

**Akademia Górniczo-Hutnicza  
im. Stanisława Staszica w Krakowie**

---

Wydział Elektrotechniki, Automatyki, Informatyki i Inżynierii Biomedycznej

KATEDRA INFORMATYKI STOSOWANEJ



**PRACA MAGISTERSKA**

**RAFAŁ STUDNICKI**

**PODSTAWOWA FUNKCJONALNOŚĆ ERLANGA DLA SYSTEMU  
FREERTOS**

PROMOTOR:  
dr inż. Piotr Matyasik

Kraków 2014

## **OŚWIADCZENIE AUTORA PRACY**

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.

.....

PODPIS

**AGH**  
**University of Science and Technology in Krakow**

---

Faculty of Electrical Engineering, Automatics, Computer Science and Engineering in  
Biomedicine

DEPARTMENT OF APPLIED COMPUTER SCIENCE



**MASTER OF SCIENCE THESIS**

**RAFAŁ STUDNICKI**

**IMPLEMENTATION OF BASIC FEATURES OF ERLANG FOR  
FREERTOS**

SUPERVISOR:  
Piotr Matyasik Ph.D

Krakow 2014



## Spis treści

<b>A. Kompilacja kodu źródłowego</b>	7
A.1. Wprowadzenie	7
A.2. Kod źródłowy	7
A.3. Preprocessing	8
A.4. Transformacje drzewa syntaktycznego	9
A.5. Kod pośredni ( <i>bytecode</i> )	10
A.6. Plik binarny BEAM	11
A.6.1. Tablica atomów	13
A.6.2. Kod pośredni	13
A.6.3. Tablica importowanych funkcji	14
A.6.4. Tablica eksportowanych funkcji	15
A.6.5. Tablica funkcji lokalnych	15
A.6.6. Tablica <i>lambda</i>	16
A.6.7. Tablica stałych	17
A.6.8. Lista atrybutów modułu	17
A.6.9. Lista dodatkowych informacji o kompilacji modułu	18
A.6.10. Tablica linii kodu źródłowego modułu	18
A.6.11. Drzewo syntaktyczne modułu	19
<b>B. Lista instrukcji maszyny wirtualnej BEAM</b>	20
B.1. Typy argumentów	20
B.2. Lista instrukcji	22
<b>Bibliografia</b>	29

## A. Kompilacja kodu źródłowego

Dodatek opisuje kolejne kroki, z jakich składa się proces otrzymywania skompilowanego kodu pośredniego maszyny wirtualnej BEAM z kodu źródłowego napisanego w języku Erlang. Oprócz tego dodatek dokumentuje, na potrzeby projektu, zawartość pliku ze skompilowanym kodem pośrednim. Format pliku nie jest objęty oficjalną dokumentacją języka ze względu na dużą zmienność pomiędzy kolejnymi wersjami kompilatora i maszyny wirtualnej.

### A.1. Wprowadzenie

Jak zostało wspomniane w ??, program napisany w języku Erlang wykonywany jest na dedykowanej do tego celu maszynie wirtualnej.

Narzędzia przeznaczone do operacji opisanych w niniejszym rozdziale zostały napisane w języku Erlang i dostępne są w aplikacji **compiler** dostarczanej wraz z maszyną wirtualną BEAM.

### A.2. Kod źródłowy

```
1 | -module(fac) .
2 |
3 | -export([fac/1]) .
4 | -define(ERROR, "Invalid argument") .
5 |
6 | -include("fac.hrl") .
7 |
8 | fac(#factorial{n=0, acc=Acc}) ->
9 |     Acc;
10 | fac(#factorial{n=N, acc=Acc}) ->
11 |     fac(#factorial{n=N-1, acc=N*Acc});
12 | fac(N) when is_integer(N) ->
13 |     fac(#factorial{n=N});
14 | fac(N) when is_binary(N) ->
15 |     fac(binary_to_integer(N));
16 | fac(_) ->
17 |     {error, ?ERROR}.
```

Listing A.1: Plik fac.erl

```
1 || -record(factorial, {n, acc=1}).
```

Listing A.2: Plik fac.hrl

## A.3. Preprocessing

```
1 || -file("fac.erl", 1).
2 ||
3 || -module(fac).
4 ||
5 || -export([fac/1]).
6 ||
7 || -file("fac.hrl", 1).
8 ||
9 || -record(factorial, {n, acc = 1}).
10 ||
11 || -file("fac.erl", 7).
12 ||
13 || fac(#factorial{n = 0, acc = Acc}) ->
14 ||     Acc;
15 || fac(#factorial{n = N, acc = Acc}) ->
16 ||     fac(#factorial{n = N - 1, acc = N * Acc});
17 || fac(N) when is_integer(N) ->
18 ||     fac(#factorial{n = N});
19 || fac(N) when is_binary(N) ->
20 ||     fac(binary_to_integer(N));
21 || fac(_) ->
22 ||     {error, "Invalid argument"}.
```

Listing A.3: Moduł fac po pierwszym przetworzeniu

```
1 || -file("fac.erl", 1).
2 ||
3 || -file("fac.hrl", 1).
4 ||
5 || -file("fac.erl", 7).
6 ||
7 || fac({factorial, 0, Acc}) ->
8 ||     Acc;
9 || fac({factorial, N, Acc}) ->
10 ||     fac({factorial, N - 1, N * Acc});
11 || fac(N) when is_integer(N) ->
12 ||     fac({factorial, N, 1});
13 || fac(N) when is_binary(N) ->
14 ||     fac(binary_to_integer(N));
15 || fac(_) ->
16 ||     {error, "Invalid argument"}.
```

```

18 module_info() ->
19     erlang:get_module_info(fac).
20
21 module_info(X) ->
22     erlang:get_module_info(fac, X).

```

Listing A.4: Moduł fac po drugim przetworzeniu

## A.4. Transformacje drzewa syntaktycznego

```

1  [{attribute,1,file,{"fac.erl",1}},
2   {attribute,1,module,fac},
3   {attribute,5,export,[{fac,1}]},
4   {attribute,1,file,{"fac.hrl",1}},
5   {attribute,1,record,
6     {factorial,
7       [{record_field,1,{atom,1,n}},
8        {record_field,1,{atom,1,acc},{integer,1,1}}]}},
9   {attribute,9,file,{"fac.erl",9}},
10  {function,10,fac,1,
11    [{clause,10,
12      [{record,10,factorial,
13        [{record_field,10,{atom,10,n},{integer,10,0}},
14         {record_field,10,{atom,10,acc},{var,10,'Acc'}}]}],
15      [],
16      [{var,11,'Acc'}]},
17     {clause,12,
18       [{record,12,factorial,
19         [{record_field,12,{atom,12,n},{var,12,'N'}},
20          {record_field,12,{atom,12,acc},{var,12,'Acc'}}]}],
21       [],
22       [{call,13,
23         {atom,13,fac},
24         [{record,13,factorial,
25           [{record_field,13,
26             {atom,13,n},
27             {op,13,'-',{var,13,'N'},{integer,13,1}}},
28            {record_field,13,
29              {atom,13,acc},
30              {op,13,'*',{var,13,'N'},{var,13,'Acc'}}}}]}],
31        {clause,14,
32          [{var,14,'N'}],
33          [{call,14,{atom,14,is_integer},{var,14,'N'}}]},
34          [{call,15,
35            {atom,15,fac},
36            [{record,15,factorial,
37              [{record_field,15,{atom,15,n},{var,15,'N'}}]}]}],
38        {clause,16,

```



```

39         [{var,16,'N'}],
40         [{call,16,{atom,16,is_binary},{var,16,'N'}}]],
41         [{call,17,
42             {atom,17,fac},
43             [{call,17,{atom,17,binary_to_integer},{var,17,'N'}}]]}],
44     {clause,18,
45         [{var,18,'_'}],
46         [],
47         [{tuple,19,[{atom,19,error},{string,19,"Invalid argument"}]}]}],
48     {eof,20}]

```

Listing A.5: Drzewo syntaktyczne modułu fac

## A.5. Kod pośredni (bytecode)

```

1  {module, fac}. %% version = 0
2
3  {exports, [{fac,1},{module_info,0},{module_info,1}]}.
4
5  {attributes, []}.
6
7  {labels, 11}.
8
9
10 {function, fac, 1, 2}.
11   {label,1}.
12     {line,[{location,"fac.erl",8}]}.
13     {func_info,{atom,fac},{atom,fac},1}.
14   {label,2}.
15     {test,is_tuple,{f,4},{x,0}}.
16     {test,test_arity,{f,4},{x,0},3}.
17     {get_tuple_element,{x,0},0,{x,1}}.
18     {get_tuple_element,{x,0},1,{x,2}}.
19     {get_tuple_element,{x,0},2,{x,3}}.
20     {test,is_eq_exact,{f,4},{x,1},{atom,factorial}}.
21     {test,is_eq_exact,{f,3},{x,2},{integer,0}}.
22     {move,{x,3},{x,0}}.
23     return.
24   {label,3}.
25     {line,[{location,"fac.erl",11}]}.
26     {gc_bif,'-',{f,0},4,[{x,2},{integer,1}],{x,0}}.
27     {line,[{location,"fac.erl",11}]}.
28     {gc_bif,'*',{f,0},4,[{x,2},{x,3}],{x,1}}.
29     {test_heap,4,4}.
30     {put_tuple,3,{x,2}}.
31     {put,{atom,factorial}}.
32     {put,{x,0}}.
33     {put,{x,1}}.

```

```

34     {move, {x, 2}, {x, 0}}.
35     {call_only, 1, {f, 2}}.
36 {label, 4}.
37     {test, is_integer, {f, 5}, [{x, 0}]}.
38     {test_heap, 4, 1}.
39     {put_tuple, 3, {x, 1}}.
40     {put, {atom, factorial}}.
41     {put, {x, 0}}.
42     {put, {integer, 1}}.
43     {move, {x, 1}, {x, 0}}.
44     {call_only, 1, {f, 2}}.
45 {label, 5}.
46     {test, is_binary, {f, 6}, [{x, 0}]}.
47     {allocate, 0, 1}.
48     {line, [{location, "fac.erl", 15}]}.
49     {call_ext, 1, {extfunc, erlang, binary_to_integer, 1}}.
50     {call_last, 1, {f, 2}, 0}.
51 {label, 6}.
52     {move, {literal, {error, "Invalid argument"}}, {x, 0}}.
53     return.
54
55
56 {function, module_info, 0, 8}.
57     {label, 7}.
58     {line, []}.
59     {func_info, {atom, fac}, {atom, module_info}, 0}.
60 {label, 8}.
61     {move, {atom, fac}, {x, 0}}.
62     {line, []}.
63     {call_ext_only, 1, {extfunc, erlang, get_module_info, 1}}.
64
65
66 {function, module_info, 1, 10}.
67     {label, 9}.
68     {line, []}.
69     {func_info, {atom, fac}, {atom, module_info}, 1}.
70 {label, 10}.
71     {move, {x, 0}, {x, 1}}.
72     {move, {atom, fac}, {x, 0}}.
73     {line, []}.
74     {call_ext_only, 2, {extfunc, erlang, get_module_info, 2}}.

```

Listing A.6: Bytecode modułu fac

## A.6. Plik binarny BEAM

Efektem przetworzenia kodu pośredniego, wyrażonego w postaci krotek, jest plik binarny w formacie IFF [2], w formacie zrozumiałym przez maszynę wirtualną BEAM. Maszyna ta wykorzystuje tego

rodzaju pliki do ładowania kodu poszczególnych modułów do pamięci. Ich źródłem może być zarówno system plików na fizycznej maszynie, na której uruchomiony został BEAM, jak i inna maszyna wirtualna znajdująca się w tym samym klastrze *Distributed Erlang*, co docelowa.

W tabeli A.1 zaprezentowana została ogólna struktura pliku binarnego ze skompilowanym modułem.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	"FOR1"															
4	32	Rozmiar pliku bez pierwszych 8 bajtów															
8	64	"BEAM"															
12	96	Identyfikator fragmentu ( <i>chunk</i> ) 1															
16	128	Rozmiar fragmentu 1															
20	160	Dane fragmentu 1															
...	...	Identyfikator fragmentu ( <i>chunk</i> ) 2															
...	...	...															

Tablica A.1: Struktura pliku modułu BEAM

Każdy plik binarny BEAM powinien zawierać przynajmniej 4 następujące fragmenty (*chunki*). Obok opisu każdego fragmentu, w nawiasie podano ciąg znaków będący jego identyfikatorem w binarnym pliku modułu:

- tablica atomów wykorzystywanych przez moduł (*Atom*);
- kod pośredni danego modułu (*Code*);
- tablica zewnętrznych funkcji używanych przez moduł (*ImpT*);
- tablica funkcji eksportowanych przez moduł (*ExpT*).

Ponadto, w pliku mogą znajdować się następujące fragmenty:

- tablica funkcji lokalnych dla danego modułu (*LocT*);

- tablica lambd wykorzystywanych przed moduł (`FunT`);
- tablica stałych wykorzystywanych przed moduł (`LitT`);
- lista atrybutów modułu (`Attr`);
- lista dodatkowych informacji o kompilacji modułu (`CInf`);
- tablica linii kodu źródłowego modułu (`Line`);
- drzewo syntaktyczne modułu (`Abst`).

W przypadku każdego rodzaju fragmentu, obszar pamięci jaki zajmuje on w pliku jest zawsze wielokrotnością 4 bajtów. Nawet jeżeli nagłówek fragmentu, zawierający jego rozmiar nie jest podzielny przez 4, obszar zaraz za danym fragmentem dopełniany jest zerami do pełnych 4 bajtów.

Warto zaznaczyć również, że sposób implementacji maszyny wirtualnej BEAM nie definiuje kolejności w jakiej poszczególne fragmenty powinny występować w pliku binarnym.

### A.6.1. Tablica atomów

Tablica atomów zawiera listę wszystkich atomów, które używane są przez dany moduł. W trakcie ładowania kodu modułu przez maszynę wirtualną, atomy, które nie występowały we wcześniej załadowanych modułach, zostają wstawione do globalnej tablicy atomów (w postaci tablicy z hashowaniem).

Ponieważ długość atomu zapisana jest na jednym bajcie, nazwa atomu może mieć maksymalnie 255 znaków.

Fragment pliku binarnego z tablicą atomów reprezentowany jest przez napis `Atom`. Struktura danych fragmentu zaprezentowana jest w tabeli A.2.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Ilość atomów w tablicy atomów															
4	32	Dł. atomu 1								Nazwa atomu 1 w ASCII							
...	...	Dł. atomu 2								Nazwa atomu 2 w ASCII							
...	...	...															

Tablica A.2: Struktura tablicy atomów w pliku BEAM

### A.6.2. Kod pośredni

Sekcja z kodem pośrednim zawiera faktyczny kod wykonywalny modułu, który jest interpretowany przez maszynę wirtualną w trakcie uruchomienia systemu. Szczegółowy opis reprezentacji i znaczenia

opkodów i ich argumentów zawarty został w dodatku B.

Fragment pliku z kodem identyfikowana jest przez napis `Code`. Struktura danych fragmentu zawarta została w tabeli A.3.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	0x000010															
4	32	Numer wersji formatu kod (w Erlangu R16 - 0x00000000)															
8	64	Maksymalny numer operacji (do sprawdzenia kompatybilności)															
12	96	Liczba etykiet w kodzie modułu															
16	128	Liczba funkcji eksportowanych z modułu															
20	160	Opkod 1								Argument 1							
...	...	...								Argument N							
...	...	Opkod 2								Argument 1							
...	...	...															

Tablica A.3: Struktura kodu pośredniego w pliku BEAM

### A.6.3. Tablica importowanych funkcji

Fragment pliku binarnego z tablicą importowanych funkcji zawiera informacje o funkcjach zaimplementowanych w innych modułach, które są wykorzystywane przez moduł.

Identyfikowany jest on przez napis `ImpT`. Struktura danych fragmentu zawarta została w tabeli A.4.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Liczba importowanych funkcji															
4	32	Indeks atomu z nazwą modułu 1															
8	64	Indeks atomu z nazwą funkcji 1															
12	96	Arność funkcji 1															

<b>16</b>	<b>128</b>	Indeks atomu z nazwą modułu 2
...	...	...

Tablica A.4: Struktura tablicy importowanych funkcji w pliku BEAM

#### A.6.4. Tablica eksportowanych funkcji

Fragment pliku binarnego z tablicą eksportowanych funkcji zawiera informacje o funkcjach z modułu, które widoczne są z poziomu innych modułów.

Identyfikowany jest on przez napis `ExpT`. Struktura danych fragmentu zawarta została w tabeli A.5.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<b>0</b>	<b>0</b>	Liczba eksportowanych funkcji															
<b>4</b>	<b>32</b>	Indeks atomu z nazwą funkcji 1															
<b>8</b>	<b>64</b>	Arność funkcji 1															
<b>12</b>	<b>96</b>	Etykieta początku kodu funkcji 1															
<b>16</b>	<b>128</b>	Indeks atomu z nazwą funkcji 2															
...	...	...															

Tablica A.5: Struktura tablicy eksportowanych funkcji w pliku BEAM

#### A.6.5. Tablica funkcji lokalnych

Fragment pliku binarnego z tablicą lokalnych funkcji zawiera informacje o funkcjach zaimplementowanych w module (w tym `lambda`), które wykorzystywane są tylko przez ten moduł i nie są widoczne z poziomu innych modułów.

Identyfikowany jest on przez napis `LocT`. Struktura danych fragmentu zawarta została w tabeli A.6.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

<b>0</b>	<b>0</b>	Liczba lokalnych funkcji
<b>4</b>	<b>32</b>	Indeks atomu z nazwą funkcji 1
<b>8</b>	<b>64</b>	Arność funkcji 1
<b>12</b>	<b>96</b>	Etykieta początku kodu funkcji 1
<b>16</b>	<b>128</b>	Indeks atomu z nazwą funkcji 2
<b>...</b>	<b>...</b>	...

Tablica A.6: Struktura tablicy lokalnych funkcji w pliku BEAM

#### A.6.6. Tablica lambda

Fragment pliku binarnego z tablicą lambda zawiera informacje o obiektach funkcyjnych, które wykorzystywane są przez ten moduł.

Lambdy identyfikowane są poprzez atomy, które powstały przez złączenie nazwy funkcji, w której zostały zdefiniowane oraz kolejny indeks lambdy zdefiniowanej w danej funkcji. Np. kolejne obiekty funkcyjne zdefiniowane w funkcji `foo/1` będą identyfikowane przez atomy `-foo/1-fun-0-`, `-foo/1-fun-1-` itd.

Fragment pliku tablicą lambda identyfikowany jest przez napis `FunT`. Struktura danych fragmentu zawarta została w tabeli A.7.

	Oktet	0										1					
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<b>0</b>	<b>0</b>	Liczba lambda w module															
<b>4</b>	<b>32</b>	Indeks atomu z identyfikatorem lambda 1															
<b>8</b>	<b>64</b>	Arność lambda 1															
<b>12</b>	<b>96</b>	Etykieta początku kodu lambda 1															
<b>16</b>	<b>128</b>	Indeks lambda 1 (0x00)															
<b>20</b>	<b>160</b>	Liczba wolnych zmiennych w lambdzie 1															
<b>24</b>	<b>192</b>	Wartość skrótu z drzewa syntaktycznego kodu lambda 1															

28	224	Indeks atomu z identyfikatorem lambdy 2
...	...	...

Tablica A.7: Struktura tablicy lambd w pliku BEAM

### A.6.7. Tablica stałych

Fragment pliku binarnego z tablicą lambd stałych zawiera informacje o stałych (listy, napisy, duże liczby) które wykorzystywane są przez ten moduł.

Właściwa lista wartości stałych (od bajtu 4 do końca fragmentu) przechowywana jest w pliku w postaci skompresowanej algorytmem **zlib**. Stałe zapisane są w formacie binarnym w formacie *External Term Format*, opisanym w dokumencie [1].

Fragment pliku z tablicą identyfikowany jest przez napis `LitT`. Struktura danych fragmentu zawarta została w tabeli A.8.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Rozmiar tablicy w bajtach															
4	32	Liczba stałych															
8	64	Rozmiar stałej 1 w bajtach															
12	96	Stała 1 w External Term Format															
...	...	Rozmiar stałej 2 w bajtach															
...	...	...															

Tablica A.8: Struktura tablicy stałych w pliku BEAM

### A.6.8. Lista atrybutów modułu

Fragment pliku binarnego z listą atrybutów modułu zawiera listę dwójek (proplistę) ze wszystkimi dodatkowymi atrybutami, z jakimi został skompilowany dany moduł (np. informacje o wersji czy autorze). Lista ta zapisana jest binarnie w postaci *External Term Format*.



Fragment ten reprezentowany jest przez napis `Attr`.

### A.6.9. Lista dodatkowych informacji o kompilacji modułu

Fragment pliku binarnego z listą informacji o kompilacji modułu zawiera proplisę z informacjami dotyczącymi kompilacji, takimi jak: ścieżka pliku z kodem źródłowym, czas kompilacji, wersja kompilatora czy użyte opcje kompilacji. Informacje te zapisane są binarnie w postaci *External Term Format*.

Fragment ten reprezentowany jest przez napis `CInf`.

### A.6.10. Tablica linii kodu źródłowego modułu

Fragment pliku binarnego z informacjami o liniach kodu źródłowego modułu zawiera informacje dla instrukcji `line/1` maszyny wirtualnej o pliku źródłowym i linii, z której pochodzi aktualnie wykonywany fragment kodu. Informacje te wykorzystywane są przy generowaniu stosu wywołań przy wystąpieniu błędu lub wyjątku. Funkcjonalność ta została wprowadzona dopiero w wersji R15 maszyny wirtualnej BEAM.

Jeżeli kompilowany plik jest na etapie preprocessingu łączony z innymi plikami z kodem źródłowym (poprzez użycie atrybutu `include`) to informacja o tych plikach zostanie zawarta w tym fragmencie. Domyślnie, kompilowany plik nie zostanie uwzględniony i zostanie przydzielony mu indeks 0.

Numer linii koduje się przy użyciu tagu `0001`, jak w przypadku argumentów instrukcji maszyny wirtualnej, opisanych w sekcji B.1. Rozróżnienie pliku, z którego pochodzi linia odbywa się za pomocą zapamiętania, z którego pliku pochodziła ostatnia linia. Domyślnie jest to plik o indeksie 0. Jeżeli dochodzi do zmiany aktualnego pliku, kolejny numer linii poprzedzony jest indeksem pliku z którego pochodzi, zakodowanym przy użyciu tagu `0010` (jak w sekcji B.1). Dlatego też numer linii może zawierać w pliku binarnym 1 lub 2 bajty.

Fragment pliku z tablicą identyfikowany jest przez napis `Line`. Struktura danych fragmentu zawarta została w tabeli A.9.

	Oktet	0								1							
Oktet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	Wersja (0x000000)															
4	32	Flagi (0x000000)															
8	64	Liczba instrukcji <code>line</code> w kodzie modułu															
12	96	Liczba linii z kodem w plikach modułu															
16	128	Liczba plików z kodem modułu															
20	160	Numer linii (1 lub 2 B)								Numer linii (1 lub 2 B)							

...	...	...	
...	...	Długość nazwy pliku 1	Nazwa pliku 1 w ASCII
...	...	...	
...	...	Długość nazwy pliku 2	Nazwa pliku 2 w ASCII
...	...	...	

Tablica A.9: Struktura tablicy linii kodu źródłowego w pliku BEAM

#### A.6.11. Drzewo syntaktyczne modułu

Plik z modułem zawiera fragment pliku źródłowego z drzewem syntaktycznym pliku z kodem źródłowym o ile został skompilowany z opcją `debug_info`. Fragment ten identyfikowany jest przez napis `Abst.`

Zawartością fragmentu jest drzewo syntaktyczne modułu, w postaci opisanej w sekcji A.4 zakodowane w formacie *External Term Format*.

## B. Lista instrukcji maszyny wirtualnej BEAM

Dodatek zawiera listę instrukcji maszyny wirtualnej BEAM, jakie może zawierać skompilowany kod pośredni przez nią wykonywany oraz sposób zapisu argumentów dla instrukcji.

Kod danej operacji zajmuje zawsze 1 bajt w pliku ze skompilowanym kodem pośrednim modułu. Argumenty mogą zajmować więcej przestrzeni, zgodnie z opisem w sekcji B.1.

Kolejność bajtów w zapisie kodu pośredniego to zawsze *big endian*.

### B.1. Typy argumentów

Argumentem jest zawsze liczba całkowita, reprezentująca taką wartość liczbową albo indeks w odpowiedniej tablicy z wartościami (pierwszym indeksem takiej tablicy jest 0). W związku z tym argumenty mogą być różnego typu. Aby rozróżnić argument jednego typu od drugiego poddaje się je odpowiedniemu tagowaniu. Operację wykonuje się bezpośrednio na argumentie, jeśli jest dostatecznie mały, lub na odpowiednim nagłówku poprzedzającym argument. Rozróżnienie to jest spowodowane oszczędnością rozmiaru kodu pośredniego, który musi być przechowywany w pamięci.

Każdy z tagów, które zostały wymienione w tabeli B.1, jest możliwy do zapisania przy użyciu 3 bitów, które zajmują najmniej znaczące bity argumentu. Jednak w kodowaniu binarnym do zapisu typu używane są dodatkowo 1 lub 2 bity. Dzięki nim możliwe jest rozróżnienie pomiędzy argumentami zapisanymi przy użyciu różnej liczby bajtów.

Tagowanie odbywa się za pomocą następującej operacji:

$$(0000XXXX \ll N)_{(2)} \oplus 000\mathbf{SSTTT}_{(2)},$$

gdzie  $XXXX_{(2)}$  jest tagowaną liczbą,  $N = 4$  lub  $5$ ,  $\mathbf{SS}_{(2)}$  są dodatkowymi bitami znakującymi rozmiar argumentu, a  $\mathbf{TTT}_{(2)}$  jest danym tagiem.

Tag		Typ
binarnie	dziesiętnie	
000	0	uniwersalny indeks, np. do tablicy stałych
001	1	liczba całkowita
010	2	indeks do tablicy atomów
011	3	numer rejestru X maszyny wirtualnej
100	4	numer rejestru Y maszyny wirtualnej

101	5	etykieta, używana w funkcjach skoku
111	7	złożone wyrażenie (np. lista, liczba zmiennoprzecinkowa)

Tablica B.1: Tagi typów danych w pliku ze skompilowanym modułem

Jeżeli tagowana liczba jest nieujemna, mniejsza od 16 (możliwe jest zapisanie jej przy użyciu 4 bitów) to argument jest zapisany przy użyciu jednego bajtu a jego postać binarna to:

$$X_1X_2X_3X_4\mathbf{0TTT}_{(2)},$$

gdzie  $X_1X_2X_3X_4$  to tagowana liczba,  $X_1$  jest jej najbardziej znaczącym bitem, a  $TTT_{(2)}$  to tag danego typu argumentu.

Na przykład, atom, który w tablicy atomów modułu ma indeks  $2_{10} = 10_2$ , po zakodowaniu będzie miał postać:

$$00100010_2 = 22_{16} = 34_{10}.$$

W przypadku, gdy liczba jest nieujemna, mniejsza lub równa 16, a mniejsza od 2048 (możliwe jest jej zapisanie przy użyciu 11 bitów), argument jest zapisany przy użyciu dwóch bajtów, których postać binarna to:

$$X_1X_2X_3\mathbf{01TTT} X_4X_5X_6X_7X_8X_9X_{10}X_{11(2)},$$

gdzie  $X_1...X_{11(2)}$  to tagowana liczba,  $X_1$  jest jej najbardziej znaczącym bitem, a  $TTT_{(2)}$  to tag danego typu argumentu.

Na przykład, liczba całkowita  $565_{10} = 010\ 00110101_2$  po zakodowaniu będzie miała postać:

$$01001001\ 00110101_2 = 4935_{16} = 18741_{10}.$$

Jeżeli argument jest liczbą ujemną lub dodatnią wymagającą w zapisie dwójkowym więcej niż 11 bitów to liczba taka zapisywana jest binarnie w kodzie uzupełnień do dwóch (U2) poprzedzona odpowiednim nagłówkiem.

Jeżeli zakodowaną liczbę można zapisać na nie więcej niż 8 bajtach, to nagłówek ma następującą postać:

$$N_1N_2N_3\mathbf{11TTT}_{(2)},$$

gdzie  $N_1N_2N_3$  to rozmiar argumentu w bajtach pomniejszony o 2 (jeżeli argument jest liczbą ujemną zajmującą 1 bajt to powinien on zostać dopełniony do 2 bajtów),  $N_1$  jest jego najbardziej znaczącym bitem, a  $TTT_{(2)}$  to tag danego typu argumentu.

Na przykład, aby zapisać na dwóch bajtach liczbę  $-21_{10} = 11111111\ 11101011_{U2}$ , jej postać binarną należy poprzedzić nagłówkiem:

$$00011001_2 = 19_{16} = 25_{10}.$$

Jeżeli do zapisania liczby w kodzie uzupełnień do dwóch potrzeba przynajmniej 9 bajtów, wtedy nagłówek jest dwubajtowy i ma postać:

$$11111\mathbf{TTT} \ N_1N_2N_3N_4\mathbf{0000}_{(2)},$$

gdzie  $N_1N_2N_3N_4_{(2)}$  to rozmiar argumentu w bajtach pomniejszony o 9,  $N_1$  jest jego najbardziej znaczącym bitem, a  $\mathbf{TTT}_{(2)}$  to tag danego typu argumentu.

Na przykład, w celu zapisania liczby  $2^{(15 \times 8) - 1} - 1$  na 15 bajtach, należy zapis tej liczby w kodzie U2 poprzedzić następującym nagłówkiem:

$$11111\mathbf{001} \ 0110\mathbf{0000}_2 = F960_{16} = 63840_{10}.$$

## B.2. Lista instrukcji

W tabeli B.2 zawarto listę instrukcji rozumianych przez maszynę wirtualną BEAM wraz z jednobajtowym kodem operacji, listą jej argumentów i krótkim opisem działania.

Instrukcje nieużywane przez kompilator Erlanga w wersji R16 zostały pominięte.

Kod operacji		Nazwa operacji i jej argumenty	Opis operacji i uwagi
hex	dec		
01	1	label Lbl	Wprowadza lokalną dla danego modułu etykietę identyfikującą aktualne miejsce w kodzie.
02	2	func_info M F A	Definiuje funkcję F, w module M o arności A.
03	3	int_code_end	Oznacza koniec kodu.
04	4	call A Lbl	Wywołuje funkcję o arności A znajdującą się pod etykietą Lbl. Zapisuje następną instrukcję jako adres powrotu (rejestr <b>CP</b> ).
05	5	call_last Arity Lbl Dest	Wywołuje prawostronnie rekursywnie funkcję o arności Arity znajdującą się pod etykietą Lbl. Nie zapisuje adresu powrotu. Przed wywołaniem zwalnia Dest słów pamięci na stosie.
06	6	call_only Arity Lbl	Wywołuje prawostronnie rekursywnie funkcję o arności Arity znajdującą się pod etykietą Lbl. Nie zapisuje adresu powrotu.
07	7	call_ext Arity Dest	Wywołuje zewnętrzną funkcję o arności Arity mającą indeks Dest w tablicy funkcji zewnętrznych. Zapisuje następną instrukcję jako adres powrotu (wskaźnik <b>CP</b> ).

08	8	call_ext_last Arity Des Dea	Wywołuje prawostronnie rekursywnie zewnętrzną funkcję o arności Arity mającą indeks Des w tablicy funkcji zewnętrznych. Nie zapisuje adresu powrotu. Przed wywołaniem zwalnia Dea słów pamięci na stosie.
09	9	bif0 Bif Reg	Wywołuje wbudowaną funkcję Bif/0. Wynik zapisywany jest w rejestrze Reg.
0A	10	bif1 Bif Arg Reg	Wywołuje wbudowaną funkcję Bif/1 z argumentem Arg. Wynik zapisywany jest w rejestrze Reg.
0B	11	bif2 Bif Arg1 Arg2 Reg	Wywołuje wbudowaną funkcję Bif/2 z argumentami Arg1, Arg2. Wynik zapisywany jest w rejestrze Reg.
0C	12	allocate SN L	Alokuje miejsce dla SN słów na stosie. Używanych jest L rejestrów <b>X</b> , gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> . Zapisuje <b>CP</b> na stosie.
0D	13	allocate_heap SN HN L	Alokuje miejsce dla SN słów na stosie. Upewnia się że na sterpie jest HN wolnych słów. Używanych jest L rejestrów <b>X</b> , gdyby w trakcie alokacji konieczne było uruchomienie <i>garbage collector</i> . Zapisuje <b>CP</b> na stosie.
0E	14	allocate_zero SN L	Tak jak allocate/2, ale zaalokowana pamięć jest wyzerowana.
0F	15	allocate_heap_zero SN HN L	Tak jak allocate_heap/3, ale zaalokowana pamięć jest wyzerowana.
10	16	test_heap HN L	Upewnia się że na sterpie jest HN wolnych słów. Używanych jest L rejestrów <b>X</b> , gdyby w trakcie konieczne było uruchomienie <i>garbage collector</i> .
11	17	init N	Zeruje N-te słowo na stosie.
12	18	deallocate N	Przywraca <b>CP</b> ze stosu i dealokuje N+1 słów ze stosu.
13	19	return	Wraca do adresu zapisanego we wskaźniku <b>CP</b> .
14	20	send	Wysyła wiadomość z rejestru <b>X1</b> do procesu w rejestrze <b>X0</b> .

15	21	<code>remove_message</code>	Usuwa aktualną wiadomość z kolejki wiadomości. Zapisuje wskaźnik do niej w rejestrze <b>X0</b> . Usuwa aktywne przeterminowanie ( <i>timeout</i> ).
16	22	<code>timeout</code>	Resetuje wskaźnik <b>SAVE</b> . Czyści flagę przeterminowanie.
17	23	<code>loop_rec Lbl Src</code>	Zapisuje kolejną wiadomość w kolejce wiadomości <b>Src</b> w rejestrze <b>R0</b> . Jeśli jest pusta wykonuje skok do etykiety <b>Lbl</b> .
18	24	<code>loop_rec_end Lbl</code>	Ustawia wskaźnik <b>SAVE</b> na kolejną wiadomość w kolejce wiadomości i wykonuje skok do etykiety <b>Lbl</b> .
19	25	<code>wait Lbl</code>	Zawiesza proces aż do otrzymania wiadomości, który zostanie wznowiony na początku bloku <code>receive</code> w etykiecie <b>Lbl</b> .
1A	26	<code>wait_timeout Lbl T</code>	Zawiesza proces jak <code>wait</code> . Ustawia przeterminowanie <b>T</b> i zapisuje następną instrukcję, która zostanie wykonana jeśli przeterminowanie się zrealizuje.
27	39	<code>is_lt Lbl Arg1 Arg2</code>	Porównuje <b>Arg1</b> z <b>Arg2</b> i wykonuje skok do <b>Lbl</b> jeśli <b>Arg1</b> jest większe lub równe od <b>Arg2</b> .
28	40	<code>is_ge Lbl Arg1 Arg2</code>	Porównuje <b>Arg1</b> z <b>Arg2</b> i wykonuje skok do <b>Lbl</b> jeśli <b>Arg1</b> jest mniejsze <b>Arg2</b> .
29	41	<code>is_eq Lbl Arg1 Arg2</code>	Porównuje <b>Arg1</b> z <b>Arg2</b> i wykonuje skok do <b>Lbl</b> jeśli <b>Arg1</b> jest arytmetycznie różne od <b>Arg2</b> .
2A	42	<code>is_ne Lbl Arg1 Arg2</code>	Porównuje <b>Arg1</b> z <b>Arg2</b> i wykonuje skok do <b>Lbl</b> jeśli <b>Arg1</b> jest arytmetycznie równe <b>Arg2</b> .
2B	43	<code>is_eq_exact Lbl Arg1 Arg2</code>	Porównuje <b>Arg1</b> z <b>Arg2</b> i wykonuje skok do <b>Lbl</b> jeśli <b>Arg1</b> jest różne <b>Arg2</b> .
2C	44	<code>is_ne_exact Lbl Arg1 Arg2</code>	Porównuje <b>Arg1</b> z <b>Arg2</b> i wykonuje skok do <b>Lbl</b> jeśli <b>Arg1</b> jest równe <b>Arg2</b> .
2D	45	<code>is_integer Lbl Arg1</code>	Sprawdza typ <b>Arg1</b> i skacze do <b>Lbl</b> jeśli nie jest on liczbą całkowitą.
2E	46	<code>is_float Lbl Arg1</code>	Sprawdza typ <b>Arg1</b> i skacze do <b>Lbl</b> jeśli nie jest on liczbą rzeczywistą.

2F	47	<code>is_number Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on liczbą.
30	48	<code>is_atom Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on atomem.
31	49	<code>is_pid Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on identyfikatorem procesu.
32	50	<code>is_reference Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on referencją.
33	51	<code>is_port Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on portem.
34	52	<code>is_nil Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on zerem ( <b>nil</b> ).
35	53	<code>is_binary Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on binarią.
37	55	<code>is_list Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on ani listą ani zerem.
38	56	<code>is_nonempty_list Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on niepustą listą.
39	57	<code>is_tuple Lbl Arg1</code>	Sprawdza typ <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest on krotką.
3A	58	<code>test_arity Lbl Arg1 Arity</code>	Sprawdza arność krotki <code>Arg1</code> i skacze do <code>Lbl</code> jeśli nie jest ona równa <code>Arity</code> .
3B	59	<code>select_val Arg Lbl Dest</code>	Skacze do etykiety <code>Dest [Arg]</code> . Jeśli nie istnieje skacze do <code>Lbl</code> .
3C	60	<code>select_tuple_arity Tuple Lbl Dest</code>	Sprawdza arność krotki <code>Tuple</code> i skacze do etykiety <code>Dest [Arity]</code> . Jeśli nie istnieje skacze do <code>Lbl</code> .
3D	61	<code>jump Lbl</code>	Skacze do etykiety <code>Lbl</code> .
3E	62	<code>catch Dest Lbl</code>	Tworzy nowy blok <code>catch</code> . Zapisuje etykietę <code>Lbl</code> na <code>Dest</code> miejscu na stosie.
3F	63	<code>catch_end Dest</code>	Kończy blok <code>catch</code> . Wymazuje etykietę na miejscu <code>Dest</code> na stosie.
40	64	<code>move Src Dest</code>	Przenosi wartość z <code>Src</code> do rejestru <code>Dest</code> .
41	65	<code>get_list Src Hd Tail</code>	Umieść głowę listy <code>Src</code> w rejestrze <code>Hd</code> i jej ogon w rejestrze <code>Tail</code> .
42	66	<code>get_tuple_element Src Elem Dest</code>	Umieść element <code>Elem</code> krotki <code>Src</code> w rejestrze <code>Dest</code> .



43	67	get_tuple_element Elem Tuple Pos	Umieść element Elem w krotce Tuple na pozycji Pos.
45	69	put_list Hd Tail Dest	Tworzy listę [Hd Tail] na szczycie sterty i umieszcza ją w rejestrze Dest.
46	70	put_tuple Dest Arity	Tworzy krotkę o arności Arity na szczycie sterty i umieszcza ją w rejestrze Dest.
47	71	put Arg	Umieszcza Arg w lokalnym kontekście.
48	72	badmatch Arg	Rzuca wyjątek badmatch z argumentem Arg.
49	73	if_end	Rzuca wyjątek if_clause.
4A	74	case_end Arg	Rzuca wyjątek case_clause z argumentem Arg.
4B	75	call_fun Arity	Woła obiekt funkcyjny o arności Arity. Zakłada, że argumenty znajdują się w rejestrach <b>X0...X(Arity-1)</b> , a lambda w rejestrze <b>X(Arity)</b> . Zapisuje następną instrukcję we wskaźniku <b>CP</b> .
4D	77	is_function Lbl Arg1	Sprawdza typu argumentu Arg1 i skacze do Lbl jeśli nie jest on funkcją.
4E	78	call_ext_only Arity Lbl	Wywołuje prawostronnie rekursywnie zewnętrzną funkcję o arności Arity mającą indeks Lbl w tablicy funkcji zewnętrznych. Nie zapisuje adresu powrotu.
59	89	bs_put_integer/5	
5A	90	bs_put_binary/5	
5B	91	bs_put_float/5	
5C	92	bs_put_string/2	
5E	94	fclearerror/0	
5F	95	fcheckerror/1	
60	96	fmove/2	
61	97	fconv/2	
62	98	fadd/4	
63	99	fsub/4	
64	100	fmul/4	
65	101	fdiv/4	
66	102	fnegate/3	
67	103	make_fun2/1	
68	104	try/2	

69	105	try_end/1	
6A	106	try_case/1	
6B	107	try_case_end/1	
6C	108	raise/2	
6D	109	bs_init2/6	
6F	111	bs_add/5	
70	112	apply/1	
71	113	apply_last/2	
72	114	is_boolean/2	
73	115	is_function2/3	
74	116	bs_start_match2/5	
75	117	bs_get_integer2/7	
76	118	bs_get_float2/7	
77	119	bs_get_binary2/7	
78	120	bs_skip_bits2/5	
79	121	bs_test_tail2/3	
7A	122	bs_save2/2	
7B	123	bs_restore2/2	
7C	124	gc_bif1/5	
7D	125	gc_bif2/6	
81	129	is_bitstr/2	
82	130	bs_context_to_binary/1	
83	131	bs_test_unit/3	
84	132	bs_match_string/4	
85	133	bs_init_writable/0	
86	134	bs_append/8	
87	135	bs_private_append/6	
88	136	trim/2	
89	137	bs_init_bits/6	
8A	138	bs_get_utf8/5	
8B	139	bs_skip_utf8/4	
8C	140	bs_get_utf16/5	
8D	141	bs_skip_utf16/4	
8E	142	bs_get_utf32/5	
8F	143	bs_skip_utf32/4	
90	144	bs_utf8_size/3	
91	145	bs_put_utf8/3	
92	146	bs_utf16_size/3	

93	147	bs_put_utf16/3	
94	148	bs_put_utf32/3	
95	149	on_load/0	
96	150	recv_mark/1	
97	151	recv_set/1	
98	152	gc_bif3 Lbl Live Bif Arg1 Arg2 Arg3 Reg	Wywołuje funkcję wbudowaną Bif/3 z argumentami Arg1, Arg2, Arg3. Wynik zapisuje w rejestrze Reg. W przypadku niepowodzenia skacze do etykiety Lbl. Uruchamia <i>garbage collector</i> jeśli jest to konieczne, zachowując Live rejestrów X.
99	153	line N	Znakuje aktualne miejsce jako linia o indeksie N w tablicy linii.

Tablica B.2: Lista operacji maszyny wirtualnej BEAM

## Bibliografia

- [1] Ericsson AB. Erlang External Term Format. [http://erlang.org/doc/apps/erts/erl\\_ext\\_dist.html](http://erlang.org/doc/apps/erts/erl_ext_dist.html), 2014. [data dostępu: 21.03.2014].
- [2] J. Morrison. EA IFF 85: Standard for interchange format files. *Amiga ROM Kernel Reference Manual: Devices (3rd edition)*, Addison-Wesley, 1(99):1, 1985.