirements of rejecting ill formed borrows for lexical scoping lifetimes but it does not work for non lexical lifetimes. I have implemented the whole borrow checker by my self using the paper on stack borrows.

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.