AGH
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Robert Cebula
kierunek studiów: **informatyka stosowana**

Opracowanie i implementacja języka programowania z wbudowanym mechanizmem wzorca obserwatora

Opiekun: **dr inż. Antoni Dydejczyk**

Kraków, styczeń 2017

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

……..

Merytoryczna ocena pracy przez opiekuna:

Merytoryczna ocena pracy przez recenzenta:

Spis treści

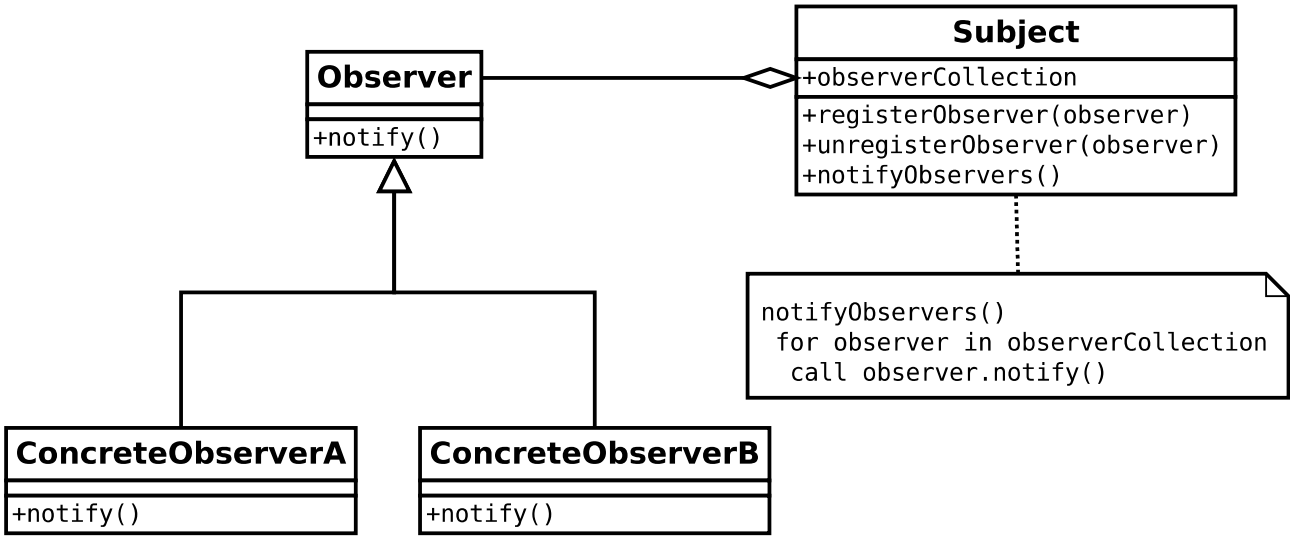
1. Wstęp

Celem niniejszej pracy jest zaprojektowanie i zaimplementowanie od podstaw języka programowania z wbudowanym mechanizmem wzorca obserwatora. Wzorzec ten jest bardzo przydatny w sytuacji gdy w naszym programie zachodzi relacja jeden do wielu pomiędzy obiektami. Wiele współczesnych języków programowania umożliwia zaimplementowanie tego wzorca, ale odbywa się to dodatkowym nakładem pracy ze strony programisty. Także czytelność takich rozwiązań jest niezadowalająca. Język BIO (taka jest nazwa opisywanego tutaj języka, pochodzi ona od pierwszych liter angielskich słów “built in observer” (tłum. wbudowany wzorzec obserwatora)) ułatwia proces tworzenia programów opartych o ten wzorzec dostarczając wbudowany mechanizm jego implementowania. Porównanie składni kilku współczesnych języków programowania i języka BIO w przypadku implementacji wzorca obserwatora znajduje się w rozdziale 2. (*Porównanie implementacji wzorca obserwatora w różnych językach programowania*). Planowana funkcjonalność języka została opisana w rozdziale 3. (*Planowana funkcjonalność języka*). Opis podstawowych elementów semantyki znajduje się w rozdziale 4. (*Semantyka*). Rozdział 5. (*Realizacja*) zawiera szczegóły implementacji kompilatora i interpretera. W rozdziale 6. (*Przykład użycia*) przedstawiony został kompletny przykład użycia języka od instalacji, poprzez napisanie programu, kompilację aż po uruchomienie. Na końcu pracy w rozdziale 7. (*Zrealizowana funkcjonalność języka*) omówiona została funkcjonalność języka jaką udało się zrealizować.

2. Porównanie implementacji wzorca obserwatora w różnych językach programowania

2.1 Opis wzorca

Wzorzec obserwatora jest wzorcem w którym jeden obiekt zwany tematem (ang. Subject) informuje inne zainteresowane nim obiekty zwane obserwatorami (ang. Observer) o zmianie stanu pewnego innego obiektu.



Rys.2.1 Schemat wzorca w przypadku obiektowego języka programowania (źródło: wikipedia.org)

2.2 Java

W javie do implementacji wzorca obserwatora można posłużyć się klasami z biblioteki standardowej: java.util.Observer i java.util.Observable. Żeby stworzyć temat dziedziczymy po klasie Observable. Posiada ona między innymi dwie metody: setChanged() i notifyObservers(). Pierwsza oznacza obiekt jako zmieniony, a druga informuje wszystkich obserwatorów o zmianie. Następnie dziedziczymy po klasie Observer, która posiada jedną metodę update() wywoływaną w momencie informowania obserwatora przez temat (metoda notifyObservers).

```
import java.util.Observable;
import java.util.Scanner;

class EventSource extends Observable implements Runnable {
    public void run() {
        while (true) {
            String response = new Scanner(System.in).next();
            setChanged();
            notifyObservers(response);
        }
    }
}
```

```
import java.util.Observable;
import static java.lang.System.out;

class MyApp {
    public static void main(String[] args) {
        out.println("Enter Text >");
        EventSource eventSource = new EventSource();

        eventSource.addObserver( (Observable obj, Object arg) -> {
            out.println("\nReceived response: " + arg);
        });

        new Thread(eventSource).start();
    }
}
```

Rys.2.2 Przykład implementacji w javie (źródło: wikipedia.org). Zamiast dziedziczenia po klasie Observer została użyta funkcja lambda

2.3 C#

W języku C# możemy użyć podobnego rozwiązania jak w javie (zasadniczo dla każdego języka obiektowego można tak zrobić), ale poza tym możemy zastąpić klasę Observable i Observator zdarzeniami. Zdarzenie w języku C# (ang. Event) to coś podobnego do wskaźnika do funkcji z języka C/C++, czyli obiekt przechowujący referencję do funkcji. Typy przyjmowanych i zwracanego argumentu określa delegacja (ang. Delegate). Najpierw tworzymy delegację, która opisuje nam sygnaturę przyjmowanych funkcji. Następnie tworzymy zdarzenie, które będzie naszym tematem. Dodajemy do zdarzenia funkcję będące obserwatorami obserwatorów. Od teraz w momencie wywołania zdarzenia, wywołane zostaną wszystkie przypisane do niego funkcje.

```
1  namespace SimpleEvent
2  {
3      public delegate void someEventDelegate(string param);
4      public class EventTest
5      {
6          private event someEventDelegate someEvent;
7
8          public void addObserver(someEventDelegate observer)
9          {
10             someEvent += observer;
11         }
12
13         public void removeObserver(someEventDelegate observer)
14         {
15             someEvent -= observer;
16         }
17
18         public void notifyObservers(string str)
19         {
20             if (someEvent != null)
21             {
22                 someEvent(str);
23             }
24         }
25     }
26
27     public class MainClass
28     {
29         public static void Main()
30         {
31             EventTest et = new EventTest();
32
33             et.addObserver(delegate (string param)
34             {
35                 System.Console.WriteLine("metoda 1: " + param);
36             });
37
38             et.addObserver(delegate (string param)
39             {
40                 System.Console.WriteLine("metoda 2: " + param);
41             });
42
43             et.notifyObservers("hello");
44         }
45     }
46 }
```

Rys.2.3 Przykład implementacji w języku C# (źródło: własne)

2.4 Python

Następny przykład będzie implementacją w języku python z wykorzystaniem kilku cech tego języka: dekoratorów, wykorzystaniu faktu, że każda funkcja jest obiektem, więc możemy do niej przypisać zmienne oraz przekazywanie do funkcji listy wartości(*args) oraz słownika kluczy-wartości(**kwargs). Implementacja ta zaoferuje funkcjonalność bardzo podobną do funkcjonalności w języku BIO, tzn. dla każdej funkcji dla której użyjemy dekoratora @Event będzie możliwe dodanie i usunięcie funkcji obserwatora. Użycie *args i **kwargs umożliwi nam wykorzystanie dekoratora @Event z funkcjami o dowolnej liczbie parametrów. Dekorator @Event tworzy funkcję, która przy wywołaniu wywołuje funkcję pierwotną, a następnie wywołuje wszystkich obserwatorów z tymi samymi parametrami. Dodatkowo dodaje do tej funkcji listę obserwatorów.

```
def attach_to_event(subject, observer):
    subject.observers.append(observer)

def detach_from_event(subject, observer):
    subject.observers.remove(observer)

def event(func):
    def inner(*args, **kwargs):
    |   func(*args, **kwargs)
    |   for observer in inner.observers:
    |   |   observer(*args, **kwargs)

    inner.observers = []
    return inner

@event
def some_event(data, data2):
    print("some_event " + str(data) + str(data2))

@event
def foo(data, data2):
    print("foo " + str(data) + str(data2))

def foo2(data, data2):
    print("foo2 " + str(data))

attach_to_event(some_event, foo)
attach_to_event(foo, foo2)
some_event("test", " wiadomosc")
```

Rys.2.4 Przykład implementacji w pythonie, która daje bardzo podobną funkcjonalność jak ta w języku BIO (źródło: własne)

2.5 BIO

Poniżej przedstawiona będzie implementacja wzorca obserwatora w języku BIO. Język BIO charakteryzuje się tym, że każdej funkcji użytkownika (tzn. zdefiniowanej w kodzie BIO) możemy przypisać inną funkcję użytkownika będącą obserwatorem. Żeby to osiągnąć wywołujemy funkcję ATTACH_TO_EVENT, której podajemy temat i obserwatora. Ilość parametrów przyjmowanych przez funkcję obserwatora musi być taka sama lub mniejsza jak ilość przyjmowana przez funkcję tematu.

```
def some_event(data, data2)
   PRINT("some_event", TO_STR(data), TO_STR(data2), "\n")
end

def foo(data, data2)
   PRINT("foo", TO_STR(data), TO_STR(data2), "\n")
end

def foo2(data)
   PRINT("foo2", TO_STR(data), "\n")
end

def onSTART()
   ATTACH_TO_EVENT(some_event, foo)
   ATTACH_TO_EVENT(some_event, foo2)
   some_event("test", "wiadomość")
   DETACH_FROM_EVENT(some_event, foo)
   some_event("test", "wiadomość")
end
```

Rys.2.5 Przykład implementacji w języku BIO (źródło: własne)

3. Planowana funkcjonalność języka

Poniżej w punktach przedstawiona została planowana funkcjonalność języka:

1. Dwuetapowy proces tworzenia i uruchamiania programu – kompilator tworzy kod pośredni, który następnie jest interpretowany i wykonywany przez interpreter.

2. Wbudowany wzorzec obserwatora – mechanizm pozwalający w łatwy sposób implementować programy oparte o ten wzorzec projektowy.

3. Dynamicznie typowany – typy zmiennych nie są znane podczas kompilacji programu.

4. Zorientowany na programowanie proceduralne – możliwość tworzenia własnych funkcji.

5. 10 wbudowanych typów danych - “array” (tablica), “bool” (wartość logiczna), “dict” (kolekcja par: <”string”, wartość>), “error” (błąd), “float” (liczba zmiennoprzecinkowa pojedynczej precyzji w formacie “IEEE-754”), “int” (liczba całkowita ze znakiem 32 bitowa, w formacie kodu uzupełnień do dwóch), “none” (brak wartości), “string” (łańcuch znaków UTF-8), “struct” (struktura), “tuple” (krotka, odpowiednik tablicy ale bez możliwości zmiany jej elementów).

6. Błędy są wartościami – jeżeli w funkcji nastąpi błąd, zwraca ona wartość typu “error”.

7. Syntaktycznie wszystko jest funkcją – między innymi instrukcje sterujące przebiegiem programu jak IF i FOR.

8. Zmienne lokalne i globalne – możliwość zdefiniowania zmiennych lokalnych widocznych tylko dla danego wywołania funkcji lub zmiennych globalnych widocznych w każdej funkcji przez całe działanie programu.

9. Preprocesor – posiadający dwie dyrektywy: #INCLUDE pozwalającą na dołączenie innego pliku i dbającą o to, żeby ten sam plik nie został dołączony dwa razy i #IMPORT pozwalająca na dołączenie opcjonalnego modułu.

4. Semantyka

Każdy język programowania ogólnego przeznaczenia musi posiadać pewne podstawowe konstrukty, takie jak instrukcje przypisania wartości do zmiennej czy instrukcje warunkowe lub pętle. Poniżej przedstawiona została kluczowa semantyka języka BIO.

4.1 Przypisanie

Przypisanie wartości do zmiennej lokalnej odbywa się poprzez wywołanie funkcji ASSIGN_LOCAL (alias AS_LOC). Przypisanie wartości do zmiennej globalnej odbywa się przy użyciu funkcji ASSIGN_GLOBAL (alias AS_GLOB). Uzyskanie wartości zmiennej lokalnej uzyskujemy poprzez napisanie jej identyfikatora, a w przypadku zmiennej globalnej musimy do tego użyć funkcji GET_GLOBAL (alias GET_GLOB).

```
AS_LOC(a, 10)
AS_GLOB(a, "text")
PRINTLN(a)
PRINTLN(GET_GLOB(a))
```

4.2 Instrukcja warunkowa IF

Instrukcja umożliwiająca wykonanie warunkowe części kodu. Syntaktycznie instrukcja warunkowa IF to funkcja, która przyjmuje 3 parametry (3 jest opcjonalny):

IF (warunek, funkcja_jeżeli_prawda, funkcja_jeżeli_fałsz)

```
IF(flag, PRINTLN("true"), PRINTLN("false"))
```

4.3 Pętla FOR

Pętla umożliwiająca powtarzalne wykonywanie części kodu dopóki warunek jest spełniony. Syntaktycznie pętla FOR to funkcja przyjmująca 4 parametry (4 jest opcjonalny):

FOR (funkcja_na_początek, warunek, ciało_pętli, funkcja_inkrementacji)
funkcja_na_początek – funkcja wywoływana jeden raz na początku wywołania pętli FOR
warunek – wartość logiczna, jeżeli false to pętla jest przerywana
ciało_pętli – funkcja wykonywana za każdym obiegiem pętli FOR, **może** być pominięta przez funkcję CONTINUE().
funkcja_inkrementacji – funkcja wykonywana za każdym obiegiem pętli, **nie może** być pominięta przez funkcję CONTINUE().

Wewnątrz pętli FOR możemy używać dwóch specjalnych funkcji BREAK() i CONTINUE(). Pierwsza z nich przerywa działanie pętli FOR, druga powoduje ominięcie ciała_pętli i przejście do funkcji_inkrementacji.

```
FOR
(
  AS_LOC(i, 0),
  LS(i, 10),
  PRINTLN(i),
  INC(i)
)
```

4.4 Definicja funkcji

Definicja funkcji wygląda następująco:

def nazwa_funkcji(argumenty_wymagane, argumenty_domyślne)
ciało funkcji
end

Argumenty domyślne muszą znajdować się za argumentami wymaganymi. Ciało funkcji to lista wywołań funkcji oddzielona białym znakiem (tj. spacją, tabulaturą, komentarzem lub znakiem nowej linii).

```
def fun_name(arg1, arg2, arg3=10)
   PRINTLN(arg1)
   PRINTLN(ADD(arg2, arg3))
end
```

4.5 Wywołanie funkcji

Aby wywołać funkcję piszemy jej nazwę, a następnie w nawiasach przekazujemy argumenty. Musimy przekazać wszystkie argumenty wymagane. W przypadku funkcji użytkownika (tj. zdefiniowanych w kodzie BIO) istnieje możliwość zapisania dla jakiego argumentu przekazujemy daną wartość poprzez napisanie nazwy argumentu, następnie znaku równości "=" i wartości dla tego argumentu.

nazwa_funkcji (argumenty, argumenty_nazywane)

```
fun_name("sum: ", arg3=10, arg2=-20.5)
```

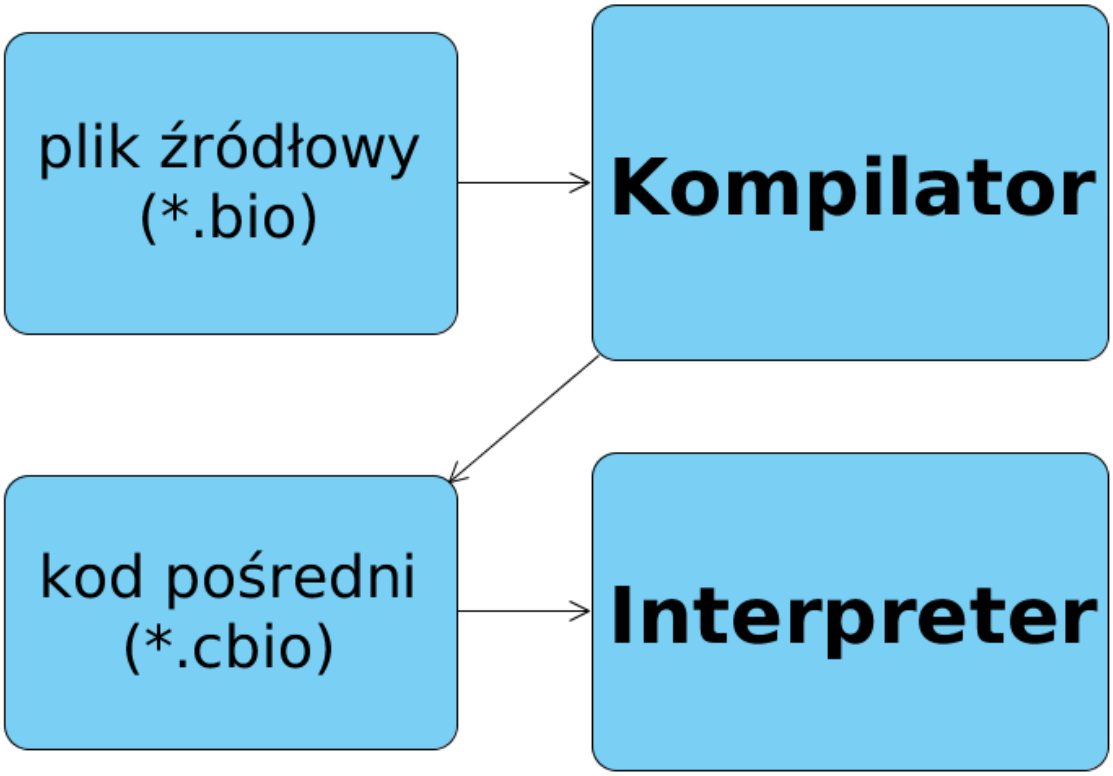
4.6 Wzorzec obserwatora

Wykorzystanie wzorca obserwatora polega na zdefiniowaniu jednej funkcji zdarzenia i jednej lub wielu funkcji obserwatorów. Funkcje obserwatorów mogą przyjmować mniej lub tyle samo parametrów co funkcja zdarzenia. Funkcja obserwatora może być jednocześnie funkcją zdarzenia i odwrotnie. Zarejestrowanie obserwatora do zdarzenia odbywa się za pomocą funkcji ATTACH_TO_EVENT. Jeżeli chcemy usunąć takie powiązanie należy użyć funkcji DETACH_FROM_EVENT. Sprawdzenie czy obserwator jest zarejestrowany do zdarzenia odbywa się przy pomocy funkcji IS_ATTACHED.

```
def some_event()
   DN()
end

def on_some_event()
   DN()
end

def onSTART()

   ATTACH_TO_EVENT(some_event, on_some_event)
   IS_ATTACHED(some_event, on_some_event)
   DETACH_FROM_EVENT(some_event, on_some_event)

end
```

5. Realizacja

Implementacja języka BIO składa się z dwóch programów napisanych w javie: kompilatora i interpretera. Kompilator dostaje na wejściu kod źródłowy języka BIO i generuje kod pośredni zapisywany w pliku binarnym o z góry określonym formacie. Następnie plik ten podawany jest jako wejście dla interpretera, który wykonuje zawarty w nim kod. Przedstawia to poniższy diagram:

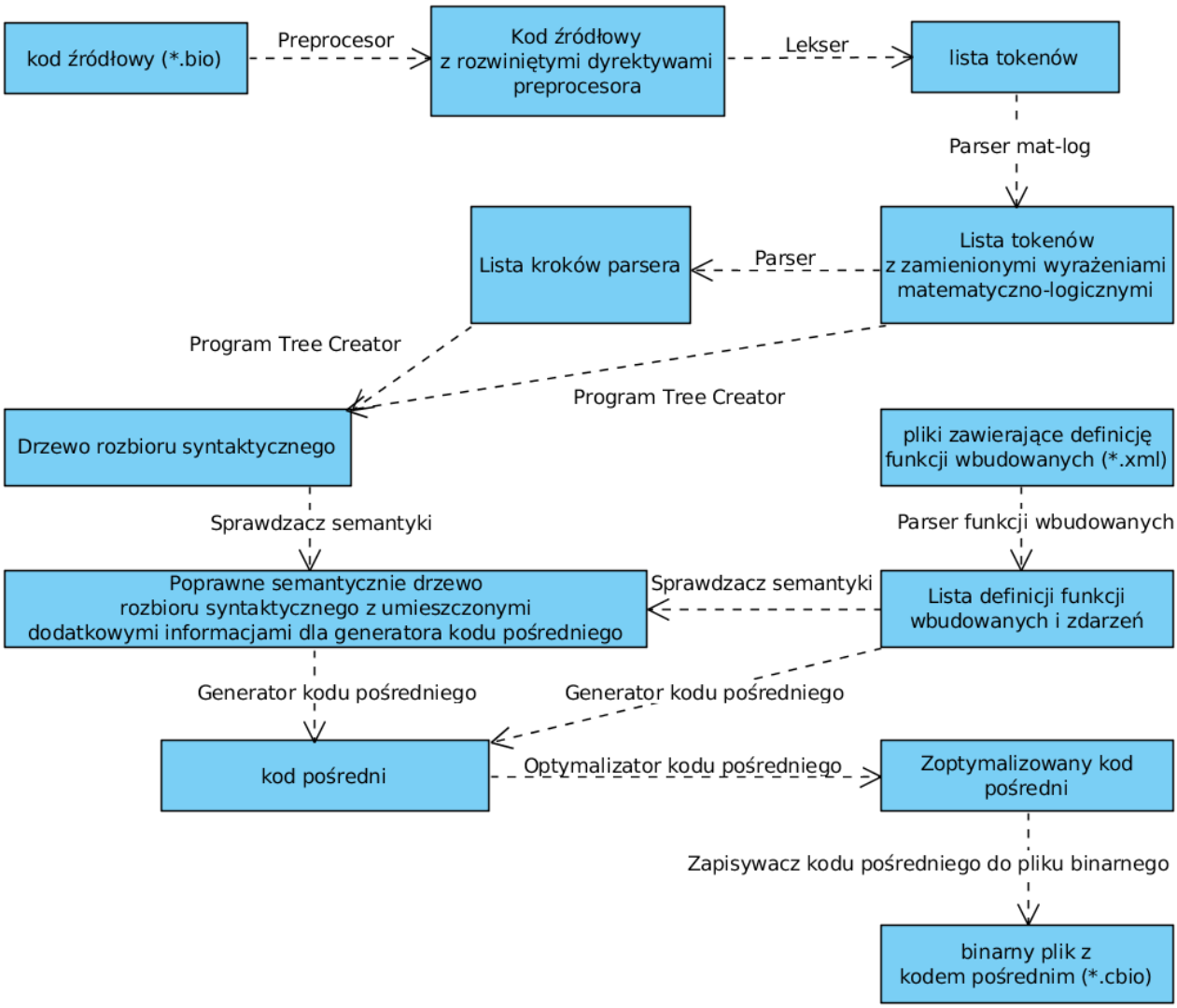


Rys.5.1. Diagram obrazujący proces zamienienia pliku źródłowego w wykonywalny kod (źródło: własne)

Dalej w tym rozdziale opisane zostało jakie kroki podejmuje kompilator, aby na podstawie pliku źródłowego stworzyć kod pośredni, a także jakie kroki podejmuje interpreter aby tak wygenerowany kod pośredni wykonać.

5.1 Kompilator

Kompilator przetwarza plik źródłowy na kod pośredni w kilku etapach. Na początek następuje przetworzenie kodu przez preprocesor, następnie analiza leksykalna, translacja wyrażeń matematyczno-logicznych na kod języka, parsowanie kodu BIO, sprawdzanie semantyki, generacja kodu pośredniego i na koniec jego optymalizacja. Proces ten przedstawia poniższy diagram:



Rys.5.1.1 diagram ukazujący proces przetwarzania (kompilacji) kodu źródłowego na kod pośredni. W niebieskich ramkach przedstawione zostały zasoby, a pomiędzy nimi moduły je przetwarzające. (źródło: własne)

Dalej opisane zostały kolejne moduły przetwarzające zasoby.

5.1.1 Preprocesor

Preprocesor jest pierwszym etapem kompilacji. Jego zadaniem jest rozwinięcie dyrektyw. Preprocesor w języku BIO posiada dwie dyrektywy. Każda dyrektywa zaczyna się znakiem "#" po którym następuje jej nazwa i parametry podane w nawiasach (składnia taka sama jak wywołanie funkcji poza początkowym znakiem "#"). Dyrektywy muszą znajdować się w osobnej lini. Dyrektywy języka:

1. #INCLUDE("ścieżka_do_pliku") - powoduje załączenie w miejscu wystąpienia pliku podanego jako argument. Ścieżki mogą być względne i bezwzględne. Kompilator dba o to, aby ten sam plik nie został załączony dwa razy. Dodatkowo możemy podać ścieżkę między znakami "<" i ">". Wtedy kompilator będzie szukał podanego pliku w lokalizacji standardowej (katalog lib).

2. #IMPORT("nazwa_modułu") - powoduje zaimportowanie modułu o podanej nazwie. Moduł to zbiór funkcji wbudowanych (np. funkcja AS_LOC należy do modułu basic, a funkcja PRINTLN do modułu io). Domyślnie pewne moduły są zaimportowane i nie musimy ich jawnie importować (choć możemy). Importować musimy jedynie moduły opcjonalne. Wykaz wszystkich modułów dostępny jest w dokumentacji języka. Jeżeli chcemy to możemy zaimportować wszystkie moduły podając jako nazwę modułu "all" (#IMPORT("all")).

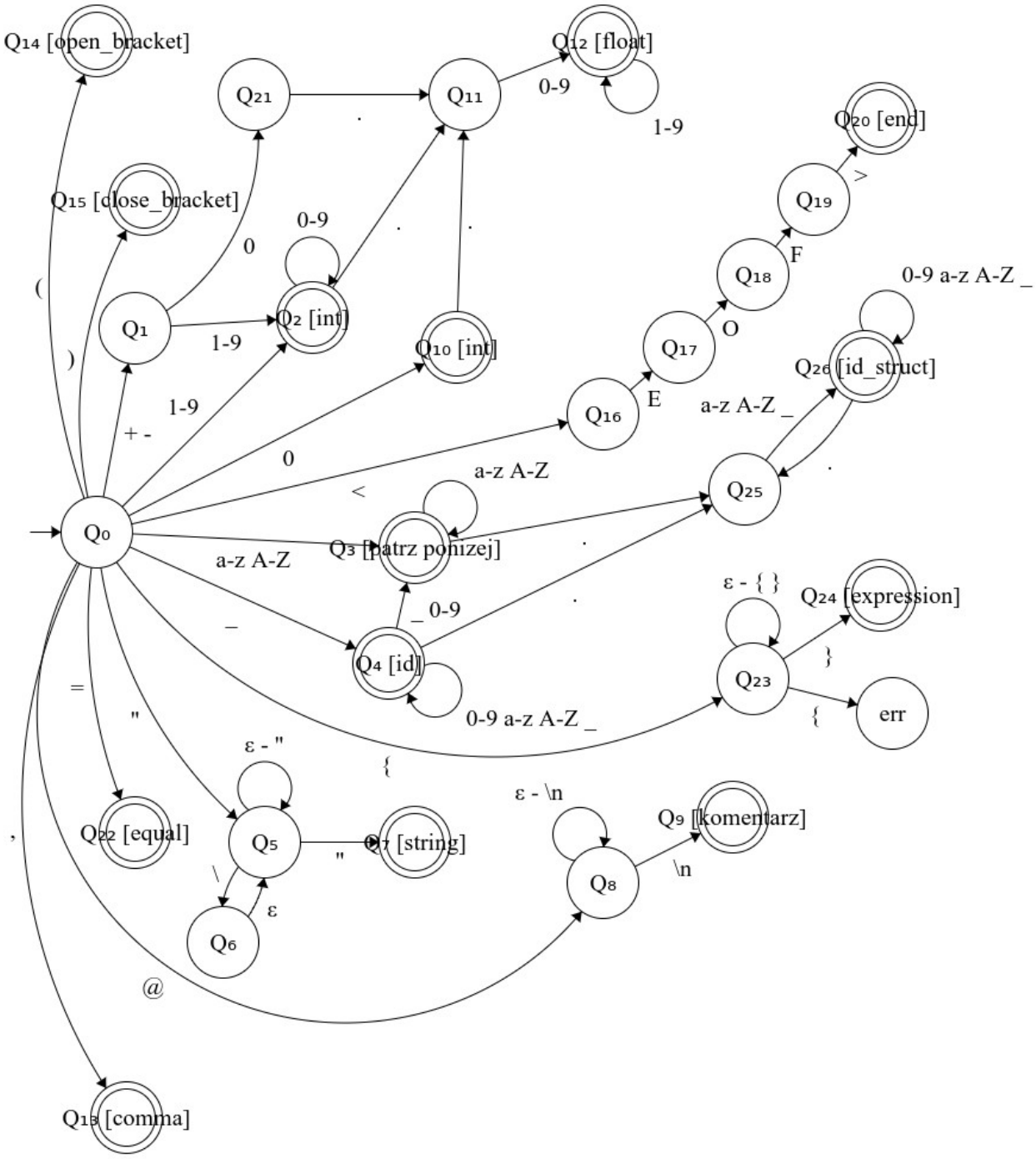
Dodatkowo preprocesor zamieszcza informację, która linia należy do którego pliku i który plik został załączony z którego. Jest to wykorzystywane potem przy raportowaniu prawie wszystkich błędów zarówno kompilacji jak i wywołania.

5.1.2 Lekser

Wyjście preprocesora podawane jest do leksera, którego zadaniem jest “podzielenie” ciągu znaków na ciąg tokenów. Typy tokenów:

Typ	**Przykład**
int	10, -5
float	-10.5, 0.26
string	“przykładowy tekst\n”
bool	True, False
none	None
open_bracket	(
close_bracket	)
comma	,
equal	=
id	wektor, a, iter, żółć
id_struct	a.b.c, wektor.x, kolor.r
end	
keyword	def, end
expression	{ a + b ^ 2 }

Wyodrębnianie tokenów jest realizowane za pomocą skończonego automatu stanów. Tabela stanów i przejść pomiędzy nimi znajduje się poniżej. Warto zauważyć, że istnieje token expression. Jest to wyrażenie matematyczno-logiczne, które zostaje poddane osobnemu procesowi dzielenia na tokeny, parsowania i generowania odpowiadającemu mu kodu języka BIO. Więcej informacji o tym znajduje się w rozdziale 5.1.3 (*Parser matematyczno-logiczny*).



Rys.5.1.2.1 diagram stanów skończonego automatu stanów leksera

Objaśnienia do powyższego diagramu:

- obok stanu w nawiasach kwadratowych znajduje się informacja token jakiego typu stan produkuje.

- komentarz jest ignorowany przez lekser.

- w stanie Q_3 możliwe jest wystąpienie więcej niż jednego tokenu: bool, none, keyword, id. Lekser przyporządkowuje typ tokenu do pierwszego pasującego wzorca z listy powyżej.

- zapis $\epsilon - \{\}$ oznacza wszystkie znaki alfabetu z wyjątkiem znaków { i }.

- zapis a-z A-Z oznacza **każdą** literę zakodowaną w UTF-8.

- w stanach $Q_{2,}Q_{3,}Q_{4,}Q_{10,}Q_{12,}Q_{26}$ dozwolone jest wystąpienie separatora, który powoduje zapisanie aktualnego tokenu, cofnięcie ostatnio pobranego znaku i przejście do stanu początkowego. Separatory to: (,), spacja, \t, \n, @, przecinek, =

- err to stan oznaczający błąd leksera.

5.1.3 Parser matematyczno-logiczny

W rzeczywistości moduł ten składa się z leksera, parsera i translatora. Zamienia on wszystkie tokeny typu expression, na odpowiadający wyrażeniu ciąg tokenów, który jest wstawiany w miejsce tokenu expression. Do zrealizowania tego modułu użyty został JavaCC, czyli generator parsera w języku Java. Jeżeli chodzi o tokeny są one prawie identycznym podzbiorem tokenów, które znajdują się w głównym lekserze języka. Poniżej znajduje się gramatyka tego parsera, zapisana w notacjii EBNF (pomiędzy znakami <> zapisane są tokeny - terminale):

```
START = EXPRESSION, <EOF>;
EXPRESSION = OR_EX;
OR_EX = AND_EX, OR_EX_1;
OR_EX_1 = [ <OR>, AND_EX, OR_EX_1 ];
AND_EX = EQ_EX, AND_EX_1;
AND_EX_1 = [ <AND>, EQ_EX, AND_EX_1 ];
EQ_EX = CMP_EX, EQ_EX_1;
EQ_EX_1 = [ (<EQ> | <NEQ>), CMP_EX, EQ_EX_1 ];
CMP_EX = ADD_EX, CMP_EX_1;
CMP_EX_1 = [ (<LS> | <LE> | <GT> | <GE>), ADD_EX, CMP_EX_1 ];
ADD_EX = MUL_EX, ADD_EX_1;
ADD_EX_1 = [ (<PLUS> | <MINUS>), MUL_EX, ADD_EX_1 ];
MUL_EX = POW_EX, MUL_EX_1;
MUL_EX_1 = [ (<TIMES> | <DIVIDE> | <MODULO>), POW_EX, MUL_EX_1 ];
POW_EX = NEG_EX, POW_EX_1;
POW_EX_1 = [ <POWER>, POW_EX ];
NEG_EX = (<MINUS> | <PLUS>) NEG_EX | NOT_EX;
NOT_EX = [ <EXCL_MARK> ], INDX_EX;
INDX_EX = PRIMARY, INDX_EX_1;
INDX_EX_1 = [ <OPEN_INDX>, EXPRESSION, <CLOSE_INDX>, INDX_EX_1 ];
PRIMARY = <INT> | <FLOAT> | <STRING> | <NONE> | <ID> | <ID_STRUCT> | <TRUE> |
<FALSE> | <OPEN_PAR>, EXPRESSION, <CLOSE_PAR>;
```

Wynikiem parsowania powyższą gramatyką jest drzewo AST, które translator zamienia na odpowiadający mu ciąg tokenów języka BIO. Następnie tokeny te zastępują token expression.

5.1.4 Parser

Parser języka BIO jest typu LL(1). Przyjmuje on na wejściu ciąg tokenów, a na wyjściu zwraca listę kroków (kolejne produkcje) jakie należy wykonać, aby stworzyć drzewo rozbioru syntaktycznego. Gramatyka tego parsera znajduje się poniżej (podkreślenie symbolizuje symbol nieterminalny):

(0) D → D_1 D_2
(1) D_1 → D_2 D_1
(2) D_1 → ϵ
(3) D_2 → def id (P) F end
(4) P → P_1
(5) P → ϵ
(6) P_1 → id P_2
(7) P_2 → , P_1
(8) P_2 → ϵ
(9) P_2 → = P_3
(10) P_3 → PDC P_4
(11) P_4 → , id = P_3
(12) P_4 → ϵ
(13) PDC → CONST
(14) PDC → id (PDC_1)
(15) PDC_1 → PDC_2
(16) PDC_1 → ϵ
(17) PDC_2 → PDC_3 PDCR
(18) PDC_3 → CONST
(19) PDC_3 → id (PDC_1)
(20) PDCR → ϵ
(21) PDCR → , PDC_2
(22) F → C F_1
(23) F_1 → C F_1
(24) F_1 → ϵ
(25) C → id (C_1)
(26) C_1 → C_2
(27) C_1 → ϵ
(28) C_2 → C_3 R

(29) R → , C_2

(30) R → ϵ

(31) C_3 → id R_1

(32) C_3 → CONST

(33) C_3 → id_struct

(34) R_1 → (C_1)

(35) R_1 → ϵ

(36) R_1 → = C_4

(37) C_4 → CONST

(38) C_4 → id_struct

(39) C_4 → id C_5

(40) C_5 → ϵ

(41) C_5 → (C_1)

(42) CONST → INT | FLOAT | STRING | BOOL | NONE

5.1.5 Program Tree Creator

Jest to moduł, który na wejściu bierze ciąg tokenów i listę kroków parsera i produkuję z nich drzewiastą strukturę programu, na której łatwo operować w następnych krokach.

5.1.6 Parser funkcji wbudowanych

Język BIO składa się z trzech typów funkcji:

- funkcje użytkownika, które są zdeklarowane i zdefiniowane w języku BIO.

- funkcje wbudowane (np. ASSIGN_LOCAL, RETURN), które są zdeklarowane w pliku .xml o odpowiednim formacie, a zdefiniowane w języku Java w interpreterze.

- funkcje wbudowane specjalne, które są zdeklarowane w pliku .xml o odpowiednim formacie, a podczas procesu kompilacji zamieniane są na odpowiedni kod pośredni (np. FOR, IF).

Oprócz funkcji wbudowanych w pliku .xml zdeklarowane mogą być również zdarzenia. Przykład pliku .xml z deklaracją funkcji wbudowanej, funkcji wbudowanej specjalnej i zdarzenia:

```
<?xml version="1.0" encoding="UTF-8"?>
<functions>
    <function>
        <name>
            ASSIGN_LOCAL
        </name>
        <alias>
            AS_LOC
        </alias>
        <params>
            <param>
                id
            </param>
            <param>
                all
            </param>
            <param repeat="true">
                id
            </param>
            <param repeat="true">
                all
            </param>
        </params>
    </function>

    <function>
        <name>
            FOR
        </name>
        <special>
            true
        </special>
        <params>
            <param>
                call
            </param>
            <param>
                all
            </param>
            <param>
                call
            </param>
            <param optional="true">
                call
            </param>
        </params>
    </function>

    <event>
        <name>
            onUNHANDLED_ERROR
        </name>
        <params>
            <param>
                all
            </param>
        </params>
    </event>
<functions>
```

Są to rzeczywiste deklaracje funkcji ASSIGN_LOCAL, FOR i zdarzenia onUNHANDLED_ERROR. Deklaracja funkcji znajduje się pomiędzy tagami <function> </function>. Musi zawierać tag <name> i <params>. Tagi <alias> i <special> są opcjonalne. Wewnątrz tagu <params> możemy umieścić zero lub więcej tagów <param>. Każdy tag param może zawierać atrybut optional lub repeat, który informuje czy dany parametr jest opcjonalny lub cykliczny. Wartościami tagu <param> może być: all, id, call. Deklaracja zdarzenia znajduje się pomiędzy tagami <event> </event>. Wewnątrz mogą znajdować się te same tagi co w przypadku deklaracji funkcji, bez tagów: <alias> i <special>. Poza tym tag <param> nie może przyjmować żadnych atrybutów, a wewnątrz niego może znaleźć się tylko wartość: all. Jest to spowodowane tym, że deklaracja zdarzenia musi się zgadzać z deklaracją funkcji użytkownika w języku BIO.

Parser funkcji wbudowanych tworzy na podstawie pliku .xml deklaracje funkcji wbudowanych, które następnie są używane w procesie sprawdzania semantyki i generowania kodu.

5.1.7 Sprawdzacz semantyki

Jak sama nazwa wskazuje, rolą tego modułu jest sprawdzenie czy semantyka programu jest poprawna. Dodatkowo przekazuje on również pewne informację do modułu generowania kodu. Może także wstawiać dodatkowy kod i modyfikować istniejący.

Zadania sprawdzacza semantyki:

1. Sprawdzenie poprawnej ilości przekazanych argumentów i ich typu (dotyczy funkcji wbudowanych) przy wywołaniu funkcji. Zawarcie informacji o ilości cykli powtórzeń w przypadku funkcji wbudowanych.

2. Sprawdzenie nazw parametrów funkcji użytkownika w poszukiwaniu powtórzeń.

3. Sprawdzenie czy funkcja o tej samej nazwie nie jest zadeklarowana dwa razy.

4. Sprawdzenie czy funkcje BREAK i CONTINUE występują wewnątrz funkcji FOR.

5. Sprawdzenie czy funkcja onSTART jest zadeklarowana i przyjmuje odpowiednią ilość parametrów (zero lub jeden).

6. Dodanie wywołania funkcji RETURN z parametrem NONE na końcu każdej funkcji użytkownika, która nie ma wywołania funkcji RETURN na końcu.

7. Zamiana aliasu funkcji na jej podstawową nazwę.

8. Sprawdzenie czy domyślne parametry zawierają poprawne wywołania (nazwy, ilości parametrów).

9. Sprawdzenie czy argumenty przekazywane przy wywołaniu funkcji spełniają poniższe zasady:

- Na początku znajdują się nie nazywane parametry, potem nazywane

- Wartości argumentów nie powtarzają się

- Podano odpowiednią ilosć parametrów (podano wszystkie parametry nie domyślne)

- Nazywane parametry mają odpowiednią nazwę

10. Uzupełnienie parametrów wywołania o wartości domyślne.

11. Wygenerowanie kolejności argumentów (potem instrukcja ORDER interpretera).

12. Sprawdzenie czy do funkcji wbudowanych nie przekazano parametrów nazywanych.

5.1.8 Generator kodu

Moduł ten odpowiada za generację niezoptymalizowanego kodu pośredniego, który następnie jest wykonywany przez interpreter. Jako wejście przyjmuję drzewo rozbioru syntaktycznego z dodatkowymi informacjami zawartymi przez sprawdzacza semantyki i zbiór deklaracji funkcji wbudowanych. Jako wyjście zwraca wspomniany wcześniej niezoptymalizowany kod pośredni.

Kod pośredni to po prostu zbiór funkcji interpretera oddzielonych znakiem nowej linii. Poniżej przedstawiona jest tabela zawierającą informację o wszystkich funkcjach interpretera:

Nazwa	Parametry	Opis
CALL	[id]	Wywołuję funkcję użytkownika o nazwie podanej jako parametr.
CALL_LOC	[id]	Wywołuję funkcję wbudowaną o nazwie podanej jako parametr.
PUSH	[all]	Odkłada na stos wartość przekazaną jako parametr. Wartościom może być: id, int, float, bool, none, string, var.
POP	[int]	Ściąga ze stosu podaną jako parametr ilość wartości i zapisuje je tymczasowo.
JMP	[int]	Skok bezwarunkowy do podanej jako parametr instrukcji.
JMP_IF_FALSE	[int]	Skok warunkowy do podanej jako parametr instrukcji. Skok jest wykonany wtedy, gdy ostatnio ściągniętą ze stosu metodą POP wartością jest false.
POPC	[int]	Ściąga ze stosu podaną jako parametr ilość wartości.
CLEAR_STACK		Czyści stos wartości.
ORDER	[int]?	Instrukcja opcjonalnie znajdującą się przed instrukcją CALL. Informuje interpreter w jakiej kolejności ma przekazać argumenty do funkcji.

Generator przechodzi drzewo rozbioru syntaktycznego zaczynając od dzieci, a na końcu generując kod dla rodzica. W zależności od tego co dany węzeł zawiera generowany jest odpowiedni kod:

1. Wywołanie funkcji użytkownika:

```
POP, ilość_argumentów

ORDER, … (opcjonalnie)

CALL, nazwa_funkcji
```

2. Wywołanie funkcji wbudowanej:

```
POP, ilość_argumentów
CALL_LOC, nazwa_funkcji
```

3. Stały argument lub identyfikator:

```
PUSH, …
```

W przypadku wbudowanych funkcji specjalnych na tym etapie są one zamieniane na odpowiedni kod pośredni. Poniżej przedstawione zostały deklaracje funkcji z odpowiadającym im kodem pośrednim. W przypadku parametrów (call i anything) są one w danym miejscu rozwijane zgodnie z powyższymi regułami.

1. BREAK():

```
JMP, etykieta_za_pętlą_for
```

2. CONTINUE():

```
JMP, etykieta_na_początku_pętli_for
```

3. DN():

```
PUSH, none:
```

4. CALL(call_1, call_2, …, call_n):

```
call_1
popc, 1
call_2
popc, 2
…
call_n
```

5. IF(anyhing, call_1, call_2):

```
anything
POP, 1
FMP_IF_FALSE, etykieta_1
call_1
POPC, 1
JMP, etykieta_2
etykieta1:
call_2
POPC, 1
etykieta_2:
PUSH, none:
```

6. FOR(call_1, anything, call_2, call_3):

```
call_1
POPC, 1
etykieta_1:
anything
POP, 1
JMP_IF_FALSE, etykieta_3
call_2
POPC, 1
etykieta_2:
call_3
POPC, 1
JMP, etykieta_1
etykieta_3:
PUSH, none:
```

W przypadku funkcji FOR i IF istnieją ich odpowiedniki bez ostatniego argumentu. Różnią się one nieznacznie od wersji przedstawionych powyżej dlatego zostały pominięte.

5.1.9 Optymalizator kodu

Jest to etap, który nie ma wpływu na funkcjonalność języka, a jedynie na jego wydajność. Efektem działania tego modułu jest zoptymalizowany kod, który zawiera mniej instrukcji. Istnieje możliwość wyłączenia optymalizacji poprzez flagę -c w kompilatorze. Optymalizacje możemy podzielić na trzy grupy:

1. Strukturalne:

Polegają na prostym wyszukaniu odpowiednich sekwencji instrukcji w kodzie pośrednim i ich zmodyfikowaniu/usunięciu. Przykładowo, taka sekwencja instrukcji:

```
PUSH, …
PUSH, …
…
PUSH, …
CLEAR_STACK
```

Może zostać zastąpiona po prostu instrukcją:

```
CLEAR_STACK
```

Optymalizacji strukturalnych jest sześć.

2. Skokowe:

Optymalizują skoki warunkowe i bezwarunkowe w kodzie pośrednim. Na przykład w wygenerowanym kodzie może zdarzyć się taka sytuacja:

```
[1] JMP_IF_FALSE, 5
…
[5] JMP, 7
…
[7] JMP, 8
```

Po pierwsze możemy usunąć ostatni skok (JMP, 8), gdyż jest on skokiem do następnej linijki. Ponadto możemy zamienić pierwszy skok (JMP_IF_FALSE, 5) na skok (JMP_IF_FALSE, 7). Usuwamy w ten sposób niepotrzebne skoki. Finalny kod wyglądałby następująco:

```
[1] JMP_IF_FALSE, 7
…
[5] JMP, 7
…
[7] …
```

3. Analiza grafu przepływu:

Polega na zbudowaniu grafu przepływu programu, a następnie wykonaniu na nim operacji optymalizacyjnych. Graf przepływu konstruujemy wedle następujących zasad:

1. Dzielimy kod na bloki podstawowe. Blok podstawowy to taki zbiór instrukcji, w którym wykonanie wszystkich instrukcji jest gwarantowane po wejściu do tego bloku (atomowa operacja).

2. Łączymy bloki podstawowe. Połączenie tworzone jest od jednego bloku do drugiego w dwóch przypadkach. Pierwszy to taki w którym jeden blok znajduje się zaraz za drugim i na końcu pierwszego nie znajduje się instrukcja JMP. Drugi przypadek to taki w którym ostatnia instrukcja w pierwszym bloku to JMP lub JMP_IF_FALSE i wskazuje ona na drugi blok.

Mając taki graf przepływu dokonujemy wyszukania nieużywanych bloków kodu i ich usunięcia. Wyszukanie nieużywanych kodów bloku polega na przejściu grafu (w jakikolwiek sposób) od bloku startowego (którym jest pierwszy blok w funkcji) i oznaczeniu wszystkich odwiedzonych bloków. Nieodwiedzone bloki są blokami nieużywanymi i możemy je bezpiecznie usunąć.

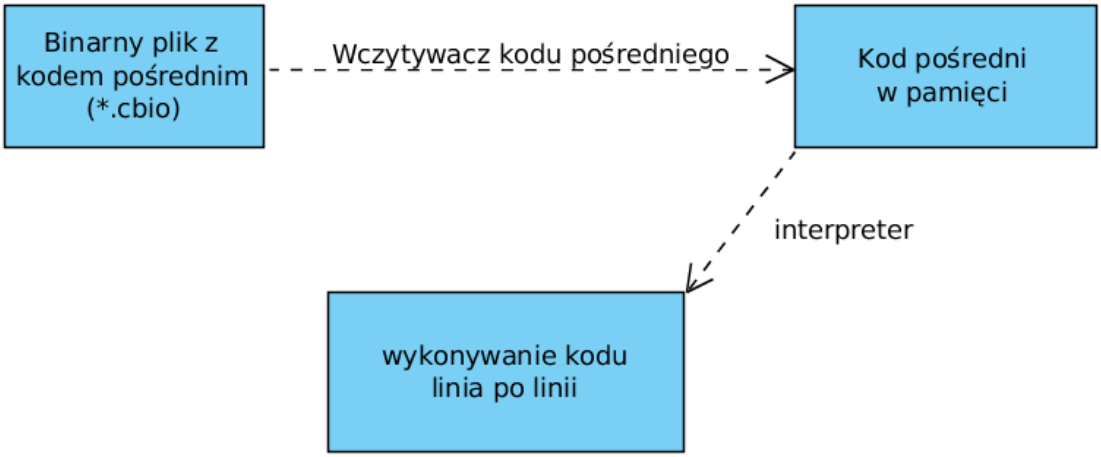
Drugą rzeczą jaką możemy zrobić jest wyszukanie nieskończonych pętli. Odbywa się to podobnie jak wyszukiwanie nieużywanych bloków kodu, ale tym razem przechodzimy graf zaczynając od każdego bloku podstawowego. Jeżeli od danego bloku nie da się dojść do bloku końcowego (to jest takiego, który zawiera instrukcję CALL_LOC, RETURN), oznacza to, że ten blok znajduje się w cyklu pętli nieskończonej.

5.1.10 Zapisywacz kodu pośredniego

Jest to ostatni etap wykonywany przez kompilator. Zapisuje on kod pośredni do pliku binarnego o odpowiednim formacie. Następnie może być on odczytany i wykonany przez interpreter.

5.2 Interpreter

Zadaniem interpretera jest wczytanie i wykonanie kodu pośredniego wygenerowanego przez kompilator. Poniżej znajduje się diagram działania interpretera. Jest on dużo prostszy niż ten dla kompilatora. Główną częścią jest wykonywanie kolejnych instrukcji. Interpreter działa jak maszyna stosowa, odkładając parametry i wartości zwracane kolejnych funkcji na stosie.



Rys.5.2.1 diagram ukazujący pracę interpretera

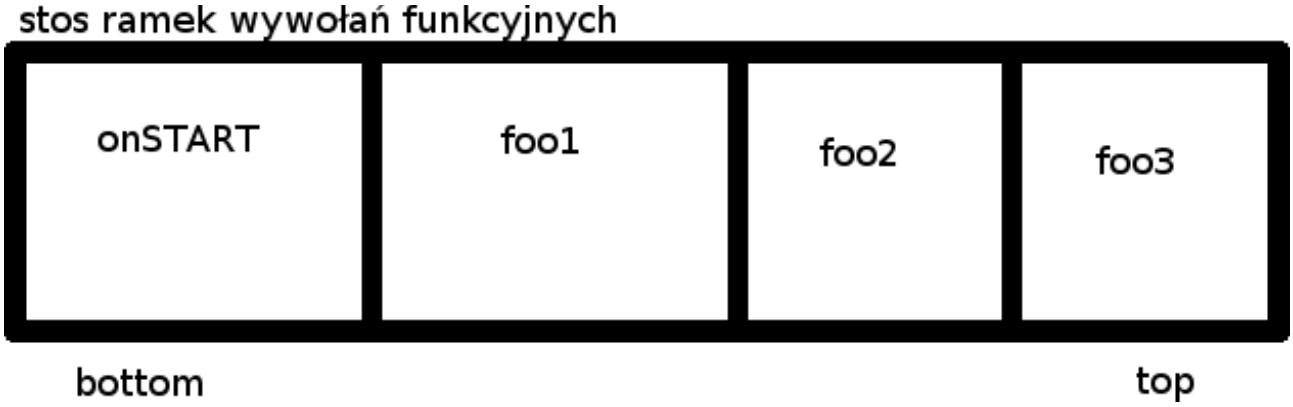
W dalszej części tego rozdziału opisane zostaną kluczowe rzeczy dla działania interpretera.

5.2.1 Wczytywacz kodu pośredniego

Zadaniem tego modułu jest wczytanie kodu pośredniego wygenerowanego przez kompilator z pliku binarnego. Kod jest w całości wczytywany do pamięci RAM. Następnie interpreter pobiera kolejne instrukcje i je wykonuje.

5.2.2 Ramka wywołania funkcyjnego

Bardzo ważny komponent w pracy interpretera. Za każdym razem gdy wywoływana jest funkcja użytkownika, tworzona jest ramka wywołania funkcyjnego i odkładana na stos ramek wywołań funkcyjnych. Stos ramek obrazuje poniższy obrazek:



Jak widać na dole stosu znajduje się funkcja onSTART (prawie zawsze, z wyjątkiem wystąpienia zdarzenia onUNHANDLED_ERROR, kiedy na dole stosu znajduje się funkcja podpięta do tego zdarzenia). Każda ramka zawiera:

- informację na temat wywoływanej funkcji - nazwę, kod/instrukcje, obserwatorów.

- wskaźnik instrukcji (ang. instruction pointer) – informacja o aktualnie wykonywanej instrukcji.

- zmienne lokalne – wszystkie zmienne stworzone wewnątrz funkcji metodą AS_LOC i przekazane do funkcji parametry.

- stos wartości – stos zawierający wartości przekazywane i zwracane pomiędzy wywołaniami funkcji.

- stos parametrów - wartości ściągnięte ze stosu wartości instrukcją POP.

- zwróc do wywołującego - informacja czy wartość zwracaną funkcji należy odłożyć na stos wartości poprzedniej ramki.

W momencie wystąpienia funkcji RETURN, ostatnio ściągnięta wartość ze stosu wartości, w przypadku gdy zwróć do wywołującego jest ustawione jako true, jest odkładana na stos wartości poprzedniej ramki znajdującej się na stosie ramek. Następnie ramka jest ściągana ze stosu.

5.2.3 Funkcje interpretera

Poniżej wymienione są wszystkie funkcje interpretera, wraz z dokładnym objaśnieniem co robią, a także parametrami, które przyjmują:

1. CALL, nazwa_funkcji – tworzy ramkę funkcyjną dla funkcji użytkownika o nazwie nazwa_funkcji i odkłada na stos ramek wywołań funkcyjnych. Kopiuje wartości ze stosu parametrów aktualnej ramki do zmiennych lokalnych nowo utworzonej ramki. Jeżeli wywoływana funkcja posiada jakichś obserwatorów, to umieszcza ich na górze stosu z tymi samymi parametrami i wartością zwróć do wywołującego ustawioną na false.

2. CALL_LOC, nazwa_funkcji – wywołuje kod funkcji wbudowanej, a wartość zwróconą odkłada na stos wartości aktualnej ramki.

3. PUSH, data – odkłada data (które może być typu: int, float, string, none, bool, id) na stos wartości. Jeżeli data jest typu var, to odkłada na stos wartości to co znajduje się w zmiennej o nazwie przekazanej wraz z tym typem.

4. POP, ilość – ściąga ze stosu wartości ilość wartości i odkłada na stos parametrów.

5. POPC, ilość – ściąga ze stosu wartości ilość wartości.

6. JMP, cel – skacze bezwarunkowo do instrukcji podanej w parametrze cel. Skok polega na przypisaniu do wskaźnika instrukcji wartości cel.

7. JMP_IF_FALSE, cel – skacze warunkowo do instrukcji podanej w parametrze jeżeli parametr na stosie parametrów jest typu bool i ma wartość false. Skok polega na przypisaniu do wskaźnika instrukcji wartości cel.

8. CLEAR_STACK – czyści stos wartośći.

9. ORDER, or1, or2, …, orN – informuje interpreter o kolejności przekazywanych argumentów do funkcji użytkownika. Może znajdować się przed instrukcją CALL.

Aby lepiej zobrazować działanie interpretera, poniżej znajduje się kod pośredni wygenerowany przez kompilator z objaśnieniem każdej linijki. Kod pośredni wygenerowany został dla takiego kodu źródłowego:

```
def add(arg1, arg2)
   RETURN({ arg1 + arg2 })
end

def onSTART()
   AS_LOC(sum, add(arg2=10, arg1=20))
   PRINTLN(sum)
end
```

Kod pośredni (kursywą wypisany został faktyczny kod, bez kursywy napisane są objaśnienia):

[0]observer,ints,strings,errors,arrays,logic,math,basic,type_check,conversion,compare,reflections,structs,io,iter,floats

[1]

Informacja o wszystkich modułach z jakich korzysta program (w tym wypadku, ponieważ nie została użyta dyrektywa #IMPORT znajdują się tu tylko wszystkie domyślne moduły).

[2] 1,test.bio

[3] -1,generated by compiler

[4]

Informacja o plikach, używana przy raportowaniu błędów.

[5] onSTART,16,5,5,1

[6] add,8,1,5,1

[7]

Spis wszystkich funkcji użytkownika występujących w programie. Zawarta jest nazwa, linia w kodzie pośrednim, linia w kodzie źródłowym, znak w kodzie źródłowym i numer pliku.

[8] add,arg1,arg2

Deklaracja, że poniżej znajduje się zbiór instrukcji dla funkcji add. Zawarta jest informacja o nazwie funkcji i nazwach parametrów jakie przyjmuje.

[9] push,var:arg1,2,13,1

[10] push,var:arg2,2,20,1

Odłożenie na stos wartości, wartości zmiennych arg1 i arg2. Trzy ostatnie liczby są używane w przypadku raportowania błędów (linia w kodzie źródłowym, znak w kodzie źródłowym, numer pliku).

[11] pop,2

Ściągnięcie ze stosu wartości dwóch wartości i skopiowanie ich na stos parametrów.

[12] call_loc,ADD,2,18,1

Wywołanie funkcji wbudowanej o nazwie ADD. Do funkcji tej zostaną przekazane parametry ze stosu parametrów, a wartość zwrócona zostanie odłożona na stos wartości.

[13] pop,1

Ściągnięcie ze stosu wartości jednego parametru i skopiowanie go na stos parametrów.

[14] call_loc,RETURN,2,4,1

Wywołanie funkcji wbudowanej RETURN, która spowoduje skopiowanie parametru ze stosu parametrów aktualnej ramki na stos wartości poprzedniej ramki i ściągnięcie aktualnej ramki ze stosu ramek.

[15]

Pusta linia informuję o tym, że jest to koniec instrukcji dla funkcji.

[16] onSTART

[17] push,id:sum,6,11,1

[18] push,int:10,6,25,1

[19] push,int:20,6,34,1

[20] pop,2

Podobnie jak wyżej, na początku jest deklaracja, że instrukcję poniżej należą do funkcji onSTART. Następnie następują trzy instrukcje odłożenia parametrów na stos wartości i na końcu instrukcja ściągnięcia ze stosu wartości dwóch parametrów i skopiowania ich na stos parametrów.

[21] order,2,1

Instrukcja informuje, że parametry ze stosu parametrów należy przekazywać w następującej kolejności do funkcji: pierwszy parametr na stosie parametrów należy przekazać jako drugi, a drugi parametr ze stosu parametrów jako pierwszy.

[22] call,add,6,16,1

[23] pop,2

[24] call_loc,ASSIGN_LOCAL,6,4,1

Wywołanie funkcji użytkownika add, które skutkuje utworzeniem nowej ramki i odłożeniem jej na stos ramek. Następnie ściągnięcie dwóch parametrów ze stosu wartości i skopiowanie ich na stos parametrów i wywołanie funkcji wbudowanej ASSIGN_LOCAL.

[25] clear_stack

Wyczyszczenie stosu wartości.

[26] push,var:sum,7,12,1

[27] pop,1

[28] call_loc,PRINTLN,7,4,1

[29] clear_stack

[30] push,none:,-1,-1,-1

[31] pop,1

[32] call_loc,RETURN,-1,-1,-1

[33]

Odłożenie na stos wartości, wartości typu var o wartośći sum. Ściągnięcie jednej wartości ze stosu wartości i skopiowanie na stos parametrów. Wywołanie funkcji wbudowanej PRINTLN. Wyczyszczenie stosu wartości. Odłożenie na stos wartości, wartości typu none, ściągnięcie jej ze stosu wartości i skopiowanie na stos parametrów. Wywołanie funkcji wbudowanej RETURN. Koniec funkcji onSTART.

5.2.4 Moduły funkcji wbudowanych

Każda funkcja wbudowana języka BIO jest zaimplementowana w interpreterze w Javie. Funkcje zgrupowane są w moduły, które dostarczają jakiejś konkretnej funkcjonalności. Na przykład moduł matematyczny (MathModule) dostarcza funkcji do operacji matematycznych. Co ważne nawet takie funkcje jak przypisanie (AS_LOC) czy zwrócenie wartości (RETURN), także są zaimplementowane w ten sposób, co umożliwia łatwą ich modyfikację gdyby zaszła taka potrzeba. Implementacja każdej funkcji sprowadza się do stworzenia klasy dziedziczącej po odpowiednim interfejsie (Ifunction). Interfejs ten wygląda następująco:

```
public interface IFunction
{
    // nazwa funkcji
    public String getName();

    // wywołanie funkcji, zwracana wartość jest odkładana na stos wartości aktualnej
ramki
    public Data call(List<Data> params, CallFrame currentFrame, Interpreter
interpreter);
}
```

W ten sam sposób implementowane są zdarzenia, przy czym w tym wypadku jedynymi informacjami są nazwa i ilość parametrów. Interfejs zdarzenia (IEvent):

```
public interface IEvent
{
    // zwraca nazwę zdarzenia
    public String getName();

    // zwraca liczbę parametrów przyjmowaną przez funkcję zdarzenia
    public int getNumberOfParameters();
}
```

6. Przykład użycia

Poniżej znajduje się kompletny przykład użycia języka, od instalacji, poprzez napisanie kilku programów, aż po kompilację i uruchomienie. Prezentowany przykład jest skierowany dla systemu operacyjnego linux, ale jest też możliwe używanie kompilatora i interpretera w systemie operacyjnym windows. Wymaganiem do używania kompilatora i interpretera jest zainstalowana java w wersjii 1.8.

6.1 Instalacja

Najprostszym sposobem instalacji kompilatora i interpretera jest użycie instalatora. Znajduje się on w załączniku pod nazwą bio_installer.jar. Po pobraniu należy go uruchomić jako administrator. Możemy to zrobić z poziomu linii poleceń wpisując komendę:

```
sudo java -jar bio_installer.jar
```

Po uruchomieniu instalator przeprowadzi nas przez proces instalacji. Poza skopiowaniem kluczowych elementów jakimi są pliki wykonywalne kompilatora i interpretera, a także stworzenie skryptów pozwalających je uruchamiać w katalogu /usr/local/bin/, mamy możliwość wybrania dodatkowych opcji. Jedną z nich jest dodatek do edytora tekstu vim, który włącza kolorowanie składni dla języka BIO. Jeżeli proces instalacji przebiegł pomyślnie do naszej dyspozycji pojawią się dwa programy: kompilator (*bioc*) i interpreter (*bio*).

6.2 Pierwszy program

Aby pokazać sposób kompilacji i uruchamiania programów, napiszemy standardową aplikację typu „hello world”. Kod tej aplikacji wygląda następująco:

```
@ program wypisze na ekran tekst "Hello world!"
def onSTART()
   PRINTLN("Hello World!")
end
```

Załóżmy, że plik z tym kodem nazywa się „hello_world.bio”. Aby go skompilować i zapisać pod nazwą „hello_world.cbio” należy wpisać następującą komendę:

```
bioc hello_world.bio -o hello_world.cbio
```

Flaga -o informuje kompilator do jakiego pliku należy zapisać wygenerowany kod pośredni. Jeżeli nie podamy tej nazwy to kompilator domyślnie zapisze kod w pliku „a.cbio”.

Aby uruchomić tak skompilowany program należy wpisać komendę:

```
bio hello_world.cbio
```

Na ekranie konsoli powinien pojawić się napis: „Hello world!”.

6.3 Aplikacja do sumowania wektorów

Drugi przykład będzie bardziej złożony i zaprezentuje większość z dostępnych funkcjonalności języka BIO. Będzie to aplikacja, której będziemy jako argumenty podawać składowe x i y wektorów, a która dokona sumowania tych składowych i wypisze tą sumę na ekranie. Przykład użycia:

```
bio sum_vect.cbio 1.5 2.5 4 6.75 0.45 -2.4
```

Wynik:

```
Vector sum: <x: 5.95, y: 6.85>
```

Aplikacja będzie się składać z dwóch plików:

1. vect.biom – moduł do działań na wektorach.

2. sum_vect.bio – główny plik z funkcją onSTART.

6.3.1 vect.biom

Oto funkcje wraz z objaśnieniami na które składa się moduł vect.biom:

1. create_vect

```
@ tworzy wektor (struktura z polami x i y)
@ domyślne wartości parametrów x i y to 0.0
def create_vect(x=0.0, y=0.0)
   @ stwórz strukturę v z polami x i y
   @ przypisz im wartości x i y
   AS_LOC(v.x, x, v.y, y)
   RET(v)
end
```

2. dist_vect

```
@ zwraca dystans pomiędzy dwoma wektorami
def dist_vect(vec1, vec2)
   RET
   (
   | @ przykład użycia parsera matematyczno-logicznego
   | @ odpowiednikiem poniższego kodu bez użycia parsera byłoby:
   | @ SQRT( ADD ( POW ( SUB ( vec1.x, vec2.x ), 2 ), POW ( SUB ( vec1.y, vec2.y ), 2 ) ) )
   | SQRT ( { (vec1.x - vec2.x)^2 + (vec1.y - vec2.y)^2 } )
   )
end
```

3. add_vector

```
@ zwraca nowy wektor będący wynikiem dodawania dwóch podanych wektorów
def add_vector(vec1, vec2)
   AS_LOC(new_vec.x, { vec1.x + vec2.x })
   AS_LOC(new_vec.y, { vec1.y + vec2.y })
   RET(new_vec)
end
```

4. sub_vector

```
@ zwraca nowy wektor będący wynikiem odejmowania dwóch podanych wektorów
def sub_vector(vec1, vec2)
   AS_LOC(new_vec.x, { vec1.x - vec2.x })
   AS_LOC(new_vec.y, { vec1.y - vec2.y })
   RET(new_vec)
end
```

5. perform_op

```
@ wykonuje funkcję podaną jako parametr na dwóch wektorach
def perform_op(vec1, vec2, op)
   @ przykład użycia mechanizmu refleksji
   @ wywołanie funkcji na podstawie jej nazwy przekazanej w zmiennej typu string
   RET(CALL_BY_NAME(op, vec1, vec2))
end
```

6. onEACH_vect_op

```
@ funkcja dla FOREACH wykonująca operację na wszystkich wektorach i zwracająca nowy
@ lista operacji:
@ 1 - dodawanie
@ 2 - odejmowanie
def onEACH_vect_op(el, obj)
   @ inicjalizacja
   IF(NOT(HAS_FIELD(obj, new_vect)), AS_LOC(obj.new_vect, create_vect()))
   @ sprawdzenie jaką funkcję mamy wywołać
   IF({ obj.add_par == 1}, AS_LOC(op, "add_vector"), AS_LOC(op, "sub_vector"))
   @ wywołanie funkcji
   AS_LOC(obj.new_vect, perform_op(obj.new_vect, el.val, op))
   @ zwrócenie nowego wektora
   RET(obj.new_vect)
end
```

6.3.2 sum_vect.bio

Oto kod pliku sum_vect.bio wraz z objaśnieniami:

```
@ załączenie pliku list.biom z standardowej lokalizacji
@ zawiera funkcje do operacji na listach (create_list, append_to_list, ...)
#INCLUDE("<stdlib/list.biom>")

@ załączenie pliku vect.biom
@ zawiera funkcje do operacji na wektorach (create_vect, add_vector, ...)
#INCLUDE("./vect.biom")

@ przykładowa funkcja-obserwator dla zdarzenia onUNHANDLED_ERROR
def error_handler_1(err)
   PRINTLN("error_handler_1", err)
end

@ przykładowa funkcja-obserwator dla zdarzenia onUNHANDLED_ERROR
def error_handler_2(err)
   PRINTLN("error_handler_2", err)
end

def onSTART(args)
   @ przypisanie obserwatora error_handler_1 do zdarzenia onUNHANDLED_ERROR
   ATTACH_TO_EVENT(onUNHANDLED_ERROR, error_handler_1)
   @ przypisanie obserwatora error_handler_2 do zdarzenia error_handler_1
   ATTACH_TO_EVENT(error_handler_1, error_handler_2)

   @ sprawdzamy czy podano dobrą ilość argumentów (nieparzystą)
   AS_LOC(args_size, SIZE(args))
   IF
   (
   |  { args_size % 2 == 0 },
   |  @ CALL to funkcja specjalna, która umożliwia nam na wykonanie więcej
   |  @ niż jednej funkcji w danym miejscu
   |  CALL
   |  (
   |  |  PRINTLN({ "Usage: " + args[0] + " x1 y1 ... xn yn" }),
   |  |  EXIT()
   |  )
   )

   @ tworzymy listę wektorów
   AS_LOC(vects, create_list())
   @ iterujemy po przekazanych argumentach od 1 co 2
   FOR
   (
   |  AS_LOC(i, 1),
   |  LS(i, args_size),
   |  CALL
   |  (
   |  |  @ pobierz współrzędna x, zamień na float i sprawdź czy nie error
   |  |  AS_LOC_RET_IF_ERR(x, TO_FLOAT({ args[i] })),
   |  |  @ pobierz współrzędna y, zamień na float i sprawdź czy nie error
   |  |  @ równoważne z kodem powyżej
   |  |  AS_LOC(y, TO_FLOAT({ args[i+1] })),
   |  |  IF(IS_ERROR(y), RET(y)),
   |  |  @ stwórz wektor i dodaj do listy
   |  |  @ przykład użycia nazywanych argumentów funkcji
   |  |  append_to_list(vects, create_vect(y=y, x=x))
   |  ),
   |  @ inkrementacja zmiennej i o 2
   |  AS_LOC(i, { i + 2 })
   )

   @ dodaj wszystkie wektory w liście
   @ przykład użycia funkcji FOREACH
   AS_LOC(new_vect, FOREACH(get_array_list(vects), onEACH_vect_op, 1))
   @ wypisanie wartości zsumowanych wektorów
   PRINT("Vector sum:", new_vect, "\n")
end
```

7. Zrealizowana funkcjonalność języka

Poniżej wymienione w punktach zostało jakie funkcjonalności języka udało się zrealizować:

1. Dwuetapowy proces tworzenia i uruchamiania programu – kompilator zamienia kod źródłowy na kod pośredni, który następnie jest wykonywany przez interpreter.

```
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/presentation $ bioc hello_world.bio
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/presentation $ bio a.cbio
Hello World!
```

2. Wbudowany wzorzec obserwatora – mechanizm pozwalający w łatwy sposób implementować programy oparte o ten wzorzec projektowy.

```
ATTACH_TO_EVENT(some_event, on_some_event)
IS_ATTACHED(some_event, on_some_event)
DETACH_FROM_EVENT(some_event, on_some_event)
```

3. Dynamicznie typowany – typy zmiennych nie są znane podczas kompilacji programu.

```
AS_LOC(a, 10)
AS_LOC(a, "text")
```

4. Zorientowany na programowanie proceduralne – możliwość tworzenia własnych funkcji.

```
def foo(arg1, arg2)
   PRINTLN(arg1, arg2)
end
```

5. 10 wbudowanych typów danych - “array” (tablica), “bool” (wartość logiczna), “dict” (kolekcja par: <”string”, wartość>), “error” (błąd), “float” (liczba zmiennoprzecinkowa pojedynczej precyzji w formacie “IEEE-754”), “int” (liczba całkowita ze znakiem 32 bitowa, w formacie kodu uzupełnień do dwóch), “none” (brak wartości), “string” (łańcuch znaków UTF-8), “struct” (struktura), “tuple” (krotka, odpowiednik tablicy ale bez możliwości zmiany jej elementów).

```
AS_LOC(arr, CREATE_ARRAY(10))
AS_LOC(flag, true)
AS_LOC(dict, CREATE_DICT("Jan", 1, "Feb", 2))
AS_LOC(err, CREATE_ERROR("Error message", none, none))
AS_LOC(float, 10.5)
AS_LOC(int, -25)
AS_LOC(none_val, none)
AS_LOC(string, "some text")
AS_LOC(person, CREATE_STRUCT(name, "Bob", age, 22))
AS_LOC(tuple, CREATE_TUPLE(10, "text", true))
```

6. Błędy są wartościami – jeżeli w funkcji nastąpi błąd, zwraca ona wartość typu “error”. Nieobsłużone błędy (tzn. nie przekazane do innej funkcji) skutkują wywołaniem zdarzenia onUNHANDLED_ERROR. Zdarzenie to jest wywoływane także w przypadku wartości typu “error” zwróconej z funkcji onSTART.

```
AS_LOC(a, TO_INT(true))
IF(IS_ERROR(a), PRINTLN("conversion error"))
PRINTLN(a)
```

7. Syntaktycznie wszystko jest funkcją – między innymi instrukcje sterujące przebiegiem programu jak IF i FOR.

```
FOR
(
  AS_LOC(i, 0),
  LS(i, 10),
  PRINTLN(i),
  INC(i)
)
```

8. Zmienne lokalne i globalne – możliwość zdefiniowania zmiennych lokalnych widocznych tylko dla danego wywołania funkcji lub zmiennych globalnych widocznych w każdej funkcji przez całe działanie programu.

```
AS_LOC(local_var, 10)
AS_GLOB(global_var, 10)
```

9. Preprocesor – posiadający dwie dyrektywy: #INCLUDE pozwalającą na dołączenie innego pliku i dbającą o to, żeby ten sam plik nie został dołączony dwa razy i #IMPORT pozwalająca na dołączenie opcjonalnego modułu.

10. Domyślna lokalizacja dla dyrektywy #INCLUDE – domyślny katalog od którego preprocesor wyszukuje pliki, jeżeli nazwę pliku zawrzemy w znakach < i >.

11. Opcjonalne moduły – możliwość dodawania opcjonalnych modułów rozszerzających możliwości języka. Moduł taki jest zdefiniowaniem i implementacją jednej lub więcej funkcji i/lub zdarzenia po stronie interpretera (a więc w przypadku tej implementacji w języku JAVA).

```
#INCLUDE("<stdlib/list.biom>")
#INCLUDE("../vect.biom")

#IMPORT("reflections")
```

12. Wbudowany parser matematyczno-logiczny – umożliwiający konstruowanie wyrażeń matematycznych i logicznych w formie znanej z innych języków programowania. Wyrażenia takie są zamieniane podczas procesu kompilacji na odpowiadający im kod języka BIO. Wyrażenia umieszczamy wewnątrz nawiasów wąsiastych: {}. Podobieństwo do trybu matematycznego w bashu.

```
AS_LOC(dist, SQRT({ vect.x ^ 2 + vect.y ^ 2 }))
```

13. Dynamiczne struktury – podobnie jak w języku MATLAB, możliwość tworzenia struktur "w locie" poprzez odwołanie do pola struktury.

```
AS_LOC(vect.x, 10, vect.y, 20)
```

14. Argumenty domyślne funkcji – możliwość definiowania funkcji o domyślnych wartościach argumentów. Domyślna wartość może być typem prostym lub wywołaniem funkcji.

```
def foo(arg1, arg2=10, arg3=CREATE_STRUCT())
```

15. Nazywane argumenty funkcji – podczas wywołania funkcji możemy podać nazwę argumentu dla którego podajemy daną wartość.

```
foo(arg2="text", arg1=true)
```

16. Argumenty cykliczne i opcjonalne funkcji wbudowanych – jako twórca modułu możliwość zdefiniowania argumentów cyklicznych (występujących zero lub więcej razy) lub opcjonalnych (występujących zero lub jeden raz).

```
UNPACK(a, b, c, RANGE(1, 3))
UNPACK(d, e, RANGE(0, 3))
AS_LOC(a, 1, b, 2, c, 3)
AS_LOC(d, 0, e, CR_TUP(1, 2, 3))
```

17. Funkcja FOREACH – umożliwiająca wywołanie przekazanej funkcji dla każdego elementu przekazanej kolekcji.

```
def onEACH_sum(el, obj)
   IF(NOT(HAS_FIELD(obj, sum)), AS_LOC(obj.sum, 0))
   AS_LOC(obj.sum, { obj.sum + el.val })
   RET(obj.sum)
end

def onSTART()
   PRINTLN(FOREACH(RANGE(1, 10), onEACH_sum))
end
```

18. Aliasy funkcji – wiele funkcji posiada alias swojej nazwy, co upraszcza pisania kodu.

```
AS_LOC(a, 10)
ASSIGN_LOCAL(a, 10)
```

19. Mechanizm refleksji – możliwość wywołania funkcji o nazwie podanej jako “string” (funkcja CALL_BY_NAME). Możliwość pobrania wszystkich funkcji użytkownika zdefiniowanych w programie (funkcja GET_USER_FUNCTIONS_NAMES).

```
CALL_BY_NAME("foo", 10, 20)
```

20. Napisany w języku BIO moduł do testów – możliwość łatwego testowania kodu przy użyciu modułu “test.biom”. Moduł definiuje 5 funkcji: bt_assert_eq, bt_assert_neq, bt_assert_true, bt_assert_false, bt_assert_error. Moduł wywołuje wszystkie funkcje zaczynające się od łańcucha “test”, i wyświetla nam podsumowanie testu.

```
#INCLUDE("<stdlib/test.biom>")

def test1()
   bt_assert_eq(10, 20)
end

def test2()
   bt_assert_true(false, "some message")
end
```

```
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/screens $ bioc tests.bio
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/screens $ bio a.cbio
test1
[In file tests.bio, line: 6, character: 4]: bt_assert_eq failed. Expected 20 got 10
test2
[In file tests.bio, line: 10, character: 4]: bt_assert_true failed: "some message"

SUMMARY
--------------
Some test passed with errors. Here is list of tests and number of errors:
test1: 1
test2: 1
```

21. Spójne raportowanie błędów – błędy preprocesora, leksykalne, syntaktyczne, semantyczne, optymalizacji i wykonania są raportowane w spójny sposób, gdzie podany jest zawsze plik w którym błąd wystąpił, z jakich plików ten plik jest dołączany, a także linia i znak wystąpienia błędu. Raportowane są wszystkie rodzaje błędów, w tym także takie jak: nieskończona pętla w programie lub nieskończona rekurencja w przypadku argumentów domyślnych funkcji.

```
def foo(arg1=foo())
   DN()
end
```

```
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/screens $ bioc compilation_err.bio
Semantic error: [In file compilation_err.bio, line: 3, character: 14]: Endless recursion when checking default parameters
```

22. Optymalizacja kodu przez kompilator – kompilator dokonuje analizy przepływu programu i eliminuje kod, który nigdy nie będzie wykonywany. Wyszukuje też nieskończone pętle i nie pozwala na skompilowanie programu, który takie pętle zawiera. Usuwa zbędne skoki. Dodatkowo dokonuje wielu pomniejszych optymalizacji tzw. strukturalnych, gdzie zastępuje lub eliminuje jedną lub więcej instrukcji mniejszą ich ilością. Możemy sprawdzić statystyki optymalizacji podając flagę: -s w kompilatorze.

```
OPTIMIZATION STATISTICS
------------------------
Optimization iterations: 3

Removed lines of code: 29.57% (34)
  Removed push sequences: 15.65% (18)
  Removed push bool jmp sequences: 2.61% (3)
  Removed popc jmp clear stack sequences: 4.35% (5)
  Removed clear stack at beginning: 0% (0)
  Removed unused code blocks lines: 6.09% (7)
  Removed jumps to next line: 0.86956520.87% (1)
Removed redundant jumps: 0
```

23. Deasemblacja kodu pośredniego – możliwość zobaczenia instrukcji kodu pośredniego zarówno po stronie kompilatora jak i intrepretera (flaga: -d).

```
DISASSEMBLE CODE
------------------------
<Modules>
observer,ints,strings,errors,arrays,logic,math,basic,type_check,conversion,compare,reflections,structs,io,iter,floats

<Error informations files>
1,hello_world.bio
-1,generated by compiler

<Functions>
onSTART
[0] push,string:"hello world!",2,12,1
[1] pop,1
[2] call_loc,PRINTLN,2,4,1
[3] clear_stack
[4] push,none:,-1,-1,-1
[5] pop,1
[6] call_loc,RETURN,-1,-1,-1
```

24. Wbudowany w interpreter profiler – uruchamiany poprzez flagę: -p profiler dostarczający informacji o czasie spędzonym w każdej funkcji (wbudowanej i użytkownika).

```
OUTPUT
------------------------
hello world!

PROFILER
------------------------
Function name        General time        Real time           Avg general time         Avg real time
onSTART              1.77503700 ms       1.20772200 ms       1.77503700 ms (1)        1.20772200 ms (1)
PRINTLN              0.55691100 ms       0.55691100 ms       0.55691100 ms (1)        0.55691100 ms (1)
RETURN               0.01040400 ms       0.01040400 ms       0.01040400 ms (1)        0.01040400 ms (1)
```

25. Obsługa UTF-8 - nazwy zmiennych i funkcji mogą zawierać dowolne znaki UTF-8, które są literami, cyframi lub znakiem podkreślenia.

```
AS_LOC(żółć, 10)
AS_LOC(æąœßż, "text")
```

26. Rozbudowana biblioteka standardowa – zawierająca 123 funkcje zorganizowane w 17 modułach. Funkcje do operacji na tablicach, relacyjne, porównujące, arytmetyczne, logiczne, bitowe, konwersji, wejścia/wyjścia, sprawdzania typów, operacji na tekście, błędach i więcej.

```
arrays.bd:21
basic.bd:9
compare.bd:7
conversion.bd:6
errors.bd:15
floats.bd:2
ints.bd:3
io.bd:3
iter.bd:1
logic.bd:3
math.bd:15
observer.bd:3
reflections.bd:2
special.bd:6
strings.bd:12
structs.bd:2
type_check.bd:13
total: 123
```

8. Podsumowanie

W pracy tej poza jednym wyjątkiem udało się osiągnąć wszystkie początkowe założenia. Wyjątkiem tym jest mechanizm wzorca obserwatora, którego można używać tylko dla funkcji użytkownika, podczas gdy pierwotnym założeniem były wszystkie funkcje (także wbudowane). Ograniczenie to spowodowane jest dużymi różnicami pomiędzy funkcjami użytkownika, a wbudowanymi, a głównie możliwością przyjmowania przez funkcje wbudowane argumentów cyklicznych. Poza tym nie ustrzeżono się również błędów przy implementacji. Jednym z nich jest zachowanie jakie dzieje się, gdy jako warunek funkcji FOR lub IF przekażemy parametr innego typu niż bool. Spodziewanym zachowaniem byłoby zwrócenie przez te funkcje wartości typu error, jednak w rzeczywistości zwracana jest wartość typu error, ale z funkcji w której zostały wywołane te funkcje. Jest to spowodowane złym zaprojektowaniem kodu pośredniego. Zaproponowane zostało rozwiązanie tego problemu poprzez wprowadzenie dwóch nowych instrukcji interpretera: **peek_jmp_if_not_bool** i **push_error**. Zaproponowane zostały także ulepszenia jakie można wprowadzić do języka. Są to między innymi:

- Dodanie zmiennych statycznych w funkcji.

- Dyrektywa #DEFINE preprocesora.

- Możliwość podawania funkcji w trybie matematyczno-logicznym.

- Nowa funkcja wbudowana: END_FOREACH, a także dodanie funkcji do obsługi plików.