Dokumentacja wstępna

Język Dixty

Autor: Dominika Ferfecka

Wprowadzenie

Dixty jest językiem ogólnego przeznaczenia, charakteryzujący się wbudowanym typem słownika z określoną kolejnością elementów tożsamą do kolejności wstawiania ich.

Oprócz podstawowych danych takich jak int, float, bool oraz string, a także wspomnianego wcześniej słownika, język udostępnia również typy danych: lista i para.

Język Dixty jest dynamicznie i silnie typowany, z mutowalnymi zmiennymi.
Program może być pisany za pomocą linijek kodu w formie skryptów - nie jest wymagana ani funkcja ani klasa main.

Dostępne operatory

W poniższej tabelce przedstawione są możliwe operatory wraz z ich priorytetem. Dodatkowo dostępne są również nawiasy, których priorytet jest najwyższy - wykonują się rekurencyjnie.

Operator	**Priorytet**
Or	1
And	2
Not	3
==	4
!=	4
<	4
>	4
<=	4
>=	4
+	5
-	5
*	6
/	6
- (negacja)	7
.	8

Złożone przykłady wykorzystania języka

Podstawowe typy danych liczbowych

Przykład tworzenia zmiennej i przypisania wartości

- utworzenie zmiennej całkowitej
- utworzenie zmiennej zmiennoprzecinkowej
- utworzenie zmiennej bool

```
liczba = 1;
ułamek = 0.5;
boolean = True;
boolean2 = False;
```

Operacje arytmetyczne

Język udostępnia podstawowe operatory:

- operator dodawania - łączny, przemienny
- operator odejmowania - łączny
- operator mnożenia - łączny, przemienny
- operator dzielenia - łączny

Funkcja print() jest funkcją wbudowaną umożliwiającą wypisywanie danych.

Operacje na liczbach całkowitych

```
a = 2;
b = 3;
c = 5;
d = 10 ;

# DODAWANIE
suma = a + b;
print(suma); # suma == 5

suma = b + a;
print(suma); # suma == 5

suma = a + b + c;
print(suma); # suma == 10

suma = (a + b) + c;
print(suma); # suma == 10
```

```
suma = a + (b + c);
print(suma); # suma == 10


# ODEJMOWANIE
wynik = a - b;
print(wynik); # wynik == -1

wynik = b - a;
print(wynik); # wynik == 1

wynik = c - b - a;
print(wynik); # wynik == 0


wynik = d - c - b - a;
print(wynik); # wynik == 0

wynik = (d - c) - (b - a);
print(wynik); # wynik == 4

wynik = d - (c - (b - a));
print(wynik) # wynik == 6

wynik = d - ((c - b) - a);
print(wynik); # wynik == 10

# MNOŻENIE

wynik = a * b;
print(wynik); # wynik == 6

wynik = b * a;
print(wynik); # wynik == 6

wynik = a * b * c;
print(wynik); # wynik == 30

wynik = a * b * c * d;
print(wynik); # wynik == 300

wynik = (a * b) * (c * d);
print(wynik); # wynik == 300

wynik = a * (b * (c * d));
```

```
print(wynik); # wynik == 300


# DZIELENIE

wynik = d / a;
print(wynik) # wynik == 5

wynik = a / d;
print(wynik); # wynik == 0.2 ?

wynik = d / a / c;
print(wynik); # wynik == 1

wynik = d / (a / c);
print(wynik); # wynik == 25


# MIESZANE

wynik = 2 - 3 * 1 - 8 / 2 - (2 - 1)
print(wynik) # -6
```

Operacje arytmetyczne na liczbach zmiennoprzecinkowych

```
a = 2.0;
b = 0.5;
c = 3.0;

result = c + a - b;
print(result); # 4.5
```

Operacje arytmetyczne na różnych typach

Język jest dynamicznie typowany, więc konwersje nie zachodzą automatycznie. Operacje na różnych typach danych, muszą być zrealizowane za pomocą funkcji wbudowanych umożliwiających użytkownikowi rzutowanie. Funkcje umożliwiające rzutowanie zostały opisane w tabelce w części “Konwersja typów”

```
a = 2.5;
b = 1;
c = "text";

result = a.ToInt() + b;
```

```
print(result); # 3

result = a + b.ToFloat()
print(result); # 3.5

result = c + b.ToString();
print(result); # text1
```

Operator kropki

Dixty udostępnia również operator kropki, dzięki któremu można wywoływać metody obiektu i odwoływać się do zmiennych.

```
a = "tekst"
result = a.len()
print(result) # 5
```

Obsługa typu znakowego

Poniżej przedstawiony jest sposób tworzenia stringa.
Znak '$' służy do escapowania znaków specjalnych.

```
tekst = "Hello";

symbol = "$n"; # znak nowej linii
symbol = "$t"; # znak tabulatora

symbol = "$$";
print(symbol); # $

tekst = "cudzyslow to $" - koniec stringa";
print(tekst); # cudzyslow to " - koniec stringa

tekst = "apostrof to $' - koniec stringa";
print(tekst); # apostrof to to ' - koniec stringa

tekst = "!@#$$/^*";
print(tekst); # !@#$/^*
```

Obsługa stringa
Możliwa jest konkatenacja oraz sprawdzenie długości stringa (funkcja wbudowana .len().

```
a = "Hello";
b = "World";

# Konkatencja

wynik = a + b;
print(wynik); # "HelloWorld"

wynik = a + " " + b + "!";
print(wynik); # "Hello World!"

# Długość stringu

wynik = a.len();
print(wynik); # 5

c = a + b;
wynik = c.len();
print(wynik); # 10
```

Wyrażenia logiczne

Dostępne operatory logiczne uporządkowane w kolejności wykonania:

- Not
- And
- Or

```
# AND
result = True And True
print(result) # True

result = True And False
print(result) # False

result = False And True
print(result) # False

result = False And False
print(result) # False

# OR
result = True Or True
print(result) # True

result = True Or False
```

```
print(result) # True

result = False Or True
print(result) # True

result = False Or False Or False
print(result) # False]

# Not
result = Not True
print(result) # False

result = Not False
print(result) # True


# mieszane
result = False And False Or True
print(result) # True

result = True And Not True
print(result) # False

result = Not False And True And False
print(result) # True

result = Not True Or True
print(result) # True
```

```
a = 1
b = 2
text = "1234"

result = ( 1 + 1 == 2 ) And ( 3 - 2 == 2 )
print(result) # False

result = ( 1 + 1 == 2 ) Or ( 3 - 2 == 2 )
print(result) # True

result = ( a - 1 )  And ( b + 1 == 3)
print(result) # False

result = ( a - 1 )  Or ( b + 1 == 3)
print(result) # True
```

```
result = ( text.len() == 4 ) And (  a + b == 3 )
print(result) # True
```

Operator porównania

Dostępne operatory porównania:

- >
- >=
- <
- <=
- ==
- !=

```
a = 1
b = 2

result = ( a == 2 )
print(result) # False

result = ( a != 2 )
print(result) # True

result = ( a >= 1 )  And ( b + 1 <= 3)
print(result) # True


result = ( a >= 1 )  And ( b + 1 <= 3)
print(result) # True

result = ( a < 1 )  Or ( b + 1 > 3)
print(result) # False

result = (a + b * 2 - 1 == 2 * 6 / 3)
print(result) # True
```

Obsługa komentarza

Język obsługuje komentarz jednolinijkowy oznaczany znakiem początkowym ‘#’ i nie wymagającym zakończenia.

Obsługuje również komentarz wielolinijkowy oznaczany znakiem początkowym ‘<#’, wymagający znaku końcowego ‘#>’

```
# To jest komentarz
To_nie_jest_komentarz = 3;

<#
      To też jest <# komentarz
      To # też jest komentarz
#>

To_nie_jest_komentarz = 3;
```

Zakresy widoczności zmiennych

Zmienna lokalna
Zmienna lokalna zdefiniowana w funkcji jest widoczna tylko w obrębie danej funkcji i nie można się do niej odwołać poza tą funkcją.

```
fun hello_world()
{
    x = 10;
    print(x);
}

hello_world();
# print(x) # this will cause error
```

Zmienna globalna
Zmienna globalna jest widoczna w całym programie - można się do niej też odwołać.

```
x = 10;
fun hello_world()
{
    print(x);
}

hello_world();
print(x);
```

Mutowalność

Zmienne w języku Dixty są mutowalne, można przypisywać do nich wartość dowolną ilość razy.

```
x = 10;
```

```
x = 5;
x = 2.5;
x = "text";
x = [1, 2, 3, 4];
```

Instrukcja warunkowa

Język Dixty umożliwia stworzenie instrukcji warunkowej if, a także obsłużenie przypadków przeciwnych za pomocą: else_if, oraz else.

```
if ( a + b == 2 And b <= 3)
{
    print("a + b jest równe 2 i b jest <= 3");
}


if ( a == 2 )
{
    print("a jest równe 2");
}
else
{
    print("a nie jest równe 2");
}


if ( a == 2)
{
    print("a jest równe 2");
}
else_if ( a == 3)
{
    print("a jest równe 3");
}
else
{
    print("a nie jest równe 2 ani 3");
}
```

Instrukcja pętli

Dostępne są również pętle for oraz while. Pętla for umożliwia iterację po elementach listy i słownika.

```
list = [1, 2, 3, 4]

for element in list
{
   print(element); # 1234
}

i = 0;
while (i < 3)
{
     print(i); # 0123
     i++;
}
```

Funkcje

Język Dixty umożliwia użytkownikowi definiowanie własnych funkcji. Parametry są przekazywane przez wartość. Nie jest możliwe przeciążanie funkcji.

```
# przykład 1

fun display()
{
    x = 2;
    print(x);
}

display(); # 2


# przykład 2
fun add(a, b)
{
    return a + b;
}

a = 2;
b = 3;
result = add(a,b);
print(result); # 5
```

Przekazywanie parametrów przez wartość

W Dixty parametry są przekazywane przez wartość.

```
fun increase(x)
{
    x = x + 1;
    return x;
}

my_x = 2;
new_x = increase(my_x);
print(new_x); # 3
print(my_x); # 2
```

Przesłanianie zmiennych

Zmienna zdefiniowana o tej samej nazwie co istniejąca nazwa, ma priorytet nad zmienną globalną zdefiniowaną wcześniej.

```
x = 10;

fun display()
{
     x = 20;
     print(x); # 20
}

display();
print(x); # 10
```

```
a = 1;
b = 2;
c = 3;

fun add(a, b)
{
    c = 10;
    return a + b + c;
}

result = add(a, b);
print(result); # 13;
```

Rekursywne wywołania funkcji

Możliwe jest rekursywne wywołanie funkcji.

```
fun factorial(x)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * factorial( x - 1 );
    }
}

result = factorial(5);
print(result); # 120
```

Lista

Listy w języku Dixty mogą zawierać w sobie elementy różnych typów.

```
list_int = [1, 2, 3, 4, 5];
list_float = [0.0, 0.25, 1.0, 3.5, 7.5];
list_str = ['a', 'b', 'c', 'd', 'e'];
list_diff = [1, 0.5, 'c', 'd', 'e'];
```

```
a = 3;
b = 5;
text = "some_text";

fun increase(a)
{
    a = a + 1;
}

list = [a, b, text, 1, increase(2)]; # [3, 5, text, 1, 3]
```

Operacje na liście

Dostępne są funkcje wbudowane umożliwiające wykonywanie operacji na liście:

- append(element) - dodanie elementu na koniec listy
- remove(index) - usunięcie elementu z miejsca wskazanego przez index
- insert(index, element) - dodanie nowego elementu w miejsce wskazane przez index

- len() - zwrócenie ilości elementów listy

```
list_int = [1, 2, 3, 4, 5]

print(list_int[0]); # 1
# add element
lista_int.append(7); # list_int = [1, 2, 3, 4, 5, 7]
# remove element from index
lista_int.remove(2); # list_int = [1, 2, 4, 5, 7]

# insert element on index (index,value)
lista_int.insert(0,3); # list_int = [3, 1, 2, 4, 5, 7]
# print length
lista_int.len() # 6
# iterate through list
for i in lista_int
{
    print(i);  # 312457
}

print(lista_int); # [3, 1, 2, 4, 5, 7]
```

```
list_str = ['a', 'b', 'c', 'd', 'e']

print(list_str[3]); # d
# add element
lista_str.append('f'); # list_str = ['a', 'b', 'c', 'd', 'e', 'f']
# remove element from index
lista_str.remove(1); # list_str = ['a', ';, 'd', 'e', 'f']

# insert element on index (index,value)
lista_str.insert(3,'x'); # list_str = ['a', 'c, 'd', 'x', 'e', 'f']
# print length
lista_str.len() # 6
# iterate through list
for element in lista_str
{
    print(element); # abcde
}
```

Zagnieżdzanie list

```
a = 1;
```

```
b = 5;

my_list = [
     [a, 2, 3],
     [4, b, 6],
    [7, 8, 9]
]


# print my list[row]
print(my_list[1]) # [4,5,6]

# print my list[row][column]
print(my_list[0][1]) # 2


# add element to first list
my_list[0].append(1)
<# my_list = [
     [a, 2, 3, 1],
     [4, b, 6],
     [7, 8, 9]
]
>#

# add new list
my_list.append([9, 8, 7])
<# my_list = [
     [a, 2, 3, 1],
     [4, b, 6],
     [7, 8, 9],
     [9, 8, 7]
]
>#


# remove element from index 1 from first list
my_list[0].remove(1)
<# my_list = [
     [a, 3, 1],
     [4, b, 6],
     [7, 8, 9],
     [9, 8, 7]
]
>#
```

```
# remove third list
my_list.remove(2)
<# my_list = [
     [a, 3, 1],
     [4, b, 6],
     [9, 8, 7]
]
>#


# length
my_list[1].len() # 3

# quantity of lists
my_list.len() # 3

#iteration
     [a, 3, 1],
     [4, b, 6],
     [9, 8, 7]

for list in my_list
{
    for element in list
    {
          print(element);
    }
     print("$n");
     <#
     131
     456
     987
     #>
}


lista_a = [1, 2, 3];
lista_b = [lista_a, 5, 6];
my_list = [
     [ lista_b, 2, 3]
]
print(my_list[0][0][0]) # 1
```

Para

Język udostępnia również typ danych Para, składający się z dwóch dowolnych typów danych sprecyzowanych przez język.

```
pair = ("value", 2);

print(pair.Key); # value
print(pair.Value); # 2

print(pair); # ("value", 2)

pair.Key = "new_value";
print(pair.Key); # new_value

pair.Value = 3;
print(pair.Value); # 3

a = 3;
text = "some_text";
pair = (a, text);
print(pair); # (3, some_text);
```

Słownik

W Dixty można tworzyć słowniki z ustaloną kolejnością elementów, która jest tożsama z kolejnością wstawiania tych elementów.
Można utworzyć słownik z różnymi typami danych.
Elementami słownika mogą być zarówno literały jak i zmienne, listy, inne słowniki oraz wartość zwrócona z funkcji.

Możliwe są podstawowe operacje na słowniku:

- dodawanie elementów
- usuwanie elementów
- odczytywanie elementów za pomocą klucza
- modyfikowanie elementów
- sprawdzanie czy dany klucz znajduje się w słowniku
- iterowanie po elementach słownika
- wykonywanie na słowniku zapytań w stylu LINQ takich jak:
    - SELECT
    - FROM
    - WHERE
    - ORDER BY

```
dict =
```

```
{
   ("value", 2),
   ("second_value", 3),
   (“third_value”, 0.5)
}

names =
{
   ("girl1", "Anna"),
   ("boy1", "Tom)"
}

data =
{
    ("name", "Tom"),
    ("age", 20)
}
```

```
dict =
{
    "value" = 1
}

dict.add("new_value", 2);
dict.add("another_value", 5);
<#
dict =
{
    ("value", 1),
    ("new_value", 2),
    ("another_value", 5)
}
#>

dict["new_value"] = 3;
<#
dict =
{
    ("value", 1),
    ("new_value", 3),
    ("another_value", 5)
}
#>
```

```
print(dict["new_value"]); # 3

dict.remove("new_value");

<#
dict =
{
    ("value", 1),
    ("another_value", 5)
}
#>


result = dict.contains_key("value");
print(result);  # True

result = dict.contains_key("value2");
print(result);  # False

for pair in dict
{
    print(pair.Key);
    print(" ");
    print(pair.Value);
    print(" ");
}
# value 1 another_value 5

print(dict) # {("value", 1), ("another_value", 5)}
```

```
a = 1;
b = 2;
list = [1, 2, 3]
dict_inside =
{
   (1, "one"),
   (2, "two")
}

fun increase(x)
{
    return x + 1;
}
```

```
dict =
{
    ("value", increase(a)),
    (b, list),
    (5, dict_inside)
}

print(dict["value"]); # 2
print(dict[b]); # [1, 2, 3]
print(dict[5][1]); # one
```

LINQ

Przykłady języka dla SELECT, FROM

Zwracana kopia wartość w liście nie zmienia się pod wpływem zmiany na słowniku

```
dict =
{
    ("one", 1),
    ("two", 2),
    ("three", 3)

}

result_select = SELECT Key
                FROM dict;
print(result_select); # ["one", "two", "three"]

result_select = SELECT Value
                FROM dict;
print(result_select); # [1, 2, 3]

result_select = SELECT (Value += 1)
                FROM dict;
print(result_select); # [2, 3, 4]


dict[“one”] = 3;
print(result_select); # [2, 3, 4]
```

```
a = 1;
b = 2;
list = [1, 2, 3]
dict_inside =
```

```
{
   (1, “one”),
   (2, “two)
}

dict =
{
    ("value", a),
    (b, list),
    (dict_inside, 5)
}

result_select = SELECT Key
                FROM dict;
print(result_select); # ["one", "two", "three"]
```

Przykłady języka dla WHERE

```
dict =
{
    ("one", 1),
    ("two", 2),
    ("three", 3)

}

result = SELECT (Key,Value)
         FROM dict
         WHERE (Value % 2 == 1);
print(result); # [ ("one", 1), ("three", 3) ]

result = SELECT (Key,Value)
         FROM dict
         WHERE (Key.Length() == 3);
print(result); # [ ("one", 1), ("two", 2) ]
```

Przykłady języka dla ORDER BY

```
dict =
{
    ("one", 1),
    ("two", 2),
```

```
    ("three", 3)

}

result = SELECT (Value)
         FROM dict
         ORDERBY (Value) DESC
print(result); # [ 3, 2, 1 ]

result = SELECT (Key,Value)
         FROM dict
         WHERE (Key.Length() == 3)
         ORDERBY (Key) ASC
print(result); # [ ("one", 1), ("three", 3), ("two", 2) ]
```

Przykłady mieszane

```
a = 1;
b = 2;
list = [1, 2, 3]
dict_inside =
{
   (a, "one"),
   (b, "two")
}

dict_inside2 =
{
   (a, "one2"),
   (b, "two")
}

dict =
{
    (a, 1),
    ("two", b),
    (list, 3),
    ("four", dict_inside)
}

result_select = SELECT (Key)
                FROM dict;

print(result_select); # [1, "two", [1, 2, 3], "four"];
```

```
dict =
{
    ("three", dict_inside),
    ("four", dict_inside2)
}

result_select = SELECT (Value[a])
                FROM dict;
                WHERE ( Value.b == "two" )

print(result_select); # ["one", "one2"]
```

Konwersja typów

Typ wejściowy	Typ wyjściowy	Funkcja wbudowana	Sposób konwersji	Przykład

int	float	.ToFloat()	Dodanie do liczby całkowitej “.0”	x = 2;
				x.ToFloat();
				print(x); # 2.0

float	int	.ToInt()	Odcięcie części ułamkowej	x = 2.6;
				x.ToInt();
				print(x); # 2

int	string	.ToString()	Zmiana liczby na tekst	x = 2;
				x.ToString();
				print(x); # 2

float	string	.ToString()	Zmiana liczby na tekst	x = 2.5;
				x.ToString();
				print(x); # 2.5

string	int	.ToInt()	Zmiana tekstu na liczbę całkowitą	x = “2”; x.ToInt(); print(x); # 2
string	float	.ToFloat()	Zmiana tekstu na liczbę zmiennoprzecinkową	x = “2.5”; x.ToFloat(); print(x); # 2.5

Obsługa błędów

Przykładowe typu komunikatów o błędach rozróżniamy na błędy leksykalne, składniowe i semantyczne. Każdy błąd skutkuje wstrzymaniem wykonania programu.

Błędy leksykalne

1) Próba utworzenia bardzo dużej liczby całkowitej wykraczającej poza zakres

```
LexicalError: cannot fit ‘int’ into an index-sized integer
File "hello_world.dx", line 2, col 3
    a = 1131313133133.....
```

2) Próba utworzenia bardzo dużego stringu wykraczającego poza zakres

```
LexicalError: cannot fit 'string' into an index-sized string
File "hello_world.dx", line 5, col 3
    a = “aaaaaaaaaaaaaaaa.....”
```

3) Podanie niezdefiniowanego znaku

```
LexicalError: illegal character
File "hello_world.dx", line 5, col 20
    print("Hello world!"); $
                           ^
```

4) Nieprawidłowa nazwa - identifier nie może się zaczynać cyfrą

```
LexicalError: spelling error
File "hello_world.dx", line 5, col 1
    3number = 3;
    ^
```

Błędy składniowe

5) Brak nawiasu zamykającego

```
SyntaxError: invalid syntax
File "hello_world.dx", line 5, col 10
    print("Hello world!";
                        ^
```

6) Brak średnika

```
SyntaxError: invalid syntax expected ';'
File "hello_world.dx", line 5, col 7
for i in list
{
    print(i)
             ^
}
```

7) Brak nawiasu otwierającego

```
SyntaxError: invalid syntax expected '{'
File "hello_world.dx", line 5, col 2
for i in list
    print(i);
   ^
}
```

8) Znak przypisania zamiast porównania

```
SyntaxError: invalid sign '=' expected '=='
File "hello_world.dx", line 5, col 4
if ( i = 2 )
       ^
{
    print("a");
}
```

9) Brak znaku FROM po znaku kluczowym SELECT

```
TypeError: expected keyword FROM in LINQ statement
File "hello_world.dx", line 25, col 8
    result = SELECT (Key,Value)
             WHERE (Key.Length() == 3);
             ^
```

Błędy semantyczne

10) Próba wykorzystania zmiennej przed jej definicją

```
NameError: name 'x' is not defined
File "hello_world.dx", line 5, col 2
    print(x);
```

11) Próba wykorzystania funkcji przed jej definicją

```
NameError: function name 'add' used but not defined
File "hello_world.dx", line 5, col 2
    add(x);
```

12) Wywołanie funkcji z niewystarczającą liczbą argumentów

```
TypeError: function add() is missing 1 required positional argument
File "hello_world.dx", line 25, col 6
    number = add(1);
```

13) Wywołanie funkcji z liczbą argumentów większą niż zdefiniowaną

```
TypeError: function add() takes 2 positional arguments but 3 were given
File "hello_world.dx", line 25, col 5
    number = add(1,2,3);
```

14) Próba odwołania się do argumentu, którego nie ma w obiekcie

```
AttributeError: 'dict' object has no attribute 'some_attribute')
File "hello_world.dx", line 25, col 6
    print(dict.some_attribute)
```

15) Próba iteracji po obiekcie, po którym nie da się iterować

```
TypeError: 'int' object is not iterable
File "hello_world.dx", line 25, col 5
   a = 5
   for i in a
       ^
   {
        print(i);
   }
```

16) Wywołanie rzutowania na obiekcie który nie da się zrzutować

```
TypeError: cannot convert list to float
File "hello_world.dx", line 25, col 5
    list.ToFloat();
```

17) Próba dzielenia przez zero

```
ZeroDivisionError: division by zero
File "hello_world.dx", line 12, col 9
    result = 10 / 0;
```

Gramatyka

```
program         ::==      {statement};

statement       ::==      for_loop_term
                          | while_loop_term
                          | fun_def_term
                          | return_term
                          | select_term
                          | assign_or_call
                          | if_term;

for_loop_term   ::== 'for' identifier 'in' expression '{' {statement} '}';

while_loop_term ::== 'while' '(' expression ')' '{' {statement } '}';


fun_def_term    ::== 'fun' identifier '(' [ identifier {',' identifier} ] ')'
                          '{' {statement}'}';

return_term     ::== ‘return’ expression ‘;’;

select_term     ::== 'SELECT' expression 'FROM' expression
                          [ 'WHERE' expression ]
                          [ 'ORDER BY' expression ['ASC' | 'DESC'] ] ‘;’;

assign_or_call  ::== object_access ['=' expression] ‘;’ ;

if_term         ::== ‘if’ ‘(‘ expression ‘)’ ‘{‘ {statement} ‘}’
                          {else_if_term} [else_term];

else_if_term    ::== ‘else_if’ ‘(‘ expression ’)’ ‘{' {statement} '};

else_term       ::== 'else' '{' {statement} '}';

expression      ::== and_term { ‘Or’ and_term}
and_term        ::== not_term { ‘And’ not_term};
not_term        ::== [‘Not’] comparison_term;


comparison_term ::== additive_term [('=='|'!='|'<'|'>'|'<='|'>=') additive_term];

additive_term   ::== mult_term {('+' | '-') mult_term};
mult_term       ::== signed_factor {('*' | '/') signed_factor}
signed_factor   ::== [sign] factor ;

factor          ::== pair_or_expr
                          | literal
                          | list_def
                          | dict_def
```

```
                        | object_access

pair_or_expr    ::==   ‘(‘ expression [‘,' expression ] ‘)’


list_def        ::== '[' [ expression {‘,’ expression} ] ']' ;
dict_def        ::== '{' [ expression {‘,’ expression} ]  '}';


object_access   ::== item {‘.’ item};
item            ::== identifier {  ‘[‘ expression ‘]’ |  ‘(‘ [arguments] ‘)’ };
arguments       ::== expression {‘,’ expression};


sign            ::== ‘-’

literal         ::== number | bool | string;

identifier      ::== letter {alphanumeric};

string          ::== '"' {character} '"';
character       ::== alphanumeric | string_escape ;

alphanumeric    ::== digit | letter;

string_escape   ::== '$$' | '$n' | '$t' | '$"' | '$\'

bool            ::== ‘True’ | ‘False’

number          ::== integer [.{digit}]
integer         ::== ‘0’ | ( digit_non_zero { digit } )
digit           ::== digit_non_zero | ‘0’
digit_non_zero  ::== '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

letter          ::==  [A-Z] | [a-z] | [other_unicode_characters]
```

Podział na moduły

Moduł odczytu

Moduł odpowiedzialny za pobieranie pojedynczo znaków z różnych źródeł. Przechowuje informację o pozycji obsługiwanego obecnie znaku. Pomija białe znaki. Przygotowuje dane by przekazać je lekserowi.

Analizator leksykalny

Moduł odpowiedzialny za przetwarzanie odczytanych danych, wyodrębnienie znaków i pogrupowanie ich w atomy leksykalne - tokeny.
Tokeny będą reprezentowane za pomocą struktury Token. Przechowuje ona typ tokenu oraz jego pozycję w źródle - numer wiersza i początek tokenu.

Analizator składniowy

Moduł odpowiedzialny za analizę składniową listy tokenów w celu utworzenia drzewa składniowego AST, które odzwierciedla strukturę programu. Sprawdza również czy tokeny pojawiające się na wejściu parsera tworzą poprawne konstrukcje składniowe.
Drzewo składniowe będą budować klasy i funkcje odpowiadające poszczególnym regułom gramatyki takie jak:

- Statement
- AssignmentTerm
- IfTerm
- ElseIfTerm
- ElseTerm
- ForLoopTerm
- WhileLoopTerm
- FunctionDefinitionTerm
- FunctionCallTerm
- ListAccess
- ObjectAccess
- Dictionary
- Pair
- List
- LINQ
- Expression
- LogicalTerm
- ComparisonTerm
- AdditiveTerm
- MultiplicationTerm
- Identifier

Interpreter

Moduł zajmujący się interpretacją drzewa składniowego i wykonywaniem programu. Zdefiniowane klasy i funkcje będą odpowiedzialne za wykonanie poszczególnych instrukcji, wyrażeń logicznych, operacji na listach, słownikach. W module będzie także struktura przechowująca zdefiniowana funkcje zarówno od użytkownika jak i wbudowane. Do realizacji zostanie zastosowany wzorzec odwiedzającego.

Tokeny

Analizator leksykalny identyfikuje tokeny z poniższej listy.

Planowane możliwe tokeny:

- Słowa kluczowe:
  - for
  - while
  - if
  - else_if
  - else
  - in
  - True
  - False
  - return
  - fun
  - SELECT
  - FROM
  - WHERE
  - ORDER BY
- Operatory logiczne:
  - And
  - Or
  - Not
- Operatory arytmetyczne
  - Dodawanie
  - Odejmowanie
  - Mnożenie
  - Dzielenie
  - Negacja
- Nawiasy
  - Nawias zwykły otwierający
  - Nawias zwykły zamykający
  - Nawias kwadratowy otwierający
  - Nawias kwadratowy zamykający
  - Nawias klamrowy otwierający
  - Nawias klamrowy zamykający

- Operatory porównawcze
    - Mniejszy
    - Mniejszy lub równy
    - Większy
    - Większy lub równy
    - Równy
    - Nie równy
- Typy:
    - String
    - Int
    - Float
    - Identyfikator
- Inne
    - Przypisanie
    - Średnik
    - EndOfText

Przykład uruchomienia

Przykład uruchomienia kodu Dixty

```
python interpreter.py source.dx [another_files.dx]
```

Biblioteki do testowania

Kod będzie testowany za pomocą testów jednostkowych i akceptacyjnych. Testy będą tworzone za pomocą biblioteki pythonowej pytest.