



**AKADEMIA GÓRNICZO-HUTNICZA  
IM. STANISŁAWA STASZICA  
W KRAKOWIE**

Wydział Elektrotechniki, Automatyki, Informatyki i  
Inżynierii Biomedycznej

**sgm-lang**

Autorzy: Borowy Szymon, Kozak Marcin, Nieużyła Grzegorz  
Kierunek studiów: Informatyka

Kraków, 2020

<b>Wstęp</b>	<b>3</b>
<b>Użyte narzędzia</b>	<b>3</b>
<b>Składnia</b>	<b>3</b>
Typy danych	3
Operatory	3
Matematyczne	3
Logiczne	3
Relacje	3
Słowa kluczowe	4
Identyfikatory	4
Pozostałe znaki specjalne	4
<b>Budowa interpretera</b>	<b>5</b>
CLI	5
Tokenizer	5
Wstęp teoretyczny	5
Realizacja	6
Tokeny	7
Przykłady działania tokenizera	7
Parser	8
Wstęp teoretyczny	8
Realizacja	8
Przykład działania parsera	10
Generator bytrecodu	10
Wstęp teoretyczny	10
Realizacja	10
Przykład działania generatora	12
Interpreter	13
Wstęp teoretyczny	13
Realizacja	13
Przykłady działania	14
<b>Future Work</b>	<b>15</b>
<b>Podział pracy</b>	<b>16</b>

# Wstęp

Sgm-lang jest prostym językiem programowania i interpreterem tego języka napisanym w Pythonie. Składnia języka przypomina grupę języków "C-podobnych".

Sgm-lang powstał jako projekt zaliczeniowy na przedmiot Teoria Kompilacji i Kompilatory.

## Użyte narzędzia

Kod w całości został napisany w języku Python.

Jako system kontroli zastosowaliśmy narzędzie git, a repozytorium było hostowane na GitHubie ([link do repozytorium](#)).

## Składnia

### Typy danych

Sgm-lang jest językiem silnie typowanym. Wspierane typy danych to:

- Typ całkowity `int` (nazwa w `sgm-lang`: `mrINTERNational`)
- Typ zmiennopozycyjny `float` (`boatWhichFloat`)
- Łańcuchy znakowe `String` (`stringiBoi`)
- Typy logiczne `bool` (`bool`)

### Operatory

#### Matematyczne

- `=` → operator przypisania
- `+` → operator dodawania
- `-` → operator odejmowania
- `*` → operator mnożenia
- `/` → operator dzielenia
- `%` → operator modulo (reszta z dzielenia)

#### Logiczne

- `&&` → koniunkcja (logiczny iloczyn, AND)
- `||` → alternatywa (logiczna suma, OR)
- `!` → negacja

#### Relacje

- `==` → porównanie (relacja równości)
- `<=` → mniejsze bądź równe

- `>=` → większe bądź równe
- `<` → mniejsze
- `>` → większe

## Słowa kluczowe

- `youSpinMeRound` → odpowiednik słowa kluczowego `while` w np: C
- `showMeYourGoods` → odpowiednik funkcji `print` w np: Pythonie
- `doItIf` → odpowiednik słowa kluczowego `if` w np: C
- `True / False` → Logiczna prawda / Logiczny Fałsz

## Identyfikatory

Identyfikator nie może być żadnym ze słów kluczowych.

Ciąg znaków jest poprawnym identyfikatorem tylko jeżeli zawiera same znaki alfanumeryczne (a-z i 0-9) oraz podkreślenia (`_`). Identyfikator nie może zaczynać się od cyfry, ani zawierać jakichkolwiek spacji.

## Pozostałe znaki specjalne

- `#` → znak początku komentarza. Komentarz kończy się wraz z końcem linii (wraz z napotkaniem znaku `\n`)
- `;` → średnik. Wskazuje koniec polecenia
- `"` → podwójny cudzysłów, wskazuje początek i koniec łańcucha znaków.
- `(, )` → nawiasy (okrągłe) wskazują kolejność wykonywania operacji, w środku znajdują się też wyrażenia do instrukcji `PRINT`, `IF`, `WHILE`
- `{, }` → nawiasy klamrowe, wyznaczają blok instrukcji. Analogiczne działanie jak w C

```
mrINTernational x = 0;

youSpinMeRound(x < 10)
{
    # Todo: add comments
    showMeYourGoods(x);
    doItIf((x % 2) == 0)
    {
        showMeYourGoods(" is even");
    }
    showMeYourGoods("\n");
    x = x + 1;
}
```

Rys 1. Przykładowy kod w sgm-lang

# Budowa interpretera

## CLI

W pliku `sgm.py` znajduje się kod odpowiedzialny za wywoływanie naszego programu (interpretera `sgm`) z linii poleceń. Do stworzenia całego interfejsu i jego mechaniki wykorzystaliśmy bibliotekę `argparse`. Pozwoliła nam ona na dodanie parametrów wywołania naszego programu oraz na stworzenie czytelnego opisu tych parametrów. Dostępne parametry i ich opis znajduje się na zdjęciu poniżej.

```
C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>python sgm -h
usage: sgm [-h] [-b] [-t] [-a] [file]

SGM language interpreter

positional arguments:
  file

optional arguments:
  -h, --help            show this help message and exit
  -b, --bytecode        Do not execute, print out bytecode
  -t, --tokenizer       Do not execute, print out tokenizer output
  -a, --ast             Do not execute, print out ast tree

C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>
```

Rys 2. Opcje interpretera sgm

## Tokenizer

### Wstęp teoretyczny

**Tokenizer/Skaner/Lekser** - wyodrębnia podstawowe symbole języka (tokeny). Usuwa białe znaki i komentarze. Wykonuje proces analizy leksykalnej.

**Analiza leksykalna**- jest to proces przetwarzania sekwencji znaków (kodu) w sekwencję tokenów, czyli nacechowanych znaczeniowo leksemów.

**Leksem** -ciąg kolejnych znaków stanowiących semantycznie niepodzielną całość.

**Token**- stała (całkowita) reprezentująca wartość wczytanego leksemu

## Realizacja

Trzon tokenizera stanowi klasa `Tokenizer`, która przechowuje kod programu (wczytany z pliku) oraz wykonuje na nim operacje w celu uzyskania ciągu tokenów.

**Wejście:** kod programu w języku sgm-lang

**Wyjście:** lista tokenów

Główną metodą jest metoda `tokenize()`, w której następują po kolei:

1. usunięcie komentarzy `deleteComments()`
  - a. usuwamy fragmenty kodu od znaku komentarza `#` do znaku nowej linii `\n`
2. Podzielenie kodu na leksemy i przechowanie ciągu leksemów w tablicy

Podzielenie kodu na leksemy odbywa się w funkcji `insertSpacesAndSplit()`. Funkcja ta ma za zadanie rozdzielić zbitki słów spacjami, wtedy i tylko wtedy, kiedy nie zmieni to semantyki danego wyrażenia. Do tego celu wykorzystuje dwie listy `splittable` i `unsplittable`, które przechowują znaki które można i których nie można rozdzielić bez zmiany w semantyce kodu.

Funkcja też uwzględnia łańcuchy znakowe i nie wstawia spacji wewnątrz Stringów.

Poniżej przykład kodu przed i po zastosowaniu na nim funkcji.

<pre>doItIf(w==12){     x=2+3*(12.3);     stringiboi x = "sqrt(12)" }</pre>	<pre>doItIf ( w == 12 ) {     x = 2 + 3 * ( 12.3 ) ;     stringiboi x = "sqrt(12)" }</pre>
---	--

Rys 3. Po lewo kod przed przetworzeniem, po lewej przetworzony przez funkcję `insertSpacesAndSplit()`

Następnie kod jest dzielony przez pythonowską funkcję `split()`, która daje w wyniku listę stringów → czyli w naszym wypadku leksemów

3. Następnie każdy z leksemów jest sprawdzany. Tutaj myślę że najlepiej będzie przedstawić kod programu który jest całkiem czytelny w tym wypadku. Prosimy zwrócić uwagę na warunki konstrukcji `IF...ELIF...ELSE`. to one decydują jaki token zostanie stworzony. Tworzenie tokenów omówimy za chwilę.

```
while self.position < len(self.splitCode):  
    word = self.splitCode[self.position]  
  
    if word in self.keyWords: # jest słowem kluczowym (IF, WHILE ...)  
        self.tokensList.append((TokenType(word), None))  
    elif word in self.dataTypes: # jest typem danych (INT, FLOAT ...)  
        self.tokensList.append((CompoundToken.DATA_TYPE, DataType(word)))  
  
    elif word == "true" or word == "false":  
        self.tokensList.append((CompoundToken.BOOL, bool(word)))  
    elif self.isParsableToInt(word): # jest liczbą 'intową'
```

```

        self.tokensList.append((CompoundToken.INT, int(word)))
    elif self.isParsableToFloat(word):
        self.tokensList.append((CompoundToken.FLOAT, float(word)))
    elif "\"" in word: # jest stringiem
        self.tokensList.append((CompoundToken.STRING, self.parseNewLines(word[1:-1])))

    elif word.isidentifier(): # jest zmienna/identyfikatorem
        self.tokensList.append((CompoundToken.ID, word))
    else:
        raise TokenizerError("Something is wrong in Tokenizer: "+ word)
    self.position += 1

```

Rys 4. główny kod tokenizera

## Tokeny

Token jest parą. Jego budowa jest następująca:

(OKREŚNIENIE\_TOKENU, INFORMACJA)

gdzie:

OKREŚNIENIE\_TOKENU → enum CompoundToken, albo TokenType. CompoundToken jest używany, kiedy token reprezentuje wartość z informacją. Np:

- `bool` → (CompoundToken.DATA\_TYPE, DataType.BOOL)
- `3.123` → (CompoundToken.FLOAT, 3.123)

TokenType zawiera wartości które nie potrzebują dodatkowych informacji. Np: nawiasy, średniki itd.

INFORMACJA → zawiera dodatkowe informacje o tokenie. Kiedy pierwsza część tokenu to wartość z CompoundToken, to INFORMACJA jest dodatkową informacją o złożonym tokenie. W przypadku kiedy OKREŚNIENIE\_TOKENU to wartość z TokenType (czyli wartości które nie potrzebują dodatkowych informacji żeby zostać poprawnie przetworzone) zawiera wartość None.

## Przykłady działania tokenizera

```

helloworld.sgm x
1 showMeYourGoods("Hello World\n");

Terminal: Local x +
C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>python sgm -t examples\helloworld.sgm
[(PRINT, None),
 (L_PAREN, None),
 (<CompoundToken.STRING: 4>, 'Hello World\n'),
 (R_PAREN, None),
 (SEMICOLON, None)]

```

Rys 5. Wynik tokenizera na przykładzie helloworld.sgm

# Parser

## Wstęp teoretyczny

**Analizator składniowy** lub **parser** – program dokonujący analizy składniowej danych wejściowych w celu określenia ich struktury gramatycznej w związku z określoną gramatyką.

**Analiza składniowa** – proces analizy tekstu, w celu ustalenia jego struktury gramatycznej i zgodności z gramatyką języka.

**Drzewo składniowe, drzewo AST** – drzewo etykietowane, wynik przeprowadzenia analizy składniowej zdania (słowa) zgodnie z pewną gramatyką.

## Realizacja

Całość parsera znajduje się w pliku `Parser.py`. Parser przekształca otrzymany od tokenizera ciąg tokenów na AST

**Wejście:** lista tokenów

**Wyjście:** AST

Na początku zostały wydzielone pewne konstrukcje języka- pewne schematy których przestrzegają wyrażenia w naszym języku. Poniżej przykład takiej konstrukcji dla operatora binarnego:

```
class BinOp(AST):
    def __init__(self, left, op, right):
        self.left = left
        self.token = self.op = op
        self.right = right

    def __str__(self):
        return f'[{self.left.__str__()} <> {self.op} <> {self.right.__str__()}']'
```

Rys 6. Konstrukcja operatora binarnego

Każda taka klasa ma na celu lepsze rozróżnienie parsowanych wyrażeń.

Obiekty danej klasy są tworzone po napotkaniu tokenu wskazującego na daną konstrukcję.

Np: obiekt klasy `IF` zostanie utworzony wtedy kiedy parser natknie się na token odpowiadający wyrażeniu `IF`. Wtedy automatycznie kolejne tokeny są parsowane (zgodnie z gramatyką języka) jako wyrażenie warunkowe.

Schemat działania parsera wygląda następująco:

1. Sprawdzane jest do którego z wyrażeń należy aktualnie parsowany fragment tokenów. Odpowiada za to funkcja `statement()`. Jeżeli analizowany token jest



tokenem reprezentującym `PRINT`, to parser wie, że kolejne tokeny mają zostać przetworzone jako wyrażenie `PRINT`. Statement jest odpowiednikiem pojedynczej instrukcji języka

2. Następnie w funkcji `statement_list()` dodajemy do wynikowej listy wyrażenie które zostało sparsowane w `statement()` i kontynuujemy parsowanie wyrażień (funkcja `statement()`) dopóki lista leksemów nie jest pusta, lub do natrafienia na pierwszy błąd.
3. Uzyskane w ten sposób `statement list` jest następnie przetwarzane na `compound statement` → jest to odpowiednik bloku instrukcji.

Uzyskujemy w ten sposób jako wynik obiekt `Compound`, który zawiera listę swoich dzieci, z których każde zawiera kolejne dzieci będące kolejnymi elementami języka.

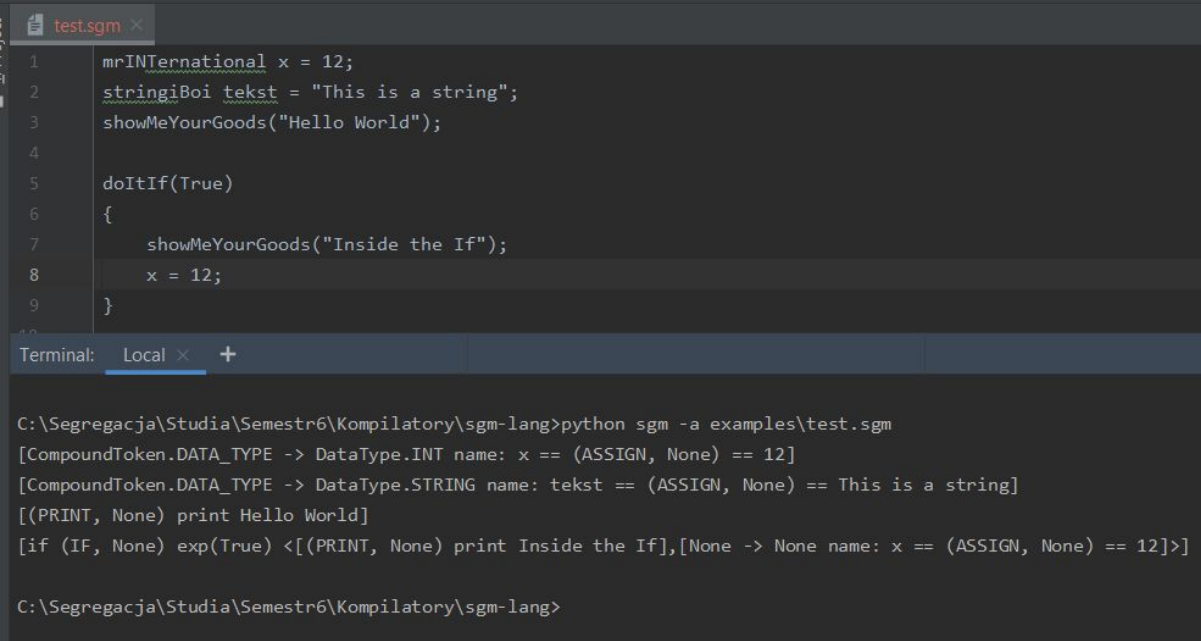
Sposób parsingu wyrażień jest złożony i zależny od konkretnego wyrażenia- inaczej parsujemy `IF`'a a inaczej parsujemy `WHILE`. Generalna składnia jest zgodna ze składnią znaną z języka C.

Poniżej przedstawiamy przykładowy parsing dla `WHILE`. Przeanalizowanie go może być pomocne w celu lepszego zrozumienia sposobu parsowania innych konstrukcji języka:

```
def while_statement(self):
    token = self.current_token
    self.eat(TokenType.WHILE) #pobierz token WHILE
    self.eat(TokenType.L_PAREN) # nawias otwierający warunek
    expression = self.expr() # sparsuj wyrażenie będące warunkiem zatrzymania pętli
    self.eat(TokenType.R_PAREN) # nawias zamykający warunek
    self.eat(TokenType.L_BRACE) # początek bloku instrukcji
    statements = self.compound_statement() # blok instrukcji - ciało petli while
    # stworzenie obiektu klasy While
    node = While(token, expression, statements)
    # nawias zamykający jest pobierany w funkcji statement_list
    return node
```

Rys 7. Parsowanie instrukcji WHILE

## Przykład działania parsera



```
test.sgm x
1  mrINternational x = 12;
2  stringiBoi tekst = "This is a string";
3  showMeYourGoods("Hello World");
4
5  doItIf(True)
6  {
7      showMeYourGoods("Inside the If");
8      x = 12;
9  }
10
Terminal: Local x +
C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>python sgm -a examples\test.sgm
[CompoundToken.DATA_TYPE -> DataType.INT name: x == (ASSIGN, None) == 12]
[CompoundToken.DATA_TYPE -> DataType.STRING name: tekst == (ASSIGN, None) == This is a string]
[(PRINT, None) print Hello World]
[if (IF, None) exp(True) <[(PRINT, None) print Inside the If],[None -> None name: x == (ASSIGN, None) == 12]>]
C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>
```

Rys 8. Przykład działania Parsera

Pierwsza linia wyniku to odpowiednik stworzenia zmiennej `x` i przypisanie jej wartości 12  
W ostatniej linii wyniku widać złożone wyrażenie- konstrukcję `IF`. dwie linie stanowiące jej ciało są zgrupowane jako de facto jej argument

## Generator bytrecodu

### Wstęp teoretyczny

**Generator** - dokonuje przekładu programu źródłowego w postaci wewnętrznej otrzymanej po analizie składniowej na kod wynikowy związany zazwyczaj z konkretną maszyną docelową. Wielu instrukcjom odpowiada pewna stała sekwencja generowanych rozkazów zwana wzorcem instrukcji.

### Realizacja

**Wejście:** AST

**Wyjście:** Lista operacji dla interpretera ( w Bytrecodzie)

Generator bytrecodu przechodzi po całym drzewie AST, tworząc bytecode dla odpowiednich konstrukcji. Faktyczny generator znajduje się w klasie `AstToBytecodeGenerator.py`  
Klasą pomocniczą przy tworzeniu kodu jest `BytecodeGenerator`, który jest odpowiedzialny za wytworzenie odpowiednich konstrukcji dla elementów języka. Poniżej przykład dla `IF`

```
def generateIf(instructions: List[Operation], condition: List[Operation]) ->
List[Operation]:
    """
    Generates if block.
    'instructions' will be executed if after executing the 'condition' instructions
    stack.pop() will be truthy
    """
    result = condition[:]
    result.append(Operation(Opcode.JMP_NOT_IF, [Parameter(ParameterType.RELATIVE,
len(instructions))]))
    result += instructions
    return result
```

Rys 9. funkcja tworząca bytecode dla konstrukcji IF

Opcode jest enumem z zdefiniowanymi kodami instrukcji. Parametr i Operation są klasami.

Poniżej Enum Opcode z wyjaśnieniem odpowiednik kodów instrukcji:

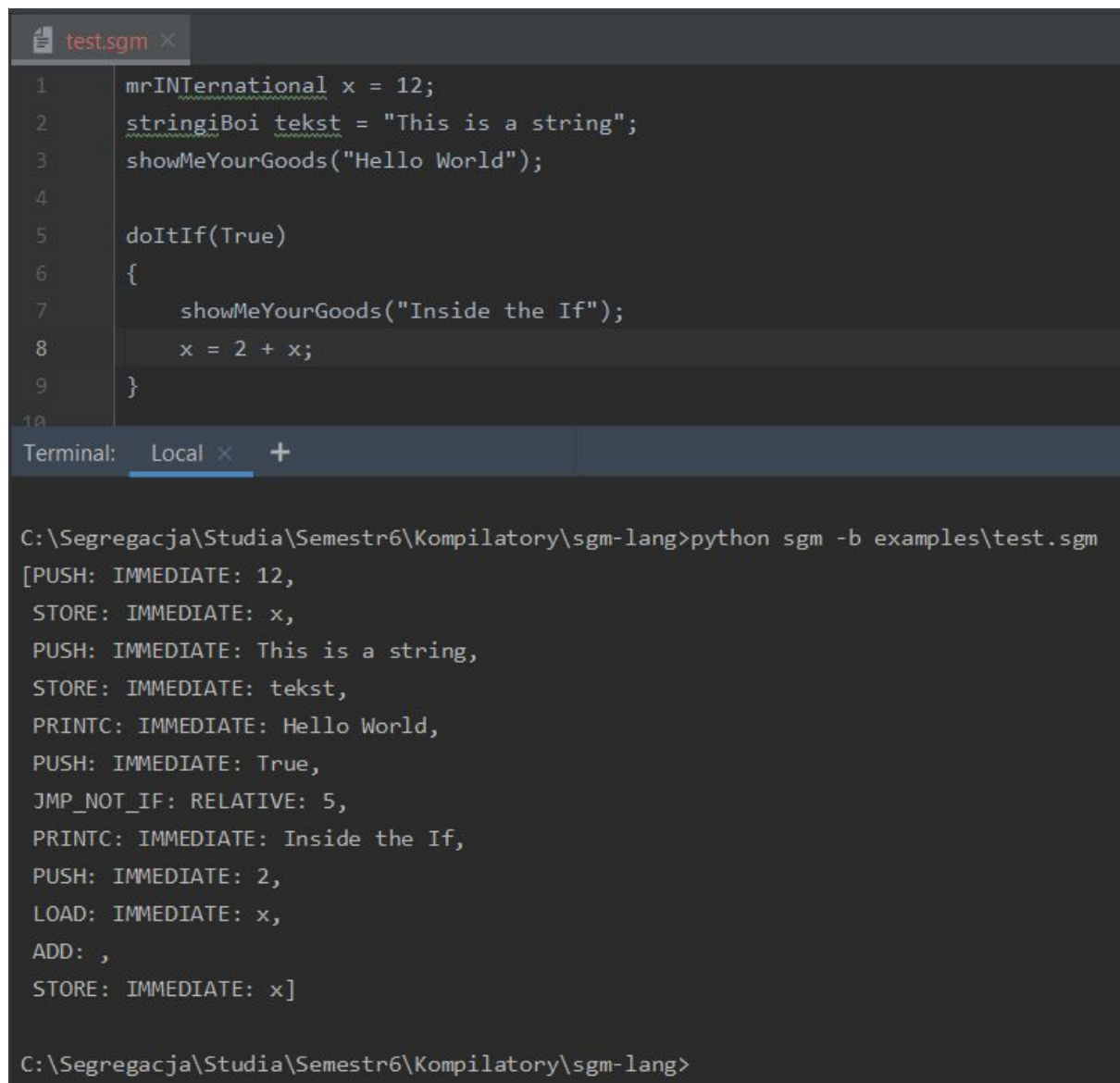
```
class Opcode(Enum):
    LOAD = auto() # push variable specified as parameter
    STORE = auto() # pops and sets as variable specified as parameter
    ADD = auto() # push stack.pop(1) + stack.pop(0)
    SUB = auto() # push stack.pop(1) - stack.pop(0)
    MUL = auto() # push stack.pop(1) * stack.pop(0)
    DIV = auto() # push stack.pop(1) / stack.pop(0)
    MOD = auto() # push stack.pop(1) % stack.pop(0)
    PRINT = auto() # prints stack[0]
    PRINTC = auto() # prints parameter
    JMP = auto() # jump
    JMP_IF = auto() # jump if stack.pop(0)
    JMP_NOT_IF = auto() # jump if not stack.pop(0)
    PUSH = auto() # push parameter
    POP = auto() # pops to variable in parameter
    EQ = auto() # pushes True if stack.pop(1) == stack.pop(0)
    NEQ = auto() # pushes True if stack.pop(1) != stack.pop(0)
    GE = auto() # pushes True if stack.pop(1) >= stack.pop(0)
    GRT = auto() # pushes True if stack.pop(1) > stack.pop(0)
    LE = auto() # pushes True if stack.pop(1) <= stack.pop(0)
    LESS = auto() # pushes True if stack.pop(1) < stack.pop(0)
    BINARY_OR = auto() # pushes stack.pop(1) || stack.pop(0)
    BINARY_AND = auto() # pushes stack.pop(1) && stack.pop(0)
    NOT = auto() # pushes !stack.pop(0)
```

Rys 10. Kody instrukcji i opisy instrukcji

Sama klasa `AstToByteCodeGenerator`, przechodzi po każdym elemencie z drzewa AST i na podstawie klasy węzła (która jest ustalana w Parserze) generuje odpowiedni ciąg instrukcji.

Przy każdej generacji sprawdzane są też odpowiednie warunki. Np jeżeli mamy do czynienia z generowaniem kodu dla przypisania (oryginalny kod był np: `x = 12`), to generator sprawdza czy zmienna `x` została już zadeklarowana. Jeżeli nie, wyrzucany jest wyjątek. Generator kontynuuje pracę dopóki nie przetworzy całego AST, lub do napotkania pierwszego błędu

## Przykład działania generatora



```
test.sgm x
1  mrINternational x = 12;
2  stringiBoi tekst = "This is a string";
3  showMeYourGoods("Hello World");
4
5  doItIf(True)
6  {
7      showMeYourGoods("Inside the If");
8      x = 2 + x;
9  }
10

Terminal: Local x +
C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>python sgm -b examples\test.sgm
[PUSH: IMMEDIATE: 12,
STORE: IMMEDIATE: x,
PUSH: IMMEDIATE: This is a string,
STORE: IMMEDIATE: tekst,
PRINTC: IMMEDIATE: Hello World,
PUSH: IMMEDIATE: True,
JMP_NOT_IF: RELATIVE: 5,
PRINTC: IMMEDIATE: Inside the If,
PUSH: IMMEDIATE: 2,
LOAD: IMMEDIATE: x,
ADD: ,
STORE: IMMEDIATE: x]
C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>
```

Rys 11. Przykład działania generatora

# Interpreter

## Wstęp teoretyczny

**Interpreter** – program komputerowy, który analizuje kod źródłowy programu, a przeanalizowane fragmenty wykonuje.

## Realizacja

**Wejście:** Lista operacji w Bytecodzie

**Wyjście:** brak - nic nie produkuje, wykonuje instrukcje

Częścią główną interpretera jest klasa `BytecodeInterpreter` z funkcją `run`, która to odpowiada za wykonanie wszystkich instrukcji które dostanie od Generатора Bytecodu. W metodzie `processInstruction` ( która jest wywoływana z metody `run`) sprawdzamy jaka jest następna operacja do wykonania- sprawdzamy pierwszy w kolejności bytecode. Dla każdego bytecodu wykonujemy operacje które zostały opisane w komentarzach w pliku `Opcode.py`. Np mamy:

```
class Opcode(Enum):
    # ...
    # push stack.pop(1) + stack.pop(0)
    ADD = auto()
    # ...

if current_op.opcode == Opcode.ADD:
    self._validateParametersLength(current_op, parameters, 0)
    a = self.stack.pop()
    b = self.stack.pop()
    self.stack.append(b + a)
```

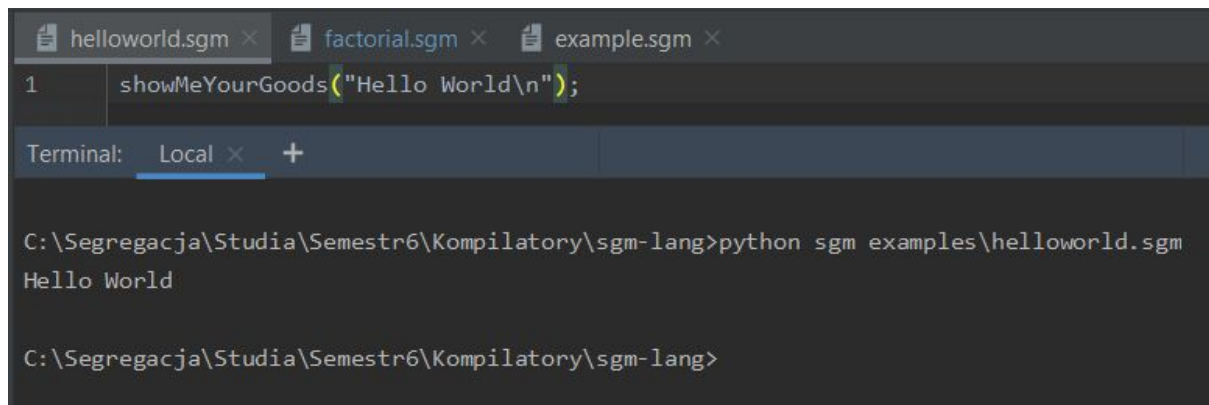
Rys 12. Przetworzenie kodu ADD w Interpreterze

Całość interpretera jest rozbudowaną instrukcją warunkową `IF...ELIF...ELSE`, która ma za zadanie sprawdzić wszystkie Opcode i wykonać odpowiednie akcję. Większość z akcji jest intuicyjna i myślimy że nie ma potrzeby ich tłumaczyć. Jednakże pojawiają się pewne operacje dla których kilka słów wyjaśnienia może być pomocnych:

1. `LOAD` → sprawdzane jest dodatkowo czy dana zmienna istnieje. Jeśli nie rzucony jest wyjątek
2. `JMP` → Dokonujemy skok do podanej instrukcji. Możemy adresować bezwzględnie (podajemy numer instrukcji na stosie), lub pośrednio (o ile miejsc stack pointer ma się przesunąć)
3. `JMP_IF` → jak `JMP`, dodatkowo skok zostanie wykonany tylko wtedy gdy warunek jest spełniony.

## Przykłady działania

Wynik dla programu helloworld.sgm



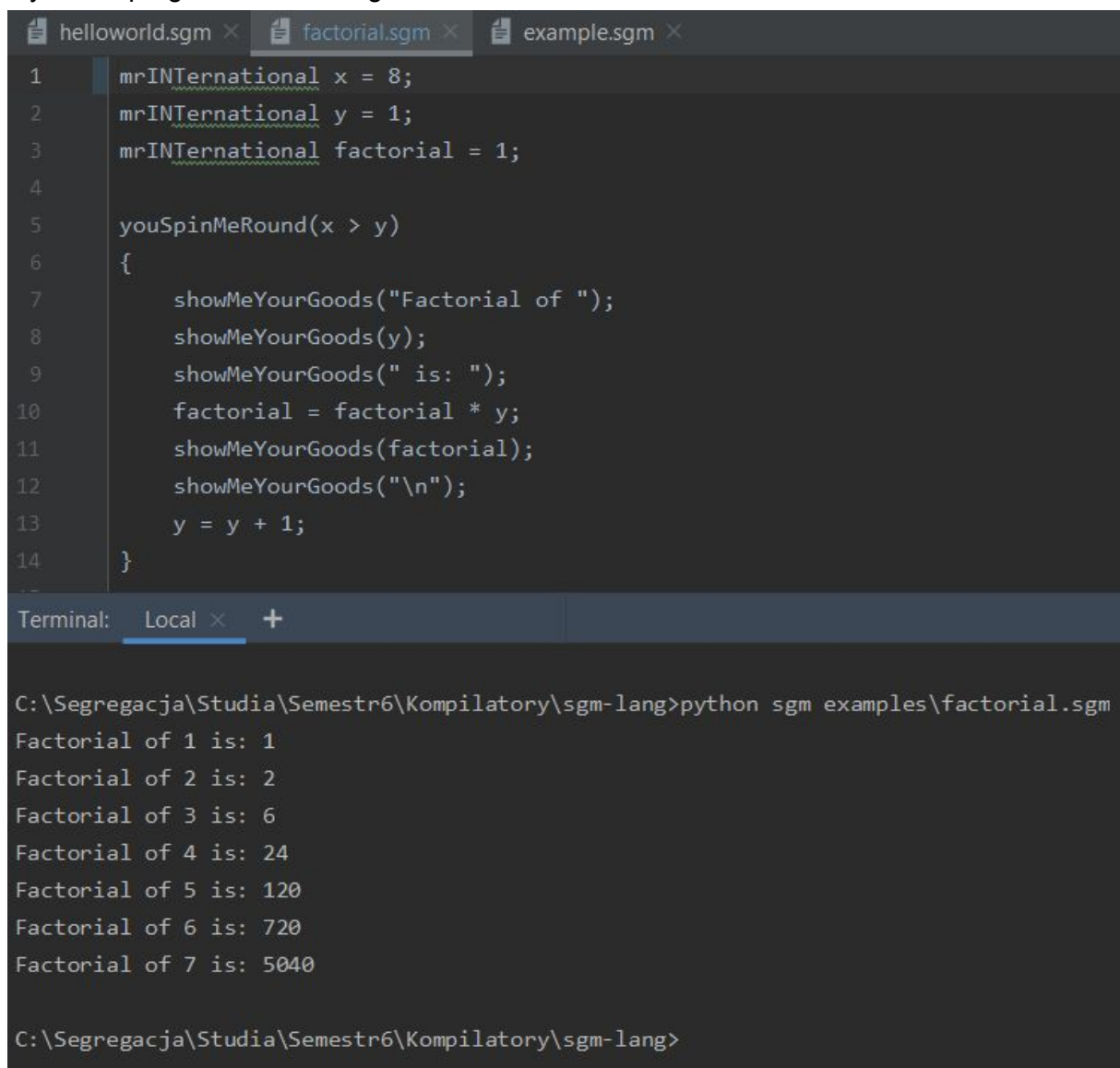
```
helloworld.sgm x factorial.sgm x example.sgm x
1 showMeYourGoods("Hello World\n");

Terminal: Local x +

C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>python sgm examples\helloworld.sgm
Hello World

C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>
```

Wynik dla programu factorial.sgm



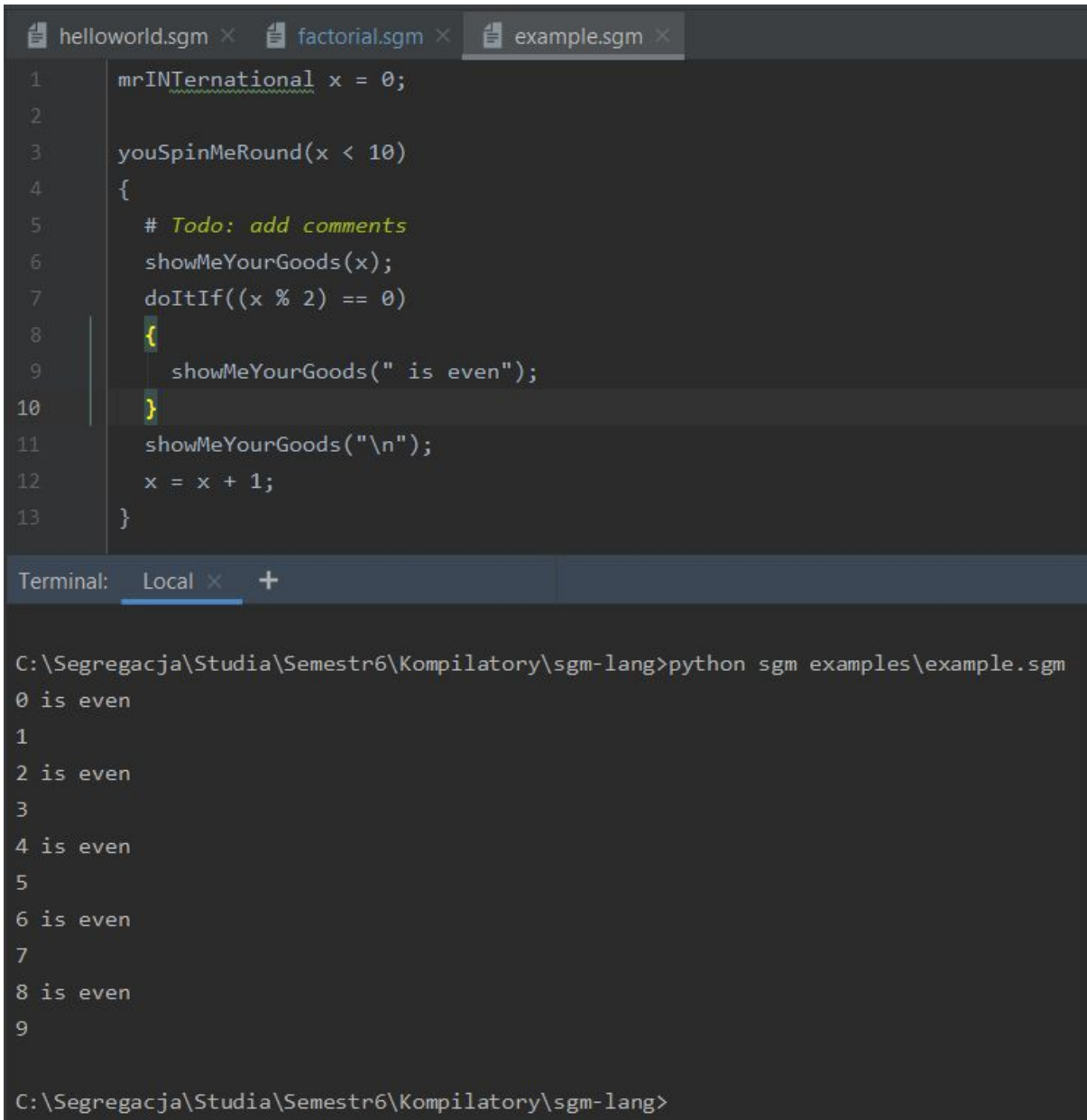
```
helloworld.sgm x factorial.sgm x example.sgm x
1 mrINTERNational x = 8;
2 mrINTERNational y = 1;
3 mrINTERNational factorial = 1;
4
5 youSpinMeRound(x > y)
6 {
7     showMeYourGoods("Factorial of ");
8     showMeYourGoods(y);
9     showMeYourGoods(" is: ");
10    factorial = factorial * y;
11    showMeYourGoods(factorial);
12    showMeYourGoods("\n");
13    y = y + 1;
14 }

Terminal: Local x +

C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>python sgm examples\factorial.sgm
Factorial of 1 is: 1
Factorial of 2 is: 2
Factorial of 3 is: 6
Factorial of 4 is: 24
Factorial of 5 is: 120
Factorial of 6 is: 720
Factorial of 7 is: 5040

C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>
```

Wynik dla programu example.sgm



The screenshot shows a code editor with three tabs: helloworld.sgm, factorial.sgm, and example.sgm. The example.sgm tab is active, displaying the following code:

```
1  mrINTERNational x = 0;
2
3  youSpinMeRound(x < 10)
4  {
5      # Todo: add comments
6      showMeYourGoods(x);
7      doItIf((x % 2) == 0)
8      {
9          showMeYourGoods(" is even");
10     }
11     showMeYourGoods("\n");
12     x = x + 1;
13 }
```

Below the code editor is a terminal window with the following output:

```
C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>python sgm examples\example.sgm
0 is even
1
2 is even
3
4 is even
5
6 is even
7
8 is even
9

C:\Segregacja\Studia\Semestr6\Kompilatory\sgm-lang>
```

## Future Work

- Możliwość definiowania funkcji
- Dodanie kolekcji (listy, zbiory, słowniki)
- Pobieranie danych od użytkownika - na obecnym etapie jedyną opcją "wprowadzania" danych jest zahardcodowanie ich w kodzie programu.

# Podział pracy

**Tokenizer** - Marcin Kozak

**Parser** - Szymon Borowy

**Generator bytencodu i interpreter bytencodu** - Grzegorz Nieużyła