Dokumentacja wstępna TKOM

Piotr Gębuś nr indeksu: 304045

1.Temat (dyn / sil / sta/ wym)

Zaimplementować interpreter prostego języka programowania, który cechuje się dynamicznym i silnym typowaniem. Zmienne są domyślnie stałe i w każdym momencie swojego istnienia posiadają prawidłową wartość (brak opcjonalności). Dodatkowo język musi zawierać w sobie mechanizm pattern matchingu.

2.Zakładana funkcjonalność:

- Odczytanie, parsowanie i analiza programów napisanych w plikach tekstowych lub w strumienia danych (wykorzystywane w przypadku testowania)
- Kontrola poprawności wprowadzanych danych i zgłaszanie wykrytych błędów
- Instrukcja warunkowa typu *if – else*
- pętla typu *while*
- operatory: *+*, *-*, ***, */*, *()*, *%*, *==*, *!=*, *>*, *>=*, *<*, *<=*, *&&*, *||*
- definiowane funkcji (brak możliwości rozdzielenia deklaracji i definicji)
- zmienne i stałe, które zawsze muszą mieć określoną wartość
- funkcja *print*() wypisujące dane, zoverloadowane dla floata,inta, stringa i boola
- Zastosowanie typowania dynamicznego i silnego
- pattern matching za pomocą słowa kluczowego *match*, gdzie symbol _ jest wartością wyrażenia podaną w wywołaniu
- przypisywanie wartości przez znak = a nie referencji
- parametry w funkcjach nie zawierają wartości domyślnych
- funkcje nie muszą zwracać wartości
- główna funkcja programu nazywa się „main”
- rzutowanie zrealizowane jako funkcje biblioteki standardowej

3. Przykłady

Deklaracja i definicji zmiennej:

```
var abc = 30;
```

Przykład konstrukcji while:

```
mut var i = 5;

while(i)

{

	print(i);

	i = i −1;
}
```

Przykład definicji funkcja i instrukcji warunkowej typu if – else:

```
fn fib(var n)
{
        if( n <= 1)
        {
                return n;
        }
        else
        {
                return fib(n-1) + fib(n-2);
        }
}
```

Przykład pattern matchingu:

```
match(command)
{
        case _ == “quit” || _ == “exit”:
        {
             print(“quit”);
        }
        case _:
        {
            print(“Unknown command “ + _);
        }
}
```

Przykład programu korzystającego z funkcji fib

```
fn main()
{
        mut var n = 9;
        print(fib(n));
        n = “dynamiczny”;
        print(n);
}
```

4.Gramatyka

```
program                    = functionDefinition, { functionDefinition };
statement                  = conditionalStatement | nonConditionalStatement;
conditionStatement         = ifStatement | whileStatement | patternStatement;
nonConditionalStatement    = (assignment | functionCall,| returnStatement |
variableDeclaration), ";" ;
functionDefinition         = "fn", identifier, "(", [ parameterList ], ")", "{", {
statement }, "}" ;
whileStatement             = "while", "(", expression, ")", "{", { statement }, "}";
patternStatement           = ("match", "(", expression, ")", "{", { caseStatement }
"}") | ("match", "(", expression, ")", "{", { caseStatement }, returnStatement "}") ;
caseStatement              = "case", expression, ":", "{", { statement }, "}";
functionCall               = identifier, "(", [argumentList] , ")" ;
parameterList              = parameter, {",", parameter};
ifStatement                = ifBlock, [ elseBlock ];
ifBlock                    = "if", "(", expression, ")", "{", {statement}, "}";
elseBlock                  = "else", "{", {statement}, "}" ;
argumentList               = expression, {",", expression } ;
parameter                  = ["mut"] , "var", identifier;
returnStatement            = "return", [expression];
variableDeclaration        = ["mut"], "var", identifer , ["=", expression] ;


expression                 = andExpression, { orOperator, andExpression } ;
andExpression              = relationExpression ,{andOperator, relationalExpression};
relationExpression         = baseLogicExpression, { relationOperator,
baseLogicExpression };
baseLogicExpression        = [ "!" ], mathematicalExpression ;
mathematicalExpression     = multiplicativeExpression, { additiveOperator,
multiplicativeExpression };
multiplicativeExpression   = baseMathematicalExpression, { multiplicativeOperator,
baseMathematicalExpression };
baseMathematicalExpression = [ "-" ], simpleExpression  ;
simpleExpression           = constant |  "(", expression, ")" | functionCall |
identifier;


nonZeroDigit               = "1" .. "9" ;
digit                      = "0" | nonZeroDigit ;
number                     = "0" | nonZeroDigit, { digit } ;
float                      = numberLiteral, ".", numberLiteral ;
boolean                    = "true" | "false" ;
letter                     = "a" .. "z" | "A" .. "Z" ;
identifier                 = ( letter | "_"), { letter | digit | "_"} ;
relationOperator           = "==" | "!=" | "<" | ">" | "<=" | ">=" ;
andOperator                = "&&" ;
orOperator                 = "||" ;
multiplicativeOperator     = "*" | "/" | "%" ;
additiveOperator           = "+" | "-";
numberLiteral              = digit, { digit } ;
```

```
character                = ? all visible characters ? ;
stringLiteral            = "“", { character – "”" | escapeSequence }, "”" | "‘", { character – ‘’’ | escapeSequence }, "’";
escapeSequence           = "\", character;
constant                 = stringLiteral | numberLiteral | float | boolean;
```

5.Informacje techniczne

Projekt zostanie zaimplementowany w języku C++ w środowisku Visual Studio.

Projekt będzie aplikacją konsolową, która jako argument będzie przyjmować ścieżkę do pliku z p kodem do interpretacji. W przypadku jakichkolwiek błędów będą one zgłaszane do użytkownika poprzez konsolę. W konsoli będą również wyświetlane komunikaty przekazywane przez funkcję „print”;

Przykładowe uruchomienie:

./interpreter pliki/przykład.tkom

6. Opis sposobu realizacji:

Moduł leksera:

Lekser jest zrealizowany jako klasa, która zwraca kolejne tokeny. Korzysta ze źródła danych, którym jest ciąg znaków tekstowych (może to być plik). Lekser próbuje po kolei utworzyć token: słowo kluczowe, napis, zmienną itd. aż dojdzie do końca pliku.

Rodzaje tokenów są określone w dodatkowej mapie.

Moduł parsera:

Parser pobiera tokeny z leksera i sprawdza, czy tworzą one wspólnie poprawną strukturę. Tworzy nowe klasy odpowiadające wczytywanym tokenom. Przykładowe klasy (reprezentajce instrukcji) to między innymi: Function, Operation, Declaration, Variable, Value etc.

Parser zajmuje się również rzucaniem wyjątków w przypadku zauważenia niepożądanej sytuacji np. brak symbolu ‘}’ na końcu odczytywanej funkcji.

Moduł interpretera:

Interpreter zajmuje wykonywaniem klas utworzonych przez parser. Aktualizuje wartości zmiennych, wypisuje ciąg znaków do konsoli w przypadku spotkania funkcji print(), dokonuje operacji matematycznych na zmiennych, czy implementuje pattern matching.

Interpreter tak samo jak parser zajmuje się również wyrzucaniem wyjątków, gdy np. będziemy chcieli zmodyfikować wartość stałej zmiennej.

7.Testowanie

Do testów zostanie wykorzystana biblioteka Boost.Test.

Dla każdego modułu będą osobne testy, gdzie dane będą wprowadzane przez ciąg znaków.

Dodatkowo w repozytorium będzie się znajdował przykładowy program w oddzielnym pliku tekstowym.