

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»**

**Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії**

**Лабораторна робота № 4
ТРАНСЛЯЦІЯ У ПОЛІЗ**

(з дисципліни «Побудова компіляторів»)

Звіт студента I курсу, групи ІП-51мн
спеціальності F2 Інженерія програмного забезпечення

Панченко С. В.

(Прізвище, ім'я, по батькові)

(Підпис)

Перевірив: доцент, к.т.н. Стативка Ю.І.

(Посада, науковий ступінь, прізвище та ініціали)

(Підпис)

Київ – 2025

Зміст

1 Мета.....	3
2 Виконання.....	4
3 Висновок.....	26

1 МЕТА

Програмна реалізація синтаксичного аналізатора (парсера) заданої мови методом рекурсивного спуску.

2 ВИКОНАННЯ

У моєму компіляторі реалізовано етап трансляції коду мови, синтаксично подібної до Lisp, у проміжне представлення (IR – Intermediate Representation). Цей етап відбувається після лексичного та синтаксичного аналізу, коли вихідний текст уже перетворено у дерево S-виразів типу `LispList`. Основною метою цього етапу є побудова внутрішньої структури, яка є зручною для подальшої генерації машинного коду та перевірки семантики програми.

IR-трансляція починається з побудови об'єктів типу `SyntaxFunctionDefinition`, що описують кожну функцію мови: її ім'я, тип повернення, список аргументів, локальні змінні та тіло програми. Далі ці об'єкти обробляються функцією `ir_fn_defs_build`, яка створює дві форми представлення: декларацію функції (`IrFnDecl`) та визначення функції (`IrFnDef`). У декларації зберігається інформація про сигнатуру функції, тоді як у визначенні формується послідовність операторів нижчого рівня, подібних до машинних інструкцій.

Кожен оператор програми відображається на свій IR-тип: присвоєння (`IrStmtSet`), умовний оператор (`IrStmtIf`), цикл (`IrStmtWhile`) або оператор повернення (`IrStmtReturn`). Для кожного оператора виконується семантична перевірка — зокрема, відповідність типів змінних, правильність логічних умов та коректність типу повернення функції. Наприклад, у конструкції `if` або `while` перевіряється, що умова має тип `BOOL`, а в операторі `set` — що тип правої частини збігається з типом лівої змінної.

Особливе місце у трансляції займають виклики функцій. Вони представлені у вигляді `IrFnCall`, який може посилатися як на користувацькі функції, так і на вбудовані (`BuiltinFunction`). Для кожного виклику перевіряється кількість аргументів і їхні типи. Якщо збіг знайдено, створюється IR-виклик, що вже містить посилання на реальну функцію та список аргументів у вигляді об'єктів

IrFnArg або Constant. Таким чином реалізується повна типова перевірка викликів і відповідність сигнатурам.

Отримане IR-подання є повністю типізованим і структурованим. Воно дозволяє не лише зручно будувати стекові фрейми та керувати регістрами при генерації коду, а й легко перевіряти коректність усієї програми до етапу асемблерної трансляції. Завдяки цьому IR виступає проміжним шаром між синтаксичним деревом і машинним кодом, забезпечуючи чітке розділення логічної та фізичної частин компіляції.

```
import sys
import itertools
import re
import enum
import typing
import pathlib
import contextlib

class TokenLparen(typing.NamedTuple):
    line_number: int
    column_number: int

class TokenRparen(typing.NamedTuple):
    line_number: int
    column_number: int

class TokenIdentifier(typing.NamedTuple):
    line_number: int
    column_number: int
    value: str

Token: typing.TypeAlias = TokenLparen | TokenRparen |
TokenIdentifier

def panic(msg: str = '') -> typing.NoReturn:
    import traceback
    traceback.print_stack()
    print(msg)
    sys.exit(-1)

def tokenize(filepath: pathlib.Path | str) ->
typing.Iterator[Token]:
    lparen_re = re.compile(r'[()]')
```

```

rparen_re = re.compile(r'[)]')
identifier_re = re.compile(r'[a-zA-Z!$%&*/+\-:;<=>?^_~\.0-9]+')
empty_re = re.compile(r'[\t\r]+')
newline_re = re.compile(r'[\n]')

with open(filepath, 'r') as file:
    data = file.read()

offset = 0
line_number = 1
last_newline_offset = 0

while offset < len(data):
    column_number = offset - last_newline_offset + 1
    subdata = data[offset:]
    if (resmatch := lparen_re.match(subdata)):
        match_len = resmatch.end()
        offset += match_len
        yield TokenLparen(line_number, column_number)
    elif (resmatch := rparen_re.match(subdata)):
        match_len = resmatch.end()
        offset += match_len
        yield TokenRparen(line_number, column_number)
    elif (resmatch := identifier_re.match(subdata)):
        match_len = resmatch.end()
        offset += match_len
        yield TokenIdentifier(line_number, column_number,
resmatch.group(0))
    elif (resmatch := empty_re.match(subdata)):
        match_len = resmatch.end()
        offset += match_len
    elif (resmatch := newline_re.match(subdata)):
        line_number += 1
        last_newline_offset = offset + 1
        offset += 1
    else:
        panic(f"Error: Unrecognized symbol at line
{line_number}, position {column_number}")

class LispList(typing.NamedTuple):
    lparen: TokenLparen
    elements: list[typing.Union['LispList', 'TokenIdentifier']]

def build_lisp_tree(lisp_list: LispList, tokenizer:
typing.Iterator[Token], lparens: list[TokenLparen]):
    for token in tokenizer:
        if isinstance(token, TokenLparen):
            lparens.append(token)
            sub_lisp_list = LispList(token, [])

```

```

        build_lisp_tree(sub_lisp_list, tokenizer, lpargs)
        lisp_list.elements.append(sub_lisp_list)
    elif isinstance(token, TokenRparen):
        lpargs.pop()
        return
    else:
        lisp_list.elements.append(token)

def at_line(token: Token | LispList) -> str:
    if isinstance(token, Token):
        return f'at line {token.line_number}, column
{token.column_number}'
    else:
        return f'at line {token.lparen.line_number}, column
{token.lparen.column_number}'

def check_lisp_element_is_type(el: list[TokenIdentifier] |
TokenIdentifier):
    assert isinstance(el, TokenIdentifier)

class VarType(enum.StrEnum):
    INT = 'int'
    FLOAT = 'float'
    BOOL = 'bool'

IDENTIFIER_RE = re.compile("[a-zA-Z!$%&*/:<=>?^_~][a-zA-Z!$%&*/:<=>?
^_~0-9]*|([+]|[-])")
INT_RE = re.compile("[+-]?[0-9]+")
FLOAT_RE = re.compile("[+-]?[0-9]+[.][0-9]+")

class VarTypePair(typing.NamedTuple):
    token: TokenIdentifier
    vartype: VarType

class SyntaxList[T](typing.NamedTuple):
    lisp_list: LispList
    syntax_list: list[T]

def syntax_parse_arg_list(arg_list: LispList | TokenIdentifier) ->
SyntaxList[VarTypePair]:
    if isinstance(arg_list, LispList):
        syntax_list = SyntaxList[VarTypePair](arg_list, [])
        for name_type_pair in arg_list.elements:
            if isinstance(name_type_pair, LispList):
                if len(name_type_pair.elements) != 2:
                    panic(f'Error: Name type pair must have 2
elements {at_line(name_type_pair.lparen)}')
                arg_name = name_type_pair.elements[0]
                if isinstance(arg_name, TokenIdentifier):

```

```

        if IDENTIFIER_RE.fullmatch(arg_name.value) is
None:
            panic(f'Error: Argument name does not match
identifier pattern {at_line(arg_name)}')
            arg_type = name_type_pair.elements[1]
            if isinstance(arg_type, TokenIdentifier):
                if arg_type.value not in VarType:
                    panic(f'Error: Argument type is not
valid {at_line(arg_type)}')

syntax_list.syntax_list.append(VarTypePair(arg_name,
VarType(arg_type.value)))
            else:
                panic(f'Error: Argument type must be an atom
{at_line(arg_type.lparen)}')
            else:
                panic(f'Error: Argument name must be an atom
{at_line(arg_name.lparen)}')
            else:
                panic(f'Error: Name type pair must be a list
{at_line(name_type_pair)}')
                return syntax_list
            else:
                panic(f'Error: Argument list must be a list
{at_line(arg_list)}')

def check_atom_identifier(lisp_element: LispList | TokenIdentifier)
-> TokenIdentifier:
    if isinstance(lisp_element, TokenIdentifier):
        if IDENTIFIER_RE.fullmatch(lisp_element.value):
            return lisp_element
        else:
            panic(f'Error: Element is not valid identifier
{at_line(lisp_element)}')
    else:
        panic(f'Error: Element must be an atom
{at_line(lisp_element.lparen)}')

class SyntaxBool(enum.StrEnum):
    TRUE = 'true'
    FALSE = 'false'

class ConstantBool(typing.NamedTuple):
    value: TokenIdentifier

class ConstantFloat(typing.NamedTuple):
    value: TokenIdentifier

class ConstantInt(typing.NamedTuple):

```

```
value: TokenIdentifier
```

```
Constant = typing.Union[ConstantInt, ConstantFloat, ConstantBool]
```

```
class SyntaxVariable(typing.NamedTuple):  
    value: TokenIdentifier
```

```
SyntaxVariableOrConstant = SyntaxVariable | Constant
```

```
def syntax_parse_var_or_const(lisp_element: LispList |  
TokenIdentifier, constants_list: list[Constant]) ->  
SyntaxVariableOrConstant:  
    if isinstance(lisp_element, TokenIdentifier):  
        if lisp_element.value in SyntaxBool:  
            constants_list.append(ConstantBool(lisp_element))  
            return constants_list[-1]  
        elif INT_RE.fullmatch(lisp_element.value):  
            constants_list.append(ConstantInt(lisp_element))  
            return constants_list[-1]  
        elif FLOAT_RE.fullmatch(lisp_element.value):  
            constants_list.append(ConstantFloat(lisp_element))  
            return ConstantFloat(lisp_element)  
        else:  
            return SyntaxVariable(lisp_element)  
    else:  
        panic(f'Error: Element must be an atom  
{at_line(lisp_element.lparen)}')
```

```
class SyntaxFunctionCall(typing.NamedTuple):  
    lisp_list: LispList  
    name: TokenIdentifier  
    arguments: tuple[SyntaxVariableOrConstant, ...]
```

```
SyntaxVarOrConstOrFuncCall = SyntaxVariableOrConstant |  
SyntaxFunctionCall
```

```
def syntax_parse_var_or_const_or_func_call(statement_argument:  
LispList | TokenIdentifier, constants_list: list[Constant]) ->  
SyntaxVarOrConstOrFuncCall:  
    if isinstance(statement_argument, TokenIdentifier):  
        return syntax_parse_var_or_const(statement_argument,  
constants_list)  
    else:  
        return SyntaxFunctionCall(  
            statement_argument,  
            check_atom_identifier(statement_argument.elements[0]),  
            tuple(syntax_parse_var_or_const(el, constants_list) for  
el in statement_argument.elements[1:])  
        )
```

```

class SyntaxStatementSet(typing.NamedTuple):
    lisp_list: LispList
    dest: TokenIdentifier
    src: SyntaxVarOrConstOrFuncCall

class SyntaxStatementReturn(typing.NamedTuple):
    lisp_list: LispList
    value: SyntaxVarOrConstOrFuncCall

class SyntaxStatementIf(typing.NamedTuple):
    lisp_list: LispList
    condition: SyntaxVarOrConstOrFuncCall
    true_branch: SyntaxList['SyntaxStatement']
    false_branch: SyntaxList['SyntaxStatement']

class SyntaxStatementWhile(typing.NamedTuple):
    lisp_list: LispList
    condition: SyntaxVarOrConstOrFuncCall
    statements: SyntaxList['SyntaxStatement']

SyntaxStatement = SyntaxStatementSet | SyntaxStatementIf |
SyntaxStatementWhile | SyntaxStatementReturn

def syntax_parse_statement_list(lisp_statement_list: LispList |
TokenIdentifier, constants_list: list[Constant]) ->
SyntaxList[SyntaxStatement]:
    if isinstance(lisp_statement_list, LispList):
        statements = SyntaxList[SyntaxStatement]
        (lisp_statement_list, [])
        for lisp_statement in lisp_statement_list.elements:
            if isinstance(lisp_statement, LispList):
                if len(lisp_statement.elements) == 0:
                    panic(f'Error: Statement must be a non-empty
list {at_line(lisp_statement.lparen)}')
                statement_name = lisp_statement.elements[0]
                if isinstance(statement_name, TokenIdentifier):
                    if statement_name.value == 'set':
                        if len(lisp_statement.elements) != 3:
                            panic(f'Error: Set statement list must
have 3 elements {at_line(lisp_statement.lparen)}')
                        statements.syntax_list.append(
                            SyntaxStatementSet(
                                lisp_statement,

check_atom_identifier(lisp_statement.elements[1]),

syntax_parse_var_or_const_or_func_call(lisp_statement.elements[2],
constants_list)

```

```

        )
    )
    elif statement_name.value == 'if':
        if len(lisp_statement.elements) != 4:
            panic(f'Error: If statement list must
have 4 elements {at_line(lisp_statement.lparen)}')
            statements.syntax_list.append(
                SyntaxStatementIf(
                    lisp_statement,

syntax_parse_var_or_const_or_func_call(lisp_statement.elements[1],
constants_list),

syntax_parse_statement_list(lisp_statement.elements[2],
constants_list),

syntax_parse_statement_list(lisp_statement.elements[3],
constants_list),

        )
    )
    elif statement_name.value == 'while':
        if len(lisp_statement.elements) != 3:
            panic(f'Error: While statement list must
have 3 elements {at_line(lisp_statement.lparen)}')
            statements.syntax_list.append(
                SyntaxStatementWhile(
                    lisp_statement,

syntax_parse_var_or_const_or_func_call(lisp_statement.elements[1],
constants_list),

syntax_parse_statement_list(lisp_statement.elements[2],
constants_list),

        )
    )
    elif statement_name.value == 'return':
        if len(lisp_statement.elements) != 2:
            panic(f'Error: Return statement list
must have 2 elements {at_line(lisp_statement.lparen)}')
            statements.syntax_list.append(
                SyntaxStatementReturn(
                    lisp_statement,

syntax_parse_var_or_const_or_func_call(lisp_statement.elements[1],
constants_list)

        )
    )
else:

```

```

        panic(f'Error: Statement name is not valid
{at_line(statement_name)}}')
    else:
        panic(f'Error: Statement name must be an atom
{at_line(statement_name.lparen)}}')
    else:
        panic(f'Error: Statement must be a list
{at_line(lisp_statement)}}')
    return statements
else:
    panic(f'Error: Statement list must be a list
{at_line(lisp_statement_list)}}')

class SyntaxFunctionDefinition(typing.NamedTuple):
    lisp_list: LispList
    name: TokenIdentifier
    return_type: VarTypePair
    arguments: SyntaxList[VarTypePair]
    variables: SyntaxList[VarTypePair]
    statements: SyntaxList[SyntaxStatement]

def syntax_parse_function_definitions(lisp_tree: LispList,
constants_list: list[Constant]) ->
typing.Iterator[SyntaxFunctionDefinition]:
    for fn_def in lisp_tree.elements:
        if isinstance(fn_def, LispList):
            if len(fn_def.elements) != 6:
                panic('')
            if isinstance(fn_def.elements[0], TokenIdentifier):
                if fn_def.elements[0].value != 'fn':
                    panic(f'Error: Function must start with fn
{at_line(fn_def.elements[0])}')
                if isinstance(fn_def.elements[1], TokenIdentifier):
                    if
IDENTIFIER_RE.fullmatch(fn_def.elements[1].value) is None:
                        panic(f'Error: Function name does not match
identifier pattern {at_line(fn_def.elements[1])}')
                    syn_fn_def_name = fn_def.elements[1]
                    if isinstance(fn_def.elements[2],
TokenIdentifier):
                        if fn_def.elements[2].value not in VarType:
                            panic(f'Error: Function return type is
not valid {at_line(fn_def.elements[2])}')
                        syn_fn_def_return_type =
VarTypePair(fn_def.elements[2], VarType(fn_def.elements[2].value))
                        syn_fn_def_arguments =
syntax_parse_arg_list(fn_def.elements[3])
                        syn_fn_def_variables =
syntax_parse_arg_list(fn_def.elements[4])

```

```

        syn_fn_def_statements =
syntax_parse_statement_list(fn_def.elements[5], constants_list)
        yield SyntaxFunctionDefinition(fn_def,
syn_fn_def_name, syn_fn_def_return_type, syn_fn_def_arguments,
syn_fn_def_variables, syn_fn_def_statements)
        else:
            panic(f'Error: Function return type must be
an atom {at_line(fn_def.elements[2].lparen)}')
        else:
            panic(f'Error: Function name must be an atom
{at_line(fn_def.elements[1].lparen)}')
        else:
            panic(f'Error: Function must start with atom
{at_line(fn_def.elements[0].lparen)}')
        else:
            panic(f'Error: Function definition must be a list
{at_line(fn_def)}')

```

```

class BuiltinFunction(typing.NamedTuple):
    name: str
    return_type: VarType
    argtypes: tuple[VarType, ...]
    asm_code: str

```

```

BULTIN_FUNCTIONS = (
    BuiltinFunction('==', VarType.BOOL, (VarType.INT, VarType.INT),
    """
cmp x0, x1
cset x0, eq
ret
    """,
    ),
    BuiltinFunction('==', VarType.BOOL, (VarType.BOOL,
VarType.BOOL),
    """
cmp x0, x1
cset x0, eq
ret
    """,
    ),
    BuiltinFunction('*', VarType.INT, (VarType.INT, VarType.INT),
    """
mul x0, x0, x1
ret
    """,
    ),
    BuiltinFunction('/', VarType.INT, (VarType.INT, VarType.INT),
    """

```

```

sdiv x0, x0, x1
ret
"""
),
    BuiltinFunction('+', VarType.INT, (VarType.INT, VarType.INT),
    """
add x0, x0, x1
ret
"""
),
    BuiltinFunction('-', VarType.INT, (VarType.INT, VarType.INT),
    """
sub x0, x0, x1
ret
"""
),

    BuiltinFunction('>', VarType.BOOL, (VarType.INT, VarType.INT),
    """
cmp x0, x1
cset x0, gt
ret
"""
),
    BuiltinFunction('<', VarType.BOOL, (VarType.INT, VarType.INT),
    """
cmp x0, x1
cset x0, lt
ret
"""
),
    BuiltinFunction('>=', VarType.BOOL, (VarType.INT, VarType.INT),
    """
cmp x0, x1
cset x0, ge
ret
"""
),
    BuiltinFunction('<=', VarType.BOOL, (VarType.INT, VarType.INT),
    """
cmp x0, x1
cset x0, le
ret
"""
),
    BuiltinFunction('*', VarType.FLOAT, (VarType.FLOAT,
VarType.FLOAT),
    """
fmul d0, d0, d1

```

```

ret
"""
),
    BuiltinFunction('/', VarType.FLOAT, (VarType.FLOAT,
VarType.FLOAT),
    """
fdiv d0, d0, d1
ret
"""
),
    BuiltinFunction('+', VarType.FLOAT, (VarType.FLOAT,
VarType.FLOAT),
    """
fadd d0, d0, d1
ret
"""
),
    BuiltinFunction('-', VarType.FLOAT, (VarType.FLOAT,
VarType.FLOAT),
    """
fsub d0, d0, d1
ret
"""
),
    BuiltinFunction('==', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
    """
fcmp d0, d1
cset x0, eq
ret
"""
),
    BuiltinFunction('>', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
    """
fcmp d0, d1
cset x0, gt
ret
"""
),
    BuiltinFunction('<', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
    """
fcmp d0, d1
cset x0, lt
ret
"""
),

```

```

        BuiltinFunction('>=', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
        """
fcmp d0, d1
cset x0, ge
ret
        """
    ),
        BuiltinFunction('<=', VarType.BOOL, (VarType.FLOAT,
VarType.FLOAT),
        """
fcmp d0, d1
cset x0, le
ret
        """
    ),
)

class IrFnArg(typing.NamedTuple):
    name: TokenIdentifier
    type_: VarType

class IrFnDecl(typing.NamedTuple):
    name: TokenIdentifier
    return_type: VarType
    arguments: tuple[IrFnArg, ...]
    variables: tuple[IrFnArg, ...]

IrFnCallArg = IrFnArg | Constant

class IrFnCall(typing.NamedTuple):
    fn: IrFnDecl | BuiltinFunction
    arguments: tuple[IrFnCallArg, ...]

def ir_get_variable(fn_def: IrFnDecl, token: TokenIdentifier) ->
IrFnArg:
    for var in fn_def.arguments:
        if var.name.value == token.value:
            return var
    for var in fn_def.variables:
        if var.name.value == token.value:
            return var
    panic(f'Error: token "{token.value}" is not a variable nor
parameter {at_line(token)}')

def ir_get_type(var: IrFnCallArg):
    if isinstance(var, IrFnArg):
        return var.type_
    elif isinstance(var, ConstantBool):

```

```

        return VarType.BOOL
    elif isinstance(var, ConstantFloat):
        return VarType.FLOAT
    else:
        return VarType.INT

def ir_fn_call_build(
    func_call: SyntaxFunctionCall,
    fn_decls: typing.Iterable[IrFnDecl],
    cur_fn_decls: IrFnDecl
) -> IrFnCall:
    arguments: list[IrFnCallArg] = []
    for arg in func_call.arguments:
        if isinstance(arg, SyntaxVariable):
            arguments.append(ir_get_variable(cur_fn_decls,
arg.value))
        else:
            arguments.append(arg)
    fn_call_types = [ir_get_type(el) for el in arguments]
    for fn_def in fn_decls:
        if func_call.name.value != fn_def.name:
            continue
        if len(func_call.arguments) != len(fn_def.arguments):
            continue
        fn_def_arg_types = list(el.type_ for el in fn_def.arguments)
        if any(t1 != t2 for t1, t2 in zip(fn_call_types,
fn_def_arg_types)):
            continue
        return IrFnCall(fn_def, tuple(arguments))
    for builtin in BULTIN_FUNCTIONS:
        if func_call.name.value != builtin.name:
            continue
        if len(func_call.arguments) != len(builtin.argtypes):
            continue
        if any(t1 != t2 for t1, t2 in zip(fn_call_types,
builtin.argtypes)):
            continue
        return IrFnCall(builtin, tuple(arguments))
    panic(f'Error: Function call does not match any functions
{at_line(func_call.lisp_list)}')

IrStmtArg = IrFnCallArg | IrFnCall

class IrStmtSet(typing.NamedTuple):
    dest: IrFnArg
    src: IrStmtArg

class IrStmtIf(typing.NamedTuple):
    cond: IrStmtArg

```

```

    branch_true: tuple['IrStmt', ...]
    branch_false: tuple['IrStmt', ...]

class IrStmtWhile(typing.NamedTuple):
    cond: IrStmtArg
    body: tuple['IrStmt', ...]

class IrStmtReturn(typing.NamedTuple):
    val: IrStmtArg

IrStmt = IrStmtWhile | IrStmtSet | IrStmtIf | IrStmtReturn

def ir_stmt_arg_build(fn_defs: typing.Iterable[IrFnDecl],
current_fn_decl: IrFnDecl, var: SyntaxVarOrConstOrFuncCall):
    if isinstance(var, SyntaxVariable):
        src_var = ir_get_variable(current_fn_decl, var.value)
        src_type = src_var.type_
    elif isinstance(var, Constant):
        src_type = ir_get_type(var)
        src_var = var
    else:
        src_var = ir_fn_call_build(var, fn_defs, current_fn_decl)
        src_type = src_var.fn.return_type
    return src_var, src_type

def ir_statement_list(fn_decls: typing.Iterable[IrFnDecl],
current_fn_decl: IrFnDecl, statements: SyntaxList[SyntaxStatement])
-> typing.Iterator[IrStmt]:
    for statement in statements.syntax_list:
        if isinstance(statement, SyntaxStatementSet):
            dest_var = ir_get_variable(current_fn_decl,
statement.dest)
            var, type_ = ir_stmt_arg_build(fn_decls,
current_fn_decl, statement.src)
            if type_ != dest_var.type_:
                panic(f'Error: Type mismatch in statement set
{at_line(statement.lisp_list)}')
            yield IrStmtSet(dest_var, var)
        elif isinstance(statement, SyntaxStatementIf):
            var, type_ = ir_stmt_arg_build(fn_decls,
current_fn_decl, statement.condition)
            if type_ != VarType.BOOL:
                panic(f'Error: Condition must be bool
{at_line(statement.lisp_list)}')
            yield IrStmtIf(
                var,
                tuple(ir_statement_list(fn_decls, current_fn_decl,
statement.true_branch)),

```

```

        tuple(ir_statement_list(fn_decls, current_fn_decl,
statement.false_branch))
    )
    elif isinstance(statement, SyntaxStatementWhile):
        var, type_ = ir_stmt_arg_build(fn_decls,
current_fn_decl, statement.condition)
        if type_ != VarType.BOOL:
            panic(f'Error: Condition must be bool
{at_line(statement.lisp_list)}')
        yield IrStmtWhile(
            var,
            tuple(ir_statement_list(fn_decls, current_fn_decl,
statement.statements)),
        )
    else:
        var, type_ = ir_stmt_arg_build(fn_decls,
current_fn_decl, statement.value)
        if current_fn_decl.return_type != type_:
            panic(f'Function does not return value of return
type {at_line(statement.lisp_list)}')
        yield IrStmtReturn(var)

class IrFnDef(typing.NamedTuple):
    decl: IrFnDecl
    body: tuple[IrStmt, ...]

def ir_fn_defs_build(syn_fn_defs:
typing.Iterable[SyntaxFunctionDefinition]):
    for fn_def in syn_fn_defs:
        for el in fn_def.statements.syntax_list[:-1]:
            if isinstance(el, SyntaxStatementReturn):
                panic(f'Error: Return statement must be the last one
{at_line(el.lisp_list)}')
            if len(fn_def.statements.syntax_list) == 0:
                panic(f'Error: Function statement list must have at
least one statement {at_line(fn_def.statements.lisp_list)}')
            if not isinstance(fn_def.statements.syntax_list[-1],
SyntaxStatementReturn):
                panic(f'Error: Last statement in function definition
statement list must be return statement
{at_line(fn_def.statements.lisp_list)}')

            it_first = itertools.chain(fn_def.arguments.syntax_list,
fn_def.variables.syntax_list)
            it_second = itertools.chain(fn_def.arguments.syntax_list,
fn_def.variables.syntax_list)
            for i_one, el_one in enumerate(it_first):
                for i_two, el_two in enumerate(it_second):
                    if i_one == i_two:

```

```

        continue
        if el_one.token.value == el_two.token.value:
            panic(f'Error: Duplicate argument name
{at_line(el_one.token)} and {at_line(el_two.token)}')

    for i_one, fn_def_one in enumerate(syn_fn_defs):
        args_one = [el.vartype for el in
fn_def_one.arguments.syntax_list]
        for i_two, fn_def_two in enumerate(syn_fn_defs):
            if i_one == i_two:
                continue
            if fn_def_one.name.value != fn_def_two.name.value:
                continue
            args_two = [el.vartype for el in
fn_def_two.arguments.syntax_list]
            if len(args_one) != len(args_two):
                continue
            if any(a1 != a2 for a1, a2 in zip(args_one, args_two)):
                continue
            panic(f'Error: Duplicate function definitions
{at_line(fn_def_one.lisp_list)} and
{at_line(fn_def_two.lisp_list)}')

    for fn_def in syn_fn_defs:
        args_one = [el.vartype for el in
fn_def.arguments.syntax_list]
        for builtin_def in BULTIN_FUNCTIONS:
            if builtin_def.name != fn_def.name.value:
                continue
            if len(args_one) != len(builtin_def.argtypes):
                continue
            if any(a1 != a2 for a1, a2 in zip(args_one,
builtin_def.argtypes)):
                continue
            panic(f'Error: Duplicate function definition with
builtin "{builtin_def.name}" {at_line(fn_def.lisp_list)}')

    def _make_fn_def(fn_def: SyntaxFunctionDefinition):
        args = tuple(IrFnArg(el.token, el.vartype) for el in
fn_def.arguments.syntax_list)
        vars = tuple(IrFnArg(el.token, el.vartype) for el in
fn_def.variables.syntax_list)
        return IrFnDecl(fn_def.name, fn_def.return_type.vartype,
args, vars)

    fn_decls = tuple(_make_fn_def(fn_def) for fn_def in syn_fn_defs)
    fn_defs = tuple(
        IrFnDef(fn_decl, tuple(ir_statement_list(fn_decls, fn_decl,
syn_fn_def.statements)))

```

```

        for fn_decl, syn_fn_def in zip(fn_decls, syn_fn_defs)
    )
    return fn_defs

# class Aarch64Register(enum.Enum):
#     X0 = enum.auto()
#     X1 = enum.auto()
#     X2 = enum.auto()
#     X3 = enum.auto()
#     X4 = enum.auto()
#     X5 = enum.auto()
#     X6 = enum.auto()
#     X7 = enum.auto()
#     X8 = enum.auto()
#     X9 = enum.auto()
#     X10 = enum.auto()

@contextlib.contextmanager
def asm_stack_frame(fn_decl: IrFnDecl):
    with contextlib.ExitStack() as exit_stack:
        exit_stack.callback(print, 'ret')
        print('sub sp, sp, #16')
        exit_stack.callback(print, 'add sp, sp, #16')
        print('str lr, [sp]')
        exit_stack.callback(print, 'ldr lr, [sp]')
        print('sub sp, sp, #16')
        exit_stack.callback(print, 'add sp, sp, #16')
        print('str fp, [sp]')
        exit_stack.callback(print, 'ldr fp, [sp]')
        print('mov fp, sp')
        exit_stack.callback(print, 'mov sp, fp')
        int_var_count = 0
        float_var_count = 0
        stack_var_count = 0
        fp_offset = 0
        arg_off: dict[IrFnArg, int] = dict()
        for arg in fn_decl.arguments:
            arg_off[arg] = fp_offset
            print('sub sp, sp, #16')
            exit_stack.callback(print, 'add sp, sp, #16')
            match arg.type_:
                case VarType.BOOL | VarType.INT:
                    if int_var_count < 8:
                        print(f'str x{int_var_count}, [fp, #-
{fp_offset}]]')
                    else:
                        print(f'ldr x9, [fp, #{32 + stack_var_count
* 16}]]')
                        print(f'str x9, [fp, #-{fp_offset}]]')

```

```

        stack_var_count += 1
        int_var_count += 1
    case VarType.FLOAT:
        if float_var_count < 8:
            print(f'str d{int_var_count}, [fp, #-
{fp_offset}}]')
        else:
            print(f'ldr d9, [fp, #{32 + stack_var_count
* 16}}]')
            print(f'str d9, [fp, #-{fp_offset}}]')
            stack_var_count += 1
            float_var_count += 1
        fp_offset += 16
    for arg in fn_decl.variables:
        arg_off[arg] = fp_offset
        print('sub sp, sp, #16')
        exit_stack.callback(print, 'add sp, sp, #16')
        print('mov x9, #0')
        print(f'str x9, [fp, #-{fp_offset}}]')
        fp_offset += 16
    yield arg_off

def asm_generate(ir_fn_defs: typing.Iterable[IrFnDef], constants:
typing.Iterable[Constant]):
    print('.data')
    const_table = dict((const, f'const{i}') for i, const in
enumerate(constants))
    for const in constants:
        if isinstance(const, ConstantBool) or isinstance(const,
ConstantInt):
            type_ = 'dword'
            val = int(const.value.value)
        else:
            type_ = 'double'
            val = float(const.value.value)
        print('.align 8')
        print(f'{const_table[const]}: .{type_} {val}')
    print('.text')
    fn_name_table = dict((fn, f'fn{i}') for i, fn in
enumerate(itertools.chain(BULTIN_FUNCTIONS, ir_fn_defs)))
    for fn_def in BULTIN_FUNCTIONS:
        print(f'// {fn_def.name}')
        print(f'.global {fn_name_table[fn_def]}')
        print(f'{fn_name_table[fn_def]}:')
        print(fn_def.asm_code.strip())
    for fn_def in ir_fn_defs:
        print(f'// {fn_def.decl.name.value}')
        print(f'.global {fn_name_table[fn_def]}')
        print(f'{fn_name_table[fn_def]}:')

```

```

        with asm_stack_frame(fn_def.decl) as arg_off:
            for stmt in fn_def.body:
                if isinstance(stmt, IrStmtSet):
                    if isinstance(stmt.src, IrFnArg):
                        match stmt.src.type_:
                            case VarType.BOOL | VarType.INT:
                                reg = 'x'
                            case VarType.FLOAT:
                                reg = 'd'
                        print(f'ldr {reg}9, [fp, #-
{arg_off[stmt.src]})
                        print(f'str {reg}9, [fp, #-
{arg_off[stmt.dest]})
                    elif isinstance(stmt.src, Constant):
                        if isinstance(stmt.src, ConstantBool):
                            reg = 'x'
                        elif isinstance(stmt.src, ConstantInt):
                            reg = 'x'
                        else:
                            reg = 'd'
                        print(f'ldr {reg}9,
={const_table[stmt.src]})
                        print(f'str {reg}9, [fp, #-
{arg_off[stmt.dest]})
                    else:
                        int_var_count = 0
                        float_var_count = 0
                        stack_arguments: list[IrFnCallArg] = []
                        for arg in stmt.src.arguments:
                            if isinstance(arg, IrFnArg):
                                match arg.type_:
                                    case VarType.BOOL | VarType.INT:
                                        if int_var_count < 8:
                                            print(f'ldr
x{int_var_count}, [fp, #-{arg_off[arg]})
                                            else:

stack_arguments.append(arg)

                                            int_var_count += 1
                                    case VarType.FLOAT:
                                        if float_var_count < 8:
                                            print(f'ldr
d{float_var_count}, [fp, #-{arg_off[arg]})
                                            else:

stack_arguments.append(arg)

                                            float_var_count += 1
                                    elif isinstance(arg, ConstantBool |
ConstantInt):

```

```

                                if int_var_count < 8:
                                    print(f'ldr x{int_var_count},
={const_table[arg]}')
                                else:
                                    stack_arguments.append(arg)
                                    int_var_count += 1
                                else:
                                    if float_var_count < 8:
                                        print(f'ldr d{float_var_count},
={const_table[arg]}')
                                    else:
                                        stack_arguments.append(arg)
                                        float_var_count += 1
                                # with contextlib.ExitStack() as exit_stack:
                                #     fp_offset = 0
                                #     stack_arguments.reverse()
                                #     for arg in stack_arguments:
                                #         print('sub sp, sp, #16')
                                #         exit_stack.callback(print, 'add
sp, sp, #16')
                                #         if isinstance(arg, IrFnArg):
                                #             match arg.type_:
                                #                 case VarType.BOOL |
VarType.INT:
                                #                                     print(f'ldr x9, [fp,
#-{arg_off[arg]}]')
                                #                                     print('str x9, [sp]')
                                #                                     case VarType.FLOAT:
                                #                                     print(f'ldr d9, [fp,
#-{arg_off[arg]}]')
                                #                                     print('str d9, [sp]')
                                #                                     print(f'ldr')
                                #                                     print(f'str ')
                                #     stmt.src.
                                #     pass
                                #     stmt.src

                                # if isinstance(stmt, IrStmtIf):
                                # if isinstance(stmt, IrStmtWhile):
                                # if isinstance(stmt, IrStmtReturn):

                                pass

def main():
    filepath = 'example.txt'
    lparens: list[TokenLparen] = []
    lisp_tree = LispList(TokenLparen(0, 0), [])
    build_lisp_tree(lisp_tree, tokenize(filepath), lparens)

```

```
if len(lpargs) > 0:
    panic(f'Error: Unmatched paren {at_line(lpargs[-1])}')
del lpargs

constants_list: list[Constant] = []
syn_fn_defs = tuple(syntax_parse_function_definitions(lisp_tree,
constants_list))
ir_fn_defs = ir_fn_defs_build(syn_fn_defs)
with open('example.s', 'w') as file:
    with contextlib.redirect_stdout(file):
        asm_generate(ir_fn_defs, constants_list)

if __name__ == '__main__':
    main()
```

3 ВИСНОВОК

У результаті розробки було реалізовано повноцінний етап трансляції Lisp-подібної мови у проміжне представлення (IR). Цей етап є ключовим для подальшої генерації асемблерного коду, оскільки саме тут виконується структуризація програми, перевірка типів і семантична валідація всіх конструкцій. IR у моєму компіляторі відображає програму у вигляді чітко типізованих інструкцій і функцій, що значно спрощує аналіз, оптимізацію та побудову машинного коду. Таким чином, етап трансляції забезпечує перехід від абстрактного синтаксичного дерева до низькорівневого подання, яке є логічною основою для подальших фаз компіляції.