TKOM

Projekt

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`
   - typ zmiennej boolowskiej `bool – true, false`

   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 zmienne boolowskie.

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 (o ile wewnątrz bloku nie zostanie zdefiniowana zmienna o tej samej nazwie, która by przysłoniła tę zewnętrzną).

   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 wzorca są wyrażeniami. Mogą się na nie składać operacje porównania z tylko prawym argumentem podanym explicite (lewy stanowi odpowiedni argument matcha), identyfikatory funkcji jednoargumentowych (argumentem podobnie staje się argument matcha), ale także zwykłe dwuelementowe elementy wyrażeń jak operacje logiczne czy matematyczne. Elementy wzorca ewaluowane są do wartości, a jej typ dyktuje dalsze postępowanie:

- Jeśli typ wynikowy jest taki sam jak wejściowy, to za trafienie uznajemy równość tych wartości
- Jeśli typ wynikowy jest boolem, a tym wejściowy jest inny, to za trafienie uznajemy wartość true wyniku
- Jeśli typy są różne w inny sposób, uznajemy to za brak trafienia.

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 – dlatego niedopasowanie żadnego wyrażenia powoduje błąd i zatrzymanie programu.

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): bool`
   - `close_file(file_descriptor: file): void`
   - `read_line(file_descriptor: file): str`

8. Obsługa błędów
   Obsługa błędów odbywa się na podstawie sprawdzania kodu powrotu funkcji, które mogą nie wykonać się poprawnie, np.: `copy(src: str, dst: str): int` może zwracać kod błędu lub wartość bool, 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 składa się z 3 głównych modłów:

   a. Lexer – pobiera kolejne znaki ze strumienia wejściowego i próbuje budować kolejne tokeny. Tokenizuje on leniwie, gdyż kolejne części programu nie potrzebują dostępu do więcej niż jednego tokenu naraz. Za budowę każdego rodzaju tokenu jest odpowiedzialna osobna funkcja. Każda z nich zwraca obiekt typu optional, dzięki czemu wiemy, która funkcja naprawdę utworzyła token. Rzucane są wyjątki gdy pojawi się sekwencja symboli, której nie przewidywano przy tworzeniu języka, a także gdy literały są zbyt duże, by przechowywać je w zmiennych używanych w programie.
   b. Parser – jest to parser rekursywnie zstępujący, zaczyna parsować od symbolu startowego i kontynuuje w nadziei, że uda mu się do tego symbolu powrócić. Kolejne definicje funkcji

zbierane są do kontenera, w definicjach funkcji są zapisywane parametry, bloki instrukcji i tak coraz głębiej. Od razu tworzone jest drzewo obiektów tworzących poszczególne funkcje. Poza funkcjami pomocniczymi, każda funkcja zaczynająca się od „try_parse" tworzy nowy obiekt, zgodnie z przypisaną jej produkcją, na stercie i przekazuje unique pointer do niego. Zgłaszane są wyjątki w przypadkach trafienia na token zły z punktu widzenia aktualnie parsowanej produkcji, bądź całkowity jego brak. Wskaźniki, w zależności czy obiekt jest logicznie w stanie przyjąć jakąś wartość, są na interfejsy IInstruction i IExpression (który dziedziczy po IInstruction). Oba interfejsy dziedziczą po interfejsie IVisitable, co umożliwia na działanie wizytatora w późniejszym etapie działania.
Kontener funkcji jest potem konwertowany na obiekt program, który odpowiada za wykrycie, czy nie ma duplikatów funkcji, oraz umieszczenie ich w mapie, za pomocą której są dostępne po nazwie.

c. Interpreter jest zaimplementowany jako wizytator. Przekazuje mu się program, strumień wyjściowy oraz wektor argumentów podawanych podczas uruchamiania programu. Działanie rozpoczyna funkcja run (która zwraca też kod powrotu), można podać jej nazwę funkcji, od której program ma zacząć działać (o ile nie przyjmuje argumentów). Używamy pomocniczych obiektów Context, który przechowuje kolejne Scopy. W Contexcie znajdują się argumenty dla pattern matchingu oraz aktualny index argumentu(umożliwia to rekursywne wywoływanie funkcji z instrukcją match); a także argumenty dla uruchamianej funkcji, przygotowywane przez funkcję wołającą.
Przy obliczaniu wartości operatorów korzystamy z map typów operatorów na lambdy, std::visit oraz wzorca overload.
Funkcje w przypadku pattern matchingu (oznaczane tylko identyfikatorem) uruchamiane są poprzez flagę oznaczającą, że operacja patern matching trwa oraz przerobienia ich na normalne zawołanie funkcji. Podobnie jest z operatorami dwu-argumentowymi, które przy match stają się jedno-argumentowe – również je konwertujemy i wywołujemy.
Funkcje wbudowane różnią się od zwykłych jedynie tym, że wskaźnik na ich blok instrukcji jest zerowy, a to oznacza że ich „treści" trzeba szukać w mapie lambd – wywołuje się je tak samo.

d. W funkcji main jedynie wczytujemy nazwę pliku jako argument, kolejne argumenty umieszczamy w wektorze, inicjujemy wszystkie obiekty i uruchamiamy funkcję run, która znajduje się w bloku try-catch. Przechwytywane są tam wszystkie błędy, i wypisywane na ekran. Poza nazwą błędu znajduje się tam też w miarę możliwości więcej informacji o błędzie oraz jego pozycja w kodzie.

10. Testowanie. Lexer i parser testowany jest za pomocą testów jednostkowych napisanych w GoogleTest. Praktycznie każda funkcja jest wywoływana (co potwierdza analiza gcovem), a pokrycie kodu jest dość dobre. Testy sprawdzają też, czy program zachowuje się jak powinien wtedy, gdy nie ma działać (np. zła sekwencja tokenów).
Interpreter testowany jest za pomocą fragmentów kodu, z których największy jest na końcu tego dokumentu.
11. Opis użytkowy. Programy uruchamiamy, uruchamiając interpreter z argumentem będącym plikiem do wykonania. Kolejne argumenty traktowane są jako argumenty programu interpretowanego. W przypadku błędów, użytkownik jest o nich informowany w zrozumiały sposób.
12. 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      = "_";
Assign          = "=";
Plus            = "+";
Minus           = "-";
Multiplication  = "*";
Division        = "/";
Modulo          = "%";
String_concat   = "|";
And             = "and";
Or              = "or";
Not             = "not";
True            = "true";
False           = "false";
Equals          = "==";
Not_equals      = "!=”;
Lt              = "<";
Gt              = ">";
Lte             = "<=";
Gte             = ">=";
Integer_type    = "int";
Floating_type   = "float";
String_type     = "str";
File_type       = "file";
Bool_type       = "bool";
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 = \\[abfnrtv\\"\?]

nonzero_digit = [1-9]

zero_digit = 0

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


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},

                          """;

Boolean_literal         = "true" | "false";


Identifier              = letter, {letter_or_under

                                 | digit};

Comment                 = "#", ? characters that are not newline sequence ?
```

Część składniowa:

```
start_symbol                    = {function_definition};

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

parameter_list_definition        = [parameter_definition, {Comma, parameter_definition}];

parameter_definition             = Identifier, Colon, type_identifier;
```

```
type_identifier                 = [Const_keywd], (Integer_type
                                                | Floating_type
                                                | String_type
                                                | File_type
                                                | Bool_type);
code_block                      = Opening_curly, { statement_or_control_block }, Closing_curly;
statement_or_control_block      = statement | control_block;
statement                       = (var_def_assign_or_funcall
                                | return_statement), Semicolon;
expression                      = logic_factor, {Or, logic_factor};
logic_factor                    = relation, {And, relation};
relation                        = [Not], math_expression, [relation_operator, math_expression];

math_expression                 = factor, {(Plus | Minus | String_concat), factor};
factor                          = term, {(Multiplication | Division | Modulo), term};
term                            = [Minus], (literal
                                           | identifier_or_funcall
                                           | ( Opening_parenth, expression, Closing_parenth));


match_expression                = match_logic_factor, {Or, match_logic_factor};
match_logic_factor              = match_relation, {And, match_relation};
match_relation                  = relation_operator, match_math_expression;

match_math_expression           = match_factor, {(Plus | Minus | String_concat), match_factor};
match_factor                    = match_term, {(Multiplication | Division | Modulo), match_term};
match_term                      = [Minus], (literal
                                         | identifier_or_funcall
                                             | ( Opening_parenth, match_expression,
Closing_parenth));


identifier_or_funcall           = Identifier, [parenths_and_args];
parenths_and_args               = Opening_parenth, argument_list, Closing_parenth;
argument_list                   = [expression, {Comma, expression}];
```

```
var_def_assign_or_funcall       = Identifier, ([Colon, type_identifier], Assign, (expression |
match_operation)| parenths_and_args);

relation_operator               = Lt

                                | Gt

                                | Lte

                                | Gte

                                | Equals

                                | Not_equals;

return_statement                = Return_keywd, [expression | match_operation];

literal                         = Integer_literal

                                | Floating_literal

                                | String_literal

                                | Boolean_literal;

control_block                   = if_block

                                | while_block

                                | match_operation;

if_block                        = If_keywd, condition, code_block, [else_block];

else_block                      = Else_keywd, (if_block | code_block);

while_block                     = While_keywd, condition, code_block;

condition                       = Opening_parenth, expression, Closing_parenth;

match_operation                 = Match_keywd, Opening_parenth, expression, {Comma, expression},
Closing_parenth, match_block;

match_block                     = Opening_curly, match_line, {Comma, match_line}, Closing_curly;

match_line                      = pattern, Colon, expression;

pattern                         = pattern_element, {Comma, pattern_element};

pattern_element                 = expression

                                | match_expression

                                | Underscore;
```

13. 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(to_string_int(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): bool {
    return match(n % 2){
        0:  true,
        1:  false
    };
}


fun is_even2(n: const int): bool {
    return match(n){
        is_even:    true,
```

```
        _:          false
    };
}

fun while_return_test(): void{
    number: int = 4;
    while(number > 0){
        if(number == 1){
            return;
        }
        print(to_str_int(number) | "\n");
        number = number - 1;
    }
}

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 | "\n");
            line = read_line(handle);
        }
        close_file(handle);
    }
}

fun add_one(number: int): void {
    number = number + 1;
}
```

```
fun multiple_matches(): int{

    return match(1, 2, 3){

        is_even2 and is_even, is_even2, is_even2: 0,

        _, is_even2, _              : 1,

        _, _, _                     : 2

    };

}


fun main(): int{

    if(arguments_number() >= 1){

        print("Opening and printing file " | argument(0) | "\n");

        print_file(argument(0));

        print("Done!\n");

    } else {

        print("No filename provided as argument, skipping...\n");

    }







    # bad_variable: int = 14.33;

    # print(to_str_int(bad_variable));



    number: int = 6;

    print("Fibonacci[" | to_str_int(number) | "] number is " | to_str_int(fibonacci(number)) | ".\n");



    print("Checking if passing by reference works - printed below should be five (4 after
incrementing)\n");

    do_dodania: int = 4;

    add_one(do_dodania);

    print(to_str_int(do_dodania) | "\n");

    #add_one(do_dodania + 1);                    #generuje error - dodanie do stalej (r-wartosci)

    #print(to_str_int(do_dodania) | "\n");




    print("2 + 3 * 2 = " | to_str_int(2+3*2) | " == 8?\n");
```

```
    if(number < 12 and is_even(number) or not is_even(number)){

        print("Logical precedence works!\n");

    }



    print("Tax for 1500zl = " | to_str_float(progressive_tax(1500.0)) | "\n");



    print("If multiple pattern elements and many levels of match calls work,\nthis should be a 1: " |
to_str_int(multiple_matches()) | "\n");



    while_return_test();



    return 0;

}
```