My First Compiler - Home Exam

Compiler Construction and Formal Languages,
D7050E

Alexander Mennborg
alemen-6@student.ltu.se

March 16, 2021

`https://github.com/Aleman778/First-Compiler`

Contents

1 Syntax

1.1 EBNF Grammar

```
⟨file⟩      ::= { ⟨function⟩ }

⟨function⟩  ::= ‘fn’ ⟨ident⟩ ‘(’ [⟨fn-arg⟩ {‘,’ ⟨fn-arg⟩ }] [‘->’ ⟨type⟩] ⟨block⟩

⟨fn-arg⟩    ::= [‘mut’] ⟨ident⟩ ‘:’ ⟨type⟩

⟨statement⟩ ::= ‘let’ ⟨ident⟩ ‘:’ ⟨type⟩ [‘=’ ⟨expr⟩] ‘;’
             |  ⟨block⟩ ‘;’
             |  ⟨expr⟩ ‘;’
             |  ⟨expr⟩

⟨block⟩     ::= ‘{’ { ⟨statement⟩ } ‘}’

⟨expr⟩      ::= ⟨atom⟩ ‘=’ ⟨expr⟩
             |  ‘if‘ <expr> <block> [‘else’ ⟨block⟩]
             |  ‘while’ ⟨expr⟩ ⟨block⟩
             |  ⟨block⟩
             |  ‘return’ ⟨expr⟩
             |  ‘break’
             |  ‘continue’
             |  ⟨atom⟩ { ⟨binop⟩ ⟨atom⟩ }
             |  ⟨atom⟩

⟨atom⟩      ::= ⟨literal⟩
             |  ‘(’ ⟨expr⟩ ‘)’
             |  ⟨ident⟩ ‘(’ [⟨expr⟩ { ‘,’ ⟨expr⟩ }] ‘)’
             |  ⟨ident⟩
             |  ⟨unop⟩ ⟨expr⟩
             |  ⟨ref⟩ ⟨expr⟩

⟨literal⟩   ::= regexp[0-9]+
             |  ‘true’
             |  ‘false’
```

⟨*ident*⟩ ::= regexp[a-zA-Z][a-zA-Z0-9_]*

⟨*binop*⟩ ::= ‘`==`’ | ‘`!=`’ | ‘`<`’ | ‘`<=`’ | ‘`>`’ | ‘`>=`’ | ‘`&&`’ | ‘`||`’ | ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’ | ‘`%`’

⟨*unop*⟩ ::= ‘`!`’ | ‘`-`’ | ‘`*`’

⟨*ref*⟩ ::= ‘`&`’ [‘`mut`’]

⟨*type*⟩ ::= ‘`i32`’ | ‘`bool`’ | ⟨*ref*⟩ ⟨*type*⟩

1.2 Demo Code

```
fn main() -> i32 {
    let x: &mut i32 = &mut 0;
    let mut i: i32 = 0;
    while i < 10 {
        i = i + 1;
        next_prime(x);
        print_int(*x);
    }
    return *x;
}

fn next_prime(n: &mut i32) {
    while true {
        *n = *n + 1;
        if is_prime(*n) {
            break;
        }
    }
}

fn is_prime(n: i32) -> bool {
    if n < 2 {
        return false;
    }

    let mut i: i32 = 2;
    while i < n {
        if n % i == 0 {
            return false;
        }
        i = i + 1;
    }

    true
}
```

where *print_int* is an intrinsic to print an *i32* to the console.

1.3 Requirements and Contributions

The parser implements all the listed requirements and more:

- Function definitions
- Commands (let, assignment, if then (else), while)
- Expressions (includig function calls)

- Primitive types (boolean, i32) and their literals
- Explicit types everywhere
- Explicit return(s)
- Operator precedence *(optional)*
- Location information *(optional)*

The only missing optional feature is error recovery thus only a single error can be reported during the entire parsing stage. There is also an issue with the error locations when they are reported from within the parser. For example missing a semicolon the **nom** parser combinator **Tag** is reported as the error message and but somewhere in unwinding the call stack the location information is lost. For example running this results in location pointing to the first character in the function:

```
firstc -r "fn main() { let x: i32 = 5 }"
<run>:1:1: error: Tag
  |
1 | fn main() { let x: i32 = 5 }
  | ^
```

However, it works fine when the error is outside the parser e.g. type errors are able to correctly locate the error:

```
firstc -r "fn main() { let x: bool = 5; }"
<run>:1:27: error: expected `bool`, found `i32`
  |
1 | fn main() { let x: bool = 5; }
  |                           ^
```

When designing the syntax for this language the Rust language was used as a guide for most of the decisions made because the goal was essentially to create a mini Rust language. Like Rust this language syntax treats most things as expressions even **if** and **while** which normally are statements (or commands) because they modify the state of the program. The only real statement in this language and like Rust is the **let** statement.

There is support for defining foreign function interfaces in the syntax but this was not actually used in the final compiler and was instead replaced by hardcoded intrinsics. That is why they were left out from the **EBNF** grammar but can still be parsed however there is no support for loading dynamic libraries so this feature is not that usefull. There are explicit return(s) in the language but like Rust implicit return(s) were also added for example:

```
firstc -r "fn main() -> i32 { 10 }"

Interpreter exited with code 10
```

Since **if** is an expression together with the fact that blocks can return values to the its parent block means that conditional ternary operator (i.e. x = true ? 10 : 20) is supported, like this:

```
firstc -r "fn main() -> i32 { let x: i32 = if true { 10 } else { 20 }; x + 5 }"

Interpreter exited with code 15
```

Another contribution that a string interner is used to avoid unnecessary copies of strings and this is used for function names, identifiers and even internal temporary names used by the compiler. This works by essentially storing each string onces in a permanent storage that only gets freed when the compiler exits. Storing references to a string is done through using symbols (i.e. integer id) and can be resolved its original string through a hash table lookup. This adds a cost for looking up names but since this only really is used for error handling it is not a big problem. It also reduces the memory footprint since string clones was previously used extensively to avoid complex borrowing situation between systems which now is replaced by cheap integer copy. This was implemented using a library called *string-interner* which provides the permanent storage and hash table lookup.

2 Semantics

2.1 Structural Operational Semantics

Evaluation of a statement s in the state σ yields new state σ':

$$\langle s, \sigma \rangle \Downarrow \sigma' \tag{1}$$

For let bindings the expression e is evaluated first to n and then assigned to the actual variable x.

$$\frac{\langle e, \sigma \rangle \Downarrow n}{\langle \textbf{let } x = e, \sigma \rangle \Downarrow \sigma[x = n]} \tag{2}$$

After previously binding a variable x if it was annotated as mutable it is possible to assign a new value to it.

$$\frac{\langle e, \sigma[x = n_1] \rangle \Downarrow n_2}{\langle x = e, \sigma[x = n_1] \rangle \Downarrow \sigma[x = n_2]} \tag{3}$$

Blocks may contain multiple statements and they have to be evaluated in the same order that they were defined in. For two statements it is important that the first is evaluated before the other statement because it needs use the new state produced by the first statement evaluation.

$$\frac{\langle s_1, \sigma \rangle \Downarrow \sigma' \;\; \langle s_2, \sigma' \rangle \Downarrow \sigma''}{\langle s_1; s_2, \sigma \rangle \Downarrow \sigma''} \tag{4}$$

Note that if-expressions are syntactically expressions but the semantics behaves more like statements because they change the state of σ. If the condition c evaluates to true then block b_1 will be evaluated and resulting in the new state σ'

$$\frac{\langle c, \sigma \rangle \Downarrow \mathit{true} \;\; \langle b_1, \sigma \rangle \Downarrow \sigma'}{\langle \textbf{if } c \ \{\ b_1\ \} \textbf{ else } \{\ b_2\ \}, \sigma \rangle \Downarrow \sigma'} \tag{5}$$

If the condition c evaluates to false then block b_2 will be evaluated and result in state σ''.

$$\frac{\langle c, \sigma \rangle \Downarrow \mathit{false} \;\; \langle b_2, \sigma \rangle \Downarrow \sigma''}{\langle \textbf{if } c \ \{\ b_1\ \} \textbf{ else } \{\ b_2\ \}, \sigma \rangle \Downarrow \sigma''} \tag{6}$$

While-loops also changes the state in a similar way except repeats, for the case where the condition c evaluates to false then nothing happens to the state.

$$\frac{\langle c, \sigma \rangle \Downarrow \mathit{false}}{\langle \textbf{while } c \ \{\ b\ \}, \sigma \rangle \Downarrow \sigma} \tag{7}$$

If the condition c evaluates to true then the semantics is recursive.

$$\frac{\langle c, \sigma \rangle \Downarrow \mathit{true} \;\; \langle b, \sigma \rangle \Downarrow \sigma' \;\; \langle \mathbf{while}\ c\ \{\ b\ \}, \sigma' \rangle \Downarrow \sigma''}{\langle \mathbf{while}\ c\ \{\ b\ \}, \sigma \rangle \Downarrow \sigma''} \tag{8}$$

Continue-expressions can only affect the current while-loop if the condition c is true. Also continue may appear anywhere inside the block b so it only gets partially evaluated denoted as b'.

$$\frac{\langle c, \sigma \rangle \Downarrow \mathit{true} \;\; \langle b', \sigma \rangle \Downarrow \sigma' \;\; \langle \mathbf{while}\ c\ \{\ b\ \}, \sigma' \rangle \Downarrow \sigma''}{\langle \mathbf{continue}, \sigma \rangle \Downarrow \sigma''} \tag{9}$$

For break-expressions it simply terminates the loop and like continue-expressions it may have partially evaluated b.

$$\frac{\langle b', \sigma \rangle \Downarrow \sigma'}{\langle \mathbf{break}, \sigma \rangle \Downarrow \sigma'} \tag{10}$$

Return-expressions terminates the entire function and can optionally return something.

$$\frac{\langle f, \sigma \rangle \Downarrow n \;\; \langle e, \sigma \rangle \Downarrow n}{\langle \mathbf{return}\ e \in f, \sigma \rangle \Downarrow n} \tag{11}$$

Example of binary addition operation uses an implementation specific function called *add* which usually translates into an *ADD* instruction in most processor architectures.

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \;\; \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow \ \mathrm{add}(n_1, n_2)}. \tag{12}$$

Calling function f cannot modify our state σ since there is only support for local variables, unless mutable references are passed into the function. The function block $f.b$ is evaluated in a new state σ' and may return r and the parameter expressions are $p_1, p_2, ..., p_n$.

$$\frac{\langle p_1, \sigma \rangle \Downarrow n_1 \;\; \langle p_2, \sigma \rangle \Downarrow n_2 \;\; \cdots \;\; \langle p_n, \sigma \rangle \Downarrow n_n \;\; \langle f.b, \sigma' \rangle \Downarrow r}{\langle f(n_1, n_2, ..., n_n), \sigma' \rangle \Downarrow r}. \tag{13}$$

Identifier x evaluates to local variable hash table lookup.

$$\frac{}{\langle x, \sigma[x = n] \rangle \Downarrow n} \tag{14}$$

Unary negation operation uses an implementation specific function called *neg* which usually translates into an *NEG* or $(0 - n)$ *SUB* instruction in most processor architectures.

$$\frac{\langle e, \sigma \rangle \Downarrow n}{\langle -e, \sigma \rangle \Downarrow \mathrm{neg}(n)}. \tag{15}$$

Borrows (&) evaluates to the memory location denoted by star symbol ($*$) of a particular variable x

$$\frac{}{\langle \&x, \sigma[x = n]\rangle \Downarrow *x} \tag{16}$$

It is also possible to borrow from a literal value v however in order to get a valid memory location it has to first be stored in memory as a temporary variable

$$\frac{}{\langle \&v, \sigma\rangle \Downarrow < *temp, \sigma[temp = v] >} \tag{17}$$

Unary dereference operation on identifier x finds the underlying value that the particular borrow points to and the operation evaluates to that value. For nested borrows this only dereferences one borrow at a time.

$$\frac{\langle y, \sigma[y = *x, x = n]\rangle \Downarrow *x}{\langle *y, \sigma[y = *x, x = n]\rangle \Downarrow n}. \tag{18}$$

2.2 Interpreting Demo Code

The state of the interpreter is represented by the *InterpContext* struct which holds things like the AST (or *File* struct), hash map of functions, callstack and the memory. The memory acts as a stack and gets automatically cleaned up after each function. When parsing is done the AST (or *File* struct) is produced which contains a list of functions that first has to be inserted into *InterpContext* hash map for access when calling functions. After this is done calling *interp_entry_point* will find the *main* function and start executing its block.

The first statement in *main* uses rules (17) and (2).

```
    let x: &mut i32 = &mut 0;
```

$$\frac{\langle \&\text{mut } 0, \sigma\rangle \Downarrow \langle *temp, \sigma[temp = 0]\rangle}{\langle \textbf{let } x = \&\text{mut } 0, \sigma\rangle \Downarrow \sigma[x = *temp, temp = 0]}.$$

The second statement evaluates to state $\sigma[x = *temp, temp = 0, i = 0]$, the third statement is a while loop which uses rules (7) when $i >= 10$ and (8) otherwise.

```
    while i < 10 {
```

$$\frac{\langle i < 10, \sigma[i >= 10]\rangle \Downarrow false}{\langle \textbf{while } c \ \{ \ b \ \}, \sigma\rangle \Downarrow \sigma}$$

$$\frac{\langle i < 10, \sigma[i < 10]\rangle \Downarrow true \quad \langle b, \sigma\rangle \Downarrow \sigma' \quad \langle \textbf{while } c \ \{ \ b \ \}, \sigma'\rangle \Downarrow \sigma''}{\langle \textbf{while } c \ \{ \ b \ \}, \sigma\rangle \Downarrow \sigma''}$$

The first statement inside the while loop increments the iterator variable i with one using semantic rules (3), (12) and (14) togheter.

```
        i = i + 1;
```

$$\frac{\langle i, \sigma[i = n]\rangle \Downarrow n \quad \langle i + 1, \sigma[i = n]\rangle \Downarrow \ \text{add}(n, 1)}{\langle i = i + 1, \sigma[i = n]\rangle \Downarrow \sigma[i = \ \text{add}(n, 1)]}$$

Next is a function call to *next_prime* which uses semantic rules (13) and (14). Calling the function will produce a new state since x is passed as a mutable reference.

```
        next_prime(x);
```

$$\frac{\langle x, \sigma[x = n]\rangle \Downarrow n}{\langle \text{next\_prime}(x), \sigma\rangle \Downarrow \sigma'}.$$

Inside *next_prime* function there is an infinite loop that increments the argument by one each time. The loop only terminates if the value is a prime number, this uses semantic rule (5). The *is_prime* is quite self explanatory so that is skipped to keep this explanation short.

```
        if is_prime(*n) {
```

$$\frac{\langle \text{is\_prime}(*n), \sigma\rangle \Downarrow \mathit{true} \quad \langle b_1, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{if}\ \text{is\_prime}(*n)\ \{\ b_1\ \}, \sigma\rangle \Downarrow \sigma'}$$

Inside the if-expression there is a break expression that will terminate the infinte loop whenever evaluated, defined by semantic rule (10).

```
              break;
```

$$\frac{\langle b', \sigma\rangle \Downarrow \sigma'}{\langle \textbf{break}, \sigma\rangle \Downarrow \sigma'}$$

Back to *main* when $\sigma[i = 10]$ the loop terminates and the resulting value of x is dereferenced and returned. This uses the semantics rules (11) and (18).

```
    return *x;
```

$$\frac{\langle *y, \sigma[y = *temp, temp = 29]\rangle \Downarrow 29}{\langle \textbf{return}\ \ * y \in f, \sigma[y = *temp, temp = 29]\rangle \Downarrow 29}$$

2.3 Requirements and Contributions

- Your interpreter should be able to correctly execute programs according to your SOS.

- Your interpreter should panic (with an appropriote error message) when encountering an evaluation error (e.g., 1 + false)

The interpreter was implemented before writing the SOS rules so the SOS rules were formed based on the code. While the interpreter by itself can report type errors it should not panic but instead report a maximum of one fatal error (usually type errors) and exit the program.

```
firstc -r "fn main() -> i32 { 10 + false }" --Znotypecheck
<run>:1:20: fatal: cannot add ‘i32‘ to ‘bool‘
  |
1 | fn main() -> i32 { 10 + false }
  |                    ^^^^^^^^^^ no implementation for ‘i32 + bool‘
```

3 Type Checker

3.1 Typing Rules

Let bindings requires that the type specified is the same as the resulting type of the expression e. It is also possible to define let binding without an expression e that instead only requires the type to be specified.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{let}\ x : \tau = e} \tag{19}$$

$$\frac{}{\Gamma \vdash \textbf{let}\ x : \tau} \tag{20}$$

Break and continue are not allowed outside of loops and will generate an error message if such case is detected.

$$\frac{\Gamma \vdash \textbf{while}\ \{\ b\ \} : none}{\Gamma \vdash \textbf{break} \in b : none} \tag{21}$$

$$\frac{\Gamma \vdash \textbf{while}\ \{\ b\ \} : none}{\Gamma \vdash \textbf{continue} \in b : none} \tag{22}$$

Assignment has two rules either assign directly through mutable variable x or first dereference it $*x$ (this is recursive and can handle any number of deferences). Note that for assigning to $*x$ then x has to be a mutable reference to the same type as the expression e. If x is something other than an identifier or deference (recursively) to identifier it will generate an invalid expression error. Note that $\tau(\text{mut})$ means that the actual underlying **let** binding is mutable.

$$\frac{\Gamma \vdash x : \tau(\text{mut})\ \Gamma \vdash e : \tau}{\Gamma \vdash x = e : none} \tag{23}$$

$$\frac{\Gamma \vdash x : \&mut\ \tau(\text{mut})\ \Gamma \vdash e : \tau}{\Gamma \vdash *x = e : none} \tag{24}$$

Binary expression can be divided into four groups arithmetic (e.g. $+$, $-$ etc.), logical (&&, $||$), equality ($==$, $!=$) and comparator (e.g. $<$, $>=$ etc.). Here are the typing rules for each respective group

$$\frac{\Gamma \vdash e_1 : \textbf{int}\ \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}} \tag{25}$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool}\ \Gamma \vdash e_2 : \textbf{bool}}{\Gamma \vdash e_1 \&\& e_2 : \textbf{bool}} \tag{26}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 == e_2 : \tau} \tag{27}$$

$$\frac{\Gamma \vdash e_1 : \textbf{int} \; \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 < e_2 : \textbf{bool}} \tag{28}$$

Calling a function requires that all the paramters passed have the same type that are defined in the function declaration.

$$\frac{\Gamma \vdash f(\tau_1, \tau_2, ..., \tau_n) : \tau \; \Gamma \vdash e_1 : \tau_1 \; \Gamma \vdash e_2 : \tau_2 \; \cdots \; \Gamma \vdash e_n : \tau_n}{\Gamma \vdash f(e_1, e_2, ..., e_n) : \tau} \tag{29}$$

Identifiers are stored in a type table and in order to use an identifier there is one check to see if the value has been initialized since let bindings doesn't require an initialization immediately.

If expressions requires that both blocks b_1 and b_2 has the same type and the condition is c is a boolean then the result will be τ.

$$\frac{\Gamma \vdash c : \textit{bool} \; \Gamma \vdash b_1 : \tau \; \Gamma \vdash b_2 : \tau}{\Gamma \vdash \textbf{if} \; c \; \{ \; b_1 \; \} \; \textbf{else} \; \{ \; b_2 \; \} : \tau} \tag{30}$$

Taking a reference either mutable or immutable only requires that the referenced expression actually has a type.

$$\frac{\Gamma \vdash e_1 : \&\tau \; \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 = \&e_2 : \&\tau} \tag{31}$$

Explicit and implicit return has to have the same type as the return type in the function declaration.

$$\frac{\Gamma \vdash f : \tau}{\Gamma \vdash \textbf{return} \; e \in f : \tau} \tag{32}$$

$$\frac{\Gamma \vdash f(...)b : \tau}{\Gamma \vdash b.last : \tau} \tag{33}$$

Unary expressions have very straightforward typing rules.

$$\frac{\Gamma \vdash e : \textbf{int}}{\Gamma \vdash -e : \textbf{int}} \tag{34}$$

$$\frac{\Gamma \vdash e : \textbf{bool}}{\Gamma \vdash !e : \textbf{bool}} \tag{35}$$

$$\frac{\Gamma \vdash e : \&\tau \text{ or } \&mut \; \tau}{\Gamma \vdash *e : \tau} \tag{36}$$

While expressions only requires that the condition c is a boolean. While is an expression but does not evaluate to any type.

$$\frac{\Gamma \vdash c : \textit{bool} \; \Gamma \vdash b : \textit{none}}{\Gamma \vdash \textbf{while} \; c \; \{ \; b \; \} : \textit{none}} \tag{37}$$

3.2 Examples

Rule 19:

```
firstc -r "fn main() { let x: i32 = false; }"
<run>:1:26: error: expected ‘i32‘, found ‘bool‘
  |
1 | fn main() { let x: i32 = false; }
  |
```

Rule 20:

```
firstc -r "fn main() { let x: i32; }" [OK]
```

Rule 21:

```
firstc -r "fn main() { break; }"
<run>:1:13: error: cannot break outside loop
  |
1 | fn main() { break; }
  |             ^^^^^^ help: remove this or move it inside a loop
```

Rule 22:

```
firstc -r "fn main() { continue; }"
<run>:1:13: error: cannot continue outside loop
  |
1 | fn main() { continue; }
  |             ^^^^^^^^^ help: remove this or move it inside a loop
```

Rule 23:

```
firstc -r "let mut x: i32 = 10; x = 20; print_int(x);" [OK]
20
firstc -r "let mut x: i32 = 10; x = false; print_int(x);"
<run>:1:22: error: expected ‘i32‘, found ‘bool‘
  |
1 | let mut x: i32 = 10; x = false; print_int(x);
  |                      ^^^^^^^^^^
firstc -r "let x: i32 = 10; x = 20; print_int(x);"
<run>:1:18: error: cannot assign twice to immutable variable ‘x‘
  |
1 | let x: i32 = 10; x = 20; print_int(x);
  |                  ^^^^^^^ cannot assign twice to immutable variable
1 | let x: i32 = 10; x = 20; print_int(x);
  |     ^ help: make variable mutable ‘mut x‘
```

Rule 24:

```
firstc -r "let x: &mut i32 = &mut 10; *x = 20; print_int(*x);" [OK]
20
firstc -r "let x: &mut i32 = &mut 10; *x = false; print_int(*x);"
<run>:1:28: error: expected ‘i32‘, found ‘bool‘
  |
1 | let x: &mut i32 = &mut 10; *x = false; print_int(*x);
  |                            ~~~~~~~~~~~
firstc -r "let x: &i32 = &10; *x = 20; print_int(*x);"
<run>:1:20: error: cannot assign through an ‘&‘ immutable reference
  |
1 | let x: &i32 = &10; *x = 20; print_int(*x);
  |                    ~~~~~~~~ help: change to ‘&mut‘ mutable reference
```

Rule 25, 26, 27 and 28:

```
firstc -r "let x: i32 = 10 + 20;" [OK]
firstc -r "let x: i32 = 10 + false;"
<run>:1:1: error: cannot add ‘i32‘ to ‘bool‘
  |
1 | let x: bool = 10 + false;
  |               ~~~~~~~~~~ no implementation for ‘i32 + bool‘
firstc -r "let x: bool = true && false;" [OK]
firstc -r "let x: bool = true && 20;"
<run>:1:1: error: cannot logical and ‘bool‘ to ‘i32‘
  |
1 | let x: bool = true && 20;
  |               ~~~~~~~~~~ no implementation for ‘bool && i32‘
firstc -r "let x: bool = 10 == 10;" [OK]
firstc -r "let x: bool = true == true;" [OK]
firstc -r "let x: bool = true == 20;"
<run>:1:1: error: cannot compare equal ‘bool‘ to ‘i32‘
  |
1 | let x: bool = true == 20;
  |               ~~~~~~~~~~ no implementation for ‘bool == i32‘
firstc -r "let x: bool = 10 < 20;" [OK]
firstc -r "let x: bool = 10 < false;"
<run>:1:15: error: cannot compare less than ‘i32‘ to ‘bool‘
  |
1 | let x: bool = 10 < false;
  |               ~~~~~~~~~~ no implementation for ‘i32 < bool‘
```

Rule 29:

```
firstc -r "print_int(10);"
10
firstc -r "print_int(false);"
<run>:1:11: error: expected ‘i32‘, found ‘bool‘
  |
1 | print_int(false);
  |           ^^^^^
```

Rule 30:

```
firstc -r "let x: i32 = if true { 10 } else { 20 };" [OK]
firstc -r "let x: i32 = if 10 { 10 } else { 20 };"
<run>:1:4: error: expected ‘bool‘, found ‘i32‘
  |
1 | let x: i32 = if 10 { 10 } else { 20 };
  |                 ^^
firstc -r "let x: i32 = if true { false } else { 20 };"
<run>:1:39: error: expected ‘bool‘, found ‘i32‘
  |
1 | let x: i32 = if true { false } else { 20 };
  |                                       ^^

<run>:1:14: error: expected ‘i32‘, found ‘bool‘
  |
1 | let x: i32 = if true { false } else { 20 };
  |              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Rule 31:

```
firstc -r "let x: &i32 = &y;"
<run>:1:16: error: cannot find value ‘y‘ in this scope
  |
1 | let x: &i32 = &y;
  |                ^ not found in this scope

<run>:1:15: error: expected ‘&i32‘, found ‘&()‘
  |
1 | let x: &i32 = &y;
  |               ^^
```

Rule 32 and 33:

```
firstc -r "fn main() -> i32 { return 10; }" [OK]
firstc -r "fn main() -> i32 { 10 }" [OK]
firstc -r "fn main() -> i32 { return false; }"
<run>:1:20: error: expected ‘i32‘, found ‘bool‘
  |
1 | fn main() -> i32 { return false; }
  |                    ^^^^^^^^^^^^^
firstc -r "fn main() -> i32 { false }"
```

```
<run>:1:20: error: expected ‘i32‘, found ‘bool‘
  |
1 | fn main() -> i32 { false }
  |                    ^^^^^
```

Rule 34, 35 and 36:

```
firstc -r "let x: i32 = -10;" [OK]
firstc -r "let x: i32 = -false;"
<run>:1:14: error: type ‘bool‘ cannot be negated
  |
1 | let x: i32 = -false;
  |              ^^^^^^ no implementation for ‘-bool‘
firstc -r "let x: bool = !false;" [OK]
firstc -r "let x: bool = !10;"
<run>:1:15: error: type ‘i32‘ cannot be logical inverted
  |
1 | let x: bool = !10;
  |               ^^^ no implementation for ‘!i32‘
firstc -r "let x: i32 = *&10;" [OK]
firstc -r "let x: i32 = *10;"
<run>:1:14: error: type ‘i32‘ cannot be dereferenced
  |
1 | let x: i32 = *10;
  |              ^^^ no implementation for ‘*i32‘
```

Rule 37 (infinite loops are not detected by the compiler):

```
firstc -r "while false {}" [OK]
firstc -r "while true {}" [OK]
firstc -r "while 10 {}"
<run>:1:7: error: expected ‘bool‘, found ‘i32‘
  |
1 | while 10 {}
  |       ^^
```

3.3 Requirements and Contributions

The type checker is complete since it is able to correclty identify type errors for the previously specified type rules. Note that the type checker was written before the type rules were specified. The type checker was designed by using mostly common sense and previous programming experience. The strict static typing and the error messages was directly inspired by the Rust compiler. The only thing the type checker is not able to do is type inference.

4 Borrow Checker

4.1 Specification

The borrow checker has a few responsibilies to ensure memory safety. Since this compiler only has primitive types that are always copied it means that move semantics is not needed. On most processesors registers hold these values and are moved between registers and stack using a single instruction which means copys (or moves) are very cheap. The borrow checker is only concerned with borrowed values through references. Here is the specification that defines the rules the borrow checker follows:

- Lifetime is defined to be an unsigned 32-bit integer.
- Lifetimes are only based on lexical scoping, there is no support for non-lexical lifetimes.
- Block scopes are assigned a lifetime from an incremental counter. This has a useful property when iterating over each scope in order where smaller lifetime values will always live longer than higher lifetime values. This is true because inner scopes are always entered last therefore will have a higher lifetime value and an inner scope cannot live longer than its outer scopes.
- Lifetimes of variables are assigned to have the same lifetime as the block it resides in.
- Whenever a value is borrowed the lifetime property is used to compare the values to ensure that the borrowed value lives long enough.
- Each let expression has its own reference counter which is used to ensure memory safety when dealing with multiple references. Every time a new borrow occurs the reference counter for the owned value has to first count the new reference and check that the following conditions are met:
    - Multiple immutable references, no mutable references
    - One mutable reference, no immutable references
- There is no concept of move semantics in this borrow checker since all types are primitives and they get copied instead of moved.

4.2 Examples

```
fn inc(x: &mut i32) {
    *x = *x + 1;
}

fn test_references_1() {
    let mut a: i32 = 10;
    let b: &i32 = &a;
    let c: &i32 = (&a);
    inc(&mut a);
}

fn test_references_2() {
    let mut a: i32 = 10;
    inc(&mut a);
    let b: &i32 = &a;
}

fn test_references_3() {
    let mut a: i32 = 10;
    let b: &mut i32 = &mut a;
    let c: &mut i32 = (&mut a);
}

fn test_returning_reference() -> &i32 {
    let a: i32 = 10;
    &a
}

fn test_reference_out_of_scope() {
    let mut a: &i32 = &0;
    {
        let b: i32 = 5;
        a = &b;
    }
    print_int(*a);
}

fn test_mutate_while_borrow() {
    let mut a: i32 = 5;
    let b: &i32 = &a;
    a = a + 5;
    print_int(*b);
}
```

Running this code shows all the borrowing errors that the compiler is capable of catching. This program is designed so only one error is reported per function.

```
firstc borrowing.sq
borrowing.sq:9:9: error: cannot borrow ‘a‘ as mutable because it is also borrowed as immutable
  |
9 |     inc(&mut a);
  |         ^~~~~~ immutable borrow occurs here

borrowing.sq:15:19: error: cannot borrow ‘a‘ as immutable because it is also borrowed as mutable
   |
15 |     let b: &i32 = &a;
   |                   ^~ immutable borrow occurs here

borrowing.sq:21:24: error: cannot borrow ‘a‘ as mutable more than once
   |
21 |     let c: &mut i32 = (&mut a);
   |                        ^~~~~~ immutable borrow occurs here

borrowing.sq:26:6: error: cannot return value borrowed from this function
   |
26 |     &a
   |      ^ borrowed value will outlive owned value

borrowing.sq:33:13: error: ‘b‘ does not live long enough
   |
33 |         a = &b;
   |             ^~ borrowed value does not live long enough

borrowing.sq:41:5: error: cannot assign to ‘a‘ because it is borrowed
   |
41 |     a = a + 5;
   |     ^~~~~~~~~~ assignment to borrowed ‘a‘ occurs here
```

4.3 Requirements and Contributions

The borrow checker is able to detect and reject ill-formed borrows using only lexical scoping. For the borrow checker a first draft of the specification was actually first written before the implementation and still remains in the source code. The specification was refined after finishing the project. The only thing not implemented was non-lexical scoping and move semantics.

5 Intermediate Representation

This intermediate representation is based of LLVM IR but does not completely follow the SSA form because there are no phi-nodes nor any control-flow graph. The intermediate reprsentation was created to make the backend codegen easier therefore it is closer to an actual assembler level language. The conversion from AST to the IR uses a pre-order walk through each node in the AST and converts each of them separately.

5.1 IR Instruction

Instructions are stored as three-address code so each instruction has up to three operands and one opcode. Also the type of the first operand in each instruction is included to help the backend to figure out how many bytes to use. This is because booleans are represented as signed byte and integers are 4-bytes. Operands can either be an identifier or a literal.

These are the supported opcodes in the IR:

```
pub enum IrOpcode {
    Nop,
    Alloca, // op1 = alloca ty
    AllocParams, // allocates all defined parameters
    Copy, // op1 = op2
    CopyFromDeref, // op1 = *op2
    CopyFromRef, // op1 = &op2 (always mutable)
    CopyToDeref, // *op1 = op2
    Clear, // op1 = 0
    Add, // op1 = op2 + op3
    Sub,
    Mul,
    Div,
    Pow,
    Mod,
    And,
    Or,
    Xor,
    Lt, // op1 = op2 < op3 (op1 always boolean)
    Le,
    Gt,
    Ge,
    Eq,
    Ne,
    IfLt, // jump op3 (if op1 binop op2 equals true)
    IfGt,
    IfLe,
    IfGe,
    IfEq,
    IfNe,
    Jump,     // jump op1
    Label,    // label op1
    Param,    // param op1 (ordered left-to-right)
    Call,     // op1 := op2(...) (#parameter stored in op3)
    Return,   // return op1 (where op1 is optional)
    Prologue, // marks beginning of function
    Epilogue, // marks end of function
}
```

5.2 Example

Here is a very simple example showing let bindings with simple binary add expression.

```
firstc -r "let x: i32 = 20 + 10;" --backend=x86 --print=ir

main:
    prologue
    alloc_params
    %x = alloca i32
    %0 = add i32 20, 10
    %x = copy i32 %0
main1:
    epilogue
```

The -r command automatically creates a *main* function if there is not one already specified. The first line is a *label* defining where *main* starts similarly to the second label *main1* which marks the end of the function. All functions starts with a prologue and ends with epilogue which is for backend specific use, mostly they serve as stack initialization and cleanup respectively. Next *alloc_params* allocates parameters, however main does not take any parameters so this does nothing. Then *alloca i32* reserves 4 bytes on the stack, then the *add* instruction performes the addition 20 + 10 and stores it in variable %0 which is a temporary register. Note all the names starting with % are local variables and if it has no name like %0 it is temporary, also each name has an extra number (e.g. *main1*) this is to prevent aliasing internally since everything is stored in hash tables. Finally the *copy* commands just copies the data of %0 like a regular assignment into %*x*. This should be enough to give an idea of how the IR works in this language so next let's look at the actual x86 machine code generated.

6 X86 Backend

The x86 backend translates each IR instruction to machine code one by one starting always with the *main* function. Note that this backend can print the assembly code but it does not actually generate any valid assembly code, the main issue is that only 64-bit register types are printed. This backend is focuses on generating valid machine code so there is no need for an external assembler. There is no guarantee that this will run on every processor it has only been tested on one x86 64-bit processor but might also work on 32-bit versions too, but it has not been tested. This part of the compiler essentially boils down to encoding instructions described in Intels manual, but also it has to generate multiple instructions since the IR instructions are not one-to-one with x86 instructions. For example there is no prologue opcode on x86 so it generates potentially multiple x86 instructions. The result of compiliation is a list of unsigned bytes which are copied to a new memory page marked with read and execution flags so it can be executed just-in-time. There is no support for generating actual executables but just-in-time was implemented by using the operating system APIs. The only problem with generating executables is that intrinsics needs to be linked in which seems to be more hassle than it is worth. This is not an optimizating backend there are some small optimizations tricks in some places but the focus of writing this backend was to learn of how x86 backends can be implemented.

6.1 Examples

Of course the *examples/demo.sq* program works as expected but it generates a lot of assembly code, so it is impractical to use for studying how this backend works. Lets start by using the very simple example used in the previous section.

```
firstc -r "let x: i32 = 20 + 10;" --backend=x86 --print=asm


main:
    push  rbp
    mov   rbp, rsp
    mov   rax, 20
    add   rax, 10
    mov   dword ptr [rbp - 4], rax
main1:
    pop    rbp
    ret
```

The *prologue* always does the first two instruction to first make sure that the base pointer is stored and the new base pointer is set the the stack pointer. The rest looks very similar to the IR except that the *add* instruction cannot take two immediates (or literals) at once. Thus the *add* instruction has to first move the left-hand side to a temporary register and then the perform addition with the temporary register. The register allocation is aware of which registers are general purpose and how many there are (64-bit mode introduces a few more and this backend can use those). It is also necessary for the register allocator to know whenever a register is no longer in use so it can be automatically freed. This is done by checking the live intervals of variables which is provided by the IR. The *alloca* was moved after the addition and stores it in stack offset 4-bytes as expected. The *epilogue* performs stack cleanup which resets the stack pointer. It is worth mentioning this now this did not allocate any stack space which was intentional as this is called a "leaf" function since it doesn't call any other functions, thus it does not need any allocated stack space.

Lets look at a slightly more complicated program with a slow version of the *fibonacci* program listed below.

```
fn main() -> i32 {
    return fib(33);
}

fn fib(x: i32) -> i32 {
    if x < 2 {
        return x;
    }
    return fib(x - 1) + fib(x - 2);
}
```

```
firstc examples/fib.sq --backend=x86 --print=asm


main:
    push  rbp
    mov   rbp, rsp
    sub   rsp, 0
    mov   rcx, 33
    call  fib
    jmp   main1
main1:
    add   rsp, 0
    pop    rbp
    ret
fib:
    push  rbp
    mov   rbp, rsp
```

```
18     sub   rsp, 16
19     mov   dword ptr [rbp - 4], rcx
20     cmp   dword ptr [rbp - 4], 2
21     jge   .if_exit
22     mov   rax, dword ptr [rbp - 4]
23     jmp   fib1
24 .if_exit:
25     mov   rax, dword ptr [rbp - 4]
26     sub   rax, 1
27     mov   rcx, rax
28     call  fib
29     mov   rbx, dword ptr [rbp - 4]
30     sub   rbx, 2
31     mov   dword ptr [rbp - 8], rax
32     mov   rcx, rbx
33     call  fib
34     mov   rcx, dword ptr [rbp - 8]
35     add   rcx, rax
36     mov   rax, rcx
37     jmp   fib1
38 fib1:
39     add   rsp, 16
40     pop    rbp
41     ret
```

The first difference is that now the stack pointer is subtracted except in *main* the value 0 was subtracted which might be considered a bug or oversight. There is another oversight on line 10 where there is a jump to the label on line 11. These do not actually affect the results of executing the program but wastes some spaces and CPU clock cycles. In *main* there is a call to *fib* and the argument is stored in **RCX** which is based of the Microsoft x64 calling convention, there is also support for sysv64 use to call intrinsics on Linux or Mac based operating systems. In both calling conventions *RAX* is used to store the return value. Next the *fib* function has an if-expression which is converted to a conditional relative jump. The condition is always inverted because when the jump should go over the first block to the second block or exit the if-expression. Since the order of the blocks are not changed it is more efficient to invert the condition otherwise more jumps would need to be added.

There is one major challange when setting jump distances which is that whenever one relative offset needes jump longer than -127 or 128 bytes. For longer relative jump distances more bytes have to be added both for the opcode and to promote to 4-byte offset. This can obviously mess up any other previously calculated relative jumps that tries to jump over this one. The code for resolving entangled relative jumps is very complicated, possibly buggy and very messy. It works similar to a plane sweep alogrithm except it starts at the bottom of the program and sweeps upwards (discretely). When the sweep line enters an event point processing is done An event point is defined for each relative jump and is the maximum between the byte position for the jump instruction and the byte position that the jump is targeted for. At every event point it adds the jump to a list of active jumps and iterates over all the current active jumps to see if any jump needs to be removed from the list. The invariant holds that whenever a jump is removed from the list it will never need to be recalculated because adding bytes before the jump won't affect relative distances. Bytes are never added before any of the active jumps and whenever bytes are added all the active jumps needs to check if they have to also jump over those bytes or not.

As previously mentioned the backend is not concerned with generating valid assembly code but instead focused on generating actual machine code which looks like this:

```
firstc examples/fib.sq --backend=x86 --print=machinecode


ff f5 48 8b ec 48 81 ec 00 00 00 00 c7 c1 21 00
00 00 e8 0c 00 00 00 eb 00 48 81 c4 00 00 00 00
8f c5 c3 ff f5 48 8b ec 48 81 ec 10 00 00 00 89
4d fc 81 7d fc 02 00 00 00 7d 05 8b 45 fc eb 2c
8b 45 fc 81 e8 01 00 00 00 8b c8 e8 d3 ff ff ff
8b 5d fc 81 eb 02 00 00 00 89 45 f8 8b cb e8 c0
ff ff ff 8b 4d f8 03 c8 8b c1 eb 00 48 81 c4 10
00 00 00 8f c5 c3

Size of code is 118 bytes
```

6.2 Performance Improvements

Just for fun lets see how much faster this backend is compared to the interpreter. Let's run the *examples/fib.sq* code which is of course not an efficient way to implement *fibonacci* sequence.

```
firstc examples/fib.sq --backend=interp --profile

Interpreter exited with code 3524578
Interpreter execution time: 7.173111 seconds
```

```
firstc examples/fib.sq --backend=x86 --profile

Program exited with code 3524578
Program execution time: 0.0255269 seconds
```

The speed up from running actual machine code is quite large compared to the interpreter. This does not count the time to construct the machine code but even that would still be way faster.

7 Learning Outcomes

Lexical analysis, syntax analysis, and translation into abstract syntax. I have learnt about lexical analysis which converts source code into a stream of tokens. Tokens combines one or multiple characters and gives them a label e.g. ”10” has two characters and is an integer literal token. I have implemented my own lexer for another compiler project.

I have also learnt about syntax analysis which uses the stream of tokens and the grammar for a language to construct an abstract syntax tree. This is not something I have practically implemented or used myself in my projects.

Regular expressions and grammars, context-free languages and grammars, lexer and parser generators. I have learnt a little bit about how grammars and context-free grammars work but this is something I haven't applied in my implementation. The EBNF grammar was specified as part of the assignment but was not used in writing the actual parser.

In this course I have implemented a parser for a mini Rust language using the parser generator library *nom*. I have also implemented my own parser using a lexer for another compiler project. While I have not used parser generators and grammars in practical implementations, I've found that parsing is a relatively simple problem if it is broken down into first lexing and then building the abstract syntax tree.

Identifier handling and symbol table organization. Type-checking, logical inference systems. As mentioned in Section 1.3 identifiers are converted into symbols which are cheap to copy and can easily be used to check if two symbols are equal by their id. Symbols are also easy to store in hash tables which is used by most compiler stages to keep track of variables.

I have learnt about type-checking which is the process of ensuring correctness of types and their usage according to the specified type rules. However as mentioned in Section 3.3 I didn't use the type rules directly when writing the type-checker but instead used Rust as a guide and common sense.

In the course we also looked at structural operational semantics for defining the rules for how the code should be interpreted and also for defining rules that the type checker should follow. However these were not used in the actual process of implementing both the interpreter and type checker.

Intermediate representations and transformations for different languages. I have looked at intermediate representations for LLVM and implemented a similar style however not fully committing to the SSA form. I do find however that LLVM is a very interesting way to bring more languages to compiled form, e.g. this is for example being done for web with webassembly.

Code optimization and register allocation. Machine code generation for common architectures. I didn't look much at code optimizations but some work was put into doing proper register allocation, however my version of the register allocator is still not very efficient. One notable optimization that I mistakingly choose to implement was the relative jumps with single byte offsets whenever possible.

I also didn't use LLVM for my project, instead I decided to build my own x86 backend because I wanted to understand how something like LLVM could work. For this I had to learn about how instructions are encoded in x86 and about how to run code just-in-time. Trying to understand LLVM by looking at their source code is probably not a good way to learn it and just using it will probably not give much insight into the inner workings either.