

Necklace

Maksymilian Demitraszek, Paweł Gmerek, Michał Kuliński

November 2021

1 Abstract

Necklace is a tiny, imperative, statically, strongly typed language with Elixir-like syntax.

2 Example syntax

```
function do_stuff(a: int, b: int) -> int do
  return 2 + 2;
end
```

```
function complex -> int do
  a: int;
  b: int;
  c: int;
  d: int;
  a = 1;
  b = 2;
  c = 3;
  d = a + b + c;
  return d;
end
```

```
function array_operations(a: *int, size: int) -> void do
  i: int;
  for (i = 0; i < size; i += 1) {
    (a + i)* +=1
  }
end
```

```
function main do
  do_stuff(1, 2);
end
```

3 Tokens

3.0.1 Regular Expressions

```
@keywords = "(function|if|else|for|while|return|break|continue|->|do|end)"
@varId = "[A-Za-z][A-Za-z0-9_]*"
@int_lit = "([0-9])+"
@bool_lit = "true|false"
@operator = "(&+|\+|-|\*|\/|\%|<|>|>=|<=|==|!=|&&|\&|\||!|=)"
@comment = "~~.*"
@special = "[\(\)\,\;\:\[\]\{\}\]"
@whitechar = "[\t\n\r\v\f\ ]"
@type = int|bool
```

4 Grammar

```
< start > → [ < function > ]
< type > → bool | int | [ < type > ] | * < type >
< return_type > → < type > | void
< function > →
    | function < name > < arguments > -> < return_type > do < function_body >
    | function < name > -> < return_type > do < function_body > end
< arguments > → ( < function_args > )
< function_args > → < name > : < type > ( , < name > : < type > )*
< function_body > → < declaration > * < statement > +
< body > → < statement > *
< statement > → < function_call >;
    | < name > = < expression >;
    | if < expr > do < body > else < body > end
    | for ( < expr > , < expr > , < expr > ) do < body > end
    | while < expr > do < body > end
    | return < expr >;
    | return;
    | break;
    | continue;
< declaration > → < name > : < type >;
< expression > → < expression > < binary_operator > < expression >
    | < unary_operator > < expression >
    | ( < expression > )
```

$$\begin{aligned}
& | \langle \textit{function_call} \rangle \\
& | \langle \textit{literal} \rangle \\
& | \langle \textit{expression} \rangle [\langle \textit{digit} \rangle^+] \\
\langle \textit{unary_operator} \rangle & \rightarrow * | - | ! \\
\langle \textit{binary_operator} \rangle & \rightarrow \langle \textit{arithmetic_operator} \rangle \\
& | \langle \textit{relational_operator} \rangle \\
& | \langle \textit{equality_operator} \rangle \\
\langle \textit{arithmetic_operator} \rangle & \rightarrow + | - | * | / | \% \\
\langle \textit{relational_operator} \rangle & \rightarrow < | > | <= | >= \\
\langle \textit{equality_operator} \rangle & \rightarrow == | != \\
\langle \textit{conditional_operator} \rangle & \rightarrow \&\& | || \\
\langle \textit{function_call} \rangle & \rightarrow \langle \textit{function_name} \rangle (\langle \textit{expr} \rangle^*) \\
\langle \textit{function_name} \rangle & \rightarrow \langle \textit{name} \rangle \\
\langle \textit{literal} \rangle & \rightarrow \langle \textit{int_literal} \rangle | \langle \textit{bool_literal} \rangle | \langle \textit{array_literal} \rangle \\
\langle \textit{int_literal} \rangle & \rightarrow - \langle \textit{digit} \rangle^+ | \langle \textit{digit} \rangle^+ \\
\langle \textit{bool_literal} \rangle & \rightarrow \textbf{true} | \textbf{false} \\
\langle \textit{array_literal} \rangle & \rightarrow [\langle \textit{expr} \rangle^*] \\
\langle \textit{identifier} \rangle & \rightarrow \langle \textit{letter} \rangle | \langle \textit{identifier} \rangle \langle \textit{letter} \rangle | \langle \textit{identifier} \rangle \langle \textit{digit} \rangle
\end{aligned}$$

5 Type system

Necklace is a strongly typed language, so all type conversions have to be explicit.

5.1 Base types

5.1.1 Boolean

Declaration

```
variable: bool;
```

$\{true, false\}$

Corresponds to LLVMs `il` <https://releases.llvm.org/9.0.0/docs/LangRef.html#integer-type>

5.1.2 Int

Declaration

```
variable: int;
```

a 32 bit signed integer type
Corresponds to LLVMs i32 <https://releases.llvm.org/9.0.0/docs/LangRef.html#integer-type>

5.1.3 Pointer

variable: *<type>;

Represents the location in memory of a variable Corresponds to LLVMs pointer type <https://releases.llvm.org/9.0.0/docs/LangRef.html#pointer-type>

5.1.4 Array

Declaration

variable: [<type>;

Represents an array of variables of specified type Corresponds to LLVMs array type <https://releases.llvm.org/9.0.0/docs/LangRef.html#array-type>

5.2 Type inference

5.2.1 '-' unary operator

$$(+): int \longrightarrow int$$

5.2.2 '!' unary operator

$$(!): bool \longrightarrow bool$$

5.2.3 '*' unary operator

$$(*): pointer < type > \longrightarrow < type >$$

5.2.4 '+' binary operator

$$(+): int \times int \longrightarrow int$$

5.2.5 '-' binary operator

$$(-): int \times int \longrightarrow int$$

5.2.6 '*' binary operator

$$(*): int \times int \longrightarrow int$$

5.2.7 '/' binary operator

$$(/): int \times int \longrightarrow int$$

5.2.8 '/' binary operator

$$(-) : int \times int \longrightarrow int$$

5.2.9 '%' binary operator

$$(\%) : int \times int \longrightarrow int$$

With behaviour defined as

$$x \% y = r \quad r = x - yk, x \in C$$

5.2.10 'toBool' conversion

$$toBool : int \longrightarrow bool$$

With behaviour defined as

$$toBool(x) = \begin{cases} false & x == 0 \\ true & otherwise \end{cases}$$

5.2.11 'toInt' conversion

$$toInt : bool \longrightarrow int$$

With behaviour defined as

$$toInt(x) = \begin{cases} 0 & x == false \\ 1 & x == true \end{cases}$$

5.2.12 '==' binary operator

$$(==) : int \times int \longrightarrow bool$$

$$(==) : bool \times bool \longrightarrow bool$$

5.2.13 '!= ' binary operator

$$(!=) : int \times int \longrightarrow bool$$

$$(!=) : bool \times bool \longrightarrow bool$$

5.2.14 '<' binary operator

$$(<) : int \times int \longrightarrow bool$$

5.2.15 '>' binary operator

$$(>) : int \times int \longrightarrow bool$$

5.2.16 '<=' binary operator

$$(<=) : int \times int \longrightarrow bool$$

5.2.17 '>=' binary operator

$(\geq) : int \times int \longrightarrow bool$

5.2.18 '&&' binary operator

$(\&\&) : bool \times bool \longrightarrow bool$

5.2.19 '||' binary operator

$(||) : bool \times bool \longrightarrow bool$

5.2.20 'if' conditional operator

if bool do < block > end

5.2.21 'while' binary operator

while bool do < block > end

5.2.22 'for' binary operator

for < type > bool < type > do < block > end

6 Compiler architecture

6.1 Overview

6.2 Lexer

The lexer is generated using the alex library for Haskell, which provides similar interface as lex.

6.3 Parser

The parser is generated using the happy library for Haskell, which provides similar interface as yacc.

6.4 Semantic checker

The language is statically and strongly checked. The compiler will perform a semantic analysis and throw errors if any of the types are not matching. TODO
1. Validate expressions and operator types
2. validate array literals are singular type
3. validate assignments have correct type
4. validate if all variables are declared

6.5 Code generation

For the generation of LLVM IR representation we use the `llvm-hs` library which provides bindings simplifying the LLVM code generation

7 References

1. Engineering a Compiler, by Keith D. Cooper Linda Troczon
2. MiT compilers course, decaf lang