Blaze

Програмски језик

Владимир Дабић

Садржај

1. Увод 2

1.1. Пример програма 2

1.2. Структура језика 3

2. Формат компајлованог програма 4

2.1. Формат константне вредности 5

2.2. Формат променљиве 5

2.3. Формат функције 6

2.4. Формат класе 6

3. Представљање инструкција 7

3.1. Табела инструкција 8

4. Процес компилације 10

4.1. Лексичка анализа 10

4.2. Синтаксна анализа 11

4.3. Стварање модул фајла 12

4.3.1. Стварање извршног кода 14

Blaze

# Увод

**Блејз** (Blaze) је интерпретован динамички програмски језик.  
Компајлује се у бајт-код који онда интерпретер извршава.

*blzc.exe* – Компајлер (енгл. Compiler)  
*blzi.exe* – Интерпретер (енгл. Interpreter)

Фајл направљен након компилације је типа *Blaze Module* (.blzm)  
У овом фајлу се чувају подаци о програму; Функције (и њихове инструкције), променљиве, класе и константне вредности.

Блејз спада у “stack-oriented” језике, то значи да користи стек (енгл. stack) помоћу ког извршава операције над објектима.

## Пример програма

Сваки програм почиње од *main* фунцкије.  
 *У овом случају да али не мора да значи, зависи од програма који позива блејз функције.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | primer.blz | | 1  2  3  4  5 | extern var console;  func main() {  console.print("Hello World");  } | |

Кад покренемо овај програм у конзоли ће се исписати „Hello World“.

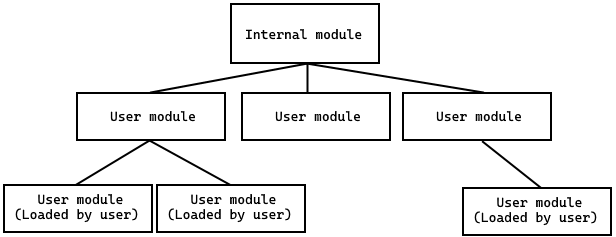
|  |
| --- |
| > .\blzc.exe -s primer.blz  Compiled to 'primer.blzm'  > .\blzi.exe -m primer.blzm  Hello World |

## Структура језика

Блејз је дизајниран првенствено за уградњу у апликације.  
Само по себи не долази са библиотекама, то апликација у коју је уграђен обезбеђује.

Интерпретер је програм у који је уграђен блејз и он обезбеђује библиотеке.  
У примеру Hello World програма видимо да је *console* екстерна глобална променљива.

Блејз функционише на систему модула, где сваки модул може да има подмодуле, чиме се прави хијерархија модула.



У интерпретеру модул који покрећемо спада у User мodule на другом нивоу (Internal module је први ниво).

Можемо приступити свим променљивима декларисане као public у internal module.  
Такође можемо приступити јавним променљивима које спадају у подмодуле нашег модула.

Променљиве декларисане са *public* су видљиве у свим модулима.  
Променљиве декларисане са *private* су видљиве само у том модулу у ком су дефинисане.

Да би приступили јавним (public) променљивима које су ван нашег модула морамо декларисати променљиву као *extern*.

Планирано је да се имплементира *internal* тип променљиве. То значи да је променљива видљива само подмодулима модула у ком је дефинисана.

# Формат компајлованог програма

Компајлован програм је бинарни фајл типа *Blaze Module* (.blzm).  
Следећа табела дефинише формат.  
 *Напомена: Little-endian encoding*

|  |  |  |
| --- | --- | --- |
| Тип | Назив | Опис |
| u32 | identifier | Број који представља овај тип фајла 0x626C7A6D ("blzm") |
| u16 | version | Верзија формата са којом је фајл сачуван 0xAABB (A - major, B - minor) |
| u16 | filename\_length | Дужина имена модул фајла |
| char[filename\_length] | filename | Име модул фајла |
| bool (u8) | debug | Дали инструкције имају број линије префикс |
| u16 | constant\_count | Број константних вредности |
| Constant[constant\_count] | contants | Низ константних вредности дужине *constant\_count* |
| u16 | variable\_count | Број дефинисаних променљивих |
| Variable[variable\_count] | variables | Низ променљивих дужине *variable\_count* |
| u16 | function\_count | Број дефинисаних функција |
| Function[function\_count] | functions | Низ функција дужине  *function\_count* |
| u16 | class\_count | Број дефинисаних класа |
| Class[class\_count] | classes | Низ класа дужине *class\_count* |

## Формат константне вредности

|  |  |  |
| --- | --- | --- |
| Тип | Назив | Опис |
| u8 | constant\_type | Тип константне вредности |
| u8[…] | data | Подаци (зависе од типа) |

Типови константне вредности:

|  |  |
| --- | --- |
| Тип (constant\_type) | Назив |
| 0 | NUMBER |
| 1 | STRING |

Подаци константе зависе од типа, следећа табела дефинише податке у зависности од типа.

|  |  |
| --- | --- |
| Тип | Назив |
| STRING | |
| u16 | string\_length |
| char[string\_length] | characters |
| NUMBER | |
| f64 (double) | value |

## Формат променљиве

|  |  |  |
| --- | --- | --- |
| Тип | Назив | Опис |
| u8 | variable\_type | Тип променљиве |
| u16 | name | Индекс константе која представља име променљиве  *name - 1* |

Типови променљивих (тј. видљивост)

|  |  |
| --- | --- |
| Тип (variable\_type) | Назив |
| 0 | PRIVATE |
| 1 | PUBLIC |
| 2 | EXTERNAL |

## Формат функције

|  |  |  |
| --- | --- | --- |
| Тип | Назив | Опис |
| u16 | name | Индекс константе која представља име функције  *name - 1*  *0 значи да нема име* |
| u8 | num\_of\_args | Број параметара фунцкије |
| u8 (bool) | varargs | Дали функција прима променљив број параметара |
| u8 | num\_of\_locals | Број локалних променљивих |
| u16 | num\_of\_instructions | Број инструкција |
| u16[num\_of\_instructions] | instructions | Низ инструкција дужине *num\_of\_instructions*  (Свака инструкција је Big-endian encoding) |

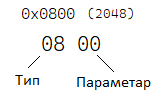
Ако је Debug укључен, онда ће инстукција бити u32 а не u16.  
Више о овоме у поглављу *Представљање инстукција*

## Формат класе

|  |  |  |
| --- | --- | --- |
| Тип | Назив | Опис |
| u16 | name | Индекс константе која представља име класе  *name - 1*  *0 значи да нема име* |
| u8 | num\_of\_members | Број променљивих класе |
| u16[num\_of\_members] | member\_list | Низ индекса константи које представљају променљиве |
| u16 | constructor | Индекс конструктора у низу функција |

# Представљање инструкција

Свака инструкција се састоји из два бајта и чува се као цео број од 16 бита.  
Леви бајт представља тип инструкције а десни параметар.



Пример сабирања два броја



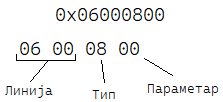
Иако је параметар присутан у свакој инструкцији то не значи да ће га инструкција искористити.

Неке инструкције захтевају већи број од 255 или више од једног параметра, да би то могло да се изведе уведена је инструкција *EXTENDED\_ARG.*Пример:

|  |  |
| --- | --- |
| Инструкција | Параметар (hex) |
| EXTENDED\_ARG | 01 |
| LDLOCAL | 02 |

Параметар за инструкцију *LDLOCAL* ће бити 0x0102 (258)  
Овај број може да значи више ствари у зависности од иинструкције.  
  
Свака инструкција може да има највише три *EXTENDED\_ARG* инструкције пре себе.  
То значи да параметар може да буде највише 4 бајта тј. цео број од 32 бита.

Ако је Debug укључен, онда се инстукција састоји из четири бајта.



Додају се два бајта испред која чувају линију изворног кода  
 *Линија се чува са little endian encoding*

## Табела инструкција

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип | Назив | Број  параметара | Stack | Опис |
| 00 | NOP | 0 | → | Нема операције |
| 01 | POP | 1 | value → | Склони #arg1 објеката са врха стека |
| 02 | EXTENDED\_ARG | 1 | → | Продужи параметар са #arg1 |
| 03 | LDNULL | 0 | → null | Стави NULL на стек |
| 04 | LDARG | 1 | → value | Стави параметар #arg1 функ. на стек |
| 05 | LDCONST | 1 | → value | Стави константу #arg1 на стек  *arg је индекс константе* |
| 06 | LDLOCAL | 2 | → value | Стави локалну променљиву #arg1 на стек  Други параметар (arg2) је ниво на коме се налази променљива (0 - default) |
| 07 | LDVAR | 1 | → value | Стави глобалну променљиву #arg1 на стек  *arg је индекс STRING константе* |
| 08 | LDFUNC | 1 | → funcref | Стави референцу функције #arg1 на стек  *arg је индекс функције* |
| 09 | LDCLASS | 1 | → classref | Стави референцу класе #arg1 на стек  *arg је индекс класе* |
| 0A | LDBOOL | 1 | → value | Стави *bool* на стек |
| 0B | STLOCAL | 2 | value → | Сачувај објекат са врха стека у локалну променљиву #arg1  Други параметар (arg2) је ниво на коме се налази променљива (0 - default) |
| 0C | STVAR | 1 | value → | Сачувај објекат са врха стека у глобалну променљиву #arg1  *arg је индекс STRING константе* |
| 0D | STARG | 1 | value → | Стави објекат са врха стека у параметар #arg1 функције |
| 0E | CALL | 1 | argn, …, arg1, callable  → value | Позови објекат са врха стека са #arg1 бројем параметара |
| 0F | RET | 0 | value → | Врати објекат са врха стека |
| 10 | ADD | 0 | left, right → result | Сабери два објекта са врха стека |
| 11 | SUB | 0 | left, right → result | Одузми два објекта |
| 12 | MUL | 0 | left, right → result | Помножи два објекта |
| 13 | DIV | 0 | left, right → result | Подели два објекта |
| 14 | INTDIV | 0 | left, right → result | Подели два објекта као целе бројеве |
| 15 | THROW | 0 | → | Изазови грешку и проследи објекат са врх стека |
| 16 | CATCH | 1 | value → | Сачувај релативну позицију (#arg1) *catch* блока у exception stack |
| 17 | TRY\_END | 0 | → | Означава излазак из *try* блока |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Тип | Назив | Број  параметара | Stack | Опис |
| 18 | EQ | 0 | left, right → result | Упореди два објекта и стави резултат на стек |
| 19 | LT | 0 | left, right → result | Ако је леви објекат мањи од десног |
| 1A | LTE | 0 | left, right → result | Ако је леви објекат мањи или једнак |
| 1B | NOT | 0 | value → negated | Негација објекта |
| 1C | JMP | 1 | → | Релативан скок унапред |
| 1D | JMPB | 1 | → | Релативан скок уназад |
| 1E | JMPA | 1 | → | Апсолутак скок |
| 1F | JMPT | 1 | value → | Релативан скок унапред уколико је објекат на стеку *true* |
| 20 | JMPF | 1 | value → | Релативан скок унапред уколико је објекат на стеку *false* |
| 21 | OR | 0 | left, right → result | Изврши логичку ИЛИ операцију на два објекта |
| 22 | AND | 0 | left, right → result | Изврши логичку И операцију на два објекта |
| 23 | DUP | 1 | value → value\*arg1 | Направи #arg1 копија објекта на врху стека |
| 24 | VARARGS | 1 | valuen … value1 → |  |
| 25 | LDLIST | 1 | valuen … value1 → array | Стави нову листу на стек |
| 26 | LDOBJ | 1 | valuen, keyn …  value1,  key1  → object | Стави нови dict на стек |
| 27 | LDINDEX | 0 | idx, object → value | Учитај вредност на индексу idx из објекта |
| 28 | STINDEX | 0 | value, idx, object → | Сачувај вредност на индексу idx у објекту |
| 29 | LDPROP | 1 | object → value |  |
| 2A | STPROP | 1 | value, object → |  |
| 2B | LDEVENT | 0 | → event | Стави нови event на стек |
| 2C | ITER | 0 | value → iterator | Стави нови итератор на стек |
| 2D | NEW | 1 | argn, …, arg1, class  → value | Направи нову инстанцу класе |
| 2E | ATTACH | 0 | value, event, → | Закачи callback функцију за event. |

Када пише први параметар мисли се на скроз десни бајт, сваки следећи параметар се чита ка лево

# Процес компилације

Процес компилације се састоји из три корака:

1. Лексичка анализа
2. Синтаксна анализа
3. Стварање модул фајла
   1. Стварање извршног кода

## Лексичка анализа

**Лексичка анализа** је процес рашчлањивања знакова приликом учитавања изворног кода рачунарског програма, са циљем да се произведе низ симбола који се зову *лексичке ознаке* (енгл. lexical tokens).

Лексичка анализа се обично врши учитавањем знакова који се деле на посебне категорије, које зависе од програмског језика (нпр. препознавање варијабли, бројева, посебних речи - наредбе).

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3 | func main() {  print("Hello World");  } | |

Низ симбола који ће се произвести лексичком анализом овог програма је:

|  |  |
| --- | --- |
| Тип | Вредност |
| FUNC |  |
| IDENTIFIER | “main” |
| OPEN\_PAREN |  |
| CLOSE\_PAREN |  |
| OPEN\_BRACE |  |
| IDENTIFIER | “print” |
| OPEN\_PAREN |  |
| STRING | “Hello World” |
| CLOSE\_PAREN |  |
| SEMICOLON |  |
| CLOSE\_BRACE |  |

Све се ово извршава у класи *Lexer* (**Lexer.cs**)

Након овог рашчлањивања, рашчлањен изворни код се провлачи кроз део програма који прегледа синтаксу програма (синтаксна анализа).

## Синтаксна анализа

**Синтаксна анализа** је процес анализирања низа *лексичких ознака* (енгл. lexical tokens) ради утврђивања структуре програма у односу на дату граматику.

Низ лексичких ознака се претвара у облик који је погодан за даљу обраду, у овом случају *апстрактно синтаксно стабло* (енгл. Abstract Syntax Tree - AST).

|  |  |
| --- | --- |
| Тип | Вредност |
| FUNC |  |
| IDENTIFIER | “main” |
| OPEN\_PAREN |  |
| CLOSE\_PAREN |  |
| OPEN\_BRACE |  |
| IDENTIFIER | “print” |
| OPEN\_PAREN |  |
| STRING | “Hello World” |
| CLOSE\_PAREN |  |
| SEMICOLON |  |
| CLOSE\_BRACE |  |

За овај низ ознака стабло ће изгледати овако (упрошћено)

FUNCTION  
 name: “main”  
 args: []  
 body: …

VARIABLE  
 name: “print”

STRING  
 value: “Hello World”

CALL  
 callee: …  
 args: …

Све ово се извршава у класи *Parser* (**Parser.cs**).

## Стварање модул фајла

Након синтаксне анализе добија се апстрактно синтаксно стабло од ког се производи *Blaze Module* фајл.  
Процес стварања модул фајла се извршава „шетањем“ стабла ка доле и исписивањем одговарајућих бајтова према формату модул фајла.

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5 | extern var print;  func main() {  print("Hello World");  } | |

За овај програм модул фајл ће изгледати овако:  
 *Напомена: Little-endian encoding*

Број константи (3)

Верзија (1.0)

“blzm”



По формату након броја констанати следи низ константних вредности.  
У овом случају су то све вредности типа STRING (01).

|  |  |  |  |
| --- | --- | --- | --- |
| Тип | Дужина | Карактери |  |
| 01 | 05 00 | 70 72 69 6Е 74 | “print” |
| 01 | 04 00 | 6D 61 69 6E | “main” |
| 01 | 0B 00 | 48 65 6C 6C 6F 20 57 6F 72 6C 64 | “Hello World” |

Затим следи низ променљивих.



Број променљивих (2)

|  |  |
| --- | --- |
| Тип | Име |
| 02 | 01 00 |
| 00 | 02 00 |

Име је индекс у низ константи  
Видимо да су типови 02 (EXTERNAL) и 00 (PRIVATE).  
 *main се води да је приватна функција иако нисмо нагласили у коду*

Затим следе дефиницијие функција.



Број функ. (2)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Име | Број  Парам. | vargs  (bool) | Број  локалних | Број  Инстр. | Инструкције |
| 00 00 | 00 | 00 | 00 | 04 00 | 08 01 0C 01 03 00 0F 00 |
| 02 00 | 00 | 00 | 00 | 06 00 | 05 02 07 00 0E 01 01 01 03 00 0F 00 |

Након броја функција следи низ функција.

Име је индекс у низ константи.  
 *Ако је 0 то значи да функција нема име.*

Примећујемо да има две функције иако смо дефинисали само једну.  
Прва функција (idx. 0) нема име (0x0000) и служи за иницијализацију променљивих у модулу.

У блејзу ову функцију називамо *статична функција* (енгл. static function) и она је позвана само једном током очитавања модула у интерпретер.

|  |  |  |  |
| --- | --- | --- | --- |
| Тип | Назив | Параметар | Опис |
| 08 | LDFUNC | 01 | Учитај функцију 1 и сачувај је у променљиву *main* (1) |
| 0C | STVAR | 01 |
| 03 | LDNULL | 00 | Врати NULL |
| 0F | RET | 00 |

Инструкције  
статичне функ.  
  
Ову функцију неби требало дирати али ако нешто баш треба да се догоди током очитавања модула могуће је урадити следеће:

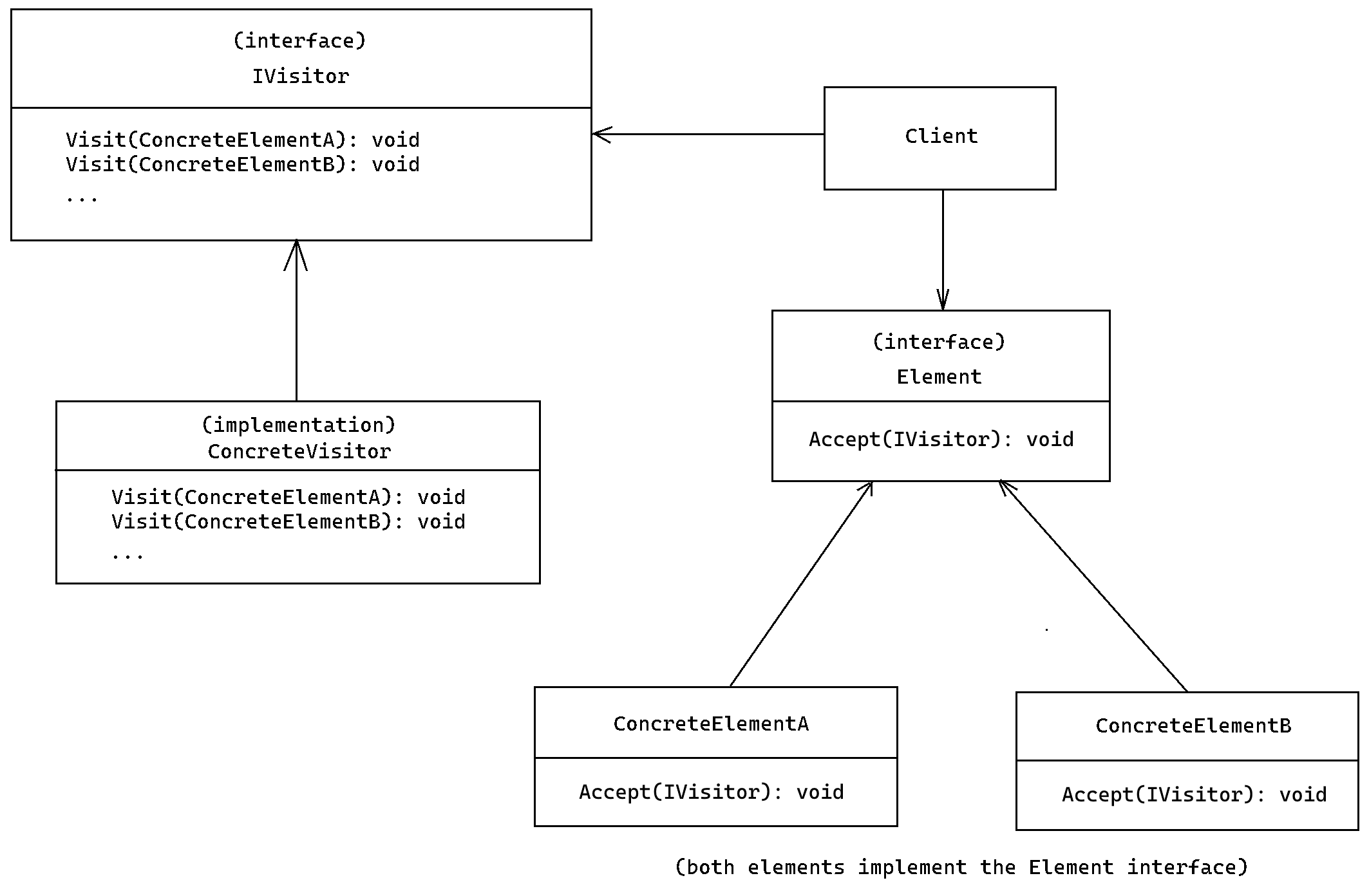
|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11 | extern var print;  var myvar;  // static blok je pozvan tokom ocitavanja  static {  myvar = 20;  }  func main() {  print(myvar); // 20  } | |

Понашање статичне функције није скроз дефинисано јер зависи од програма који очитава  
блејз модуле.  
*нпр. не можемо приступити print функцији у статичном блоку док у неком програму који очитава исти тај модул можемо*.

### Стварање извршног кода

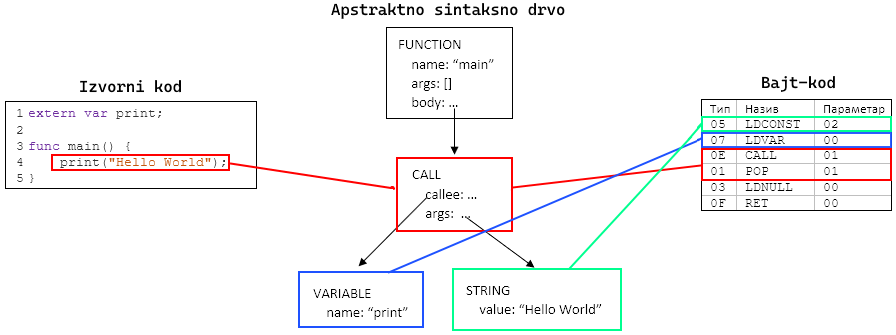
Стварање извршног кода се извршава у класи *Generator* (**Generator.cs**).  
Класа *Generator* користи помоћну класу *Module* која скрива логику писања и читања бајтова према формату модул фајла (сериализација) чиме нам олакшава рад са њим.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | Пример употребе Module класе (C#) | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Module module = new Module();  *// Definisi funkciju 'add' koja ima dva parametra*  Function addf = module.CreateFunction("add", 2);  *// Izvrsni kod*  addf.Emit(Opcode.LDARG, 0); *// ucitaj prvi parametar na stack*  addf.Emit(Opcode.LDARG, 1); *// ucitaj drugi*  addf.Emit(Opcode.ADD); *// dodaj ih*  addf.Emit(Opcode.RET); *// vrati rezultat*  *// Sacuvaj modul*  using(var stream = File.Open("compiled.blzm", FileMode.Create, FileAccess.Write))  {  BinaryWriter writer = new BinaryWriter(stream);  module.ToBinary(writer);  } | |

Класа Generator је имплементација *Visitor* шаблона понашања.  


У Visitor шаблону понашања логика се одваја од структуре објекта.  
Користи се када треба да извршимо операцију над групом сличних објеката.

Генератор пролази кроз сваку грану апстрактног синтаксног стабла помоћу овог шаблона.



Спуштањем низ стабло генератор долази до CALL гране која има две подгране.  
Да би генерисали CALL инструкцију морамо прво проћи кроз подгране.

По табели инструкција, CALL инструкција очекује да функција буде на врху стека а после ње параметри. Да би ово испунили морамо прво генерисати параметре уназад, па онда функцију коју позивамо.

У овом случају то значи да прво генеришемо константу “Hello World” па онда променљиву print (функција се чува у поменљивој).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | Имплементација у компајлеру (*Generator.cs*) | | 1  2  3  4  5  6  7  8  9  10  11 | public void Visit(Expression.Call call)  {  for(int i = call.Arguments.Count - 1; i >= 0; i--)  {  Evaluate(call.Arguments[i]);  }  Evaluate(call.Callee);  \_function.Emit(Opcode.CALL, (byte)call.Arguments.Count);  } | |

*Evaluate функција позива Accept функцију објекта и за IVisitor прослеђује генератор (this).*