Dokumentacja

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 >= 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.

```
# To 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;
    return a;
}

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(element) - usunięcie pierwszego napotkanego takiego elementu
- 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(3); # 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('b'); # list_str = ['a', 'c', '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
}
```

Zagnieżdżanie 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([7, 8, 9]);
# 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[0]); # value
print(pair[1]); # 2
```

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

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 == 1);
print(result); # [ ("one", 1)) ]

result = SELECT (Key,Value)
         FROM dict
         WHERE (Key.Len() == 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	Sposób	Przykład

		wbudowana	konwersji	

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

```
IntLimitExceeded: Attempted to create integer bigger then maximum range
[9223372036854775807]
Invalid integer position: row: [1], column: [5]

    a = 92233720368547758077567
```

2) Próba utworzenia bardzo dużego stringu wykraczającego poza zakres

```
StringLimitExceeded: Attempted to create string bigger then maximum
range: [9223372036]
```

```
Invalid string position: row: [1], column: [5]

a = “aaaaaaaaaaaaaaaa.....”
```

3) Próba utworzenia bardzo dużego identyfikatora wykraczającego poza zakres

```
IdentifierLimitExceeded: Attempted to create identifier bigger then
maximum range: [1000]
Invalid string position: row: [1], column: [1]

asdasdadsasdasdasdasdasdadsasdasdasd = 1
```

4) Nie podanie cudzysłowu, który zamknie string

```
StringNotFinished: Could not find closing " to the opened string
Not closed string position: row: [1], column: [5]

a = “aaaaa
```

5) Podanie nieistniejącego znaku po znaku ucieczki

```
UnexpectedEscapeCharacter: Invalid character [%] after escape sign.
Expected character: 'n', 'r', 't', '\\'
Invalid escape character position: line 5, col 20

print("Hello\%");
```

6) Podanie niezdefiniowanego znaku

```
TokenNotRecognized: Token starting with character [$] was not recognized
Invalid token position: {position} line 5, col 20

print("Hello world!"); $
```

Błędy składniowe

7) Brak nawiasu zamykającego lub innego oczekiwanego statementu. Gdy błąd występuje w statemencie, wyświetlany jest dziedziczący błąd z nazwą statementu:
    - InvalidFunctionDefinition
    - InvalidWhileLoop
    - InvalidForLoop
    - InvalidIfStatement

- InvalidElseIfStatement
- InvalidElseStatement
- InvalidReturnStatement
- InvalidAssignmentStatement

```
MissingExpectedStatement: Expected token [;], received token [(] at line
5, col 10

print("Hello world!";
                      ^
```

8) Brak średnika na końcu wyrażenia

```
SemicolonMissing: Missing semicolon after end of statement at position
row: [1], column: [4]

for i in list
{
    print(i)
             ^
}
```

9) Brak nawiasu otwierającego

```
InvalidIfStatement: Expected token [TokenType.BRACKET_OPENING], received
token [TokenType.INT] at row: [1], column: [4]
if i == 2
{
    print("a");
}
```

10) Znak przypisania zamiast porównania

```
InvalidIfStatement: Expected token [TokenType.BRACKET_CLOSING], received
token [TokenType.ASSIGN] at row: [1], column: [7]
if ( i = 2 )
{
    print("a");
}
```

11) Brak znaku FROM po znaku kluczowym SELECT

```
MissingExpectedStatement: Expected token [TokenType.FROM], received
```

```
token [TokenType.WHERE] at row: [1], column: [29]
    result = SELECT (Key,Value)
             WHERE (Key.Length() == 3);
```

12) Próba zdefiniowania funkcji o nazwie która już istnieje

```
FunctionAlreadyExists: Function with name [display] is already defined.
Tried to define again at position: row: [1], column: [29]

fun display()
{
   print(1);
}
```

13) Błędny typ w słowniku

```
DictInvalidElement: Elements in Dict must be Pair type. Object with
invalid type int at row: [1], column: [29]

a = {1};
```

14) Podanie wyrażenia które nie jest żadnym zdefiniowanym wyrażeniem

```
UsedNotRecognizedStatement: Not recognized statement detected at [1],
column: [29]

2 + 2;
```

Błędy semantyczne

15) Próba wykorzystania zmiennej przed jej definicją

```
VariableNotExists: Tried to use not defined variable: [x] at row [1],
column: [29]
    print(x);
```

16) Próba wykorzystania funkcji przed jej definicją

```
FunctionNotDeclared: Tried to use not defined function: [add] at row
```

```
[1], column: [29]
    add(x);
```

17) Wywołanie funkcji z nieodpowiednią liczbą argumentów

```
IncorrectArgumentsNumber: Got incorrect number of parameters for
function [add], expected: [2], received: [1] - at row [1], column: [29]
    number = add(1);
```

18) Próba wykonania operacji dodawanie, odejmowanie, mnożenie, dzielenie na nieodpowiednich typach obiektów, odpowiednio dla operacji zdefiniowane dziedziczące po UnsupportedTypesToMakeOperation errory:
    - CannotAddUnsupportedTypes
    - CannotSubUnsupportedTypes
    - CannotMultUnsupportedTypes
    - CannotDivUnsupportedTypes

```
CannotAddUnsupportedTypes: Cannot make operation [add] on given types:
[int] and [float], at row [1], column: [29]
    print(2 + 3.0);
```

19) Próba wykonania operacji porównywania na typach obiektów, które nie są tego samego typu :

```
CannotCompareUnsupportedTypes: Cannot make operation [add] on given
types: [int] and [float], at row [1], column: [29]
    print(2 < 3.0);
```

20) Próba wykonania operacji Or, And, Not na obiektach, które nie są boolami, odpowiednio dla operacji zdefiniowane dziedziczące po UnsupportedTypesToMakeOperation errory:
    - CannotMakeOrOnNotBoolTypes
    - CannotMakeAndOnNotBoolTypes
    - CannotMakeNotOnNotBoolTypes

```
CannotMakeAndOnNotBoolTypes: Cannot make operation [And] on given types:
[int] and [float], at row [1], column: [29]
    print(2 And 3.0);
```

21) Wywołanie rzutowania na obiekcie który nie da się zrzutować

```
CannotConvertType: Cannot convert from [list] to [float] at row [1],
column: [29]
    list.ToFloat();
```

22) Próba dodania do słownika klucza który już istnieje

```
AlreadyExistingDictKey: Tried to add key to dict which already exists:
[“one”] at row [1], column: [29]
```

23) Próba usunięcia klucza ze słownika który nie istnieje

```
NotExistingDictKey: Tried to remove key which not exist in dict: [“one”]
at row [1], column: [29]
```

24) Próba usunięcie wartości która nie istnieje z listy

```
NotExistingListValue: Tried to remove value which not exist in list:
[“one”] at row [1], column: [29]
```

25) Próba wywołania metody na obiekcie który nie ma tej metody

```
AttributeError:Tried to access attribute [append] not available in
object type: [int] at row [1], column: [29]
```

26) 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);
   }
```

27) Przekroczono limit rekursji

```
RecursionLimitExceeded: Exceeded recursion max limit [1000] at row [1],
column: [29]
```

Gramatyka

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

statement       ::==      for_loop_statement
                          | while_loop_statement
                          | fun_def_statement
                          | return_statement
                          | assign_or_call
                          | if_statement;

for_loop_statement   ::== 'for' identifier 'in' expression block;

while_loop_statement ::== 'while' '(' expression ')' 'block;


fun_def_statement    ::== 'fun' identifier '(' [ parameters ] ')' block;

return_statement     ::== ‘return’ expression ‘;’;


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


if_statement    ::== ‘if’ ‘(‘ expression ‘)’ block
                          {else_if_statement} [else_statement];

else_if_statement      ::== ‘else_if’ ‘(‘ expression ’)’ block;

else_statement       ::== 'else' 'block;

parameters   ::== identifier {',' identifier}

block        ::== '{' {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
```

```
                        | select_term
                        | object_access;

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


list_def        ::== '[' [ expressions_list ] ']' ;
dict_def        ::== '{' [ expressions_list ]  '}';

expressions_list     ::== expression {‘,’ expression}

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


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

sign            ::== ‘-’

literal         ::== number | bool | string;

identifier      ::== letter {alphanumeric};

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

alphanumeric    ::== digit | letter;

string_escape   ::== '\n' | '\t' | '\"' | '\\' | ‘\r’

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

Obsługuje odczytywanie tekstu z abstrakcyjnego źródła zdefiniowanego w klasie Source - może to być plik lub String. Radzi sobie również ze znakami spoza Unicode. Obsługuje kilka rodzajów znaku nowej linii takich jak: ‘\n’, ‘\n\r” oraz “\r\n” . Obsługiwane znaki nowej linii (oprócz podstawowego ‘\n’) zostały zdefiniowane w słowniku NEWLINE w pliku standards.py.

W pliku standards.py jest również słownik STRING_ESCAPE przechowujący znaki specjalne w stringu, oraz są zdefiniowane stałe globalne ETX oraz EOL oznaczające jak lekser oznacza znak końca linii i końca tekstu.

Dodatkowo została stworzona klasa Position, służąca do przechowywania kolumny i wiersza danego elementu.

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 wartość, typ tokenu oraz jego pozycję w źródle - numer wiersza i początek tokenu.

Słowa kluczowe są przechowywane w słowniku KEYWORDS w pliku keywords.py.
Operatory przechowywane są w słowniku OPERATORS w pliku operators.py

Analizator leksykalny identyfikuje tokeny z poniższej listy. Zdefiniowane są one w klasie TokenType w pliku token_types.py

Planowane możliwe tokeny:

- Słowa kluczowe:
    - for
    - while
    - if
    - else_if
    - else
    - in
    - True
    - False
    - return
    - fun
    - SELECT
    - FROM
    - WHERE
    - ORDER BY
    - ASC
    - DESC

- Operatory logiczne:
    - And
    - Or
    - Not
- Operatory arytmetyczne
    - Dodawanie (PLUS)
    - Odejmowanie (MINUS)
    - Mnożenie (ASTERISK)
    - Dzielenie (SLASH)
- Nawiasy
    - Nawias zwykły otwierający (BRACKET_OPENING)
    - Nawias zwykły zamykający (BRACKET_CLOSING)
    - Nawias kwadratowy otwierający (SQUARE_BRACKET_OPENING)
    - Nawias kwadratowy zamykający (SQUARE_BRACKET_CLOSING)
    - Nawias klamrowy otwierający (BRACE_OPENING)
    - Nawias klamrowy zamykający (BRACE_CLOSING)
- Operatory porównawcze
    - Mniejszy (LESS)
    - Mniejszy lub równy (LESS_OR_EQUAL)
    - Większy (MORE)
    - Większy lub równy (MORE_OR_EQUAL)
    - Równy (EQUAL)
    - Nierówny (NOT_EQUAL)
- Operatory inne
    - Kropka (DOT)
- Typy:
    - String
    - Int
    - Float
    - Identyfikator
- Inne
    - Przypisanie (ASSIGN)
    - Średnik (SEMICOLON)
    - Przecinek (COMMA)
    - Koniec tekstu (END_OF_TEXT)
    - Komentarz (COMMENT)

Analizator składniowy

Moduł odpowiedzialny za analizę składniową listy tokenów w celu utworzenia drzewa składniowego, 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:

- Program
- ForStatement
- WhileStatement

- FunStatement
- ReturnStatement
- IfStatement
- ElseIfStatement
- ElseStatement
- OrTerm
- AndTerm
- NotTerm
- ComparisonTerm
- EqualsTerm
- LessTerm
- MoreTerm
- LessOrEqualTerm
- MoreOrEqualTerm
- AddTerm
- SubTerm
- MultTerm
- DivTerm
- SignedFactor
- Number
- String
- Bool
- List
- Pair
- Dict
- ObjectAccess
- Item
- IndexAccess
- FunCall
- Identifier
- Assignment
- Block
- SelectTerm

Dodatkowo została utworzona klasa printer.py pozwalająca wypisać utworzone przez parser drzewo składniowe.

Interpreter

Moduł zajmujący się interpretacją drzewa składniowego i wykonywaniem programu. Zdefiniowane klasy i funkcje a odpowiedzialne za wykonanie poszczególnych instrukcji, wyrażeń logicznych, operacji na listach, słownikach.

Funkcje wbudowane są zdefiniowane i przechowywane w słowniku BUILTINS oraz w pliku builtins.py. Funkcje wbudowane wraz z funkcjami zdefiniowanymi przez użytkownika są przechowywane w polu self._functions klasy Interpreter.

Do realizacji został zastosowany wzorzec wizytatora. Klasa wizytatora jest zdefiniowana w pliku parser/visitor.py

Tworzone zmienne są przechowywane w obszarach w odpowiednich kontekstach zdefiniowanych w klasie Context w pliku context.py. Kolejny kontekst tworzony jest w momencie wywołania funkcji, a usuwany jest w momencie wyjścia z niej.
W każdy kontekście przechowywany jest stos obszarów zdefiniowanych w klasie Scope w pliku scope.py.

Przykład uruchomienia

Przykład uruchomienia kodu Dixty:

```
python dixty.py source.dx
```

Możliwe flagi do dodania:

- –int-limit - liczba, maksymalna wielkość liczby którą można utworzyć
- –string-limit - liczba, maksymalna wielkość napisu który można utworzyć
- –identifier-limit - liczba, maksymalna wielkość identyfikatora który można utworzyć
- –recursion-limit - liczba, maksymalna ilość możliwych rekursji (domyślnie 200)
- –source-type - ‘file’ lub ‘source’ - zdefiniowanie jakiego typu jest podawane źródło, domyślnie jest plik

Przykład uruchomienia z podanymi flagami ustawiającymi limity

```
python dixty.py test_file_interpreter.dx --int-limit 300 --string-limit
200 --identifier-limit 100 --source-type file
```

Testowanie

Kod był testowany za pomocą testów jednostkowych i akceptacyjnych. Testy były tworzone za pomocą biblioteki pythonowej pytest.
Testy zostały podzielone na foldery testujące odpowiednie moduły. Testy parsera i interpretera, zostały dodatkowo podzielone na jednostkowe i integracyjne. Struktura testów:

```
tests/
      interpreter/
```

```
            integration_tests/
            unit_tests/
      lexer/
      parser/
            integration_tests/
            unit_tests/
```

Całkowita ilość testów jednostkowych i integracyjnych w całym interpretatorze Dixty wynosi 444.