Necklace

Maksymilian Demitraszek, Paweł Gmerek, Michał Kuliński

November 2021

1 Abstract

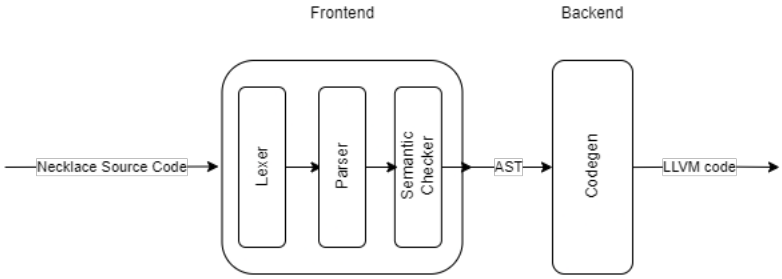
Necklace is a tiny, imperative, statically, strongly typed language with Elixir-like syntax. Made for Group Project 1 & 2 at Jagiellonian University

2 Compiler architecture

2.1 Overview

Necklace is a 2 phase compiler

- Frontend - Lexer, Parser, Semantic checker
- Backend - code generation to LLVM



2.2 Lexer

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

2.3 Parser

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

2.4 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.

2.5 Code generation

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

3 Example syntax

Every Necklacke program needs to have a main function that returns an integer.

```
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 -> int do
    do_stuff(1, 2);
    return 0;
end
```

4 Tokens

4.0.1 Regular Expressions

```
@keywords = "(function|if|else|for|while|return|break|continue|->|do|end|alloc|free)"
@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
```

5 Grammar

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

$$
\begin{aligned}
&| < function\_call > \\
&| < literal > \\
< unary\_operator > &\longrightarrow \ \mathbf{*} \mid \mathbf{-} \mid \mathbf{!} \\
< binary\_operator > &\longrightarrow < arithmetic\_operator > \\
&| < relational\_operator > \\
&| < equality\_operator \\
< arithmetic\_operator > &\longrightarrow \ + \mid \mathbf{-} \mid \mathbf{*} \mid / \mid \% \\
< relational\_operator > &\longrightarrow \ < \mid > \mid <= \mid >= \\
< equality\_operator > &\longrightarrow \ == \mid \mathbf{!=} \\
< conditional\_operator > &\longrightarrow \ \mathbf{\&\&} \mid || \\
< function\_call > &\longrightarrow < function\_name > (< expr >^*) \\
< function\_name > &\longrightarrow < name > \\
< literal > &\longrightarrow < int\_literal > \ | \ < bool\_literal > \\
< int\_literal > &\longrightarrow \ - < digit >^+ \mid < digit >^+ \\
< bool\_literal > &\longrightarrow \ \mathbf{true} \mid \mathbf{false} \\
< identifier > &\longrightarrow < letter > \ | \ < identifier >< letter > \ | \ < identifier >< digit >
\end{aligned}
$$

5.1 Operators Precedence

Priority	Category	Operator	Associativity
1	Postfix	[ ]	Left to right
2	Unary	$-$, $*$, !	Right to left
3	Multiplicative	$*$, /, %	Left to Right
4	Additive	$+$, $-$	Left to right
5	Relational	$<$, $>$, $<=$, $>=$	Left to right
6	Equality	$==$, $!=$	Left to right
7	Logical AND	&&	Left to right
8	Logical OR	$\|\|$	Left to right
9	Assignment	$=$	Right to left

6 Type system

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

6.1 Base types

6.1.1 Boolean

Declaration

```
variable: bool;
```

$$\{true, false\}$$

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

6.1.2 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`

6.1.3 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`

6.1.4 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`

6.2 Type inference

6.2.1 '-' unary operator

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

6.2.2 '!' unary operator

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

6.2.3 '*' unary operator

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

6.2.4 '+' binary operator

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

6.2.5 '-' binary operator

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

6.2.6 '*' binary operator

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

6.2.7 '/' binary operator

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

6.2.8 '/' binary operator

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

6.2.9 '%' binary operator

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

With behaviour defined as

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

6.2.10 'toBool' conversion

$$toBool : int \longrightarrow bool$$

With behaviour defined as

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

6.2.11 'toInt' conversion

$$toInt : bool \longrightarrow int$$

With behaviour defined as

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

6.2.12 '==' binary operator

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

6.2.13 '! =' binary operator

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

6.2.14 '<' binary operator

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

6.2.15 '>' binary operator

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

6.2.16 '<=' binary operator

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

6.2.17 '>=' binary operator

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

6.2.18 '&&' binary operator

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

6.2.19 '||' binary operator

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

6.2.20 'if' conditional operator

$$if\ bool\ do\ <block>\ end$$

6.2.21 'while' binary operator

$$while\ bool\ do\ <block>\ end$$

6.2.22 'for' binary operator

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

7 Tests

Necklace test suite includes unit tests for lexer and context sensitive analysis and integration tests for basic language functionalities.

8 References

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