Wydział Elektroniki
i Technik Informacyjnych

POLITECHNIKA WARSZAWSKA

Dokumentacja końcowa projektu z przedmiotu Techniki Kompilacji

Autor:
Magdalena Majkowska

Opiekun:
mgr inż. Witold Wysota

7 czerwca 2021

1 Opis funkcjonalności i przykłady kodu

1.1 Ogólny opis języka

Projektowany przeze mnie język programowania można nazwać roboczo "M++", chciałabym, żeby był on podobny w swoich założeniach do C++ (ponieważ jest to jeden z moich ulubionych języków programowania), ale rozszerzony o nowy podstawowy typ danych - macierze. Język ma wbudowane wszystkie podstawowe operacje macierzowe tzn. dodawanie/odejmowanie macierzy, mnożenie macierzy przez macierz, mnożenie macierzy przez skalar, wyznaczanie wyznacznika, wyznaczanie macierzy odwrotnej, transponowanie macierzy, liczenie rzędu
macierzy.
Ze zmian, które wprowadziłam :

- brak klamerek do wyznaczania logicznych zakresów w programie - uważam, że w językach takich jak np. Python, kod wygląda dużo czyściej i schludniej oraz dodatkowo wymuszamy na programiście uporządkowaną indentację
- brak średnika jako zakończenie linii - z podobnego powodu co wyżej, kod wygląda schludniej
- Zmienna zawsze przy utworzeniu musi zostać zainicjalizowana - dzięki temu jesteśmy w stanie uniknąć wielu błędów związanych z nieuwagą programisty
- Operatory logiczne w postaci : and zamiast &&, or zamiast ||, not zamiast !

1.2 Przykłady obrazujące konstrukcje językowe oraz semantykę

```
integer age = 45
//dwa sposoby inicjalizacji macierzy
matrix[2][3] matrix1 = [1,2,3,4,5,6]
matrix[2][3] matrix2 = [1,2,3,4,5,6]
matrix[2][3] matrix3 = matrix1 * matrix2 //mnożenie macierzy
matrix[2][3] matrix4 = matrix3 * age // mnożenie macierzy przez
    ↪ skalar

det matrix4 // wyznacznik macierzy

////////////////////////////////////////////////

text name = 'Ala' // operacje na stringach
text surname = 'Kowalska'
text fullName = name + ' ␣' + surname

////////////////////////////////////////////////

integer number = 100;
void function count(integer number):
    print(number)
```

```
count(2) // powinno wypisać '2'
print(number) // powinno wypisać 100

////////////////////////////////////////////////

integer price1 = 20
double price2 = 0.5

double price3 = price1 + price2 // konwersja price1 na double
print(price3) // wypisuje 20.5

////////////////////////////////////////////////

integer age = 60
double price = 0.0
condition:
    case(age < 10):
        print('Child')
    case(age >= 10 and age < 20):
        print('Teenager')
    case(age >= 20 and age < 60):
        print('Adult')
    case(age >= 60):
        print('Senior')
    default: // tutaj wpadamy gdy nic nie pasuje
        print('Error')

if(age < 10 or age > 10):
    price = 25.50
otherwise:
    price = 30.0

////////////////////////////////////////////////

// wypisze 10 razy 'Hello'
loop(1:10:1):
    print('Hello') // nie ma dostępu do licznika pętli

integer counter = 10

// wypisze 1, 2, 3 .. 10
loop(1:counter:1):
    print(counter) // możemy wypisać licznik pętli

// wypisze 1, 6
// ostatni parametr loop to krok z jakim ma iść pętla
loop(1:counter:5):
    print(counter)

// wypisze 1, 2, 3 .. 10
```

```
// parametr kroku jest domyślnie 1, dlatego nie zawsze musimy go
   ↪ podawać
loop(1:counter):
    print(counter)


//wypisze 10, 9, 8 ... 1
asLongAs(counter > 0):
    print(counter)
    counter -= 1

//nieskończona pętla
asLongAs(1):
    print(counter)

////////////////////////////////////////////////
```

2 Poprawiona formalna specyfikacja i składnia

```
        program ::= {statement}

      statement :: = if statement | cond statement | loop statement | aslas statement
                | fun statement | fun call | statement | expression | comment;

   if statement ::= 'if', parentheses expr , ':', indent ,{statement} [other statement];

 other statement ::= 'otherwise', ':', indent, {statement};

  cond statement ::= 'condition', ':', indent, {case statement},[default statement];

  case statement ::= 'case', parentheses expr, ':', indent, {statement};

default statement ::= 'default', ':', indent, { statement };

  loop statement ::= 'loop', '(', expression, ':', expression, [':', expression]')',
                ':', indent, {statement};

 aslas statement ::= 'asLongAs', parentheses expr, ':', indent, {statement};

   fun statement ::= type, 'function', identifier, '(',arguments,')',':', indent, {statement},[
                statement];

      arguments ::= type, identifier {',', type, identifier};

       fun call ::= identifier, '(', [fun argument, [{',' fun argument}]], ')';

   fun argument ::= expression

 return statement ::= 'return', expression;

 parentheses expr ::= '(' expression ')' ;

     expression ::= test expression | assign expression
```

```
assign expression ::= identifier, '=' , expression;
  test expression ::= sum | sum, ('<' | '>' | 'not'), sum, [{('and' | 'or')] test expression}];
              sum ::= term | sum, {('-'|'+') sum};
             term ::= factor, {('*'| '/'), factor};
           factor ::= base { ['^',exponent] }
             base ::= '(' sum ')' | identifier | number
         exponent ::= '(' sum ')' | identifier | number;
            value ::= string | number | matrix value ;
             type ::= 'text' | 'integer' | 'matrix' | 'double';
           matrix ::= 'matrix',{'[', expression ,']'}, identifier, '=', matrix value;
     matrix value ::= {expression, [',',expression]};
         new type ::= alphanumeric char, {alphanumeric char | underscore};
          comment ::= comment delimiter, visible char, newline;
comment delimiter ::= '\\' | "#";
           string ::= string delimiter, {visible char}, string delimiter;
 string delimiter ::= ''| '"';
       identifier ::= alphanumeric char, {alphanumeric char | digit | underscore};
           number ::= integer,[ '.', digit, {digit}], [('E'|'e'), ['+' | '-'], {digit}];
          integer ::= zero | non zero integer;
 non zero integer ::= non zero digit,{digit};
            digit ::= zero | non zero digit;
     visible char ::= /* any member of the source character set or visible character*/;
alphanumeric char ::= /* any alphanumeric char */;
       underscore ::= '_';
          newline ::= /* newline sign */;
           indent ::= newline, {space};
            space ::= ' ';
   non zero digit ::= '1' ... '9';
             zero ::= '0';
```

3 Analiza wymagań funkcjonalnych i niefunkcjonalnych

3.1 Wymagania funkcjonalne

- Obsługa podstawowych typów danych liczbowych
- Obsługa operacji na stringach - konkatenacja
- Obsługa komentarzy
- Obsługa tworzenia zmiennych, przypisywania wartości do nich i odczytywania:
  - typowanie silne
  - typowanie statyczne
  - zmienne mutowalne
- Dwie operacja warunkowe - if... otherwise ... oraz condition z case'ami
- Dwie operacje pętli: loop oraz asLongAs - odpowiedniki pętli for i while
- Obsługa definiowania funkcji przez użytkownika wraz z obsługą zmiennych lokalnych
- Możliwość rekursywnego wywoływania funkcji
- Obsługa błędów poprzez mechanizm wyjątków - patrz niżej

3.2 Wymagania niefunkcjonalne

- Błąd w języku nie będzie powodował wyłączenia się programu - odpowiednia obsługa wyjątków
- Możliwość przetwarzania danych pobranych z gniazd, z pliku lub ze stringa - odpowiednia abstrakcyjność źródła, tzn w zależności od podanego typu źródła znaków program będzie miał oddzielną jego obsługę.

3.3 Obsługa błędów

Obsługa błędów jest zrealizowana za pomocą wyjątków - w pliku helpers/exception.hpp zdefiniowane są różne klasy wyjątków, które są rzucane w trakcie działania programu, gdy napotkany jest odpowiedni błąd/problem. Każdy z analizatorów tzn. leksykalny i semantyczny ma swoje oddzielne klasy. W obecnym stanie nie jest realizowana komunikacja pomiędzy modułami za ich pomocą i wszystkie błędy leksykalnei składniowe kończą się obecnie tak samo - wypisaniem na konsolę błędu z odpowiednim komentarzem co się wydarzyło i miejscem, w którym to się stało.

```
class Exception : public std::runtime_error {
 public:
  Exception(std::string m) : std::runtime_error(m) {}
};
```

```
class SocketProblemException : public Exception {
 public:
  SocketProblemException(std::string m) : Exception(m) {}
};
class WrongIndentException : public Exception {
 public:
  WrongIndentException(std::string m) : Exception(m) {}
};
class WrongKeywordException : public Exception {
 public:
  WrongKeywordException(std::string m) : Exception(m) {}
};
class WrongFlagsException : public Exception {
 public:
  WrongFlagsException(std::string m) : Exception(m) {}
  ...
};
```

4 Sposób uruchomienia

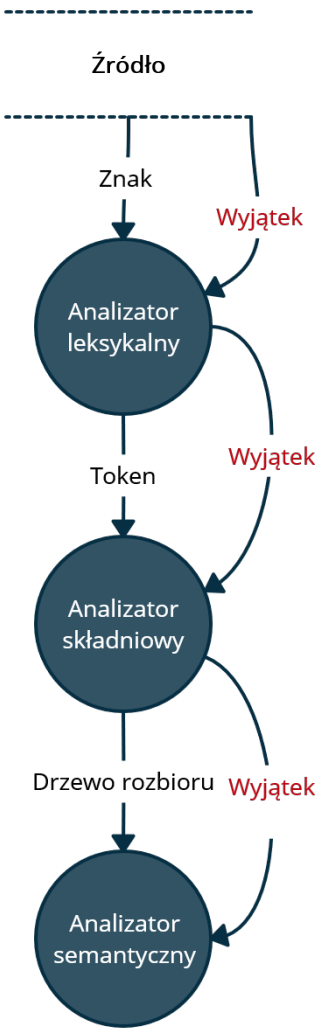
Interpreter nie posiada GUI .Wywołuje się go z konsoli poleceniem: `$/ m++ [flaga] [plikwejściowy.mpp | string | socket ]`

- m++ - nazwa programu po skompilowaniu
- flaga - –file/-f plik tekstowy, –socket/–sc socket TCP, –string/–s czytanie ze stringa
- plikwejściowy.mpp - źródło, z którego będzie można odczytać kod do parsowania
- string - string do przeparsowania

5 Opis sposobów realizacji

5.1 Opis komponentów i ich interfejsów

- Program - przestrzeń nazw udostępniająca interfejs do rozpoczęcia toku przetwarzania i zakończenia go.
- Źródło - *pojemnik* na tekst do analizy. Jego jedynym zadaniem jest podawanie do analizatora leksykalnego kolejnych znaków.
  Interfejs zawiera metodę getChar(), która zwraca strukturę NextCharacter.
- Analizator leksykalny - jego zadaniem jest budowanie tokenów na podstawie znaków ze *Źródła*, jest odpowiedzialny również za np. budowanie liczb czyli zamianę stringu '1000' na jego wartość liczbową.
  Interfejs zawiera publiczną metodę getToken(), która zwraca obiekt klasy Token odpowiadającą prawidłowo rozpoznanemu tokenowi.



Rysunek 1: Data flow diagram: główna pętla programu

- Analizator składniowy - jest odpowiedzialny za budowanie drzewa składniowego
- Ewaluator - interpreter oraz analizator semantyczny. Przed ewaluowaniem każdego wyrażenia najpierw sprawdza jego poprawnośc a następnie je wykonuje. Jest oparty na modelu wizytatora, który wizytuje kolejne obiekty klasy Node w programie.
  Interfejs prawie wszystkie jego metody są publiczne i nazywają się evaluate z argumentem wskaźnika na odpowiedni Node. Wewnątrz zawiera również ScopeStack, czyli stos przechowujący nazwy zmiennych lokalnych na danym poziomie, VariableMap czyli mapa zmiennych lokalnych wraz z ewaluowanymi wartościami i FunctionMap czyli mapa funkcji.

5.2 Rozpoznawane tokeny

- Typy danych:
  - MatrixToken
  - IntegerToken
  - TextToken
  - DoubleToken
- Literały
  - StringLiteralToken
  - IntegerLiteralToken
  - DoubleLiteralToken
  - MatrixLiteralToken
- Operatory:
  - AdditiveOperatorToken
  - MultiplicativeOperatorToken
  - MatrixOperationToken
  - RelationalOperatorToken
  - LogicalOperatorToken
  - AssignmentOperatorToken
  - ExponentiationOperatorToken
  - DetToken
  - TransToken
  - InvToken
  - AndToken
  - OrToken
  - NotToken
- Struktury językowe:

  - IfToken
  - OtherwiseToken
  - LoopToken
  - AsLongAsToken
  - FunctionToken
  - ConditionToken
  - CaseToken
  - DefaultToken
- Inne:
  - IdentifierToken
  - OpenRoundBracketToken
  - CloseRoundBracketToken
  - OpenSquareBracketToken
  - CloseSquareBracketToken
  - ColonToken
  - OpenBlockToken
  - CloseBlockToken
  - CommaToken
  - CommentToken
  - PointToken
  - RootToken
  - EndOfFileToken
  - NextLineToken
  - ReturnToken
  - PrintToken

5.3 Wykorzystywane struktury danych

```
// struktutra pozwalająca na dokładne określenie położenia w
   ↪ parsowanym tekście przekazywana ze źródła do analizatora
   ↪ leksykalnego

struct NextCharacter {
  NextCharacter() = default;
  NextCharacter(char letter, uint64_t aPos, uint64_t cPos, uint64_t
     ↪ lPos)
      : nextLetter(letter),
        absolutePosition(aPos),
        characterPosition(cPos),
        linePosition(lPos) {}
```

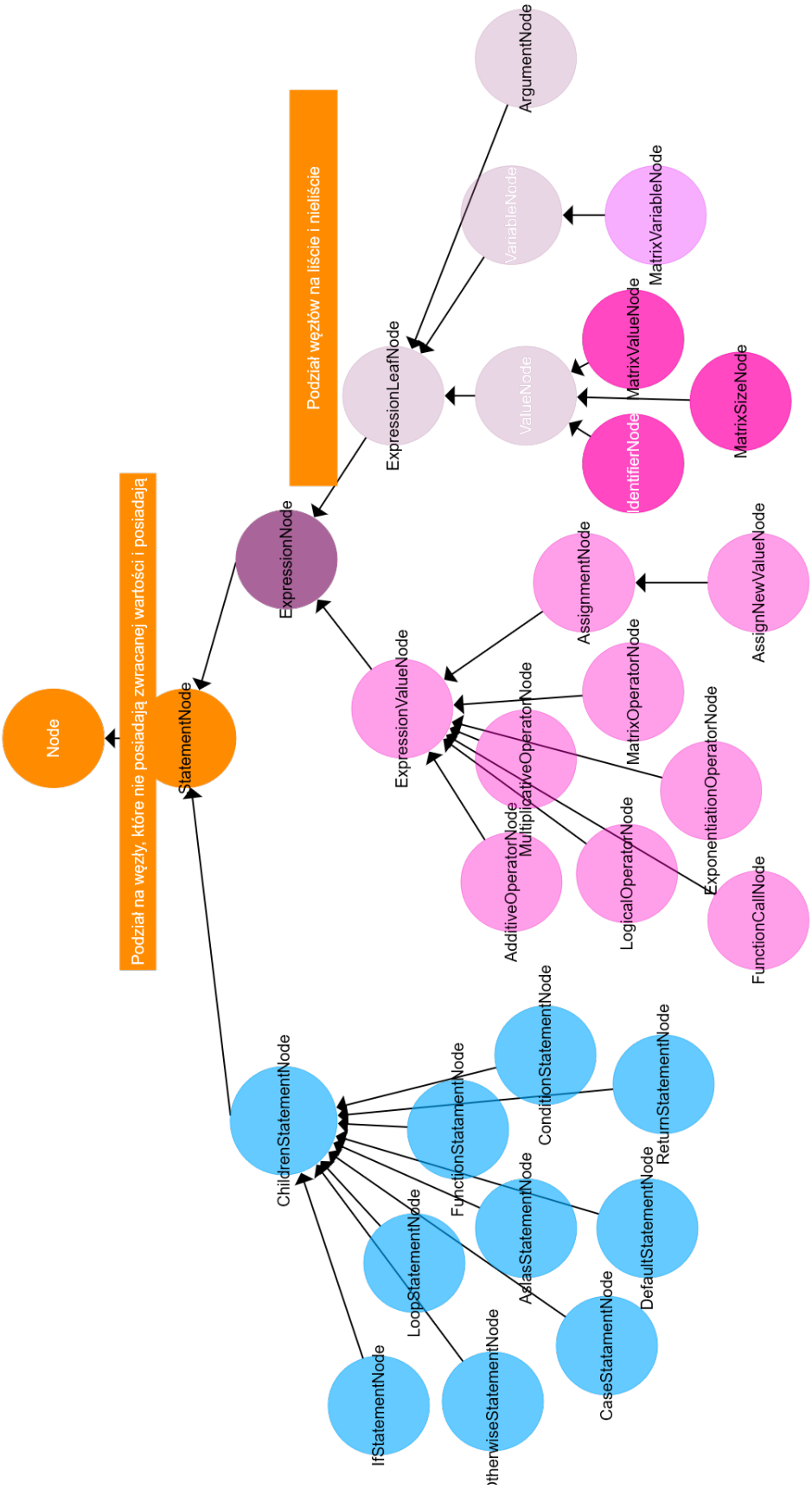
```
  std::string getLinePosition() const {
    return std::to_string(characterPosition) + ":" +
           std::to_string(linePosition);
  }

  char nextLetter;
  uint64_t absolutePosition;
  uint64_t characterPosition;
  uint64_t linePosition;
};

// klasa pozwalająca na zbudowanie drzewa składniowego. Będzie
   ↪ przekazywana z analizatora leksykalnego do semantycznego.
   ↪ Jest tylko jeden rodzaj tokenu - w tym przypadku postanowiłam
   ↪  nie budować drzewa hierarchi i polimorfizm (niepoprawnie)
   ↪ jest oparty na rodzaju i podrodzaju tokenu

// wartość tokena jest przechowywana w wariancie:
using TokenVariant =
    std::variant<std::monostate, int64_t, double, std::string, Matrix
       ↪ >;

class Token {
 public:
  enum class TokenType {
    MatrixToken,
    MatrixLiteralToken,
    IntegerToken,
    TextToken,
    DoubleToken,
    AdditiveOperatorToken,
    MultiplicativeOperatorToken,
    ExponentiationOperatorToken,
    LogicalOperatorToken,
    AssignmentOperatorToken,
    IfToken,
    OtherwiseToken,
    LoopToken,
    AsLongAsToken,
    FunctionToken,
    ConditionToken,
    CaseToken,
    IdentifierToken,
    OpenRoundBracketToken,
    CloseRoundBracketToken,
    OpenSquareBracketToken,
    CloseSquareBracketToken,
    ColonToken,
    OpenBlockToken,
    CloseBlockToken,
```

```
  CommaToken,
  PointToken,
  VoidToken,
  DefaultToken,
  CommentToken,
  RootToken,
  EndOfFileToken,
  UnindentifiedToken,
  NextLineToken,
  AndToken,
  OrToken,
  NotToken,
  StringLiteralToken,
  IntegerLiteralToken,
  DoubleLiteralToken,
  DetToken,
  TransToken,
  InvToken,
  ReturnToken,
};

enum class TokenSubtype {
  PlusToken,
  MinusToken,
  GreaterOrEqualToken,
  LessOrEqualToken,
  LessToken,
  GreaterToken,
  EqualToken,
  NotEqualToken,
  DivisionToken,
  MultiplicationToken,
};

Token(TokenType type, TokenVariant value, const NextCharacter &
    ↪ firstCharacter)
    : type(type),
      value(value),
      characterPosition(firstCharacter.characterPosition),
      absolutePosition(firstCharacter.absolutePosition),
      linePosition(firstCharacter.linePosition) {}
Token(TokenType type, TokenSubtype subtype, TokenVariant value,
      const NextCharacter &firstCharacter)
    : type(type),
      subtype(subtype),
      value(value),
      characterPosition(firstCharacter.characterPosition),
      absolutePosition(firstCharacter.absolutePosition),
      linePosition(firstCharacter.linePosition) {}
Token(TokenType type) : type(type) {}
```

```
  Token(TokenType type, TokenVariant value) : type(type), value(value
    ↪ ) {}
  TokenType getType() const { return type; }
  TokenSubtype getSubtype() const { return subtype; }

  TokenVariant getValue() const { return value; }
  int64_t getInt() const { return std::get<int64_t>(value); }
  double getDouble() const { return std::get<double>(value); }
  std::string getString() const { return std::get<std::string>(value)
    ↪ ; }

  uint64_t getCharacterPosition() const { return characterPosition; }
  uint64_t getAbsolutePosition() const { return absolutePosition; }
  uint64_t getLinePosition() const { return linePosition; }
  std::string getLinePositionString() const {
    return std::to_string(linePosition + 1) + ":" +
           std::to_string(characterPosition + 1);
  }

 private:
  TokenType type;
  TokenSubtype subtype;
  TokenVariant value;
  uint64_t characterPosition = 0;
  uint64_t absolutePosition = 0;
  uint64_t linePosition = 0;

//Zdefiniowane są również operatory dla Tokenów, aby w wygodniejszy
    ↪  sposób na naich operować
  friend bool operator==(Token const &lhs, Token const &rhs) {
    return lhs.type == rhs.type && lhs.value == rhs.value;
  };

  friend bool operator!=(Token const &lhs, Token const &rhs) {
    return !operator==(rhs, lhs);
  };

  friend bool operator==(Token const &lhs, Token::TokenType const &
    ↪ rhs) {
    return lhs.type == rhs;
  };

  friend bool operator==(Token::TokenType const &lhs, Token const &
    ↪ rhs) {
    return lhs == rhs.type;
  };
};

//klasa bazowa Node, po której dziedziczą wszystkie węzły drzewa sk
    ↪ ładniowego. Na niej pracuje parser.
```

```
class Node {
 public:
  Node() : token(Token(Token::TokenType::RootToken)){};
  explicit Node(Token token) : token(token){};
  Node& operator=(Node&& other) = default;
  Node(const Node& other) = default;
  Node(Node&& other) = default;
  Token getToken() const { return token; }

  virtual bool isLeaf() const {
    if (token == Token::TokenType::StringLiteralToken ||
        token == Token::TokenType::IntegerLiteralToken ||
        token == Token::TokenType::DoubleLiteralToken)
      return true;
    return false;
  }

  virtual void buildTreeStringStream(int64_t depth,
                                     std::stringstream& tree) const {
    std::string indent(depth, ' ');
    tree << indent << LexicalTable::token2StringTable.at(token.
        ↪ getType())
         << '\n';
  }
  virtual ~Node() = default;

  const std::string getPrintTree() const {
    std::stringstream ss;
    buildTreeStringStream(0, ss);
    return ss.str();
  }

 protected:
  Token token;
};
```

6 Drzewo hierarchii klasy Node



Rysunek 2: Drzewo hierarchii klasy Node - strzałki oznaczają kierunek dziedziczenia

Początkowo chciałam wyróżnić jedną klasę Node, podobnie jak w przypadku Tokenów. Niestety szybko okazało się to być niemożliwe, kiedy zauważyłam, że do interpretera będzę potrzebować klas różniących się od siebie metodami i składowymi. Po dyskusja z prowadzących zastosowałam podział bazowego Node na dwie podklasy węzłów:

- Te które posiadają zwracaną wartość - Expression
- Te które tej wartości nie posiadają - Statement

Następnie węzły klasy Expression dzielą się na:

- Te które są liściami w drzewie - zmienne, argumenty
- Te które mogą posiadać potomków - operatory, przypisania

Każdy z węzłów posiada wirtualną metodę Evaluate, której używa interpreter, potrafią one również wypisać siebie oraz swoje dzieci do stringstream'a, co bywa bardzo przydatne przy testowaniu. Ostatecznie podczas implementacji zostałam zmuszona jednak zastosować dla wszystkich węzłów tą samą metodę Evaluate ze zwracaną wartością, dlatego, że chciałam, aby StatementChildrenNode mogły przechowywać w sobie wskaźniki na obiekty typu Statement (wspólny przodek Expression) i miałam pewne problemy z funkcjami wirtualnymi. Obiekty klasy StatementChildrenNode i pochodne zwracają wartość pustą, zaś pochodne Expression wartość zdefiniowaną w wariancie Value.

7 Sposób testowania

Testowanie projektu zrealizowałam za pomocą testów jednostkowych, napisanych w bibliotece GoogleTest, zgodnie z wskazówkami prowadzącego.

- Analizator leksykalny
  - Dla każdego tokenu oddzielny test jednostkowy sprawdzający jego prawidłowe wyprodukowanie
  - Testy również dla przypadków podstawowych błędów takich jak:
    zła indentacja, niedomknięcie cudzysłowu, brak ':' na koniec jakiejś złożonej instrukcji, niedomknięcie nawiasu
- Analizator składniowy
  - Oddzielny test jednostkowy dla każdej z produkcji wraz z alternatywami
  - Testy jednostkowe również dla niepoprawnej sekwencji tokenów
  - Testy jednostkowe dla podstawowych błędów programistycznych
    takich jak np: pomylenie przypisania z porównaniem, pominięcie separatora argumentów wywołania funkcji, złe przypisanie wartości do zmiennej typu, złe użycie operatora porównania np. porównywanie integera
    ze stringiem itp.
  - Testy jednostkowe na prawidłowe zbudowanie drzewa rozbioru programu na podstawie podawanego stringa
- Interpreter

  - Testy potoku przetwarzania tzn. prawidłowe rozpoznanie tokenów, zbudowanie prawidłowego drzewa
  - Testowanie gotowego programu z plikiem jako wejście

8 Zmiany w stosunku do dokumentacji wstępnej

- Tam gdzie w porpzedniej gramatyce pojawiały się integer/non_zero_integer tzn. np. w pętli loop albo w wartościach macierzy teraz jest expression. Również widać tą zmianę w kodzie - wszędzie gdzie są warunki, wartości parsowane jest Expression.
- Zrezygnowałam z wartości domyślnych argumentów funkcji - głównie przez brak czasu na ich implementację pod koniec projektu
- Wartości prawda/fałsz są przedstawiane w logice języka jako liczby całkowite, podobnie jak w języku C. Wszystko poza 0 to prawda, 0 to fałsz.
- Zrezygnowałam z interaktywnego interpretera - z tego samego powodu, co z wartości domyślnych.
- Obsługa błędów spoczywa na użytkowniku tzn. jeśli zrobi on jakiś błąd semantyczny/leksykalny nie naprawiam go, lecz rzucony zostanie wyjątek i przerwany potok przetwarzania. W środku programu również zrezygnowałam z obsługi wyjątków jako mechanizmu komunikacji.
- Dodana funkcja wbudowana print(), która przyjmuje argumenty (integer, double, Matrix, string czyli wszystkie te, które naturalnie występują w języku, a także wyrażenia!) i wypisuje je na standardowe wyjście