Jakub Bryl - 293085
TKOM - 20L

Opis projektu

Projekt ma na celu wykonanie interpretera prostego języka który składnią ma przypominać język Scala. Język ten ma być wyposażony w zmienne z zasięgiem,podstawową konstrukcje sterującą (instrukcja warunkowa) oraz możliwość definiowania funkcji i klas. Powinny być również obsługiwane wyrażenia matematyczne i logiczne uwzględniające priorytet operatorów.

Wymagania funkcjonalne

1. Odczytywanie, parsowanie i analiza skryptów zapisanych w plikach tekstowych
2. Kontrola poprawności wprowadzonych danych oraz zgłaszanie błędów wykrytych na każdym z etapów analizy.
3. Poprawne wykonywanie poprawnie zapisanych instrukcji w pliku skryptowym
4. Możliwość definiowania własnych zmiennych, funkcji.
5. Wykonywanie wyrażeń logicznych uwzględniając priorytet operatorów ((), &&, ||, ==, !=)
6. Wykonywanie wyrażeń matematycznych uwzględniając priorytet operatorów ((), +, -, /, *)
7. Możliwość używania instrukcji warunkowych
8. Silna typizacja (<- Tego nie ma. Typy są dynamiczne, mimo to użytkownik przy pisaniu skryptu musi zapisywać jakiego typu oczekuje.)
9. Możliwość definicji własnej klasy - definiowanie nazwy, listy parametrów oraz metod do niej należących.
10. Możliwość tworzenia instancji obiektów na podstawie zdefiniowanych klas.

Wymagania niefunkcjonalne

1. Po uruchomieniu aplikacji z niepoprawnymi parametrami powinno nastąpić poinformowanie użytkownika o możliwych prawidłowych parametrach startowych
2. Komunikaty o błędach powinny wskazywać gdzie błąd wystąpił oraz co jest jego przyczyną
3. Prezentowanie wyników wykonania skryptów w przejrzysty sposób

Środowisko

Projekt został zaimplementowany w języku Scala, a do testów jednostkowych wykorzystana została biblioteka ScalaTest.

Obsługa programu

Program jest aplikacją konsolową uruchamianą wraz z parametrem reprezentującym ścieżkę do pliku ze skryptem do interpretacji. Wynik działania programu oraz ewentualne informacje o błędach są wyświetlane na standardowym wyjściu.

Sposób uruchomienia:

<w repozytorium jest Dockerfile więc można uruchamiać w kontenerze bez instalacji sbt>

> sbt

[sbt] > run [path_to_script]

Sposób realizacji

Program będzie złożony z modułów odpowiedzialnych za kolejne fazy analizy plików wejściowych oraz moduł wykonujący wygenerowane na podstawie analizy instrukcje.

Moduły główne:

1. FileHandler- moduł zapewniający operacje odczytywania pliku.
2. Lexer - odpowiedzialny za utworzenie tokenów na podstawie pliku wejściowego. Będzie pobierał od FileHandlera kolejne znaki aż do odczytania całej sekwencji odpowiadającej jednemu z akceptowanych tokenów języka(np. “else”).
3. Parser - moduł sprawdzający czy kolejno otrzymane od Lexera tokeny są ułożone zgodnie z gramatyką języka i na ich podstawie tworzy drzewo składniowe. Drzewo składniowe składa się z obiektów klas implementujących interfejsy Expression oraz Statement.
4. Interpreter - ma za zadanie wykonywanie instrukcji zawartych w drzewie uzyskanym na poprzednich etapach analizy. Wykorzystuje wzorzec projektowy Wizytatora. Na tym poziomie mogą pojawić się błędy spowodowane odwoływaniem się do nieistniejących wartości lub związane z wartościami zmiennych(np. dzielenie przez zero) lub redefinicją zmiennych/klas/funkcji.

Interfejsy Wizytatorów:

```
trait StatementVisitor[R] {

 def execute(statement: Statement) =
   statement.accept(this)

 def visitBlockStmt(stmt: Block): R
 def visitClassStmt(stmt: ClassStatement): R
 def visitExpressionStmt(stmt: ExpressionStatement): R
 def visitFunctionStmt(stmt: FunctionStatement): R
 def visitIfStmt(stmt: IfStatement): R
 def visitPrintStmt(stmt: PrintStatement): R
 def visitReturnStmt(stmt: ReturnStatement): R
 def visitVarStmt(stmt: VarStatement): R
}

trait ExpressionVisitor[R] {
 def evaluate(expr: Expression) =
   expr.accept(this)
 def visitSetExpression(set: SetExpression): R
 def visitAssignExpression(expr: AssignExpression): R
 def visitBinaryExpression(expr: BinaryExpression): R
 def visitCallExpression(expr: CallExpression): R
 def visitGetExpression(expr: GetExpression): R
 def visitGroupingExpression(expr: ParenthExpression): R
 def visitLiteralExpression(expr: LiteralExpression): R
 def visitLogicalExpression(expr: LogicalExpression): R
 def visitThisExpression(expr: ThisExpression): R
 def visitUnaryExpression(expr: UnaryExpression): R
 def visitVariableExpression(expr: VariableExpression): R
}
```

Moduł pomocniczy

1. Moduł obsługi błędów - moduł w sposób czytelny prezentujący użytkownikowi błędy otrzymane w modułach głównych

Testowanie:

Do każdego z modułów głównych utworzyłem testy jednostkowe.
Jednak w przypadku Parsera i Interpretera są one wybrakowane - nie sprawdzają poprawności działania tych modułów, a jedynie to, czy został lub nie został rzucony wyjątek oraz, niestety, korzystają z implementacji modułów na których polegają.
Poprawność wyników działania tych modułów sprawdzałem ręcznie.

Opis języka:

Gramatyka:

root -> declaration*
declaration = varDecl | classDecl | funDecl | statement
varDecl -> "var" IDENTIFIER "=" expression
classDecl -> "class" IDENTIFIER “(“ parameters “)” classBlock
funDecl -> "def" IDENTIFIER “(“ parameters “)” ofType “=” statement
type -> IDENTIFIER
parameters -> ((parameter “,”)* parameter)?
parameter -> IDENTIFIER ofType
ofType -> :type

statement ->
exprStmt
| ifStmt
| printStmt
| returnStmt
| blockStmt

expressionStmt = assignment
returnStmt -> “return” expression
ifStmt -> "if" "(" expression ")" statement ("else" statement)?
expression = logic_or
blockStmt -> "{"(declaration*, statement)? statement* "}"
classBlock -> “{“ funDecl* “}”
printStmt -> "print""(” logic_or ")”
assignment → (call ".")? IDENTIFIER "=" expression | expression;
logic_or → logic_and ("or" logic_and)*
logic_and → equality ("and" equality)*
equality → comparison (("!=" | "==") comparison)* ;
comparison → addition ((">" | ">=" | "<" | "<=") addition)* ;

```
addition       → multiplication ( ( "-" | "+" ) multiplication )* ;
multiplication → unary ( ( "/" | "*" ) unary )* ;
unary          → ( "!" | "-" ) unary
               | call ;
call           → primary ( "(" arguments ")" | "." IDENTIFIER )*
primary        → "(" expression ")"
| literal | IDENTIFIER
arguments -> ((parse_or ",")* parse_or)?
literal    → NUMBER | STRING | "true" | "false" | "nil" ;
grouping   → "(" expression ")" ;
unary      → ( "-" | "!" ) expression ;
operator   → "==" | "!=" | "<" | "<=" | ">" | ">="
           | "+"  | "-"  | "*" | "/" ;
```

Blok - lista instrukcji.

Zakres bloku

- zawiera informacje o dostępnych dla obecnie wykonywanej instrukcji zmiennych, funkcjach lub definicjach klas.
- każdy utworzony zakres zawiera odwołanie do zakresu "wyższego" rzędu.
- podczas wywoływania funkcji do utworzonego na potrzeby jej wywołania zakresu dodawane są zmienne o identyfikatorach odpowiadających parametrom z definicji funkcji.
- podczas wykonywania metody należącej do **instancji klasy**, przy tworzeniu jej zakresu dodawana jest zmienna **this** która jest odwołaniem do wspomnianej wcześniej instancji.
- po wykonaniu bloku instrukcji, zakres utworzony na jego potrzeby jest usuwany.
- możliwe jest dodawanie do zakresu obiektów o identyfikatorach istniejących w wyższych zakresach. W takim wypadku podczas odwoływania się do obiektu o danym identyfikatorze zwrócony zostanie identyfikator z najbliższego zakresu.
- nie ma ustalonego limitu głębokości zakresu

Instrukcje podstawowe:

- **Wypisanie na wyjście** – zdefiniowana funkcja print, która jako parametr przyjmuje tekst, liczbę, zmienną lub wywołanie funkcji.
- **Operacja arytmetyczna** – program potrafi przetwarzać podstawowe operacje arytmetyczne, takie jak dodawanie, odejmowanie, mnożenie i dzielenie. Możliwe są także wyrażenia nawiasowe.
- **Warunek** – wartość wyrażenia logicznego(np. 4 > 5).
- **Przypisanie** – lewa strona „równania" to zmienna o typie dynamicznym natomiast prawa to przypisywana do niej wartość. Wartością może być liczba, napis otoczony znakami **"** lub instancja klasy.
- **Utworzenie zmiennej** - definiowana słowem kluczowym **var** po którym następuje identyfikator tworzonej zmiennej, znak **: ,** umowny typ zmiennej, znak **=** oraz przypisywana wartość inicjująca zmienną.

- **Odwołanie** - zawołanie wcześniej zdefiniowanej funkcji, zmiennej, konstruktora klasy lub atrybutów wcześniej zdefiniowanych obiektów.
- **Komentarz** – reprezentowany przez **//**, reszta linii nie jest analizowana przez lexer.

Instrukcje złożone to takie instrukcje, które tworzą nowy blok. Zakres bloku definiowany jest poprzez klamry { }, ale istnieje też możliwość tworzenia jednoliniowych bloków(np po warunku w instrukcji if).

W jednoliniowych blokach nie ma możliwości definiowania czegokolwiek ponieważ definiowanie czegoś co nie będzie używane i ma zaraz zniknąć nie ma sensu.

Instrukcje złożone:

- **Warunek if** - definiowany przez słowo kluczowe **if** oraz warunek zamknięty w nawiasach po którym znajduje się blok instrukcji do wykonania w razie gdy warunek zachodzi. Po tym bloku możliwe jest zdefiniowanie alternatywnego bloku do wykonania który jest uruchamiany gdy wspomniany wcześniej warunek nie zachodzi. Aby ten alternatywny blok zdefiniować należy przed nim umieścić słowo kluczowe **else**.
- **Definicja funkcji** - definiowana przez słowo kluczowe **def** po którym należy umieścić identyfikator funkcji, listę parametrów otoczoną nawiasami otwartymi, znak **:** , zwracany typ a następne znak **=** i blok.
- **Definicja klasy** - definiowana przez słowo kluczowe **class** po którym tak jak w funkcji należy umieścić jej identyfikator oraz listę parametrów otoczoną nawiasami oraz blok **(w tym przypadku blok musi być w klamrach, a w samym bloku możliwe jest jedynie definiowanie funkcji)**.
-

Typizacja

Obecnie język jest typowany dynamicznie, jednak osoba z niego korzystająca musi zapisywać jakiego typu wartości w danym miejscu umownie oczekuje.

Sposób realizacji definiowania i wywoływania funkcji

Podczas interpretacji definicji funkcji tworzony jest, obiekt klasy **MyFunction** implementujący **MyCallable** która zawiera w sobie deklarację funkcji.
Obiekt ten udostępnia też metodę **call** która sprawdza czy podano poprawną liczbę argumentów, tworzy nowy zakres i dodaje do niego podane argumenty - <nazwa parametru z deklaracji. argument>, opcjonalnie gdy wywołujemy metodę należącą do jakiegoś obiektu - dodaje do zakresu parę <"this", obiekt> po czym zaczyna interpretację bloku z deklaracji funkcji.

Jeżeli podczas interpretacji bloku z deklaracji funkcji natkniemy się na instrukcję return, rzucany jest obiekt Return dziedziczący po klasie Throwable który jest łapany, a następnie zwracany przez **call**.

Sposób realizacji obiektowości języka

Podczas interpretacji definicji klasy tworzony jest, a następnie dodawany do Scope'a, obiekt klasy **MyClass** implementujący **MyCallable** która zawiera w sobie

informacje o nazwie nowej klasy, liście parametrów potrzebnych do jej utworzenia i listę metod jakie ta klasa posiada. Obiekt ten udostępnia też metodę call która zwraca nową instancję tej klasy.

Instancja klasy zawiera w sobie mapę<identyfikator, atrybut> w której identyfikatory to lista parametrów klasy, a atrybuty to wartości jakie zostały przekazane jako lista argumentów przy wołaniu call na Klasie.

Przykład skryptu:

```
class List(head: Int, tail: List) {
  def get(x: Int): Int = {
    if (x <= 0)
      return this.head
    if (this.tail != nil)
      return this.tail.get(x-1)
    else
      return nil
    }

  def add(x: Int): List =
    return List(x, this)

  def length(): Int = {
    if (this.tail == nil) return 1
    else return 1 + this.tail.length()
  }
}

var lst: List = List(4, List(3,List(4,List(5,nil))))
def haha(): Unit =
   print("haha")

print(lst.add(4))
haha()
```