TKOM

Projekt wstępny

Bartosz Okoń, 304093

Przedmiotem projektu jest interpretowany język ogólnego przeznaczenia o zmiennych o wymaganej wartości, domyślnie mutowalnych; silnym i statycznym typowaniu.

1. Typy
   Obsługiwane przez język typy:
   - liczby całkowite `int – 0, 123, 9989…`
   - liczby zmiennoprzecinkowe `float – 0.001, 0.0, 121.2121`
   - zmienne łańcuchowe `str – ”tekst”, ”magi\”a)-_\n”`
   - sztuczny typ `void` oznaczający brak wartości zwracanej przez funkcję
   - typ deskryptora pliku `file`

   Dodatkowo każdy typ może zostać oznaczony jako const, co oznacza, że zmienna nim oznaczona nie może zostać zmieniona.

   Obsługiwane operacje na typach:
   a. liczbowych:
      i. arytmetyczne `+, -, *, /`
      ii. porównania `<, >, <=, >=, ==`
      iii. logiczne `not, and, or` (zero traktowane jak fałsz, wszystkie inne wartości jako prawda)
   b. dodatkowo całkowitych
      i. reszta z dzielenia `%`
   c. łańcuchowych
      i. konkatenacja `|`
      ii. porównanie `==`

   Wynikami operacji porównania i logicznych są liczby całkowite 0 - fałsz lub 1 - prawda.

2. Zmienne
   Zmienne są typowane statycznie silnie oraz muszą mieć wartość w każdej chwili działania programu. Oznacza to, że zmiennej należy nadać wartość w miejscu definicji.
   Identyfikator zmiennej musi się zaczynać literą i zawierać litery, liczby i podkreślenia.
   Definicja zmiennej:
   ```
   identyfikator: typ = wartość;
   ```

   Zmienne mają zakres nie wychodzący poza blok, w którym zostały zdefiniowane, ale są widoczne w blokach wewnątrz tego bloku.

   Przypisanie `zmienna1 = zmienna2;` oznacza skopiowanie drugiej zmiennej i przypisanie tej kopii do zmiennej pierwszej.

3. Funkcje
   Funkcje definiowane są poprzez słowo kluczowe fun, po którym następuje identyfikator funkcji, w nawiasach lista argumentów i ich typów, typ zwracany oraz blok stanowiący ciało funkcji:
   `fun nazwa(arg1: typ, arg2: typ,): typ {}`
   Argumenty funkcji zawsze dostarczane są jako referencje. Jeśli zajdzie potrzeba, można utworzyć kopię lokalną argumentu, przypisując go nowej zmiennej.
   Podczas uruchomienia programu uruchamiana jest domyślnie funkcja `main`.
4. Konstrukcje języka:
   a. pętla `while(warunek){kod do wykonania}`
   b. instrukcja `if(warunek){kod do wykonania}`
   c. instrukcja `if(warunek){kod do wykonania} else {…}`
   d. ciąg instrukcji `if(warunek){kod do wykonania} else if …`
   e. instrukcja `match`, przybierająca wartość na podstawie dopasowania do wzorca.
5. Pattern matching
   Jest on osiągany za pomocą konstrukcji `match`:
   `match(wyrażenie, wyrażenie, …){`

```
wzorzec1: wyrażenie1,
wzorzec2: wyrażenie2…}
```
Wzorzec jest postaci listy elementów o długości takiej samej, jak parametry `match`, oddzielonych przecinkami. Elementy mogą być wyrażeniami, funkcjami zwracającymi 0 lub 1, a także prostymi operacjami przyrównania (wartość argumentów wstawiana jako argument funkcji lub na początek porównania). Za pomocą znaku _ możemy określić, że dopasowanie danego elementu nie ma dla nas znaczenia. (przykładowy kod na dole dokumentu)
Blok sterujący `match` przybiera wartość dopasowanego wyrażenia.

6. Komentarze
   Komentarze zaczynają się od znaku #, a kończą na znaku nowej linii.
7. Uruchamianie oraz wejście i wyjście
   Program uruchamiany jest poprzez dostarczenie interpreterowi nazwy pliku jako argumentu. Kolejne argumenty przekazywane są uruchamianemu programowi, a dostęp do nich zapewniają funkcje `arguments_number()` oraz `argument(index: int): str.`

   Wydruk tekstu na ekran zapewnia funkcja `print`. Dostępne są też funkcje `to_str`, `to_int` i `to_float`.

   Dostęp do plików za pomocą funkcji bibliotecznych:
   - `open_file(filename): file`
   - `bad_file(file_descriptor: file): int`
   - `close_file(file_descriptor: file): void`
   - `read_line(file_descriptor: file): str`
   - `read(file_descriptor: file): str`
   - `write(file_descriptor, to_read: str): int`
8. Obsługa błędów
   Obsługa błędów odbywa się na podstawie sprawdzania kodu powrotu funkcji, które mogą nie wykonać się poprawnie, np.: `to_float(src: str, dst: float): int` może zwracać kod błędu, a wartość zapisywać w `dst` dzięki przekazywaniu parametrów przez referencję.
   Jeśli chodzi o błędy leksykalne, składniowe czy typu, interpreter wyświetli stosowny komunikat wyjaśniający błąd, miejsce jego wystąpienia, a następnie zakończy działanie programu.
9. Szczegóły techniczne
   Interpreter będzie napisany w języku C++. Pierwszym elementem będzie lexer budujący kolejne tokeny wedle opisu leksyki języka. Jego interfejs będzie umożliwiał pobieranie kolejnych tokenów. Tokeny różnych typów będą zawierały ich miejsce w tekście, i zależenie od typu, jego wartość.

   Parser rekursywnie zstępujący będzie pobierał tokeny i budował drzewo wyprowadzenia dla zdania będącego kodem źródłowym.

10. Formalna specyfikacja języka:
    Warstwa leksyki (elementy zaczynające się wielką literą są tokenizowane, symbole pomocnicze małymi literami i z wykorzystaniem wyrażeń regularnych):

```
Opening_parenth = "(";

Closing_parenth = ")";

Opening_curly   = "{";

Closing_curly   = "}";

Colon           = ":";

Semicolon       = ";";

Comma           = ",";
```

```
Underscore      = "_";
Hash            = "#";
Assign          = "=";
Plus            = "+";
Minus           = "-";
Multiplication  = "*";
Division        = "/";
Modulo          = "%";
String_concat   = "|";
And             = "and";
Or              = "or";
Not             = "not";
Equals          = "==";
Lt              = "<";
Gt              = ">";
Lte             = "<=";
Gte             = ">=";
Integer_type    = "int";
Floating_type   = "float";
String_type     = "str";
File_type       = "file";
Void_type       = "void";
Return_keywd    = "return";
Function_keywd  = "fun";
If_keywd        = "if";
Else_keywd      = "else";
While_keywd     = "while";
Const_keywd     = "const";
Match_keywd     = "match";


letter = [A-Za-z]
escaped_character = \\[abefnrtv\\'"\?]
nonzero_digit = [1-9]
```

```
zero_digit = 0

other_character = [^a-zA-z\\"'0-9]

char_not_newline = [^\r\n]

newline_chr = [\r\n]



letter_or_under         = letter

                        | underscore;

digit                   = zero_digit

                        | nonzero_digit;

Integer_literal         = zero_digit

                        | (nonzero_digit, {digit});

Floating_literal        = (zero_digit

                         | (nonzero_digit, {digit})),

                          ".", (zero_digit

                              | ({digit}, nonzero_digit));

String_literal          = """, {letter

                              | digit

                              | escaped_character

                              | other_character},

                          """;



Identifier              = letter, {letter_or_under

                                 | digit};

Comment                 = hash, {char_not_newline}, newline_chr;

Newline                 = newline_chr;
```

Część składniowa:

```
start_symbol                    = {function_definition};

function_definition             = Function_keywd, Identifier, Opening_parenth,
argument_list_definition, Closing_parenth, Colon, (type_identifier | Void_type), code_block;

argument_list_definition        = [argument_definition, {Comma, argument_definition}];

argument_definition             = Identifier, Colon, type_identifier;

type_identifier                 = [Const_keywd], (Integer_type

                                                | Floating_type

                                                | String_type
```

```
                                        | File_type);

code_block                  = Opening_curly, {statement

                                        | control_block},

                              Closing_curly;

statement                   = (var_def_assign_or_funcall

                            | return_statement), Semicolon;

math_expression             = factor, {(Plus | Minus | String_concat), factor};

factor                      = term, {(Multiplication | Division | Modulo), term};

term                        = [Minus], (literal

                                    | identifier_or_funcall

                                    | Opening_parenth, logic_expression, Closing_parenth);

logic_expression            = logic_factor, {Or, logic_factor};

logic_factor                = relation, {And, relation};

relation                    = [Not], math_expression, relation_operator, math_expression;

identifier_or_funcall       = Identifier, {parenths_and_args};

parenths_and_args           = Opening_parenth, argument_list, Closing_parenth;

argument_list               = [math_expression, {Comma, math_expression}];

var_def_assign_or_funcall   = Identifier, ([Colon, type_identifier], Assign,
(math_expression | match_operation)

| parenths_and_args);

relation_operator           = Lt

                            | Gt

                            | Lte

                            | Gte

                            | Equals;

return_statement            = Return_keywd, [math_expression];

literal                     = Integer_literal

                            | Floating_literal

                            | String_literal;

control_block               = if_block

                            | while_block

                            | match_operation;

if_block                    = If_keywd, Opening_parenth, logic_expression, Closing_parenth,
code_block, [else_block];

else_block                  = Else_keywd, (if_block | code_block);
```

```
while_block                     = While_keywd, Opening_parenth, logic_expression, Closing_parenth,
code_block;

match_operation                 = Match_keywd, Opening_parenth, match_arguments, Closing_parenth,
match_block;

match_arguments                 = math_expression, {Comma, math_expression};

match_block                     = Opening_curly, match_line, Closing_curly;

match_line                      = pattern, Colon, math_expression, Comma;

pattern                         = pattern_element, {Comma, pattern_element};

pattern_element                 = math_expression

                                | (relation_operator, math_expression)

                                | Underscore;
```

11. Przykłady użycia języka:

```
fun fibonacci(n: const int): int {

    return match(n){

        0:  0,

        1:  1,

        _:  fibonacci(n - 1) + fibonacci(n - 2),

    };

}



fun fizzbuzz(n: const int): void {

    temp: int = 1;

    while(temp <= n){

        match(temp % 3, temp % 5){

            0, 0:   print("fizzbuzz"),

            0, _:   print("fizz"),

            _, 0:   print("buzz"),

            _, _:   print(temp),

        }

        temp = temp + 1;

    }

}



fun progressive_tax(salary: const float): float {   # not accurate

    tax_rate: float = match(salary){
```

```
        < 1000.0:   0.1,
        < 2500.0:   0.3,
        _:          0.5,
    };
    return salary * tax_rate;
}

fun is_even(n: const int): int {
    return match(n % 2){
        0:  1,
        1:  0,
    };
}

fun is_even2(n: const int): int {
    return match(n){
        is_even:    1,
        _:          0,
    };
}



fun print_file(filename: const str): void {
    handle: file = open_file(filename);
    if(bad_file(handle)){
        return;
    }
    else{
        line: str = read_line(handle);
        while(line != EOF_MARKER){
            print(line);
            line = read_line(handle);
        }
        close_file(handle);
```

```
    }
}


fun main(): void{
    if(arguments_number() < 1){
        exit(1);
    }
    print_file(argument(1));


    number: int = 6;
    print(to_str(number) | ". Fibonacci number is " | to_str(fibonacci(number)));


    print("2 + 3 * 2 = " | to_str(2+3*2) | " == 8?");


    if(number < 12 and is_even(number) or not is_even(number)){
        print("Logical precedence works ?")
    }


    print("Tax for 1500zl = " | tax_rate(1500.0));
}
```