[TKOM] Projekt Hexgrider
Dokumentacja projektowa

Opis użytkowy

Koncepcja języka

Język służy do wykonania operacji na siatkach heksagonalnych. Typowanie jest silne i dynamiczne. Występuje 5 typów zmiennych: *int*, *float*, *string*, *array* i *hexgrid*. *Array* jest wektorem, który może jednocześnie przechowywać wartości dowolnych typów. Po inicjalizacji *Array* nie można zmieniać, dodawać lub usuwać z niego wartości. *Hexgrid* odpowiada siatce heksagonalnej, z sześciennym układem współrzędnych - położenie na siatce identyfikuje się trzema liczbami całkowitymi q, r, s przy czym suma tych liczb musi się równać 0 (q+r+s=0). Do siatki można dodać wartość na wybraną pozycję, zdjąć wartość z wybranej pozycji lub przełożyć wartość z pozycji siatki do zmiennej lub do innej pozycji na innej lub tej samej siatce. Na siatce można wykonywać operacji wyszukujące. Można odnaleźć wartość na danej pozycji klatki, odnaleźć pozycje klatek o podanej wartości lub znaleźć pozycje istniejących sąsiadów danej klatki. Dostępna jest pętla *foreach* za pomocą której można iterować po wartościach wektorów i pozycjach siatek. Iterowana wartość zapisywana jest do zmiennej zadeklarowanej w wyrażeniu *foreach*, do nowego zakresu zmiennych. Jeśli wartość iterowana jest innego typu niż zmienna zadeklarowana w wyrażeniu *foreach*, to wartość iterowana jest pomijana. Dostępne jest wyrażenie if elif else. Wartości typu PRAWDA/FAŁSZ reprezentowane są jako liczby całkowite, FAŁSZ = 0, PRAWDA=reszta. Możliwe jest zdefiniowanie funkcji, typ wartości wyjściowej i parametrów. Wartością wyjściową musi być jeden z pięciu typów zmiennych, ale funkcja może nic nie zwrócić, wtedy potencjalne przepisanie wartości wyjściowej wywołanie funkcji zostanie pominięte. Funkcja nie może być zdefiniowana w środku innej funkcji. Wszystkie funkcję są wczytywane przed interpretowaniem treści programu, a więc możliwe jest użycie funkcji przed jej zdefiniowaniem. Na wyjście standardowe jest wypisuje się wartość wskazana przez instrukcję return poza jakąkolwiek funkcją. Poza funkcją instrukcja return może być użyta dowolną ilość razy.

Opis semantyki

Zdefiniowanie zmiennej

```
int a;
float b;
string c;
array d;
hexgrid e;
```

Przypisanie wartości do zmiennej

```
a = 2;
b = 2.;
c = "2";
d = [2];
```

```
e = <"cell name" at [1, 0, -1]>;
```

Inicjalizacja zmiennej

```
int a = 100;
float b = 30.490;
string c = "Hello";
array d = ["1", 2, 3.4, [5, 6.7]];
hexgrid e = <"cell name" at [0, 0, 0], 100 at [4, 5, -9]>;
```

Dodawanie klatki do siatki heksagonalnej

```
hexgrid a = <>;
add "cell name" to a at [-3, -4, 7];
```

Usuwanie klatki z siatki heksagonalnej

```
hexgrid a = <"cell name" at [0, 0, 0], 100 at [4, 5, -9]>;
remove [0, 0, 0] from a;
```

Przemieszanie klatki siatki heksagonalnej

```
hexgrid a = <"cell name" at [0, 0, 0], 100 at [4, 5, -9]>;
string b;
move [0, 0, 0] from a to b;
hexgrid c;
move [4, 5, -9] from a to c at [0, 0, 0];
```

Definicja i wywołanie funkcji

```
func int double(int a){ return a*2;}
int b = double(3);
```

Instrukcja foreach

```
hexgrid h = <2 at [0, 0, 0], "dsa" at [0, -2, 2], 4 at [4, -3, -1]>;
int x = 0;
foreach array y in h { x = x + 1; }
```

Instrukcja if elif else

```
int y = 10;
if (y > 4) {y = y - 3;}
elif(y >2) { y = y +4;}
```

```
else {y = 10;}
```

Operacje logiczne

```
int a = 1 and 0;
int b = 1 or 0;
int c = !0;
```

Operacje porównania

```
int a = 100 > 10;
int b = -4 < 1;
inr c = 6 >= 5;
int d = 7 <= 9;
int e = 8 == 9-1;
int f = 8 != 16 /2;
```

Operacje arytmetyczne

```
int a = 2 + 3;
float b = 4.5 - 3;
int c = 2 * 3;
float d = 6/4;
int e = 5%2;
int f = -10;
```

Operacje na siatce heksagonalnej

Wyszukiwanie wartości na pozycji.

```
hexgrid a = <4 at [0, 0, 0]>;
int b = a on [0, 0, 0];
```

Wyszukiwanie pozycji z wartością

```
hexgrid a = <4 at [0, 0, 0], 5 at [1, 0, -1], 4 at [5, -2, -3]>;
array b = a by 4;
```

Wyszukiwanie pozycji istniejących sąsiadów

```
hexgrid a = <4 at [0, 0, 0], 5 at [1, 0, -1], 4 at [5, -2, -3]>;
array b = a beside [0, 0, 0];
```

Operacje indeksowania

```
array a = [1, 2, 3, 4];
int b = a[0];
```

Przykładowe kody

```
hexgrid example_hexgrid = <"blue"   at [0, 0, 0],
                           "red"    at [0, -1, 1],
                           "blue"   at [1, 0, -1],
                           "red"    at [-1, 1, 0],
                           "yellow" at [1, -1, 0],
                           "red"    at [2, 0,-2],
                           "blue"   at [-1, 0, 1]>;
int blue_count = 0;
foreach array pos in example_hexgrid beside [0, 0, 0]
{
    if (example_hexgrid on pos == "blue")
    {
        blue_count = blue_count + 1;
    }
}
int out = blue_count;
```

```
hexgrid example_hexgrid = <"blue"   at [0, 0, 0],
                           "red"    at [0, -1, 1],
                           "blue"   at [1, 0, -1],
                           "red"    at [-1, 1, 0],
                           "blue" at [1, -1, 0],
                           "red"    at [2, 0,-2],
                           "blue"   at [-1, 0, 1]>;
foreach array pos in example_hexgrid by "blue"
{
    if (pos[0] > 0) {remove pos from example_hexgrid;}
}
hexgrid out = example_hexgrid;
```

Sposób korzystania z interpretera

1. Zbudować projekt wpisując “scons” w katalogu projektu. Pojawi się plik wykonywalny “hexgrider”
2. Uruchomić interpreter podając kod programu na wejście standardowe. Np.: “hexgrider < plik_z_kodem_programu”

Finalna wersja gramatyki

```
Program = {Statement | FunctionDefinition};
Statement = StatementsWithOutSemicolon | StatementsWithSemicolon;
FunctionDefinition = "func", Declaration, "(", ParamList, ")",
StatementBlock;
StatementsWithOutSemicolon = IfStatement | ForeachStatement;
StatementsWithSemicolon = ( FunctionCallOrAssignment
                          | DeclarationOrInitialization
                          | ReturnStatement
                          | AddStatement
                          | RemoveStatement
                          | MoveStatement
                          ), ";";
Declaration = VariableType, Identifier;
ParamList = Declaration, {",", Declaration};
StatementBlock = "{", {Statement}, "}";
IfStatement = IfBlock, {ElifBlock}, [ElseBlock];
ForeachStatement = "foreach", Declaration, "in", Expression,
StatementBlock;
FunctionCallOrAssignment = Identifier, (Assignment | FunctionCall);
DeclarationOrInitialization = Declaration, ("=", Expression);
ReturnStatement = "return", Expression;
AddStatement = "add", Expression, MoveTarget, "at", Expression;
RemoveStatement = "remove", Expression, MoveSource;
MoveStatement = "move", Expression, MoveSource, MoveTarget, ["at",
Expression];
IfBlock = "if", ConditionBlock;
ElifBlock = "elif", ConditionBlock;
ElseBlock = "else", StatementBlock;
Assignment = "=", Expression;
FunctionCall = "(", ElementList, ")";
ConditionBlock = Condition, StatementBlock;
Condition = "(", Expression, ")";
ElementList = Expression, {",", Expression};
Expression = OrExpression;
OrExpression = AndExpression, {"or", AndExpression};
AndExpression = ComparisonExpression, {"and", ComparisonExpression};
ComparisonExpression = HexgridExpression, {ComparisonOperator,
HexgridExpression};
HexgridExpression = AddSubExpression, {("on"|"by"|"beside"),
AddSubExpression};
AddSubExpression = MulModDivExpression, {("+"|"-"),
MulModDivExpression};
MulModDivExpression = ArithmeticNegation, {("*"|"%"|"/"),
ArithmeticNegation};
```

```
ArithmeticNegation = ["-"], LogicalNegation;
LogicalNegation = ["!"], IndexingExpression;
IndexingExpression = Term, {"[", Expression, "]"};
Term  = IntegerLiteral
      | DecimalLiteral
      | TextLiteral
      | VariableReferenceOrFunctionCall
      | Array
      | Hexgrid
      | SubExpression
      ;
VariableReferenceOrFunctionCall = Identifier, [FunctionCall];
Array = "[", ElementList, "]";
Hexgrid = "<", HexgridCellList, ">";
HexgridCellList = HexgridCell, {",", HexgridCell};
HexgridCell = Expression, "at", Term;
SubExpression = "(", Expression, ")";
MoveSource = "from", Identifier;
MoveTarget = "to", Identifier;
VariableType = ("int" | "float" | "string" | "array" | "hexgrid");
ComparisonOperator = (">"|"<"|">="|"<="|"=="|"!=");
```

Opis implementacji

Projekt składa się z trzech części: Lexer, Parser i Interpreter.

Lexer

Lexer na zawołanie parsera przechodzi po znakach tekstu wejściowego i składa z nich token, który przekazuje do parsera.

Parser

Parser dostaje tokeny od lexera i na ich podstawie buduje drzewo rozbioru składniowego.

Interpreter

Interpreter ma postać visitora, który odwiedza górny węzeł drzewa rozbioru składniowego, będącego węzłem programu. Interpreter wczytuje wszystkie zdefiniowane w programie funkcje i następnie odwiedza po kolei każdą instrukcję w programie.

Opis testowania

Testy przeprowadzone zostały za pomocą biblioteki Boost Test. Testy stworzone są w trzech grupach: testy lexera, testy parsera i testy interpretera. Testy lexera są zupełnie niezależne

od innych komponentów, sprawdzają wyłącznie działanie lexera: poprawne budowanie tokenów, wywołanie wyjątków przy napotkaniu nie odpowiednich danych wejściowych oraz zapisywaniu położenia początku i końca tokenu w tekście wejściowym. Testy parsera zakładają poprawne działanie lexera, a testy interpretera zakładają poprawne działanie lexera i parsera. Testy są uruchamiane przy budowaniu projektu.