Dokumentacja kalkulatora

Kompilatory, 2020

Autorzy:

1. Andrzej Ratajczak
2. Michał Szkarłat

Spis treści

Opis sposobu użycia języka kalkulatora **3**
Przykład użycia z poziomu konsoli 3

Opis zaimplementowanych funkcjonalności **4**

Szczegóły implementacji **6**
Priorytety operatorów 6
Obsługa tokenów dla liczb całkowitych i rzeczywistych 7
Obsługa tokenów dla funkcji unarnych 7
Obsługa tokenów dla funkcji binarnych 7
Implementacja instrukcji oddzielonych średnikiem 8
Obsługa typów string i boolean 8
Realizacji pętli while i for 9
Budowa drzewa składniowego 10
Prześledźmy jak zostało zbudowane drzewo. 11

Ograniczenia języka kalkulatora **12**

Przykłady użycia języka kalkulatora **12**
Zaawansowany przypadek: obliczanie ciągu fibonacciego 13

1. Opis sposobu użycia języka kalkulatora

Nasz kalkulator to proste narzędzie pozwalające na ewaluację obliczeń arytmetycznych wzbogacony o między innymi instrukcje warunkowe, pętle czy deklaracje zmiennych.
Aby skorzystać z kalkulatora należy obsłużyć funkcję z dostępnego API (w pliku **calc.py**):

```
def parse( input )
```

Funkcja ta przyjmuje na wejściu napis zawierający tekst do ewaluacji. Od użytkownika zależy jak ją obsłuży: może zrobić interfejs CLI, może też wczytywać tekst z pliku, albo wprowadzać wywołania za pomocą konsoli python. Funkcja **parse(input)** zwraca zbudowane drzewo AST, oraz wypisuje na standardowe wyjście rezultat wprowadzonych obliczeń. W przypadku błędu (na przykład błędu parsowania, niepoprawnej deklaracji zmiennej lub błędu arytmetycznego), program wypisze wygenerowany błąd na standardowe wyjście, oraz jeśli to możliwe, zwróci poprawnie zbudowane drzewo składniowe.

Przykład użycia z poziomu konsoli

W katalogu głównym projektu uruchom konsolę python wskazując jako moduł plik **calc.py**.
Przykładowe uruchomienie wygląda następująco:

```
user@pc: ~/rootDir$ python3 -im calc
>>>
```

Uruchomiliśmy w tym momencie program w trybie interaktywnym. Aby korzystać z udostępnionego API należy wywołać wspomnianą wyżej funkcję:

```
>>> ast = parse(“ begin print(2+2); end ”)
4
>>>
```

Aplikacja wypisze na standardowe wyjście wynik parsowania. Pod zmienną **ast** przypisane zostało drzewo składniowe AST. Jeśli chcemy zobaczyć jak ono wygląda możemy je wydrukować na ekran:

```
>>> str(ast)
```

2. Opis zaimplementowanych funkcjonalności

Każda z przedstawionych poniżej funkcjonalności została przetestowana przy pomocy testów jednostkowych, testów integracyjnych oraz testów end-2-end. Każde z szczegółów implementacyjnych zostały zawarte w punkcie trzecim.

Lp	Zaimplementowana funkcjonalność
1	Obsługa tokenów dla liczb: ○ całkowitych ○ rzeczywistych
2	Obsługa tokenów dla funkcji specjalnych: ○ sin ○ cos ○ exp ○ log ○ sqrt ○ power ○ intToFloat ○ floatToInt ○ eq ○ noteq ○ lt ○ gt ○ lteq ○ gteq ○ print
3	Obsługa tokenów dla operatora: ○ dodawania ○ odejmowania ○ mnożenia ○ dzielenia ○ grupowania
4	Implementacja działań: ○ potęgowania ○ funkcji specjalnych ○ działań relacyjnych ○ zmiany znaku
5	Implementacja instrukcji oddzielonych średnikiem
6	Kontynuacja parsowania kolejnych instrukcji w przypadku błędu

7	**Możliwość wykonywania obliczeń dla odwrotnej notacji polskiej** Przedstawiona funkcjonalność nie jest częścią głównego (zintegrowanego ze wszystkich laboratoriów) programu. Kalkulator dla odwrotnej notacji polskiej został dołączony jako osobny moduł (plik).
8	Instrukcje warunkowe i pętle
9	Wizualizacja drzewa składniowego
10	Deklarowanie typów dla zmiennych
11	Sprawdzanie typów
12	Instrukcja przypisania
13	Konwersja typów za pomocą dodatkowego operatora
14	Definiowanie funkcji
15	Definiowanie bloków instrukcji
16	Definiowanie zmiennych globalnych i lokalnych
17	Instrukcja wywołania funkcji
18	Automatyczna konwersja typów
19	Zagnieżdżone wywołania funkcji
20	Pomijanie zbędnych instrukcji
21	Optymalizacja wykonania wspólnych podwyrażeń
22	Optymalizacje algebraiczne

Tab 1.1 Zaimplementowane funkcjonalności

3. Szczegóły implementacji

Priorytety operatorów

Priorytety zostały zdefiniowane w lexerze. Wyglądają one następująco:

```
precedence = (
      ('left', '+', '-'),
      ('left', '*', '/'),
      ('left', 'sin', 'cos', 'log', 'exp', 'sqrt',
               'intToFloat', 'floatToInt', 'print'),
      ('right', 'power'),
      ('right', 'UMINUS'),
)
```

Kolejność priorytetów jest następująca:

Priorytet	Operator
1	umius
2	operator potęgowania
3	operatory: ○ sin ○ cos ○ log ○ exp ○ sqrt ○ intToFloat ○ floatToInt ○ print
4	operatory mnożenia i dzielenia
5	operatory dodawania i odejmowania

Tab 3.1 Priorytety operatorów

Obsługa tokenów dla liczb całkowitych i rzeczywistych

Rodzaj	Typ
Liczby całkowite	`int`

	`int i = 2;`
	
	`define add(int a, int b) = a + b;`
	
	`for(int i = 1; i < 10; i = i + 1) i;`

Liczby rzeczywiste	`float`

	`float f = 2.0;`
	
	`define toInt(float a) = floatToInt(a)`

Typy w kalkulatorze są silnie typowane. Oznacza to, że do typu int możemy przypisać tylko liczbę całkowitą - **przypisanie int i = 2.0 spowoduje wyświetlenie błędu**.

Obsługa tokenów dla funkcji unarnych

Wszystkie funkcje wymienione w punkcie 2 (tabela 1.1, podpunkt 2) są obsługiwane w podobny sposób. Oczekują one jednego argumentu, który jest dynamicznie typowany **(poza funkcjami specjalnymi intToFloat oraz floatToInt)**. Umożliwia to wywołanie sinusa w następujący sposób:

```
begin sin(2); end
begin sin2; end
begin sin 2.0; end
begin sin 2; end
```

Obsługa tokenów dla funkcji binarnych

Występuje w sposób analogiczny. Możliwe są następujące wywołania:

```
begin print(1 == 1); end
begin 1+2; end
begin 2 ** 3; end
begin print(-3 - -3); end
begin print(1!=4); end
```

Implementacja instrukcji oddzielonych średnikiem

Realizowana jest na poziomie lexera:

```
def p_casual_statements(p):
      """
      casual_statement : if_statement ';'
                       | for_statement ';'
                       | while_statement ';'
                       | define ';'
                       | statement ';'
      """
      p[0] = ast.P_casual_statements(p[1])
```

Pozwala to na budowę rozbudowanych instrukcji, takiej jak:

```
begin
  define fun(int i, int j) = i + j;
  define fun2(int i, int j) = fun(i, j) * i;
  int res = fun(3, 4);
  print(res);
end
```

Lexer spodziewa się aby każdy **casual_statement** oraz **statement** kończył się średnikiem.

Obsługa typów string i boolean

Rodzaj	Typ
Ciągi znaków	`string`
	`string` `i = “moj string“;` `define add(`**`string`**` a) = a + “concat“;`
Wartości logiczne	`boolean`

	`boolean` `f = False;`
	`boolean` `t = True;`
	`if(`**`t`**`) print(True);`
	`int i = 0; while(`**`i < 1`**`) i = i + 1;`

Realizacji pętli while i for

Budowa pętli:

```
def p_while_statement(p):
      """
      while_statement : WHILE '(' expression ')' statement
      """
      condPr = p[3]
      cond = condPr.fun
      statementPr = p[5]
      statement = statementPr.fun

      def res(local):
      while cond(local)[1] if callable(cond) else cond:
            # print(statement(local))
            statement(local)

      p[0] = ast.P_while_statement(lambda local: res(local) if res(local)
else ast.P_empty().fun(local), condPr, statementPr)
```

```
def p_for_statement(p):
      """
      for_statement : FOR '(' statement ';' expression ';' statement ')'
statement
      """

      initialPr = p[3]
      initial = initialPr.fun
      condPr = p[5]
      cond = condPr.fun
      updatePr = p[7]
      update = updatePr.fun
      actionPr = p[9]
      action = actionPr.fun

      def loop(local):
      initial(local)
      while cond(local)[1]:
            # print(action(local))
            action(local)
            update(local)

      p[0] = ast.P_for_statement(lambda local: loop(local) if loop(local)
else ast.P_empty().fun(local), initialPr, condPr, updatePr, actionPr)
```

Obie pętle mogę realizować w sekcji **action** tylko jedną instrukcję (**statement**). Oznacza to, że nie jest możliwe wywoływanie bloku **begin end** w ramach kolejnych iteracji pętli. Jest to jedyne ograniczenie. Pod warunek condition pętla przyjmuje dowolną wartość typu boolean.

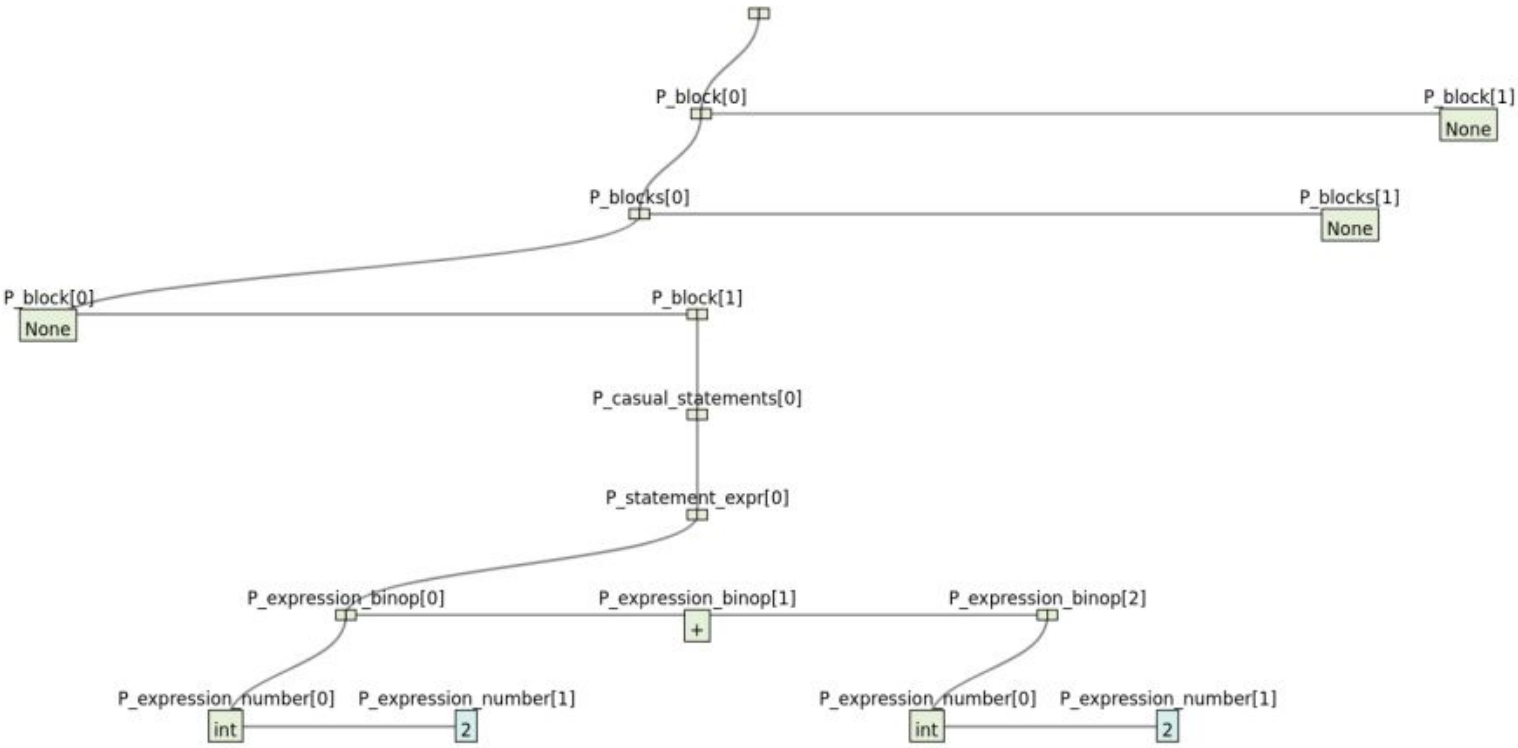
Budowa drzewa składniowego

Definicja drzewa trzymana jest w pliku **ast.py**. Budowane drzewo jest bardzo niestandardowe. Wynika to z zastosowania **funkcyjnych deklaracji** niektórych z instrukcji - co sprawia, że drzewo przechowuje nie tylko wprowadzoną wartość, ale i funkcję (która zawiera obietnicę ewaluacji wyniku w przyszłości). Każde z liści poza wskazaniem na trzymany typ (lub None) może zawierać wspomnianą funkcję. Podczas parsowania, z każdego liścia ściągamy informacje o zadeklarowanej funkcji i ewaluujemy ją według potrzeby. Takie podejście pozwala na łatwą implementację deklarowania własnych funkcji w kalkulatorze oraz na korzystanie z różnych rodzajów dostępu do zmiennych - globalnych i lokalnych. Minusem tego aspektu jest złożona i skomplikowana struktura, którą trzeba czytać w określonym porządku.

Warto również wspomnieć o sposobie przechowywania kolejnych elementów drzewa. Spojrzymy na strukturę po sprasowaniu takiego ciągu:

```
begin begin 2+2; end end
```

Drzewo prezentuje się następująco:



Każdy z nodeów został zbudowany jako osobna klasa, która w zależności od pełnionej funkcji przyjmuje określoną liczbę parametrów. Na przykład klasa **P_start** (która jest nadrzędną klasą) przyjmuje tylko jeden - **P_block**. P_block natomiast oczekuje **P_blocks** i **P_casual_statement**.

Prześledźmy jak zostało zbudowane drzewo.

→P_start
(korzeń, główna klasa w którą zapakowane jest każde wywołanie)

→ P_block
(akceptowany przez P_start, oznacza wywołanie bloku **begin end**. Z takiej implementacji wynika, że każde parsowane wyrażenie **musi** zaczynać się od deklaracji begin-end)

→P_blocks
→P_block
(P_block przyjmuje dwa parametry, P_blocks - informuje, że występuje kolejna - bezpośrednia - deklaracja begin-end, a P_block wskazuje, na kolejny blok typu P_block).

Rozwiązanie to pozwala na zastosowanie wielokrotnych definicji bloków begin-end. Jeśli występuje ich więcej niż jedno P_block zawiera wskazanie na P_blocks, które zawiera wskazanie na kolejne P_block (a te może zawierać koleje P_blocks). Powstaje rekurencyjna struktura, która na każdym węźle przechowuje informacje o lokalnym bloku do ewaluacji oraz wskaźnik na następne begin-end. Takie rozwiązanie pozwala na łatwą implementację wielokrotnych bloków przy wykorzystaniu biblioteki lex.

→P_casual_statement
(Teraz zaczyna się parsowanie wyrażenia **2 + 2;** Kalkulator zbudował odpowiednią strukturę P_bloków i w jednej z nich przetrzyma teraz informacje o dodawaniu)

→P_expression_binop
→P_expression_number
(ostanie dwa wywołania przechowują rodzaj operacji - “+” - oraz prasowaną liczbę - “2”)

4. Ograniczenia języka kalkulatora

Kalkulator posiada również kilka ograniczeń. Podczas łączenia poszczególnych ćwiczeń należało pójść na kompromisy, tak by zachowywał się jednoznacznie i deterministycznie. Dlatego w finalnej wersji kalkulatora nie znalazły się takie funkcjonalności jak:

- Odwrotna Notacja Polska - aby móc z niej korzystać, należy skorzystać ze snapshotu związanego z tymże ćwiczeniem
- Explicite konwersje danych - zdecydowaliśmy się na automatyczne konwersje pomiędzy poszczególnymi typami danych w czasie ich ewaluacji.
- Przemieszczanie kodu w pętli - z braku czasu zdecydowaliśmy się porzucić rozwój tej funkcjonalności
- Automatyczna korekta błędów w tokenach - zdecydowaliśmy się, że eliminowanie błędów w tokenach kłóci się z założeniami sprawdzania poprawności składniowej
- Pętle które mogą wykonywać tylko jedną instrukcję (brak bloków)

5. Przykłady użycia języka kalkulatora

```
begin 2 + 2; end
```

```
begin
  int i = 2;
  int j = 3;
  print(i + j);
end
```

```
begin
  int i = 0;
  define fun(int i, int j) = i + j;
  fun(2, 3);
end
```

```
begin
  define fun(int i, int j) = i + j;
  define fun2(int i, int j) = fun(i, j) * i;
  int res = fun(3, 4);
  print(res);
end
```

```
begin
  define fun(int i, int j) = i + j;
  define fun2(int i, int j) = i * j;
  int res = fun(fun2(2, 3), 4);
end
```

```
begin
  for(int i = 0; i < 10; i = i + 1) print(i);
end
```

Zaawansowany przypadek: **obliczanie ciągu fibonacciego**

```
begin
  define fib(int n) = if (n < 2) n else fib(n - 1) + fib(n - 2)
  print(fib(10));
end
```

Powyższe przykłady można znaleźć ewaluowane w testach tests.py