Techniki Kompilacji - Projekt Wstępny

Bartosz Nowak, 325201

1. Opis zakładanej funkcjonalności, przykłady obrazujące dopuszczalne konsrukcje językowe oraz ich semantykę

1.1 Typy danych i Operatory

1.1.1 Typy Danych

Zaimplementowane zostaną następujące typy danych:

1. Integer:
   - `int`
2. Float
   - `flt`
3. `string`
4. `bool`
   - `true`
   - `false`

1.1.2 Operatory

Zdefiniowana została następująca kolejność operatorów:

priorytet	operatory
0.	dekorator: `@`
1.	bind-front: `->>`
2.	wywołanie, ciąg wywołań: `->`, nawiasy
3.	unarne: `!`, `-`, `+`
4.	mnożenie, dzielenie: `*`, `/`
5.	dodawanie, odejmowanie: `+`, `-`
6.	porównywanie: `>`, `<`, `>=`, `<=`, `!=`, `==`
7.	logiczne AND, OR: `&&`, `||`
8.	przypisanie: `=>`

1.1.3 Konwersja typów

- `int` -> `flt`
  - Konwersja typu całkowitego na zmiennoprzecinkowy. (`12 -> 12.0`)

- `flt` -> `int`
  - Zaokrąglenie w dół. (`12.5 -> 12`)
- `int` -> `string` , `flt` -> `string`
  - Zamiana liczby na postać tekstową. (`123 -> "123"`)
- `int` -> `bool` , `flt` -> `bool`
  - Jeśli wartość jest różna od 0 - zamiana na true (`1 -> true, -1 -> true`)
  - Jeśli wartość jest równa 0 - zamiana na false (`0 -> false, 0.0 -> false`)
- `bool` -> `int` , `bool` -> `flt`
  - `true -> 1, false -> 0`
- `string` -> `bool`
  - Zamienia na true jeśli string nie jest pusty, w przeciwnym wypadku false
- `bool` -> `string`
  - `true -> "true", false -> "false"`
- `string` -> `int` , `string` -> `flt`
  - zakończy się błędem

Tabela konwersji typów dla operacji arytmetycznych:

L\R Oznacza pozycję względem operatora

- Operator `+`

L\R (+)	int	flt	string	bool
int	int	flt	string	int
flt	flt	flt	string	flt
string	string	string	string	string
bool	int	flt	string	bool

- Operator `-`

L\R (-)	int	flt	string	bool
int	int	flt	-	int
flt	flt	flt	-	flt
string	-	-	-	-
bool	int	flt	-	int

- Operator `*`

L\R (*)	int	flt	string	bool
int	int	flt	string	int
flt	flt	flt	-	flt

L\R (*)	int	flt	string	bool
string	string	-	string	string
bool	int	flt	string	bool

- Operator `/`

L\R (/)	int	flt	string	bool
int	int	flt	-	-
flt	flt	flt	-	-
string	-	-	-	-
bool	-	-	-	-

- Dla operatorów równości (`==` , `!=`) oraz logicznego AND i OR zawsze nastąpi konwersja obu stron do typu `bool`
- Dla operatorów porównania: (`<, >, <=, >=`) oba czynniki zostaną zrzutowane na następujący typ:

	int	flt	string	bool
int	int	flt	-	int
flt	flt	flt	-	flt
string	-	-	-	-
bool	int	flt	-	int

1.2 Definiowanie zmiennych

Definicja zmiennych odbędzie się przy użyciu operatora `=>` w następujący sposób:

```
5 => int b;  // odpowiednik int b = 5
```

Po lewej stronie operatora mogą być obecne bardziej złożone wyrażenia

```
5 + 10 * 2 => int c;
```

Język będzie **słabo typowany**, zatem w następującej sytuacji:

```
5.5 * 2 + 2 => int a;
```

Zostaną wykonane następujące operacje:

1. Zrzutowanie literału 2 -> 2.0
2. Wykonanie operacji mnożenia
3. Zrzutowanie literału 2 -> 2.0

4. Wykonanie dodawania

Operatory `-` oraz `/` nie mogą działać z typem danych `string` (po żadnej ze stron, patrz tabelka w punkcie 1.1.3)

1.2.1 Czas życia zmiennych

Czas życia zmiennych określony jest przez blok kodu, w którym zostały zdefiniowane i jest ograniczony za pomocą znaków `{}`.

1.2.2 Przesłanianie zmiennych

W języku nie jest obecny mechanizm przesłaniania zmiennych, Zdefiniowanie zmiennej w bloku zagnieżdzonym o takiej samej nazwie jak zmienna znajdująca się w bloku nadrzędnym skutkować będzie wsytąpieniem błędu.

```
int main::{
    5 => int a;
    if (a > 0) {
        10 => int a;  // error, identifier `a` is already defined.
    }
}
```

1.2.3 Widoczność zmiennych

Zmienne widoczne są w blokach, w których zostały zdefiniowane oraz w blokach podrzędnych.

1.3 Mutowalność

Zmienne w języku są domyślnie **niemutowalne**, w celu zdefiniowania mutowalnej zmiennej, konieczne będzie dodanie słowa kluczowego `mut` przed definicją typu.

```
5 => mut int a;
```

wartość zmiennej mutowalnej możemy zmienić za pomocą operatora przypisania.

```
10 => a;
```

1.4. Funkcje

1.4.1 Operator ->

Operator *wołania funkcji* jest operatorem używanym w celu wywoływania funkcji i przekazywania ich wyniku do następnych funkcji. Ma on postać `->`.

Zmienne przekazywane są do funkcji przez **referencję**, zatem wyrażenie:

```
(5) -> increment => int a;
```

utworzy tymczasową zmienną niemutowalną z literału 5, a następnie przekaże ją jako referencję do funkcji increment.

Operator `->` może być użyty w celu tworzenia łańcuchów wywołań funkcji:

```
(5)
    -> increment
    -> increment
    -> decrement
=> int b;
```

1.4.2 Argumenty funkcji

Argumenty wywołania funkcji muszą znaleźć się z **lewej** strony operatora przekazania i **muszą** być zamknięte w nawias.

```
(5) -> increment => int a;  // a = 6
(5, 6) -> add => int b; // b = 11
() -> no_argument_function => int c;
```

Jeżeli wektor argumentów funkcji zostanie przekazany na końcu łańcucha, będzie on potraktowany jako wyrażenie, przy założeniu, że posiada jedynie **1** argument.

```
(5) -> increment -> (_ + 10) => int c;
```

(Znak "_" jest słowem kluczowym `wildcard` i został dokładniej opisany w sekcji 1.7)

1.4.3 Funkcje jako argument

Funkcje mogą zostać przekazywane jako argumenty innych funkcji. W celu dokonania takiego przekazania należy użyć nazwy funkcji.

```
(5, increment) -> apply_function => int a;
```

1.4.4 Typy zmiennych funkcji

Typ funkcji-zmiennych zapisywany jest w formie: `[<ret_type>::<args>]`

```
increment => [int::int] increment_function;
```

1.4.5 Definicja funkcji

Funkcje definiowane są przy użyciu operatora `::`, z którego lewej strony powinna znaleźć się nazwa funkcji poprzedzona typem danych zwracanych przez funkcję. Z prawej strony operatora powinny się znaleźć argumenty wywołania.

```
mut int add :: int a, int b {
    a + b -> ret;
```

```
}
```

Słowo kluczowe `ret` odpowiada za zwrot z funkcji, operator przekazania przekazuje mu wartość, którą zwróci funkcja.

1.4.6 Przekazywanie przez referencję

Zmienne przekazywane są do funkcji poprzez **referencję**, co oznacza, że następująca funkcja:

```
void increment :: mut int a {
    a + 1 => a;
    ret;
}
```

Nie zwróci żadnej wartości, ale w przypadku wywołania jej za pomocą:

```
5 => int foo;
(foo) -> increment;
(foo) -> stdout;
```

Zostanie wyświetlona wartość 6, zmienna `foo` zostanie zmodyfikowana wewnątrz funkcji. Ze względu na przekazywanie przez referencję, błędnym jest przekazanie literałów do funkcji oczekujących mutowalnych argumentów:

```
(5) -> increment;  // ERROR! Expected mutable argument: (int) -> void increment :: mut
        int
```

1.4.7 Przeciążanie funkcji

Przeciążanie funkcji nie jest dozwolone, próba zdefiniowania wielu funkcji o tej samej nazwie zakończy się błędem

1.4.8 Wywoływanie w miejscu

Funkcję otrzymaną z innej funkcji możemy wywołać w miejscu za pomocą ciągu wywołań:

```
() -> () -> get_func;
```

W przypadku, gdy znalezione zostaną dwa wektory argumentów obok siebie w łańcuchu wywołań - kompilator założy, że użytkownik próbuje wywołać funkcję, którą zwraca wywołanie "wewnętrzne"

```
() -> () -> get_func;
      ^     ^
      2 wektor argumentów z rzędu - użytkownik wywołuje zwróconą funkcję
            |
            jeśli get_func nie zwróci funkcji - otrzymamy błąd
```

1.5 If

If-statement definiowany jest następująco:

```
if (a == b) {
    5 -> ret;
} elif (a == c) {
    6 -> ret;
} else {
    1 -> ret;
}
```

Operatory `==, >=, <=, !=, >, <` używane są do porównywania, warunki mogą być łączone za pomocą operatorów `&&` (AND) oraz `||` (OR).

1.6 Pętle

Język oferuje dwie konstrukcje pętli - `for` oraz `while`

1.6.1 For

Konstrukcja pętli `for` prezentuje się nastepująco:

```
for (<mutable variable>; <condition>) {
    ...
} -> <update expression>;
```

w praktyce mamy zatem:

```
for (0 => mut int a; a < 10) {
    ...
} -> increment;
```

Zmienna, po której będziemy iterować **musi** być zmienną mutowalną, w przeciwnym przypadku otrzymamy błąd:

```
5 => int a;
for (a; i < 10) {  // error: Expected mutable argument: (int) -> for (mut any; condition)
    ...
} -> increment;
```

Konieczne jest podanie zmiennej oraz warunku działania pętli, możliwe jest pominięcie wyrażenia wywoływanego po każdej iteracji (w przykładzie: `increment`)

1.6.2 While

Konstrukcja pętli `while` prezentuje się następująco:

```
while (<condition>) {
    ...
}
```

```
while (a < 10) {
    ...
}
```

W pętli `while` mogą zostać użyte zmienne niemutowalne (ale może to spowodować powstanie nieskończonej pętli)

```
10 => mut int a;
12 => int b;
while (a < b) {
    a -> increment;
}

while (b) {
    ...
}  // effectively a while (true) loop
```

1.7. "_"

Słowo kluczowe `_` może zostać użyte w celu sprecyzowania dokładnej pozycji w wektorze argumentów funkcji, na którą przekierowane będzie wyjście poprzednio wywołanej funkcji.

```
12 => int a;
10 => int b;
(a, b) -> add -> (a, _) -> add => int c;  // otrzymamy a + (a + b) = c
```

Zapis będzie oczywiście równy zapisowi:

```
(a, (a, b) -> add) -> add => int c;
```

Ale w zależności od sytuacji, może okazać się bardziej czytelny.

1.8 Operacje I/O

Funkcja wbudowana `stdout` pozwala na wypisanie ciągu znaków na wyjście standardowe.

```
("wynik to" + 5) -> stdout;
```

```
wynik to 5
```

Z kolei funkcja `stdin` pozwala na wczytanie ciągu znaków z wejścia standardowego.

```
("Podaj liczbe:") -> stdin => string a;
```

1.9 Funkcje wyższego rzędu - dekoratory

Dekoratory "opakowują" funkcje w inne funkcje.

```
[int::int] my_decorator :: [i32::i32] other_function {
    int wrapper :: int a {
        ("hello from decorator before function!\n") -> stdout;
        5 -> other_function => int a;
        ("\nhello from decorator after function!\n") -> stdout;
        ret a;
    }
    ret wrapper;
}
```

Funkcje dekorujące **muszą** przyjmować funkcję opakowywaną jako swój pierwszy i jedyny argument.

Funkcje mogą być dekorowane w następujący sposób:

```
my_function @ my_decorator => [i32::i32] decorated_function;


(5) -> decorated_function;
```

```
hello from decorator before function!
6
hello from decorator after function!
```

Możliwa jest również jednorazowa dekoracja funkcji:

```
(5) -> increment @ log => int a;
```

Ze względu na priorytet operatorów, funkcja zostanie najpierw udekorowana, a następnie wywołana z wartością 5. Jako, że operator przypisania rozpatrywany jest na sam koniec, wartość przypisana do zmiennej `a` będzie wartością zwróconą przez udekorowaną funkcję.

1.10 Funkcje wyższego rzędu - bind front

Dla funkcji N argumentowej możliwe jest przypisanie stałych wartości dla n pierwszych argumentów za pomocą mechanizmu bind front. Przyjmijmy, że mamy funkcję `add` przyjmującą 2 argumenty i zwracającą ich sumę.

```
int add :: int a, int b {
    ret a + b;
}
```

Możemy użyć operatora `->>`, aby przypisać zmienne do pierwszych n argumentów funkcji

```
(5) ->> add => [int::int] add5;
```

Możemy do tego również użyć zmiennych mutowalnych

```
5 => mut int a;
(a) ->> add => [int::int] adda;
5 -> adda;  // 10
1 => a;
5 -> adda;  // 6
```

Bind front **MUSI** zakończyć się przypisaniem, inaczej zostanie zwrócony błąd kompilacji: `unassigned bind-front`

1.11 Złożone konstrukcje

1. `Hello World`

```
int main:: {
    ("Hello World") -> stdout;
    ret 1;
}
```

2. `quadratic`

```
int main:: {
    ("Enter a:") -> stdout;
    stdin => flt a;
    ("Enter b:") -> stdout;
    stdin => flt b;
    ("Enter c:") -> stdout;
    stdin => flt c;

    b * b - 4 * a * c => flt delta;

    if (delta > 0) {

        // ciąg wywołań rozbity na wiele linijek
        (delta)
            -> sqrt
            -> (-b + _ / (2 * a))   // wildcard otrzyma wynik pierwiastkowania
        => flt x1;  // wynik wyrażenia zostanie przekazany do zmiennej x1

        (delta)
            -> sqrt
            -> (-b - _ / (2 * a))
        => flt x2;

        ("Result: ") -> stdout;
        (x1 + " and " + x2 " \n") -> stdout;

    } elif (delta == 0) {
        -b / 2 * a => flt x1;

        ("Result: ") -> stdout;
        (x1 + "\n") -> stdout

    } else {
        ("no real solution\n") -> stdout;
```

```
    }
}
```

3. Recursive sum of n natural numbers

```
int recur_sum::int n {
    if (n <= 1) {
        ret n
    }
    ret (n - 1) -> recur_sum -> (_ + n)
}

int main:: {
    stdin => int n;
    (n)
        -> recur_sum
        -> ("suma to: " + _)
        -> stdout;
}
```

2. Formalna specyfikacja i składnia (EBNF)

```
start_symbol        = {statement}

function_def        = type_mut, identifier, function_sign_op, function_def_params, block
function_def_params = function_param, {",", function_param}
function_param      = type_non_void_mut, identifier

statement           = operation
                    | if_statement
                    | for_loop
                    | while_loop
                    | return_statement
                    | function_def

return_statement    = ret, [expression], ";"

while_loop          = while, "(", expression, ")", block
for_loop            = for, for_loop_args, block, call_op, statement, ";"
for_loop_args       = "(", (identifier | assignment), ";", expression, ")"

if_statement        = if, "(", expression, ")", block, {elif_st}, [else_st]
else_st             = else, block,
elif_st             = elif, "(", expression, ")", block

block               = "{", {statement}, "}"

operation           = [function_call], [assignment], ";"
assignment          = assignable, assign_op, [type_non_void_mut], identifier

assignable          = expression

bind_front          = function_args, bindfrt_op, identifier

expression          = logical_expression
```

```
logical_expression  = comp_expression, [logical_and_or, comp_expression]
comp_expression     = additive_expression, [comp_operator, additive_expression]
additive_expression = term, {add_sub_op, term}
term                = unary_factor, {mult_div_op, unary_factor}
unary_factor        = [unary_operator], factor
factor              = function_call
                    | wildcard
                    | number
                    | identifier
                    | string
                    | "(" expression ")"

function_call       = single_call, [call_chain]
single_call         = identifier_arg_call, call_op, identifier_arg_call
call_chain          = {call_op, identifier_arg_call}
identifier_arg_call = function_args | bind_front
function_args       = "(", [func_arg_list], ")"
bind_front          = func_arg_list, [bindfrt_op, decorator]

func_arg_list       = expression, {",", expression}

decorator           = identifier, [decoration_op, identifier]


type_non_void_mut   = type_non_void [mut]
type_mut            = type [mut]
type                = type_non_void | void
type_non_void       = int
                    | flt
                    | string
                    | bool
                    | func_type
```

```
func_type       = "[", type_mut, function_sign_op, func_type_args, "]"
func_type_args  = type_non_void_mut, {",", type_non_void_mut}

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

string          = '"', {character | escape} '"'
escape          = escape_op, escape_sequence

character       = letter
                | digit
                | special_character

number          = float
                | integer

float           = integer, ".", {digit}
integer         = zero | non_zero, {digit}

funcn_sign_op   = "::"
if              = "if"
elif            = "elif"
else            = "else"
while           = "while"
```

```
for             = "for"
ret             = "ret"

assign_op       = "=>"
bindfrt_op      = "->>"
decoration_op   = "@"
call_op         = "->"

int             = "int"
flt             = "flt"
string          = "string"
bool            = "bool"
void            = "void"

mut             = "mut"

bool_type       = "true"
                | "false"

wildcard        = "_"

unary_operator  = "!"
                | "-"

comp_operator   = ">"
                | "<"
                | ">="
                | "<="
                | "!="

logical_and_or  = "&&"
                | "||"

mult_div_op     = "*"
                | "/"

add_sub_op      = "-"
                | "+"

escape_op = "\"

escape_sequence = "n"
                | "t"
                | "r"
                | "\"
                | '"'

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

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

```
digit           = non_zero | zero
non_zero        = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
zero            = "0"
```

3. Obsługa błędów

Każdy z etapów kompilacji będzie zgłaszał wyjątki, lecz w większości sytuacji nie będzie powodowało to zatrzymania programu, aby możliwe było przeanalizowanie całego pliku źródłowego i jednoczesne zgłoszenie największej możliwej ilości błędów kompilacji

Poniższe przykłady reprezentują jedynie typowe błędy i przykładowe zachowanie w przypadku ich wystąpienia

3.1 Analizator Leksykalny (Przykłady błędów i zachowanie)

1. Niepoprawny identyfikator (*Illegal Identifier*)

```
Illegal identifier

5 => int ab$
         ^^^
```

- Nielegalne znaki identyfikatora zostaną zastąpione znakiem "_"

2. Identyfikator jest słowem kluczowym

```
Reserved keyword used as identifier

5 => int if
```

- Dodany zostanie znak "_" na końcu nazwy identyfikatora

3. Nieznane symbole

```
Unknown symbol "π"

"pi is π" => string pi;
       ^
       Unknown symbol
```

- Nieznany symbol zostanie zastąpiony symbolem "_"

4. Niewłaściwy operator

```
Unexpected token "%*"

(5 %* 1) => int a;
   ^^
   Unexpected Token
```

- Wyrażenie zostanie zastąpione wyrażeniem `0`

3.2 Analizator Składniowy

```
Expected "{" in <block>, got "ret a + 5;"!

int some_fun :: int a ret a + 5;
                      ^^^^^^^^^^^
```

- Kompilator założy, że użytkownik zapomniał rozpocząć definicji bloku

```
(@fun * 2) => int a;
 ^^^^
 Expected factor
```

- Kompilator zastąpi niewłaściwy factor wartością 0

3.3 Analizator Semantyczny

```
string type is not compatible with "-" operator

"abc" - 1 => result;
^^^^^^^
```

- Wartość wyrażenia zostanie zastąpiona wartością 0

```
Undeclared variable

5 => a;
     ^
     a is undeclared here
```

- Zostanie zdefiniowana zmienna `a`

```
Undeclared function

(5) -> a;
       ^
       a is not a known function
```

- Wywołanie funkcji zostanie usunięte lub zastąpione wartością 0

```
Function expected n arguments

(5) -> add;
^^^^^^^^^^
Function add [int::int,int] expected 2 arguments. (got 1)

(5) ->> add => [int::int] x;
++++++++++++++++++++++++++
Perhaps you wanted to bind-front the value?
```

- kompilator doda wartość "0"

```
(5, 2, 3) -> add;
^^^^^^^^^^^^^^^^
Function add [int::int,int] expected 2 arguments. (got 3)

(5, 2) -> add;
++++++
```

- Dodatkowe argumenty zostaną usunięte

```
Unreachable code

    ret 5;
    ^^^^^
    function returns here

    5 -> increment;
```

- Nieosiągalne elementy drzewa zostaną pominięte (usunięte z drzewa składniowego)

4. Sposób uruchomienia, I/O

Kompilator przyjmie na wejście plik źródłowy lub tekst z wejścia standardowego.

Sposób uruchamiania dla plików źródłowych będzie następujący:

```
<nazwa programu> <pliki źródłowe> <flagi>
```

Flaga `-o` określi nazwę pliku wyjściowego

Dla strumienia danych wejściowych będzie to:

```
<nazwa programu> -S <flagi>
```

5. Zwięzła analiza wymagań funkcjonalnych i niefunkcjonalnych

Wymagania języka

1. Słabe, Statyczne typowanie
    - Typ zmiennych jest do nich jednorazowo przypisany
    - Nie jest wymagane jawne rzutowanie typów przy operacjach na różnych typach.
2. Domyślnie stałe
    - Zmienne są domyślnie niemutowalne, ich wartośc może być do nich jednorazowo przypisana
    - Mutowalne zmienne muszą być jawnie zadeklarowane przy użyciu słowa kluczowego `mut`
3. Przekazywanie przez referencję
    - Zmienne przekazywane są do funkcji poprzez referencję. Możliwe jest zatem zmienienie wartości zmiennej mutowalnej wewnątrz funkcji, do której została przekazana

Analizator Leksykalny

1. Obsługa dwóch źródeł - plik oraz ciąg znaków
    - Analizator leksykalny pobierać będzie nowe znaki ze strumienia, co pozwoli na implementację wielu niezależnych od siebie interfejsów pobierających dane wejściowe i przekazujących je do właściwego analizatora.
2. Leniwa tokenizacja
    - Lekser generuje tokeny dynamicznie, na żądanie parsera.
3. Lekser zwracając tokeny ma zwracać liczbę, a nie postać tekstową
4. Ciągi tekstowe muszą obsługiwać *escaping*
    - Realizacja za pomocą znaku "
    - Znaki będą w razie potrzeby zamieniane na odpowiednie znaki ASCII ("\n" -> "LF")
    - Znaki bez dopasowanego escape sequence będą ignorowane: """ spowoduje że drugi cudzysłów nie zamknie stringa.
5. Parametryzowalne pograniczenia na długość identyfikatorów
    - Możliwe za pomocą odpowiedniej flagi przy uruchomieniu
6. Przekazywanie pozycji tokenu w tekście źródłowym
7. Testy jednostkowe dla każdego typu tokenu
    - Realizacja z użyciem biblioteki Google Test
8. Testy jednostkowe dla złożonych kodów źródłowych

Analizator składniowy

1. Nazwy funkcji parsujących i struktur danych zgodnie z gramatyką
2. Testy jednostkowe z wykorzystaniem sekwencji tokenów
3. Możliwość zrzucania struktury drzewa na wyjściu konsoli
    - Wzorzec wizytatora

Interpreter

1. Funkcje wbudowane traktowane tak samo jak funkcje użytkownika

6. Zwięzły opis sposobu realizacji

6.1 Rozróżniane tokeny

Lekser będzie rozróżniał następujące tokeny, a następnie przekazywał je do parsera:

1. Słowa Kluczowe
   - `int` - dla typów całkowitych
   - `flt` - dla typów zmiennoprzecinkowych
   - `string` - dla łańcuchów znaków
   - `void` - typ void - dla funkcji
   - `ret` - zwrot z funkcji
   - `while` - pętla while
   - `for` - pętla for
   - `if` - wyrażenie warunkowe
   - `elif` - else if wewnątrz wyrażenia if
   - `else` - else wewnątrz wyrażenia if
   - `mut` - słowo kluczowe czyniące zmienną mutowalną
2. Operatory (Opisane w punkcie 1)
   - `->>` , `@`
   - `->`
   - `=>`
   - `! - +`
   - `*` , `/`
   - `+` , `-`
   - `&&` , `||`
   - `<` , `>` , `>=` , `<=` , `!=`
3. Identyfikatory
   - Nazwy zmiennych/funkcji niezaczynające się cyfrą składające się ze znaków alfabetu angielskiego oraz cyfr.
4. Literały
   - `number` - Liczby stało oraz zmiennoprzecinkowe
   - `string` - Literały tekstowe zamknięte w podwójny cudzysłów
5. Nawiasy
   - `(` , `)` - do grupowania wyrażeń i przekazywania argumentów funkcji
   - `{` `}` - do wyznaczania ciała funkcji/wyrażeń
   - `[` `]` - do adnotacji typów funkcji
6. Interpunkcja
   - `;` - do kończenia złożonych wyrażeń (statementów)
   - `,` - do rozdzielania argumentów funkcji
7. Wildcard
   - `_` - znak na który przekierowane zostanie wyjście uprzednio wywołanej funkcji.
8. Komentarze

    - znaki `//`. Zawartość między tymi znakami, aż do następnego znaku nowej linii będzie ignorowana przez lekser.

6.2 Realizacja przetwarzania między komponentami

funkcja `main` programu zawierała będzie interfejs użytkownika i odpowiedzialna będzie, w zależności od przekazanych flag, za przekazanie zadanego wejścia (strumień lub plik wejściowy) do interfejsu `InputSource`, który zwróci strumień tekstu, z którego Lekser (`Lexer`) będzie pobierać kolejne znaki i uformuje z nich tokeny.

Lekser będzie rozpoznawał tokeny za pomocą deterministycznych automatów skończonych. Rozpoznane i dopasowane tokeny będą, po uprzedniej konwersji (jeśli wymagana) przekazywane do `Parsera` jako struktura `Token`, zawierająca właściwy token oraz jego pozycję w tekście. Tokeny przekazywane będą w sposób leniwy, tzn. po rozpoznaniu tokenu, a nie po analizie całego pliku.

Parser zbierze tokeny, a następnie utworzy z nich Drzewo składniowe w formie struktury `SyntaxTree`. Struktura `SyntaxTree` zawierać będzie drzewo składniowe wraz z pozycjami w pliku źródłowym każdego elementu. Drzewo to zostanie następnie przekazane do analizatora semantycznego (`SemanticAnalyzer`) w celu sprawdzenia, czy otrzymane struktury składniowe są właściwe w języku programowania.

Analizator semantyczny doda również operacje rzutowania, ze względu na słabe typowanie języka, zweryfikuje typy argumentów dla operatorów na podstawie struktury `OperatorTypes`, która będzie je przechowywać.

Analizator semantyczny przekaże następnie drzewo do interpretera, który wykona instrukcje w nim zawarte.

7. Zwięzły opis sposobu testowania

Na każdym etapie będzie prowadzone testowanie jednostkowe z użyciem frameworku `Google Test`.

Testowane zostaną zarówno przypadki z oczekiwanym pozytywnym rezultatem oraz sytuacje błędne.

- Analizator Leksykalny:
    - Testy dla każdego tokenu
    - Testy dla typowych błędów językowych
- Analizator Składniowy:
    - Testy jednostkowe dla każdej możliwej produkcji
    - Testy dla błędów składniowych
- Interpreter
    - Testy dla kompletnego potoku przetwarzania