Professor Per Lindgren
Date February 15, 2021

Viktor From, 910227 vikfro-6

Compiler construction home exam

Luleå tekniska universitet

1. Syntax

The answers in this home exam has been answered using the Operational Semantics document, see (van Bakel, 2002), as a reference to ENBF and SOS grammar.

1.1 Give an as complete as possible EBNF grammar for your language

Program = { *Expr* };

Int = ? *Rust i*32 ?;

Bool = *"true"* | *"false"*;

Var = ? *Rust String* ?;

AriOp = " + " | " − " | " ∗ " | "/";

AssOp = " = " | "+ = " | "− = " | "/ = ";

LogOp = "&&" | "||";

RelOp = " == " | "! = " | " ≥ " | " ≤ " | " > " | " < ";

Type = *"i*32*"* | *"bool"* | *"void"*;

BinExpr = *IntExpr* | *BoolExpr*

VarExpr = *Expr*, *AssOp*, *Expr*;

IntExpr = *Expr*, *AriOp* | *RelOp*, *Expr*;

BoolExpr = *Expr*, *LogOp* | *RelOp* ,*Expr*;

Parens = "(", *Expr*, ")";

Block = "{", {*Expr*, ";" },"}";

Params = "(", { *Param* { , *Param* } }, ")";

Param = *Var*, " : ", *Type*;

Args = "(", *Arg* , *Arg* , ")";

Arg = *Int* | *Bool* | *Var*;

Let = *"let"*, *Var*, " : ", *Type*, *Expr*;

$If = "if", Expr, Block;$

$IfElse = "if", Expr, Block, Block;$

$While = "while", Expr, Block;$

$Fn = "fn", Var, Params, Type, Block;$

$FnCall = Var, Args;$

$Return = "return", [\ Expr\ ];$

$Expr = Int \mid Var \mid Bool \mid Bin\_Expr \mid Var\_Expr \mid Parens \mid If \mid IfElse \mid While \mid Let \mid Fn \mid FnCall;$

1.2 Give an example that showcases all rules of your EBNF. The program should "do" something as used in the next exercise.

```
fn testfn1(a: bool) -> i32 {
    let b: i32 = (((1 + 2 + 3)));
    if a {
        let c: i32 = 1;
        return c
    } else {
        return (b)
    };
    return 7
}

fn testfn2() -> i32 {
    {{{ return testfn1(true); }}}
}

fn testfn3(d: bool, e: i32) -> i32 {
    let f: bool = d && true;
    let n: i32 = e;
    if f == true {
        n += 1;
        f = false;
    };
    return n;
}

fn main() -> i32 {
    let g: i32 = testfn2();
    let h: i32 = testfn3(true, 1);
```

```
    let i: i32 = g + h;
    return i
}
```

1.3 If you support pointers, make sure your example covers pointers as well.

There is no pointer support in the language at the moment.

1.4 Compare your solution to the requirements (as stated in the README.md). What are your contributions to the implementation.

The parser is implemented using the "Nom, eating data byte by byte" parser combinators library. The parser can handle operator precedence as well as parenthesized sub expressions for both integer and boolean expressions.

2. Semantics

2.1 Give an as complete as possible Structural Operational Semantics (SOS) for your language

Expressions are represented as e_i, numbers as n_i, boolean as b_i, where $i = 1, 2, 3...$ Further, stored programs are represented as σ.

2.1.1 I32

$$\langle e,\ \sigma\rangle \Downarrow n \tag{1}$$

2.1.2 Bool

$$\langle e,\ \sigma\rangle \Downarrow n \tag{2}$$

2.1.3 Var-assignment

$$\frac{}{\langle x := n,\ \sigma\rangle \Downarrow \sigma[x := n]} \tag{3}$$

2.2 Arithmetic operators

2.2.1 Addition

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle e_1 + e_2,\ \sigma\rangle \Downarrow n_1 + n_2} \tag{4}$$

2.2.2 Subtraction

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle e_1 - e_2,\ \sigma\rangle \Downarrow n_1 - n_2} \tag{5}$$

2.2.3 Multiplication

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle e_1 \cdot e_2,\ \sigma\rangle \Downarrow n_1 \cdot n_2} \tag{6}$$

2.2.4 Division

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle \frac{e_1}{e_2},\ \sigma\rangle \Downarrow \frac{n_1}{n_2}} \tag{7}$$

2.3 Logical operators

2.3.1 And

$$\frac{\langle b_1,\ \sigma\rangle \Downarrow false \langle b_2,\ \sigma\rangle \Downarrow true}{\langle b_1 \&\& b_2,\ \sigma\rangle \Downarrow false} \tag{8}$$

$$\frac{\langle b_1,\ \sigma\rangle \Downarrow true \langle b_2,\ \sigma\rangle \Downarrow false}{\langle b_1 \&\& b_2,\ \sigma\rangle \Downarrow false} \tag{9}$$

$$\frac{\langle b_1,\ \sigma\rangle \Downarrow false \langle b_2,\ \sigma\rangle \Downarrow false}{\langle b_1 \&\& b_2,\ \sigma\rangle \Downarrow false} \tag{10}$$

$$\frac{\langle b_1,\ \sigma\rangle \Downarrow true \langle b_2,\ \sigma\rangle \Downarrow true}{\langle b_1 \&\& b_2,\ \sigma\rangle \Downarrow true} \tag{11}$$

2.3.2 Or

$$\frac{\langle b_1,\ \sigma\rangle \Downarrow false \langle b_2,\ \sigma\rangle \Downarrow true}{\langle b_1 || b_2,\ \sigma\rangle \Downarrow true} \tag{12}$$

$$\frac{\langle b_1,\ \sigma\rangle \Downarrow true \langle b_2,\ \sigma\rangle \Downarrow false}{\langle b_1 || b_2,\ \sigma\rangle \Downarrow true} \tag{13}$$

$$\frac{\langle b_1,\ \sigma\rangle \Downarrow true \langle b_2,\ \sigma\rangle \Downarrow true}{\langle b_1 || b_2,\ \sigma\rangle \Downarrow true} \tag{14}$$

$$\frac{\langle b_1,\ \sigma\rangle \Downarrow false \langle b_2,\ \sigma\rangle \Downarrow false}{\langle b_1 || b_2,\ \sigma\rangle \Downarrow false} \tag{15}$$

2.4 Relational operators

2.4.1 Equal

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle e_1 == e_2,\ \sigma\rangle \Downarrow n_1 == n_2} \tag{16}$$

2.4.2 Not equal

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle e_1 == e_2,\ \sigma\rangle \Downarrow n_1 == n_2} \tag{17}$$

2.4.3 Lesser than

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle e_1 < e_2,\ \sigma\rangle \Downarrow n_1 < n_2} \tag{18}$$

2.4.4 Greater than

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle e_1 > e_2,\ \sigma\rangle \Downarrow n_1 > n_2} \tag{19}$$

2.4.5 Lesser than equal

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle e_1 \leq e_2,\ \sigma\rangle \Downarrow n_1 \leq n_2} \tag{20}$$

2.4.6 Greater than equal

$$\frac{\langle e_1,\ \sigma\rangle \Downarrow n_1 \langle e_2,\ \sigma\rangle \Downarrow n_2}{\langle e_1 \geq e_2,\ \sigma\rangle \Downarrow n_1 \geq n_2} \tag{21}$$

2.5 If-statement

$$\frac{\langle b,\ \sigma\rangle \Downarrow true\ \ \langle c,\ \sigma\rangle \Downarrow \sigma''}{\langle \text{if b then c},\ \sigma\rangle \Downarrow \sigma''} \tag{22}$$

$$\frac{\langle b,\ \sigma\rangle \Downarrow false\ \ \langle c,\ \sigma\rangle \Downarrow \sigma''}{\langle \text{if b then c},\ \sigma\rangle \Downarrow \sigma''} \tag{23}$$

2.6 If-else-statement

$$\frac{\langle b,\ \sigma\rangle \Downarrow true\ \ \langle c_1,\ \sigma\rangle \Downarrow \sigma' \langle c_2,\ \sigma\rangle \Downarrow \sigma''}{\langle \text{if b then } c_1, \text{ else } c_2,\ \sigma\rangle \Downarrow \sigma''} \tag{24}$$

$$\frac{\langle b,\ \sigma\rangle \Downarrow false\ \ \langle c_1,\ \sigma\rangle \Downarrow \sigma' \langle c_2,\ \sigma\rangle \Downarrow \sigma''}{\langle \text{if b then } c_1 \text{ else } c_2,\ \sigma\rangle \Downarrow \sigma''} \tag{25}$$

2.7 While-statement

$$\frac{\langle b,\ \sigma\rangle \Downarrow true\ \ \langle c,\ \sigma\rangle \Downarrow \sigma' \langle \text{while b do c},\ \sigma'\rangle \Downarrow \sigma''}{\langle \text{while b do c},\ \sigma\rangle \Downarrow \sigma''} \tag{26}$$

$$\frac{\langle b,\ \sigma\rangle \Downarrow false}{\langle \text{while b do c},\ \sigma\rangle \Downarrow \sigma} \tag{27}$$

2.8 Command

Where C_i range over programs of the language, big-step iteration over a scope,

$$\frac{\langle C_1,\ \sigma\rangle \Downarrow \sigma'}{\langle C_1;\ C_2,\ \sigma\rangle \Downarrow \langle C_2,\ \sigma'\rangle} \tag{28}$$

2.9 Function call

Where a_i is the argument of a function call, r the return value and f_c is the function call,

$$\frac{(\langle e_1,\ \sigma\rangle \Downarrow n_1\ \ldots\ \langle e_i,\ \sigma\rangle \Downarrow n_i),\ \sigma' = [a_1 := n_1,\ \ldots,\ a_i := n_i],\ \langle f_c,\ \sigma'\rangle \Downarrow r}{\text{call f}(e_1, \ldots, e_i)\ \ \Downarrow r} \tag{29}$$

2.10 Return

$$\frac{\langle e,\ \sigma\rangle \Downarrow r}{\langle \text{return e},\ \sigma\rangle \Downarrow r} \tag{30}$$

2.11 Explain (in text) what an interpretation of your example should produce, do that by dry running your given example step by step. Relate back to the SOS rules. You may skip repetitions to avoid cluttering.

Looking at the example, see 1.2, the code can be explained using SOS. The example program starts from the "main" function by going through the AST and executing each function before it returns the value of said functions.

1. testfn1(a: bool)
   (a) A variable "b" is assigned to 6, see eq. 3.
   (b) A variable "a" is retrieved from memory and compared, see eq. 24.
   (c) A variable "c" is assigned to 1, see eq. 3.
   (d) Either variable "b", "c" or 7 is returned (variable "c" in this case), see eq. 30.
2. testfn2()
   (a) The function call of testfn1(true) is returned with the use of parameter injection, see eq. 30 and eq. 29.
3. testfn3(d: bool, e: bool)
   (a) A variable "f" retrieves the parameter value of "d" and sets the value to true, see eq. 3.
   (b) A variable "n" is assigned to the value of "e", see eq. 3.
   (c) The value of "f" is retrieved from memory and compared, see eq. 22.
   (d) The value of "n" is incremented by 1, see eq. 27.
   (e) The value of "f" is set to false, see eq. 4.
   (f) Variable "n" is returned, see eq. 30.
4. main()
   (a) A variable "g" is assigned to the return value of testfn2(), see eq. 3.
   (b) A variable "h" is assigned to the return value of testfn3(true, 1) with the use of parameter injection, see eq. 3 and eq. 29.
   (c) A variable "i" is assigned to the value of "g" and "h", see eq. 3.
   (d) Finally the value of "i" is returned, see eq. 30.

2.12 Compare your solution to the requirements (as stated in the README.md). What are your contributions to the implementation.

The interpreter should be able to interpret any code following the SOS documentation stated above or panic otherwise.

2.13 Typechecker

Where e_i represents the expression of each statement.

2.13.1 Arithmetic operations

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\Gamma e_1 + e_2 : i32} \tag{31}$$

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\Gamma e_1 - e_2 : i32} \tag{32}$$

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\Gamma e_1 \cdot e_2 : i32} \tag{33}$$

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\frac{\Gamma e_1}{e_2} : i32} \tag{34}$$

2.13.2 Logical operations

$$\frac{\Gamma e_1 : bool \;\; \Gamma e_2 : bool}{\Gamma e_1 \&\& e_2 : i32} \tag{35}$$

$$\frac{\Gamma e_1 : bool \;\; \Gamma e_2 : bool}{\Gamma e_1 || e_2 : i32} \tag{36}$$

2.13.3 Relational operations

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\Gamma e_1 \leq e_2 : bool} \tag{37}$$

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\Gamma e_1 < e_2 : bool} \tag{38}$$

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\Gamma e_1 \geq e_2 : bool} \tag{39}$$

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\Gamma e_1 == e_2 : bool} \tag{40}$$

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\Gamma e_1 == e_2 : bool} \tag{41}$$

$$\frac{\Gamma e_1 : bool \;\; \Gamma e_2 : bool}{\Gamma e_1 == e_2 : bool} \tag{42}$$

$$\frac{\Gamma e_1 : i32 \;\; \Gamma e_2 : i32}{\Gamma e_1 != e_2 : bool} \tag{43}$$

$$\frac{\Gamma e_1 : bool \;\; \Gamma e_2 : bool}{\Gamma e_1 != e_2 : bool} \tag{44}$$

2.13.4 Assignment

$$\frac{\Gamma x : bool \;\; \Gamma e : i32}{\Gamma x := e : i32} \tag{45}$$

$$\frac{\Gamma x : bool \;\; \Gamma e : bool}{\Gamma x := e : bool} \tag{46}$$

2.13.5 If-statement

Where b is the condition of and if/else-statement,

$$\frac{}{\Gamma b : bool} \tag{47}$$

2.13.6 While-statement

Where b is the condition of and while-statement,

$$\frac{}{\Gamma b : bool} \tag{48}$$

2.13.7 Function call

Where a_i is the argument type of a function call,

$$\frac{}{\Gamma e_1 = \Gamma a_1 \;\; ... \;\; \Gamma e_i = \Gamma a_i} \tag{49}$$

2.13.8 Function return type

Where t is the return type of a function,

$$\frac{\Gamma return \;\; e == t}{t} \tag{50}$$

2.14 Demonstrate the cases of ill formed borrows that your borrow checker is able to detect and reject.

2.14.1 Arithmetic operations

```
let a = 1 + 2; // has the type Int
let b = 1 - 2; // has the type Int
let c = 1 * 2; // has the type Int
let d = 1 / 2; // has the type Int
let e = 1 + bool; // panics
```

2.14.2 Logical operations

```
let a = true && true; // has the type Bool
let b = true || false; // has the type Bool
let c = true || 5; // panics
```

2.14.3 Relational operations

```
let a = true == true; // has the type Bool
let b = true != false; // has the type Bool
let c = 1 < 2; // has the type Bool
let d = 1 > 2; // has the type Bool
let e = 1 <= 2; // has the type Bool
let f = 1 >= 2; // has the type Bool
let g = 1 == true; // panics
```

2.14.4 Assignment

```
let a: i32 = 1 // has the type Int
let b: bool = true // has the type Bool
```

2.14.5 If-statement

```
if true { ... } // runs the code inside the scope
if false { ... } // does not runs the code inside the scope
if (1 < 2) { ... } // runs the code inside the scope
```

2.14.6 If-statement

```
if true { ... } else { ... } // runs the code inside the first scope
if false { ... } else { ... } // runs the code inside the second scope
```

2.14.7 while-statement

```
while true { ... } // runs the code inside the scope
while false { ... } // does not runs the code inside the scope
while (1 < 2) { ... } // runs the code inside the scope
```

2.14.8 Function declaration

```
fn testfn(a: i32, b: bool) { ... } // runs the code inside the scope if correct type args.
```

2.14.9 Function declaration

```
return 1; // returns a value Int(1)
return 1 + 2 + 3; // returns a value Int(6)
return true; // returns a value Bool(true)
return true && true; // returns a value Bool(true)
return testfn(); // returns the value returned from the fn.
return testfn(true, 1); // Fn with input parameters
```

2.15 Compare your solution to the requirements (as stated in the README.md). What are your contributions to the implementation.

The type checker will detect if the input-program is valid according to the type rules. However, the program will simply panic and tell the user the program did not compile if it's invalid. Specific information about where and what went wrong is not detailed.

3. Borrow checker

3.1 Give a specification for well versus ill formed borrows. (What are the rules the borrow checker should check)

The borrow checkers main task is to verify if a variable is mutable or non-mutable. There can be as many references as you'd like, since none of them are writing. However, as we can only have one &mut at a time it is impossible to get a data race, which is the reason Rust prevents such issues to occur at compile time. Further, it should also check that the reference does not outlive the reference of the variable, see (Steve Klabnik, 2018).

3.2 Compare your solution to the requirements (as stated in the README.md). What are your contributions to the implementation?

Neither borrow checker or references has been implemented in this project.

4. LLVM

4.1 Let your backend produces LLVM-IR for your example program

```
define i32 @testfn1(i1 %0) {
testfn1:
  %c = alloca i32, align 4
  %b = alloca i32, align 4
  %a = alloca i32, align 4
  store i1 true, i32* %a, align 1
  %a1 = load i32, i32* %a, align 4
  store i32 5, i32* %b, align 4
  %a2 = load i32, i32* %a, align 4
  br i32 %a2, label %block1, label %block2

block1:                                           ; preds = %testfn1
  store i32 1, i32* %c, align 4
  %c3 = load i32, i32* %c, align 4
  ret i32 %c3
  br label %cont

block2:                                           ; preds = %testfn1
  %b4 = load i32, i32* %b, align 4
  ret i32 %b4
  br label %cont

cont:                                             ; preds = %block2, %block1
  %iftmp = phi i32 [ 1, %block1 ], [ 0, %block2 ]
  ret i32 7
}

define i32 @testfn2() {
```

```
testfn2:
  %testfn1 = call i32 @testfn1(i1 true)
  ret i32 %testfn1
}

define i32 @testfn3(i1 %0, i32 %1) {
testfn3:
  %n = alloca i32, align 4
  %f = alloca i1, align 1
  %e = alloca i32, align 4
  %d = alloca i32, align 4
  store i1 true, i32* %d, align 1
  %d1 = load i32, i32* %d, align 4
  store i32 1, i32* %e, align 4
  %e2 = load i32, i32* %e, align 4
  %d3 = load i32, i32* %d, align 4
  store i32 %d3, i1* %f, align 4
  %e4 = load i32, i32* %e, align 4
  store i32 %e4, i32* %n, align 4
  %f5 = load i1, i1* %f, align 1
  %Eq = icmp eq i1 %f5, true
  br i1 %Eq, label %then, label %cont

then:                                             ; preds = %testfn3
  %n6 = load i32, i32* %n, align 4
  %add = add i32 %n6, 1
  store i32 %add, i32* %n, align 4
  %f7 = load i1, i1* %f, align 1
  store i1 false, i1* %f, align 1
  br label %cont

cont:                                             ; preds = %then, %testfn3
  %if = phi i32 [ 1, %then ], [ 0, %cont ]
  %n8 = load i32, i32* %n, align 4
  ret i32 %n8
}

define i32 @main() {
main:
  %i = alloca i32, align 4
  %h = alloca i32, align 4
  %g = alloca i32, align 4
  %testfn2 = call i32 @testfn2()
  store i32 %testfn2, i32* %g, align 4
  %testfn3 = call i32 @testfn3(i1 true, i32 1)
  store i32 %testfn3, i32* %h, align 4
  %g1 = load i32, i32* %g, align 4
  %h2 = load i32, i32* %h, align 4
  %add = add i32 %g1, %h2
  store i32 %add, i32* %g, align 4
```

```
  %g3 = load i32, i32* %g, align 4
  store i32 %g3, i32* %i, align 4
  %i4 = load i32, i32* %i, align 4
  ret i32 %i4
}
llvm-result: 3
```

4.1.1 Describe where and why you introduced allocations and phi nodes?

Phi node allocations are performed for function declarations and let-statements when compiled. LLVM places a "alloca"-instruction at the start of each block within a let-statement belongs. With function declarations the "alloca"-instruction is position at the start of the function scope. No phi nodes were added.

4.1.2 If you have added optimization passes and/or attributed your code for better optimization (using e.g., noalias

No optimization passes have been added.

4.1.3 Compare your solution to the requirements (as stated in the README.md). What are your contributions to the implementation?

LLVM is able to handle all statements going through type-checker and interpreter. However, no optimization has been performed. Code from Per Lindgren and Inkwells kaleidoscope example, see (TheDan64, 2020) has been used to complete this part of the project.

5. Overall course goals and learning outcomes

I've learnt how to build a parser in order to put together an abstract syntax tree, an interpreter, a type-checker and how to put together a working LLVM back-end. My knowledge about EBNF and SOS is on the other hand rather limited. However, this is very understandable as only a small portion of the course was spent on this part.

6. References

Steve Klabnik, C. N. (2018). *The rust programming language.* Retrieved from `https://doc.rust-lang.org/1.8.0/book/README.html` (2020-12-16)

TheDan64, s. (2020). *Inkwell, kaleidoscope.* Retrieved from `https://github.com/TheDan64/inkwell/tree/master/examples/kaleidoscope` (2020-12-16)

van Bakel, S. (2002). *Operational semantics.* Department of Computing Imperial College of Science, Technology and Medicine. Retrieved from `http://www.doc.ic.ac.uk/~svb/AssuredSoftware/notes.pdf` (2020-11-21)