Strona tytułowa

strona tytułowa c.d.

Merytoryczna ocena pracy przez opiekuna:

Merytoryczna ocena pracy przez recenzenta:

Spis treści

1. Wstęp....6
2. Porównanie implementacji wzorca obserwatora w różnych językach programowania....7
2.1 Opis wzorca....7
2.2 Java....8
2.3 C#....9
2.4 Python....10
2.5 BIO....11
3. Planowana funkcjonalność języka....12
4. Semantyka....13
4.1 Przypisanie....13
4.2 Instrukcja warunkowa IF....13
4.3 Pętla FOR....14
4.4 Definicja funkcji....14
4.5 Wywołanie funkcji....15
4.6 Wzorzec obserwatora....15
5. Realizacja....16
5.1 Kompilator....17
5.1.1 Preprocesor....18
5.1.2 Lekser....19
6. Zrealizowana funkcjonalność języka....22

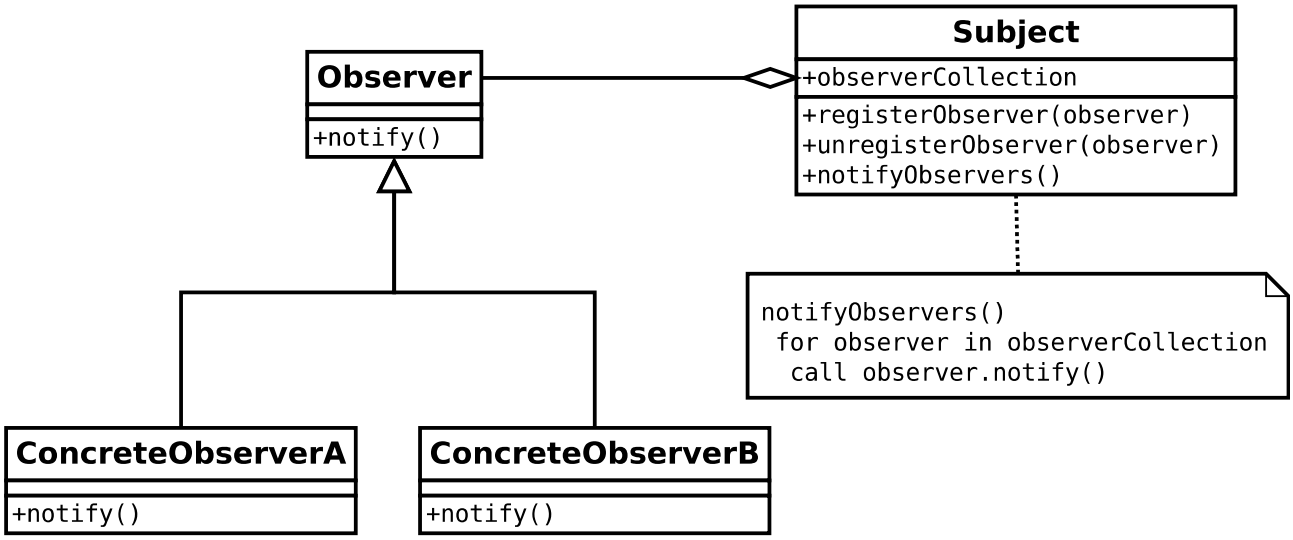
1. Wstęp

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*). Oprócz tego język BIO zawiera inne przydatne funkcjonalności ułatwiające pisanie w nim programów opisane w rozdziale 3. (*Funkcjonalność języka*). Pełne omówienie składni języka następuje w rozdziale (TODO). Sposób realizacji opisany został w rozdziale (TODO). Na końcu pracy w rozdziale (TODO) zostaje podany przykład instalacji, uruchomienia, a także dwa przykłady funkcjonalności.

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.

```
1  def attach_to_event(subject, observer):
2      subject.observers.append(observer)
3
4  def detach_from_event(subject, observer):
5      subject.observers.remove(observer)
6
7  def event(func):
8      def inner(*args, **kwargs):
9          func(*args, **kwargs)
10         for observer in inner.observers:
11             observer(*args, **kwargs)
12
13     inner.observers = []
14     return inner
15
16 @event
17 def some_event(data, data2):
18     print("some_event " + str(data) + str(data2))
19
20 def foo(data, data2):
21     print("foo " + str(data) + str(data2))
22
23 def foo2(data, data2):
24     print("foo2 " + str(data))
25
26 attach_to_event(some_event, foo)
27 attach_to_event(some_event, foo2)
28 some_event("test", " wiadomosc")
29 detach_from_event(some_event, foo)
30 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 możemy przypisać inną funkcję będącą obserwatorem (także funkcją wbudowanym jak PRINT). Żeby to osiągnąć wywołujemy funkcję ATTACH_TO_EVENT, której podajemy temat i obserwatora. Ilość parametrów przyjmowanych przez funkcję obserwatora musi być taka sama jak przyjmowana przez funkcję tematu.

```
1  def some_event(data, data2)
2     PRINT("some_event", TO_STR(data), TO_STR(data2), "\n")
3  end
4
5  def foo(data, data2)
6     PRINT("foo", TO_STR(data), TO_STR(data2), "\n")
7  end
8
9  def foo2(data, data2)
10    PRINT("foo2", TO_STR(data), "\n")
11 end
12
13 def onSTART()
14    ATTACH_TO_EVENT(some_event, foo)
15    ATTACH_TO_EVENT(some_event, foo2)
16    some_event("test", "wiadomość")
17    DETACH_FROM_EVENT(some_event, foo)
18    some_event("test", "wiadomość")
19 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)
```

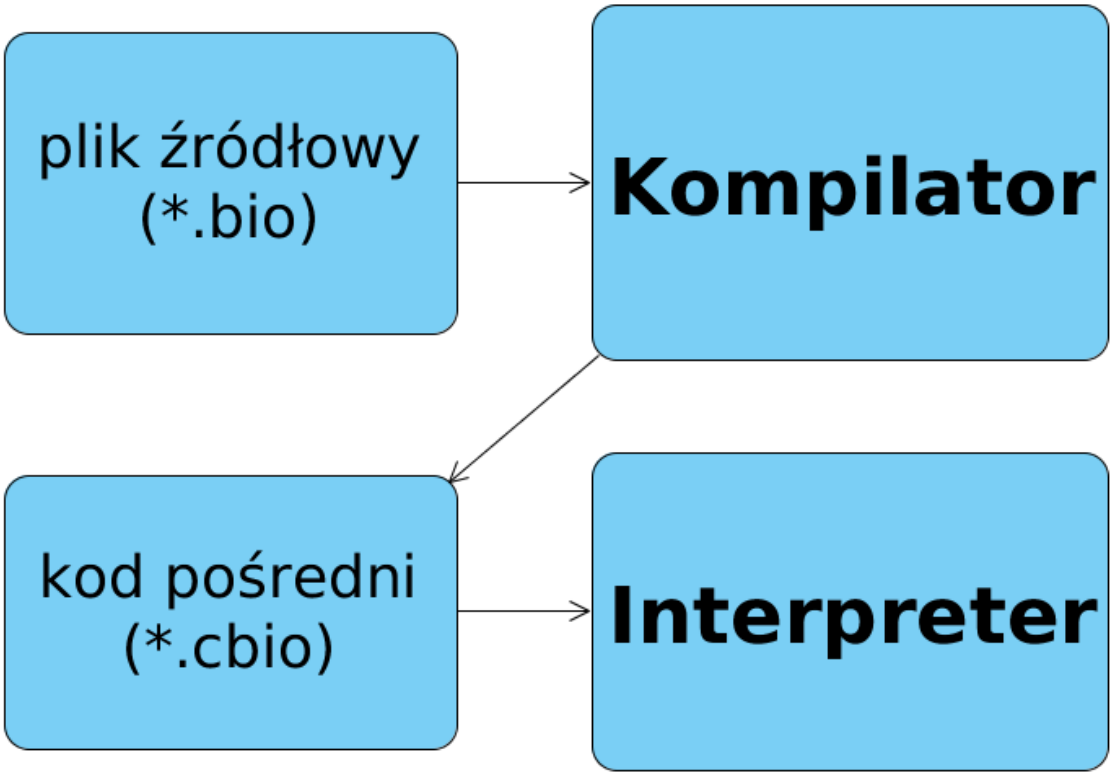
4.6 Wzorzec obserwatora

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:

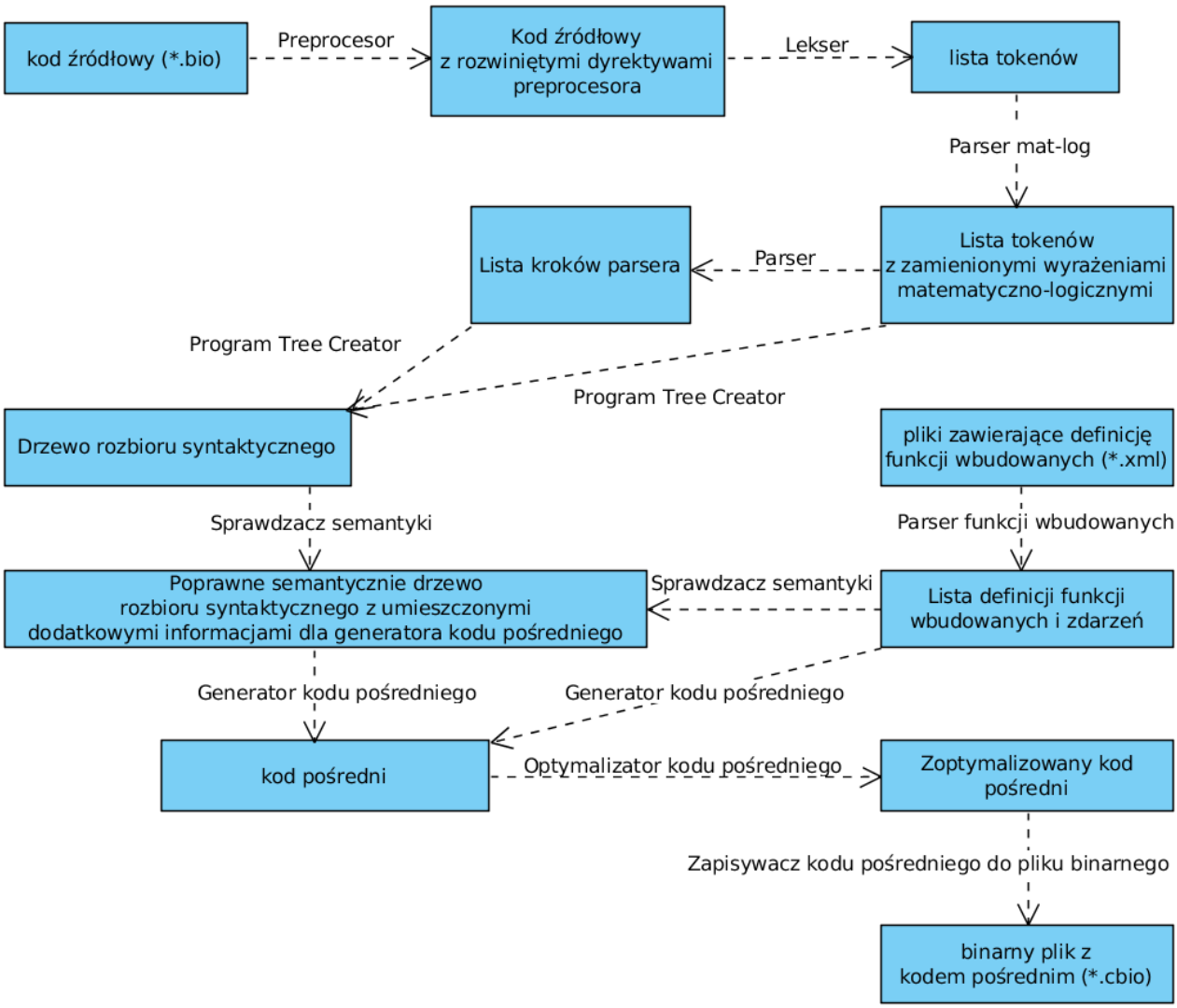


Rys.5.1. Diagram obrazujący proces zamienienia pliku źródłowego w wykonywalny kod (źródło: własne)

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")).

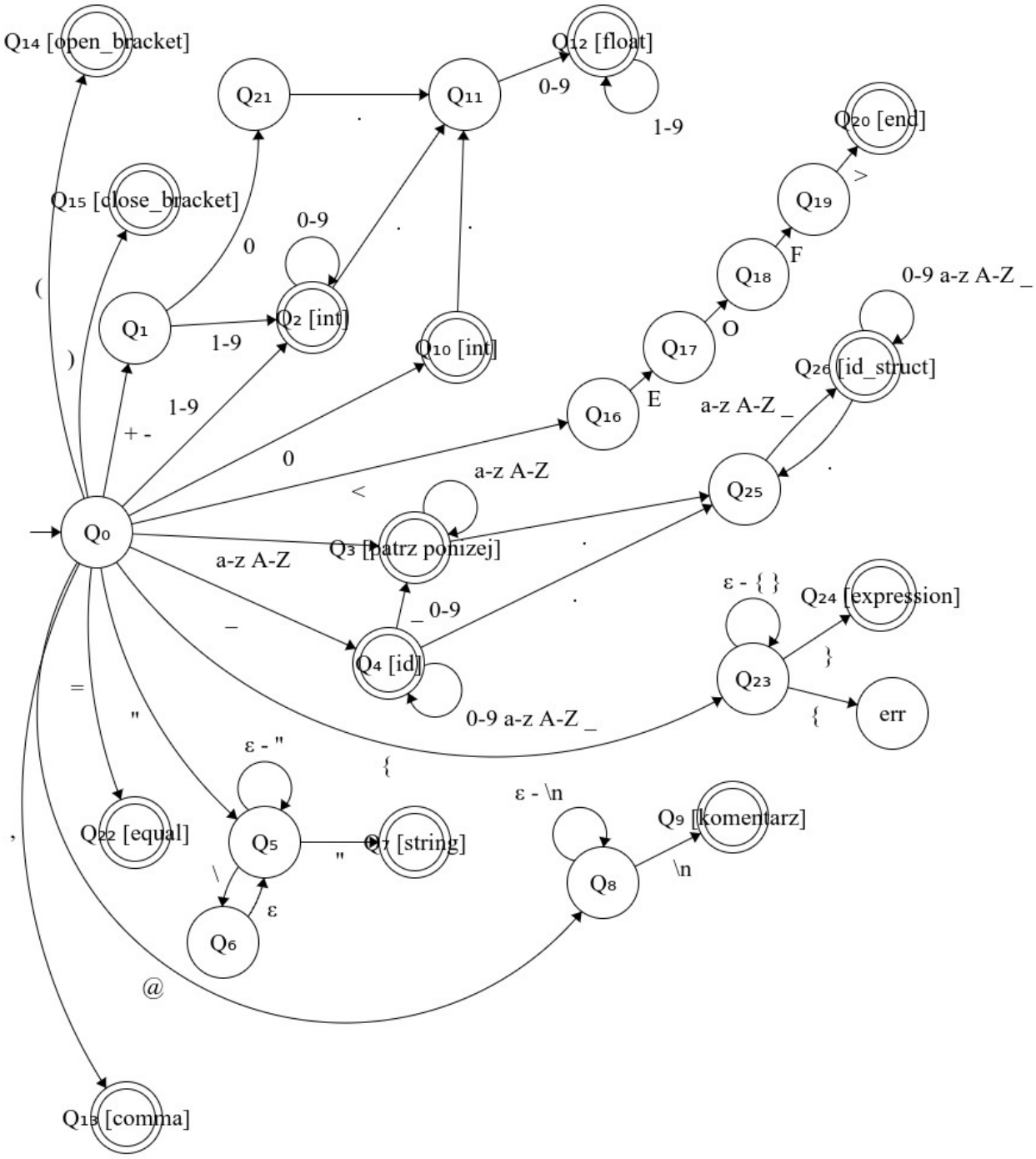
Dodatkowo preprocesor zamieszcza informację, która linia należy do którego pliku i który plik został załączony z którego. Jest to wykorzystywane potem przy raportowaniu prawie wszystkich błędów zarówno kompilacji jak i wywołania.

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 (TODO).



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.

6. 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/features $ bioc hello_world.bio
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/features $ bio a.cbio
hello world!
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/features $
```

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"))
```

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/features $ bioc tests.bio
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/features $ bio a.cbio
test1
[In file tests.bio, line: 5, character: 4]: bt_assert_eq failed. Expected 20 got 10
test2
[In file tests.bio, line: 9, 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
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/features $
```

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 linię 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/features $ bioc compilation_err.bio
Semantic error: [In file compilation_err.bio, line: 2, character: 14]: Endless recursion when checking default parame
ters
```

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).

```
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/features $ bio -d a.cbio
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).

```
robert@robert-desktop ~/programowanie/BIO/bsc_thesis/features $ bio -p a.cbio
OUTPUT
------------------------
hello world!

PROFILER
------------------------
Function name         General time         Real time            Avg general time          Avg real time
onSTART               1.82801900 ms        1.19559900 ms        1.82801900 ms (1)         1.19559900 ms (1)
PRINTLN               0.62293900 ms        0.62293900 ms        0.62293900 ms (1)         0.62293900 ms (1)
RETURN                0.00948100 ms        0.00948100 ms        0.00948100 ms (1)         0.00948100 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
```