

# Techniki kompilacji 2024Z

Jan Górski

## Informacje podstawowe

Tematem projektu jest napisanie interpretera języka programowania ogólnego przeznaczenia.

- Język wykorzystany w projekcie: C++
- Wykorzystane biblioteki:
  - Boost.

## Opis zakładanej funkcjonalności

- kod programu napisane w języku ewaluowany linijka po linijce - brak punktu wejściowego, jak np. funkcja `main`.
- wartości są domyślnie stałe: wartość zmiennej zadeklarowanej bez użycia słowa kluczowego `mut` nie może być zmieniona;
- typowanie dynamiczne - zmienna może w czasie działania programu zmienić swój typ,
- typowanie silne - zmienna w każdej chwili działania programu ma przypisany typ,
- zmienne są przekazywanie przez referencję (domyślnie stałą),
- pętla `while`,
- instrukcje warunkowe `if`, `else`,
- wyrażenia warunkowe zwracają wartość,
- funkcje są obiektami pierwszej klasy: mogą być przekazywana jako parametr innej funkcji, przypisywane do zmiennych itd.,
- typy danych:
  - całkowitoliczbowy (`int`),
  - zmiennopozycyjny (`dbl`),
  - logiczny (`bol`)
  - ciąg znaków (`str`),
  - funkcje,
  - dynamiczne tablice,
- funkcje wyższego rzędu: *lambda* oraz *map*.

## Formalna specyfikacja i składnia EBNF realizowanego języka

```
program = features ;
```

```
features = { feature } ;
```

```

        feature = {
            | one_line_comment
            | multiline_comment
            | statement
        } ;

one_line_comment = "//" { character - new_line } new_line ;

multiline_comment = "/*" { ( character character ) - "*/" } "*/" ;

        statement = variable_declaration
            | while_loop
            | return_statement
            | expression_statement ;

variable_declaration = "let" [ "mut" ] identifier [ "=" expression ] ";" ;

        while_loop = "while"
            "(" equality_expression ")"
            "{" { statement } "}" ;

        return_statement = "return" [ expression ] ";" ;

expression_statement = [ expression ] ";" ;

        expression = function_declaration
            | if_expression
            | literal
            | identifier
            | assignement_expression
            | function_call ;

        if_expression = if_clause { else_if_clause } [ else_clause ]

else_if_clause = "else" if_clause ;

        else_clause = "else" "{" statements [ expression ] "}" ;

        if_clause = "if"
            "(" relational_expression ")"
            "{" statements [ expression ] "}" ;

assignement_expression = equality_expression
            | primary_expression "=" equality_expression ;

map_expression = relational_expression

```

```

| "<>" identifier "<" parameter_list ">"

equality_expression = relational_expression
| equality_expression ( "==" | "!=" )
relational_expression

relational_expression = additive_expression
| relational_expression ( "<" | ">" | "<=" | ">=" )
additive_expression ;

additive_expression = multiplicative_expression
| additive_expression ( "+" | "-" )
multiplicative_expression ;

multiplicative_expression = cast_expression
| multiplicative_expression ( "+" | "-" )
cast_expression ;

cast_expression = postfix_expression
| cast_expression "<-" type_name ;

postfix_expression = primary_expression
| postfix_expression "[" expression "]"

primary_expression = identifier
| literal
| "(" expression ")" ;

identifier = letter { letter | digit | "_" } ;

literal = bool_literal
| float_literal
| integer_literal
| string_literal
| array_literal ;

bool_literal = "true" | "false" ;

float_literal = integer_literal "." { digit } ;

integer_literal = ( non_zero_digit { digit } ) | "0" ;

string_literal = "" {
    special_string_character | normal_string_character
} "" ;

```

```

array_literal = "[" [ expression { "," expression } ] "]" ;

function_call = identifier "(" { expression } ")" ;

function_declaration = lambda_function
                      | normal_function ;

normal_function = "fn" identifier
                 function_parameter_list function_body ;

lambda_function = "\" parameter_list "->" function_body;

enclosed_parameter_list = "(" parameter_list ")" ;

parameter_list = [
    [ "mut" ] identifier
    { "," [ "mut" ] identifier }
] ;

function_body = "{"
               { statement - return_statement }
               [ return_statement | expression ]
               "}" ;

normal_string_character = character - ( "\"" | "\"" ) ;

special_string_character = "\"" | "\"n" | "\"\" | "\"b";

character = alphanumeric_character
          | special_character
          | white_character ;

white_character = "\"n" | "\"t" | " " ;

alphanumeric_character = letter | digit ;

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G"
        | "H" | "I" | "J" | "K" | "L" | "M" | "N"
        | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
        | "V" | "W" | "X" | "Y" | "Z" | "a" | "b"
        | "c" | "d" | "e" | "f" | "g" | "h" | "i"
        | "j" | "k" | "l" | "m" | "n" | "o" | "p"
        | "q" | "r" | "s" | "t" | "u" | "v" | "w"
        | "x" | "y" | "z" ;

digit = "0" | non_zero_digit ;

```

```

non_zero_digit = "1" | "2" | "3" | "4" | "5"
                | "6" | "7" | "8" | "9" ;

special_character = "~" | "!" | "@" | "#" | "$" | "%" | "^" | "&" | "*"
                  | "(" | ")" | "-" | "_" | "+" | "=" | "{" | "}" | "["
                  | "]" | "|" | "\" | ":" | ";" | "\"" | "'" | "<" | ">"
                  | "," | "." | "?" | "/"

new_line = ( "\n" | "\r" | "\r\n" ) ;

```

## Obsługa błędów, przykłady komunikatów

Na każdym etapie przetwarzania danych znajdowała się będzie struktura, do której będą zapisywane komunikaty dotyczące wykrytych błędów. Na sam koniec procesu interpretacji lub gdy wykryty błąd będzie niemożliwy do pominięcia użytkownik zostanie powiadomiony o tym przez wyjście standardowe, a proces interpretacji zakończy się błędem i nie zostanie ukończony.

## Sposób uruchomienia, wej/wyj.

```

$ ./my-language --help
Usage: ./my-language [OPTIONS] INPUT
Options:
  -h [ --help ]           produce help message
  -l [ --lexer ]          create a file with lexer output (with
                           .l extention)
  -p [ --parser ]         create a file with result of syntax
                           analysis (with .p extention)
  -s [ --semantic ]       create a file with result of semantic
                           analysis (with .s extention)
  -i [ --intermediate_representation ] create a file with result of
                           intermediate representation(with .ir
                           extention)

```

## Lista tokenów:

- ERROR, ERROR\_NUMBER, ERROR\_STRING, UNFINISHED\_COMMENT,
- END\_OF\_FILE,
- ONE\_LINE\_COMMENT, MULTILINE\_COMMENT,
- PLUS, MINUS, STAR, SLASH, ASSIGN,
- EQUALS, LESS, LEQ, GREATER, GEQ,
- LPARENT, RPARENT, LBRACE, RBRACE, LBRACKET, RBRACKET,

- SINGLE\_QUOTE, DOUBLE\_QUOTE, DOUBLE\_QUOTE, SEMI-COLON, COMMA, DOT, RARROW, LARROW,
- IDENTIFIER, STRING, INTEGER, DOUBLE,
- FN, LET, MUT, RETURN,
- IF, ELSE, WHILE,
- INT, STR, DBL, BOL,
- TRUE, FALSE,
- NOT\_A\_KEYWORD,
- MAP.

Lista słów kluczowych:

- fn, let, return, mut,
- if, else, while,
- int, str, bol, dbl,
- true, false.

Lista węzłów drzewa AST:

- planowanym jest, aby lista węzłów drzewa AST w znaczącym stopniu odwzorowywała elementy występujące w składni **EBNF** języka.

## Wykorzystane struktury danych

- tablica dynamiczna,
- graf acykliczny (drzewo),
- tablica symboli,

## Przykład wykorzystania struktury danych

- tablica dynamiczna: przechowywanie lexemów, przechowywanie informacji o błędach,
- graf acykliczny (drzewo): struktura w analizatorze składniowym.
- tablica symboli: dla każdego **zasięgu** (bloku kodu) oddzielna tablica mająca postać hash-mapy. Same zaś zasięgi będą ustrukturyzowane poprzez użycie stosu zasięgów.

### Formy pośrednie

1. Kod źródłowy -> analizator leksykalny
  - analizator leksykalny pobiera wejściowe dane ze strumienia (`std::istream`)
2. analizator leksykalny -> analizator składniowy
  - analizator składniowy pobiera obiekty typu `Leksem` (`tokeny`) z analizatora leksykalnego
3. analizator składniowy -> analizator semantyczny
  - analizator składniowy generuje drzewo AST.

## Opis sposobu testowania

Proces testowania będzie opierał się na stopniowym kolejkowym testowaniu - zawsze wejściem testu będzie strumień znaków.

- Analizator leksykalny:
  - wejście - ciąg znaków do przeprowadzenia analizy leksykalnej,
  - wyjście - tablica leksemów + tablica błędów,
  - test - porównywanie poszczególnych wartości leksemów i błędów wynikowych z oczekiwanymi,
- Analizator składniowy:
  - wejście - ciąg znaków do przeprowadzenia analizy leksykalnej,
  - wyjście - postać drzewa wyjścia parsera sprowadzona do postaci wyświetlonej (jako `string`),
  - test - porównanie wyjścia z oczekiwanym wynikiem.

## Przykłady obrazujące dopuszczalne konstrukcje językowe i semantykę

### 1. Komentarze

```
// These are one line comments.  
/*  
these  
are  
multiline  
comments  
*/
```

### 2. Podstawowe typy danych

```
let a = 12; // int - const  
let mut b = 12.12; // flt - mutowalny  
let c = "duck"; // str - const  
let mut d = true; // bol - mutowalny
```

### 3. Podstawowe operacje matematyczne, dodawanie zmiennych tego samego typu

```
let mut a = 1;  
let mut b = 2;  
  
a+b; // 3 ok.  
a-b; // -1 ok.  
a*b; // 2 ok.  
a/b; // 0 ok.
```

```

a = "Hello ";
b = "World";
let c = a + b; // c=="Hello World"
let d = a + b + "!"; // d=="Hello World!"

```

#### 4. Wyrażenia warunkowe

```

let i = 12;
let mut result = 0;

// if statement is an expression so it returns value
let if_result = if (i < 2) {
    result = 1;
    null
} else if (i < 6) {
    result = 2;
} else {
    result = 3;
};

let if_result_not_null = if (i < 2) {
    result = 1;
    result
} else if (i < 6) {
    result = 2;
    result
} else {
    result = 3;
    result
};
result = 12;
if (true) {
    let result = 21;
    // result=21
};
// result=12

```

#### 5. Pętla while

```

let mut i = 1;

while (i < 10) {
    i = i + 1;
}

```

#### 6. Listy

```

let kaczka = []; // empty list
let rozne = [1, 2, 2.3, "int"];

```



```

let kaczki = ["kaczka1", "kaczka2", "kaczka3"];
let mut i = 0;
while (i < 3) {
    kaczki[i] = i;
    i = i + 1;
}

```

## 7. Funkcje

```

fn example_function(arg) {
    print(arg);
}

fn gets_function(function) {
    return function;
}

gets_function(example_function)("Hello") // `Hello`

fn function_with_return_value() {
    let x = 1 <- int;
    return x; // or just: x
}

fn function_without_return_value() {
    let x = 1<-int;
}

fn function_without_return_value_with_keyword() {
    return;
}

fn change(mut param1, param2) {
    param1 = 12; // ok.
    // param2 = 12;
}

```

## 7. Funkcje wyższego rzędu, operatory przyjmujące funkcje jako argument

```

let a = [1,2,3,4,5];
let b = [1,1,1,1,1];

fn add(first, second) {
    return first + second;
}

fn double(num) {
    return 2 * num;
}

```

```
// map operator: <> function <list1, list2, ..., listn>
let c = <> add <a, b>; // c == [2,3,4,5,6]

// lambda: \x1, x2, x3, ..., xn -> {...};
let x = \function, argument -> {function(argument)};
let y = x(double, 2); // 4

// example: combination of both
let d = <> \first, second -> {first + second} <a, b>; // c == [3,3,4,5,6]
```