Techniki Kompilacji - Dokumentacja Końcowa

Bartosz Nowak, 325201

1. Opis Języka

1.1 Typy danych i Operatory

1.1.1 Typy Danych

Zaimplementowane zostały 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`
  - Konwersja do typów liczbowych, jeżeli takowa jest możliwa (za pomocą `std::stoi` , `std::stod`)

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

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

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

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

1.2 Definiowanie zmiennych

Definicja zmiennych odbywa 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 jest **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

Przesłanianie zmiennych nie jest możliwe, próba zdefiniowania zmiennej o nazwie, która istnieje już w danej funkcji zostanie zakończona wyjątkiem:

```
int main {
    5 => int a;
    if (a > 0) {
        10 => mut int a; // niemożliwe, wystąpi błąd
    }
    ret 0;
}

```sh
Error: Exception thrown at 4:9 - Variable a is already defined in this frame
  3 |     if (a > 0) {
> 4 |         10 => mut int a; // niemożliwe, wystąpi błąd
                ^
                Exception thrown at 4:9 - Variable a is already defined in this frame
  5 |     }
```

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. 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.

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;
```

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;
}
```

Operator `::` powinien znaleźć się w definicji funkcji **jedynie** gdy definiujemy funkcję posiadającą argumenty, w innym przypadku należy tego unikać

```
int main {
...
}
```

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

Zostanie wyświetlona wartość 6, zmienna `foo` zostanie zmodyfikowana wewnątrz funkcji. Ze względu na przekazywanie przez referencję. Do funkcji oczekujących mutowalnych argumentów, możemy przekazać również literały, ale musimy wtedy liczyć się z tym, że utracimy część wyniku w momencie wyjścia z funkcji. (Czas życia tych zmiennych zakończy się w momencie wyjścia z funkcji)

Z kolei w przypadku próby przekazania referencji do zmiennej niemutowalnej, zostanie rzucony wyjątek

```
Error: Exception thrown at 3:9 - Type mismatch
  2 |     10 => int a;
> 3 |     (a)->increment;
              ^
              Exception thrown at 3:9 - Type mismatch
  4 |     ret a;
```

1.4.7 Przeciążanie funkcji

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

1.5 If

If-statement definiowany jest następująco:

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

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

1.4.8 Wywoływanie w miejscu

Możliwe jest wywołanie funkcji w miejscu, taka funkcja będzie miała postać ciągu wywołań, przykładowy program dokonujący takiego wywołania znajduje się poniżej:

Parser rozpatruje ciąg wywołań funkcji początkowo rozwijając nazwę funkcji

```
int main {
    (((5) -> (0) -> add_something) + "\n") -> stdout;
    // powyższy kod oznacza:
    // zaaplikuj `0` do `add_something`, otrzymując funkcję
    // zaaplikuj `5` do otrzymanej funkcji
    // dodaj `"\n"` do wyniku
    // wywołaj jako argument funkcji `stdout`

    ret 0;
}

[int::int] add_something :: int a {  // funkcja zwraca funkcję some_func
    ret some_func;
}

int some_func :: int a { // funkcja zwraca zadany argument
    ret a;
}
```

Powyższy kod wypisze na ekran cyfrę `5`.

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;
```

Konieczne jest podanie zmiennej, warunku działania pętli oraz funkcji wywoływanej po każdej iteracji

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
```

1.8 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 przyjmując na wejście ciąg znaków.

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

1.9 Funkcje wyższego rzędu - dekoratory

Dekoratory "opakowują" funkcje w inne funkcje.

```
int main {
    ret () -> add @ my_decorator;
}

int my_decorator :: [int::int] other_function {
    ("hello from decorator before function! the value is: ") -> stdout;
    (5) -> other_function => int a;
    (a + "\n") -> stdout;
    ("hello from decorator after function!\n") -> stdout;
    ret a;
}

int add::int some_value {
    ret some_value + 2;
}
```

Powyższy kod zwróci wynik

```
hello from decorator before function! the value is: 7
hello from decorator after function!
```

Funkcje dekorujące **muszą** przyjmować funkcję o danej sygnaturze jako swój pierwszy argument.

Funkcje mogą być dekorowane w następujący sposób:

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


(5) -> decorated_function;
```

Możliwa jest również jednorazowa dekoracja funkcji:

```
(5) -> add @ 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 W przypadku bind-frontów zmienne **zawsze** zostaną przekazane poprzez wartość

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

1.11 Złożone konstrukcje

1. `Hello World`

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

2. `quadratic`

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

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

    if (delta > 0) {
        (delta)->sqrt;
        (-b - delta) / (2 * a) => flt x1;
        (-b + delta) / (2 * a) => flt x2;

        ("Result: ") -> stdout;
        (x1 + " and " + x2 + "\n") -> stdout;
    } elif (delta == 0) {
        -b / 2 * a => flt x1;
        ("Result: " + x1 + "\n") -> stdout;
    } else {
```

```
        ("No real solution\n") -> stdout;
    }
    ret 0;
}
```

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 {
    ("Podaj n: ") -> stdin => int n;
    ("suma to: " + (n)->recur_sum + "\n")->stdout;
    ret 0;
}
```

2. Formalna specyfikacja i składnia (EBNF)

```
start_symbol        = {statement}

function_def        = func_signature, block
func_signature      = type, identifier, [function_sign_op, function_def_params]
function_def_params = function_param, {",", function_param}
function_param      = type_non_void, 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, bind_front, ";"
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}, "}"

assignment          = assignable, assign_op, [type_non_void_mut], 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       = function_args, call_op, callable
callable            = bind_front | function_call
function_args       = "(", [func_arg_list], ")"

bind_front          = [func_arg_list, bindfrt_op], decorator
decorator           = identifier, [decoration_op, identifier]

func_arg_list       = expression, {",", expression}

type                = type_non_void | void
type_non_void       = var_type | func_type

var_type_mut        = [mut], var_type
var_type            = 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 zgłasza wyjątki

Poniżej przedstawiono wyciąg z programu w przypadku wystąpienia typowych błędów kompilacji

3.1 Analizator Leksykalny (Przykłady błędów i zachowanie)

1. Niepoprawny identyfikator (*Unexpected Token*)

```
[TOKEN: [Type: T_INT_TYPE, Value: None, Position: 1:1]
[TOKEN: [Type: T_IDENTIFIER, Value: main, Position: 1:5]
[TOKEN: [Type: T_LBLOCK, Value: None, Position: 1:10]
[TOKEN: [Type: T_INT, Value: 0, Position: 2:5]
[TOKEN: [Type: T_ASSIGN, Value: None, Position: 2:7]
[TOKEN: [Type: T_INT_TYPE, Value: None, Position: 2:10]
[TOKEN: [Type: T_IDENTIFIER, Value: a, Position: 2:14]
ERROR: Unexpected token
```

2. Identyfikator jest słowem kluczowym

```
Error: Exception thrown at 2:14 - Expected identifier
  1 | int main {
> 2 |     0 => int if;
                   ^
                   Exception thrown at 2:14 - Expected identifier
  3 |     ret 0;
```

- Nieznany symbol zostanie zastąpiony symbolem "_"

3. Niewłaściwy operator

```
[TOKEN: [Type: T_INT_TYPE, Value: None, Position: 1:1]
[TOKEN: [Type: T_IDENTIFIER, Value: main, Position: 1:5]
[TOKEN: [Type: T_LBLOCK, Value: None, Position: 1:10]
[TOKEN: [Type: T_INT, Value: 0, Position: 2:5]
ERROR: Unexpected token
```

- Wyrażenie zostanie zastąpione wyrażeniem `0`

3.2 Analizator Składniowy

```
Error: Exception thrown at 3:5 - Expected ';'
  2 |     0 => int sth
> 3 |     ret 0;
```

```
          ^
          Exception thrown at 3:5 - Expected ';'
  4 | }
```

```
Error: Exception thrown at 1:10 - Expected ':'
> 1 | int main abc
               ^
               Exception thrown at 1:10 - Expected ':'
```

```
Error: Exception thrown at 1:13 - Expected type
> 1 | int main :: {
                  ^
                  Exception thrown at 1:13 - Expected type
  2 |     ret 0;
```

3.3 Analizator Semantyczny

```
Error: Exception thrown at 2:5 - Unsupported operator for type
  1 | int main {
> 2 |     "abc" - 1 => int result;
          ^
          Exception thrown at 2:5 - Unsupported operator for type
  3 | }
```

```
Error: Exception thrown at 2:5 - Variable not in scope: a
  1 | int main {
> 2 |     5 => a;
          ^
          Exception thrown at 2:5 - Variable not in scope: a
  3 | }
```

```
Error: Exception thrown at 2:12 - Unknown identifier
  1 | int main {
> 2 |     (5) -> some_func;
                ^
                Exception thrown at 2:12 - Unknown identifier
  3 | }
```

```
Error: Exception thrown at 2:9 - Invalid argument vector size
  1 | int main {
> 2 |     (5) -> some_func;
              ^
              Exception thrown at 2:9 - Invalid argument vector size
  3 | }
```

```
Error: Exception thrown at 2:15 - Invalid argument vector size
```

```
  1 | int main {
> 2 |     (1, 2, 3) -> some_func;
                     ^
                     Exception thrown at 2:15 - Invalid argument vector size
  3 | }
```

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>
```

5. Opis sposobu testowania

Wszystkie testy zostały wykonane z użyciem biblioteki `gtest`, sposób testowania oraz przetestowane elementy kodu zostały opisane poniżej

5.1 Analizator Leksykalny

Przetestowane zostały wszystkie możliwe typy tokenów oraz sytuacje w których działanie leksera kończy się błedem.

5.2 Analizator Składniowy

Testy parera obejmują 3 części:

- złożone konstrukcje
- proste konstrukcje i ich kombinacje
- sytuacje zwracające wyjątki

Każda z tych części została przetestowana przy użyciu wizytatora zliczającego wystąpienia poszczególnych elementów drzewa. Testy sparametryzowane obejmują prawie 200 permutacji wyrażeń oraz zdań oraz bardzo dużą ilość błędnie napisanych programów przy których spodziewamy się rzuczenia wyjątku.

Złożone konstrukcje obejmują większe programy, takie jak opisany wyżej program obliczający rozwiązanie równań kwadratowych

5.3 Interpreter

Testy interpretera obejmują zbiór przykładowych programów zawierających wszystkie konstrukcje w języku, sparametryzowane testy wyrażeń oraz przykładowe błędne programy. Sprawdzają one poprawność (lub niepoprawność) wyników zwracanych przez programy.