Cel projektu

Celem projektu było stworzenie kompilatora do prostego języka z wykorzystaniem parsera rekursywnego stępującego i LLVM w celu kompilacji do formy pośredniej IR lub kompilacji JIT.

Ostateczna specyfikacja gramatyki

```
Program = FunctionDecl, { FunctionDecl | VariableDeclStatement } ; // po dodaniu
zmiennych
globalnych

FunctionDecl = “fn”, Identifier, “(“, DeclArgumentList, “)”, “->”, VarType,
Block ;
ExternFuncDecl = "extern" “fn”, Identifier, “(“, DeclArgumentList, “)”, “-
>”,VarType, ";";
DeclArgumentList = [ Identifier, “:”, VarType, { “,”, Identifier, “:”, VarType }
] ;
VarType = “int” | “string” | VarType, “*”;
ConditionalExpression= UnaryLogicalExpr, { (“&&” | “||”), ConditionalExpression}
;
UnaryLogicalExpr = [“!”], LogicalExpr;
LogicalExpr = ArithmeticalExpr, { ( “<” | “>” | “<=” | “>=” | “==” | “!=”),
LogicalExpr } ;
ArithmeticalExpr = AdditiveExpr, { ( “&” | “|” | “^” |”<<”|”>>”),
ArithmeticalExpr} ;
AdditiveExpr = MultiplicativeExpr, { ( “+”, “-” ), AdditiveExpr } ;
MultiplicativeExpr = UnaryExpression, { ( “*”, “/”, “%”), MultiplicativeExpr } ;
UnaryExpression = { “*”, “&”, “~”} , Factor, [ “[“, ArithmeticalExpr, “]” ] ;
Factor = Identifier | IntegerConst | StringConst | FunctionCall | “(“,
ConditionalExpression, “)” ;
FunctionCall = Identyfier, “(“, CallArgumentList, “)” ;
CallArgumentList = [ ArithmeticalExpr, { “,” , ArithmeticalExpr} ] ;

Statement =
    (
        IfStatement |
        ForStatement |
        WhileStatement |
        ReturnSatetemnt |
        VariableDeclStatement |
        AssignStatement |
        ConditionalExpression, “;”
```

```
    );
AssignStatement = ConditionalExpression, { “=”, ConditionalExpression }, “=”,
ArithmeticalExpr ;

Block = “{“, { Statement }, “}” ‘;
IfStatement = “if”, ConditionalExpression, Block, { “elif”,
ConditionalExpression, Block } , [“else”, Block ];
ForStatement = “for”, Identifier, “in”, ArithmeticalExpr, “..”,
ArithmeticalExpr, [“..”, ArithmeticalExpr],
Block;
WhileStatement = “while”, ConditionalExpression, Block ;
ReturnStatement = “return”, ArithmeticalExpr;
VariableDeclStatement = “let”, Identifier, [ “=”, ArithmeticalExpr], { “,”,
Identifier, [ “=”,
ArithmeticalExpr] } , “:”, VarType, “;” ;
Letter = “a” | “b” | “c” | “d” | ... | “X” | “Y” |”Z” ;
FirstDigit = “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” ;
Digit = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” ;
SpecialChar = “`” | “~” | “!” | “@” |”#” | “$” | “%” | “^” | “&”| “*” | “(“ |
“)” | “-” | “=” | “+” | “[“ | “]”
| ‘\”’ | “;” | “:” | “’” | “<” | “>” | “,” | ”.” | “?” | “/” | “|” | “{“ | “}” ;
IntegerConst = [“-”], Firstdigit, { Digit } ;
StringConst = ‘”’, { Letter | Digit | SpecialChar }, ‘”’ ;
Idetifier = Letter, { Letter | Digit } ;
```

W porównaniu do gramatyki z dokumentacji wstępnej dodana została produkcja definiująca deklaracje nagłówka funkcji w celu umożliwienia korzystania z funkcji z bibliotek dynamicznych np malloc, free itd. Uproszczona została również reguła operatora przypisania, zamiast wyrażenia MutableExpression zostały podstawione ConditionalExpression aby maksymalnie uogólnić gramatyka. Ze względu na ogólność reguł, ich poprawność jest sprawdzana na późniejszym etapie analizy semantycznej.

Przykłady kodu

```
extern fn putwchar(chr : int) -> int;

fn putstr(str: string) -> int {
    let i=0 : int;
    while str[i] {
        putwchar(str[i]);
        i = i + 1;
    }
    return i;
}

fn main() -> int {
    putstr("OKI");
    return 0;
}
```

```
extern fn putwchar(chr : int) -> int;

fn putint(num : int) -> int {
    if num == 0 {
```

```
        return 0;
    }
    let digit = num % 10 : int;
    putint(num / 10);
    putwchar(digit + 48);
    return 0;
}

fn putstr(str: string) -> int {
    let i=0 : int;
    while str[i] {
        putwchar(str[i]);
        i = i + 1;
    }
    return i;
}

fn main() -> int {
    for i in 1..20 {
        putint(i);
        putstr("\n");
    }
    return 0;
}1
```

```
extern fn putwchar(chr : int) -> int;

fn putstr(str: string) -> int {
    let i=0 : int;
    while str[i] {
        putwchar(str[i]);
        i = i + 1;
    }
    return 0;
}

fn is_prime(num : int) -> int {
    for i in 2..(num / 2 + 1) { # chciałem zrobić sqrt ale nie mam floatów
        if num % i == 0 {
            return 0;
        }
    }
    return 1;
}

fn main() -> int {
    let num=2 : int;
    while num < 10 {
        if is_prime(num) {
            putwchar(num+48);
        } else {
            putstr("-");
        }
        num = num + 1;
    }
    return 0;
}
```

Oraz większy przykład interpreter języka brainfuck

```
extern fn putwchar(chr : int) -> int;
extern fn malloc(size : int) -> int*;
extern fn getchar() -> int;
extern fn free(ptr : int*) -> int;
extern fn memset(ptr : int*, val : int, size : int) -> int*;

let code : int*;
let ram : int*;
let ptr = 0 : int;
let ip = 0 : int;

fn putstr(str: string) -> int {
    let i=0 : int;
    while str[i] {
        putwchar(str[i]);
        i = i + 1;
    }
    return 0;
}

fn readstr(ptr : int*, size : int) -> int* {
    for i in 0..size {
        let ch=getchar() : int;
        if ch == 10 {
            return ptr;
        }
        ptr[i] = ch;
    }
    return ptr;
}

fn read_char() -> int {
    ram[ptr] = getchar();
    return 0;
}

fn print_char() -> int {
    putwchar(ram[ptr]);
    return 0;
}

fn inc() -> int {
    ram[ptr] = ram[ptr] + 1;
    return 0;
}

fn dec() -> int {
    ram[ptr] = ram[ptr] - 1;
    return 0;
}

fn next() -> int {
    ptr = ptr + 1;
    return 0;
```

```
}

fn prev() -> int {
    ptr = ptr - 1;
    return 0;
}

fn go_to_loop_beg() -> int {
    ip = ip - 1;
    let count = 1 : int;
    while count > 0 {
        if code[ip] == 91 {
            count = count - 1;
        } elif code[ip] == 93 {
            count = count + 1;
        }
        ip = ip - 1;
    }
    return 0;
}

fn go_to_loop_end() -> int {
    if ram[ptr] == 0 {
        ip = ip + 1;
        let count = 1 : int;
        while count > 0 {
            if code[ip] == 91 { # '['
                count = count + 1;
            } elif code[ip] == 93 { # ']'
                count = count - 1;
            }
            ip = ip + 1;
        }
        ip = ip - 1;
    }
    return 0;
}

fn debug(opcode : int) -> int {
    putstr("ip "); putint(ip); putstr("\n");
    putstr("ptr "); putint(ptr); putstr("\n");
    putstr("opcode "); putwchar(opcode); putstr("\n");
    return 0;
}

fn execute() -> int {
    while code[ip] {
        let opcode = code[ip] : int;
        #debug(opcode);
        if opcode == 44 { # ','
            read_char();
        } elif opcode == 46 { # '.'
            print_char();
        } elif opcode == 43 { # '+'
            inc();
        } elif opcode == 45 { # '-'
            dec();
        } elif opcode == 62 { # '>'
```

```
            next();
        } elif opcode == 60 { # '<'
            prev();
        } elif opcode == 91 { # '['
            go_to_loop_end();
        } elif opcode == 93 { # ']'
            go_to_loop_beg();
        } else {
            putstr("Invalid opcode ");
            putwchar(opcode);
            return -1;
        }
        ip = ip + 1;
    }
    return 0;
}

fn main() -> int {
    putstr("Podaj program brainfucka: ");
    let size = 32768 : int;
    ram = malloc(size);
    code = malloc(size);
    memset(ram, 0, size);
    memset(code, 0, size);
    readstr(code, size);
    execute();

    free(ram);
    free(code);
    return 0;
}
```

Zaimplementowałem też w moim kompilatorze obsługe znaków UTF-8 dzięki czemu można używać tych znaków jako nazw funkcji / zmiennych (poza funkcją "main", ta nazwa musi pozostać).

```
extern fn putwchar(chr : int) -> int;

fn putstr(str: string) -> int {
    let ło=0 : int;
    while str[ło] {
        putwchar(str[ło]);
        ło = ło + 1;
    }
    return ło;
}

fn こにちは() -> int {
    putstr("Hello world!");
    return 0;
}

fn main() -> int {
    こにちは();
    return 0;
}
```

Użycie

Program umożliwia:

1. Kompilacje JIT

```
loczek@loczek-pc ~ $ ./rc --input-file=brainfuck.r --jit

Podaj program brainfucka: ++++++++++[>+++++++>++++++++++>+++>+
<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.>.
Hello World!
```

2. Kompilacje z wypisaniem pseudoassemblera LLVM

```
loczek@loczek-pc ~ $ ./rc --input-file=brainfuck.r --print-ir

; ModuleID = 'top'
source_filename = "top"
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128"
target triple = "x86_64-unknown-linux-gnu"

@0 = common global i32* null
@1 = common global i32* null
@2 = common global i32 0
@3 = common global i32 0
@4 = private unnamed_addr constant [17 x i8] c"i\00\00\00p\00\00\00
\00\00\00\00\00\00\00\00", align 1
@5 = private unnamed_addr constant [9 x i8] c"\0A\00\00\00\00\00\00\00",
align 1
@6 = private unnamed_addr constant [21 x i8] c"p\00\00\00t\00\00\00r\00\00\00
\00\00\00\00\00\00\00\00\00", align 1
@7 = private unnamed_addr constant [9 x i8] c"\0A\00\00\00\00\00\00\00",
align 1
```

3. Kompilacje z zapisaniem pseudoassemblera do pliku (można go potem uruchomić narzędziem lli lub skompilować do postaci binarnej clangiem)

```
loczek@loczek-pc ~ $ ./rc --input-file=brainfuck.r --output-file=code.ll --ir
loczek@loczek-pc ~ $ lli code.ll
Podaj program brainfucka: ++++++++++[>+++++++>++++++++++>+++>+
<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.>.
Hello World!
loczek@loczek-pc ~ $ clang code.ll -o bf
warning: overriding the module target triple with x86_64-pc-linux-gnu [-
Woverride-module]
1 warning generated.
loczek@loczek-pc ~ $ ./bf
Podaj program brainfucka: ++++++++++[>+++++++>++++++++++>+++>+
<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.>.
Hello World!
```

4. Kompilacje do bytecode'u LLVM

```
loczek@loczek-pc ~ $ ./rc --input-file=brainfuck.r --output-file=code.bc --bc
loczek@loczek-pc ~ $ lli code.bc
Podaj program brainfucka: ++++++++++[>+++++++>++++++++++>+++>+
<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.>.
Hello World!
loczek@loczek-pc ~ $ clang code.bc -o bf
warning: overriding the module target triple with x86_64-pc-linux-gnu [-
Woverride-module]
1 warning generated.
loczek@loczek-pc ~ $ ./bf
Podaj program brainfucka: ++++++++++[>+++++++>++++++++++>+++>+
<<<<-]>++.>+.+++++++..+++.>++.<<+++++++++++++++.>.+++.------.--------.>+.>.
Hello World!
```

Parsing:

Po leksykalizacji następuje parsowanie do drzewka składającego się z różnych strkutur opisujących konstrukcje językowe. W celu lepszej prezentacji zamieściłem wynik uzyskany z PrintVisitora, który wiernie odwzorowuje zawartość węzłów w drzewie AST.

1. Stałe:

- liczba 0 zostanie zapisana jako `[ int `0` ]`
- napis "OKI" zostanie zapisany jako `[ string `OKI` ]`

2. Wyrażenia są zapisywane jako

<operator> : {
  agrumenty
}

np wyrażenie 2+3+4 :

```
Plus : {
     Plus : {
          [ int `2` ]
          [ int `3` ]
     }
     [ int `4` ]
}
```

3. Referencja do zmiennej

```
[ get var `n` ]
```

4. Pętla for

```
for i in 0..n {

}
```

```
[ for loop variable name = `i`
     start = {
          [ int `0` ]
     },
     end = {
          [ get var `n` ]
     },
     increase = default;
     with body = {

     },
 end for ],
```

5. Petla while

```
while 1 {

}
```

```
[ while loop condition = {
          [ int `1` ]
     },
     with body = {

     },
end while ],
```

6. Funkcja

```
fn t(n : int ) -> int {
    for i in 0..n {

    }
    return 0;
}
```

```
[ make function name = `t`; return type = `Int`; args = {
           name = `n`; type = `Int`,
      }
      with body = {
                [ for loop variable name = `i`
                     start = {
                          [ int `0` ]
                     },
                     end = {
                          [ get var `n` ]
                     },
                     increase = default;
                     with body = {

                     },
                 end for ],
                Return : {
                     [ int `0` ]
```

```
                },
        }
]
```

Analiza semantyczna

Do analizy semantycznej wykorzystuje wizytator Analyser.

Ze względu na ograniczoną ilość typów do int, int* i string, analiza poprawności wyrażeń uprasza się do sprawdzenia typu wyrażenia z wymaganym typem dla danego operatora:

```
case BinaryOperator::Plus:
case BinaryOperator::Minus:
case BinaryOperator::Multiply:
case BinaryOperator::Divide:
case BinaryOperator::Modulo:
case BinaryOperator::And:
case BinaryOperator::Xor:
case BinaryOperator::Or:
case BinaryOperator::ShiftLeft:
case BinaryOperator::ShiftRight:
    require(SemanticAnalyser::ExprType::Int,
SemanticAnalyser::ExprType::IntReference);
    require(SemanticAnalyser::ExprType::Int,
SemanticAnalyser::ExprType::IntReference);
    yield(SemanticAnalyser::ExprType::Int, pos);
    break;
```

W przypadku większej ilości typów wymagana byłaby tablica operatorów, gdzie każdy element zawierałby informacje o wymaganych typach wraz z mechanizmem niejawnej zmiany typów (tego mechanizmu też nie am w moim języku).

Analizie podlega również wartość zwracana funkcji na podstawie typu wpisanego w deklaracji oraz czy wszystkie ścieżki kończą się instrukcją "return".

Zmienne i scoping

W celu zapewnienia poprawnego działania Shadowingu i rejestracji zmienncyh w analizatorze zdefiniowany jest scope zmiennych jako :

```
std::deque<std::unordered_map<std::wstring, BuiltinType>> scopes
```

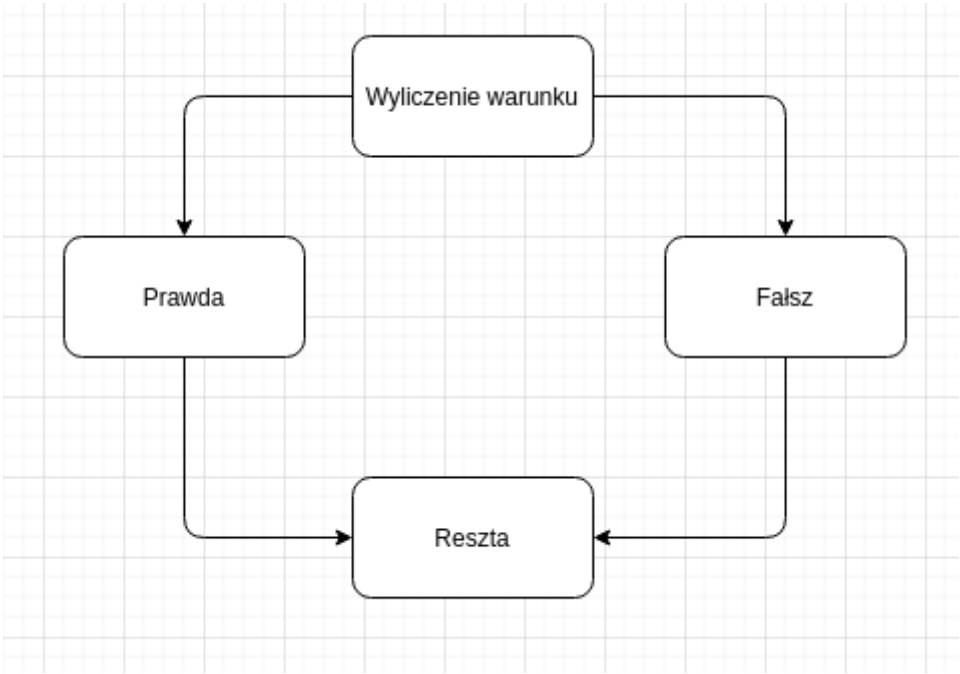
w momęcie wejścia w blok na koniec dodawana jest na koniec nowa mapa, a przy wyjściu ostatnia jest usuwana. Szukanie zmiennych następuje od końca. Przyjąłem konwencje że pierwszy scope jest scopem zmiennych globalnych a ostatni jest lokalnym obecnego bloku.

Interfejs do LLVM

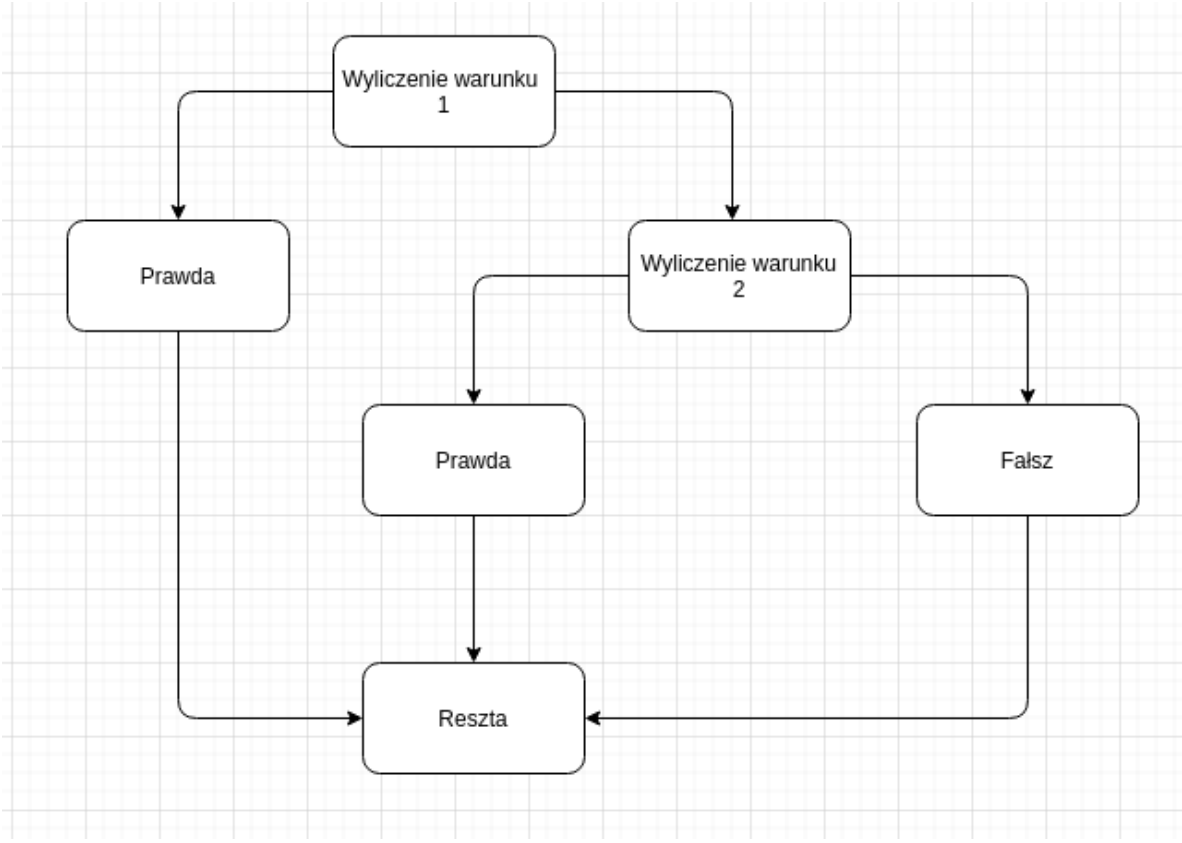
W projekcie do generowania pseudoassemblera LLVM wykorzystuje llvm::IRBuilder do generowania wyrażen oraz llvm::BasicBlock do agregacji instrukcji w bloki.

Odwzorowanie konstrukcji językowych w blokach assemblera:

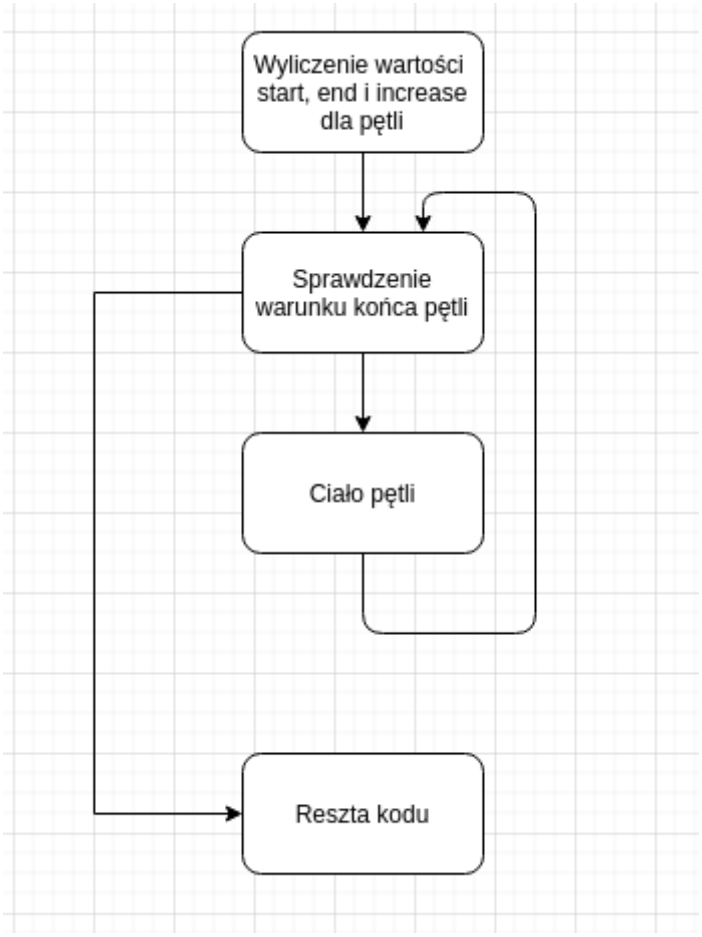
1. if



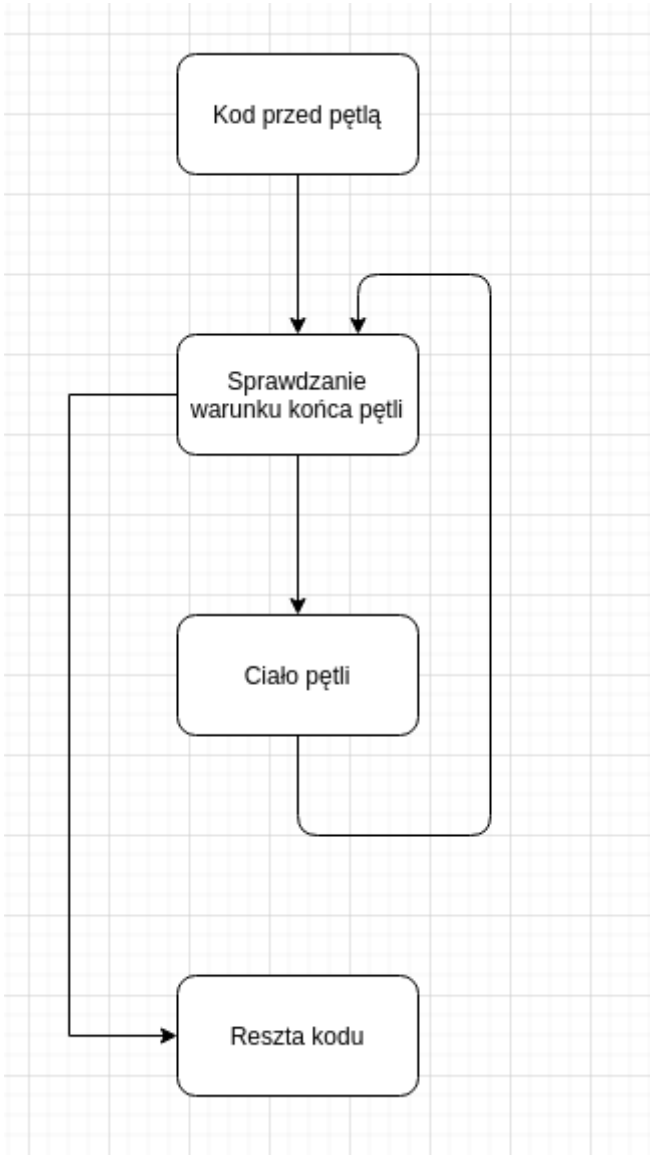
2. if w if'ie



3. Pętla for



4. Pętla while



Po skompilowaniu drzewa AST otrzymujemy moduł llvm'a który jest poddawany optymalizacji. Na początku usunięciu martwych instrukcji, okazuje się że llvm bardzo nielubi jak w danym bloku są np 2 instrukcje ret lub br. Następnie uruchamiam typowe przejścia optymalizacyjne z slajdu z wykładu.

```
    auto FPM = std::make_unique<llvm::legacy::FunctionPassManager>
(module.get());
    FPM->add(llvm::createInstructionCombiningPass());
    FPM->add(llvm::createReassociatePass());
    FPM->add(llvm::createGVNPass());
    FPM->add(llvm::createCFGSimplificationPass());
    FPM->doInitialization();

    for (auto & function : functions) {
        FPM->run(*function.second.llvm_ptr);
    }
```

W przypadku wypisania assemblera wykonywane jest

```
void LLVMCompiler::print_ir() {
    module->print(llvm::outs(), nullptr);
}
```

Zapisanie do pliku:

```
void LLVMCompiler::save_ir(const std::string& path) {
    std::error_code ec;
    llvm::raw_fd_ostream fd(path, ec, llvm::sys::fs::F_None);
    fd << *module;
}

void LLVMCompiler::save_bc(const std::string& path) {
    std::error_code ec;
    llvm::raw_fd_ostream fd(path, ec, llvm::sys::fs::F_None);
    llvm::WriteBitcodeToFile(*module, fd);
}
```

I na końcu kompilacja JIT, dzięki LLVM wystraczy skorzystać z llvm::ExecutionEngine, znaleźć funkcje main i ją wykonać:

```
int LLVMCompiler::execute() {
    llvm::InitializeNativeTarget();
    llvm::InitializeNativeTargetAsmPrinter();
    llvm::InitializeNativeTargetAsmParser();
    std::string err;
    llvm::EngineBuilder engine_builder(std::move(module));
    engine_builder.setEngineKind(llvm::EngineKind::JIT).setErrorStr(&err);
    llvm::ExecutionEngine *execution_engine = engine_builder.create();
    if (!execution_engine) {
        report_jit_creation_error(err);
    }
    std::vector<llvm::GenericValue> noargs;
    return static_cast<int>(execution_engine->runFunction(entrypoint_function,
noargs).IntVal.getLimitedValue());
}
```

Użyte w kodzie powyżej "entrypoint_function" jest specjalnie syntezowaną funkcją gdzie następuje inicjalizacja zmiennych globalnych i wołanie funkcji "main". Pełni to rolę funkcji "__libc_start_main" oraz listy inicjalizacyjnej występującej w tradycyjnym pliku wykonywalnym.