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

$$
\begin{array}{rl}
< type > \longrightarrow & \textbf{bool} \mid \textbf{int} \mid [\, < type > \,] \mid \textbf{*} < type > \\
< return\_type > \longrightarrow & < type > \mid \textbf{void} \\
< function > \longrightarrow & \\
 & \mid \textbf{function} < name >< arguments > \textbf{->} < return\_type > \textbf{do} \;\; < function\_body > \\
 & \mid \textbf{function} < name > \textbf{->} < return\_type > \textbf{do} \;\; < function\_body > \;\; \textbf{end} \\
< arguments > \longrightarrow & \textbf{(} < function\_args > \textbf{)} \\
< function\_args > \longrightarrow & < name > \textbf{:} < type > (\textbf{,} < name > \textbf{:} < type >)^{*} \\
< function\_body > \longrightarrow & < declaration >^{*} < statement >^{*} \\
< body > \longrightarrow & < statement >^{*} \\
< statement > \longrightarrow & < function\_call >; \\
 & \mid \; < name > \textbf{=} < expression > \textbf{;} \\
 & \mid \textbf{if} \; < expr > \textbf{do} < body > \textbf{else} < body > \;\; \textbf{end} \\
 & \mid \textbf{for} \; \textbf{(} < expr > \textbf{,} < expr > \textbf{,} < expr > \textbf{)} \; \textbf{do} < body > \textbf{end} \\
 & \mid \textbf{while} < expr > \textbf{do} < body > \textbf{end} \\
 & \mid \textbf{return} < expr > \textbf{;} \\
 & \mid \textbf{return;} \\
 & \mid \textbf{break;} \\
 & \mid \textbf{continue;} \\
< declaration > \longrightarrow & < name > \textbf{:} < type > \textbf{;} \\
< expression > \longrightarrow & < expression >< binary\_operator >< expression > \\
 & \mid \; < unary\_operator >< expression > \\
 & \mid (< expression >) \\
 & \mid \; < function\_call >
\end{array}
$$

$$
\begin{aligned}
& \mid \; < literal > \\
< postfix\_expression > & \longrightarrow < expression > \\
& \mid \; < postfix\_expression > [< identifier >] \\
& \mid \; < postfix\_expression > [< digit >^{+}] \\
< unary\_operator > & \longrightarrow \; \textbf{*} \mid \textbf{-} \mid \textbf{!} \\
< binary\_operator > & \longrightarrow < arithmetic\_operator > \\
& \mid \; < relational\_operator > \\
& \mid \; < equality\_operator \\
< arithmetic\_operator > & \longrightarrow \; \textbf{+} \mid \textbf{-} \mid \textbf{*} \mid \textbf{/} \mid \textbf{\%} \\
< relational\_operator > & \longrightarrow \; \textbf{<} \mid \textbf{>} \mid \textbf{<=} \mid \textbf{>=} \\
< equality\_operator > & \longrightarrow \; \textbf{==} \mid \textbf{!=} \\
< conditional\_operator > & \longrightarrow \; \textbf{\&\&} \mid || \\
< function\_call > & \longrightarrow < function\_name > (< expr >^{*}) \\
< function\_name > & \longrightarrow < name > \\
< literal > & \longrightarrow < int\_literal > \; \mid \; < bool\_literal > \mid < array\_literal > \\
< int\_literal > & \longrightarrow \; - < digit >^{+} \mid < digit >^{+} \\
< bool\_literal > & \longrightarrow \; \textbf{true} \mid \textbf{false} \\
< array\_literal > & \longrightarrow \; [< expr >^{*}] \\
< identifier > & \longrightarrow < letter > \; \mid \; < identifier >< letter > \; \mid \; < identifier >< digit > \\
< letter > & \longrightarrow a \mid b \mid .. \mid z \mid A \mid B \mid ... \mid Z \\
< digit > & \longrightarrow 0 \mid 1 \mid .. \mid 9
\end{aligned}
$$

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 i1 `https://releases.llvm.org/9.0.0/docs/LangRef.html#integer-type`

5.1.2 Int

Declaration

```
variable: int;
```

a 32 bit singed 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

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

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