# Zadanie zaliczeniowe z kompilatorów - dokumentacja

Autorzy: Jacek Gosztyła, Ignacy Grudziński

## Opis sposobu użycia języka kalkulatora

Należy uruchomić `python3 parser3.py` i wpisać dowolne wspierane przez język wyrażenie.

## Opis zaimplementowanych funkcjonalności

Kolejne instrukcje języka muszą być oddzielone średnikiem `;`. Kalkulator wspiera wszystkie podstawowe działania matematyczne oraz relacyjne (`2+5`, `3^7`, `4%2`, `3<4`, ... ). Możemy korzystać z funkcji matematycznych (`sin(0.1)`, `tan(0.0)`, `exp(2.3)`, ...) W celu zdeklarowania nowej zmiennej (globalnej czy to lokalnej), należy podać jej typ, np. `int a`. Do zmiennej możemy przypisać wartość zgodną z jej typem w momencie deklaracji `int a = 5` lub po uprzednim zdeklarowaniu `int a; a = 5`. Zmienną możemy wykorzystywać wielokrotnie w obliczeniach. Powyższe instrukcje możemy wykorzystywać w blockach kodu zajdujących się pomiędzy klamrami `{}` Instrukcja warunkowa `if` ma postać:

```
if(warunek){
    instrukcja 1;
    instrukcja 2;
    ...
}else{
    instrukcja 1;
    ...
}
```

Blok `else` jest opcjonalny. W kalkulatorze mamy do dyspozycji pętle `while` oraz `for`. Mają one postać:

```
while(warunek){
    instrukcje...
}
```

```
for(instrukacja poczatkowa; warunek zakonczenia; instrukcja po wykonaniu
petli){
    instrukcje...
}
```

Możemy definiować własne funkcje w postaci:

```
zwracany_typ nazwa_funkcji(typ_1 arg_1, typ2 arg_2, ...){
    instrukcje...
```

```
}
```

Język wspiera zagnieżdżanie funkcji, np. `int x(int g){return g^2};int y(int h){return x(h)}`

## Szczegóły implementacji

Na kalkulator składają się:

- lexer (`lexer3.py`)
- parser (`parser3.py`)
- interpeter (`interpreter3.py`)

### Lexer

Słownik `reserverd` definiuje słowa kluczowe z których nie można korzystać do nazywania zmiennych.

Lista `tokens` zawiera zdefiniowane w dalszej części lexera tokeny które są wykorzystywane nasępnie przez parser.

### Parser

Tupla `precedence` pomaga zredukować shift/reduce konflikty poprzez zdefiniowanie kolejności parsowania różnego radzaju danych.

W dalszej częsci w funkcjach postaci `p_nazwa` definiowane są sposoby przechowywania wszystkich dających się zinterpretować struktur językowych, wykorzystywanych później przez interpreter. Struktury te zostały opisane w funkcjonalnościach.

Dla przykładu

```
def p_instruction_while(p):
    '''instruction : WHILE condition block'''
    p[0] = [('WHILE', (p[2], p[3]))]
```

definiuje strukturę while, które składa się ze słowa kluczowego `while`, warunku oraz bloku. Z czego warunek ma postać zdefiniowaną

```
def p_condition(p):
    '''condition : OPAREN expression CPAREN'''
    p[0] = p[2]
```

tzn. wyrażenie zawierające się pomiędzy nawiasami `(` oraz `)`. Itd.

### Interpreter

Interpreter evaluuje wyrażenia zwrócone przez parser. Posiada obiekt `Scope` który ma stos scopów, pozwalający na zagnieżdżone wywołania funkcji. Metoda `evaluate` korzysta z metod pommocniczych

`eval_nazwa`. Np. w

```
def eval_assign(expr: tuple) -> Symbol:
        name, assigned = expr
        assigned_symbol = evaluate(assigned, scope)
        if not scope.contains(name):
            raise Exception(f"{name} is not defined!")
        assigned_to = scope.get(name)
        type_assertion(assigned_symbol, assigned_to.typ)
        scope.update(name, assigned_symbol)
        return assigned_symbol
```

kolejno rozbijamy wyrażenie zwrócone przez parser, ewaluujemy wyrażenie, które ma zostać przypisane do zmiennej, sprawdzamy, czy zmienna do której ma następić przypisanie isnieje, porównoujemy typy a na końcu dokonujemy faktycznego przypisania (o ile wcześniej nie wystąpił błąd). Pomocniczo obiekty `Symbol` przechowują typ i wartość.

Do testowania pastera napisaliśmy własny tester.

### Parser tester

`parser_tests/parser2_tester.py` pozwalaja na symulowanie wpisywania kodu na input parsera i porównywanie generowanego przez niego outputu z oczekiwanym rezultatem. Wykorzystujemy je w odpowiednio `parser_tests/lab2_test.py`.

Intepreter jest testowany wewnątrz.

## Ograniczenia

- Pętle można pisać w jednej linijce.
- Kalkulator nie wspiera negacji.
- Jeżeli string nie ma wartości, zostanie automatycznie scastowany do napisu 'None'
- nie można definiować własnych typów, obiektów
- w pętli for nie działa przeciążanie operatora

## Przykłady użycia

```
int a = 1; a = 3; a;
bool b; b = 2 < 6; b
int x(int g){return g^2};x(5)
string a = "global"; if (2>1) { a = "overwritten"}; a
int add(int a, int b){return a+b}; add(12, 5)
string a = "a"; int b = 5; a+
int x(int g){return g^2};int y(int h){return x(h)};y(5)
```

## Realizacja konkretnych zadań

W głównym folderze projektu znajdują się kolejne komponenty kompilatora, przystosowane do kolejnych ćwiczeń z laboratorium. Np. `parser2.py` oznacza, że jest to parser napisany na potrzeby zadania 2 z przedmiotu kompilatory.

## Zadanie 1

Testy do zadania w `md.py`.

## Zadanie 2

Zrealizowane jako `parser2.py`. W `parser_tests/lab2_test.py` napisane zostały testy to parsera.

## Zadania 3,4,5,6

Zrealizowane jako `interpreter3.py`. Testy umieściliśmy na końcu powyższego pliku. Dodatkowo wizualizacja drzewa składniowego w `ast.py`.