Blaze

Програмски језик

Владимир Дабић

Садржај

[1. Увод 2](#_Toc150079619)

[1.1. Пример програма 2](#_Toc150079620)

[2. Формат компајлованог програма 3](#_Toc150079621)

[2.1. Формат константне вредности 3](#_Toc150079622)

[2.2. Формат променљиве 4](#_Toc150079623)

[2.3. Формат функције 4](#_Toc150079624)

[2.4. Формат класе 4](#_Toc150079625)

[3. Представљање инструкција 5](#_Toc150079626)

[3.1. Табела инструкција 6](#_Toc150079627)

[4. Процес компилације 7](#_Toc150079628)

[4.1. Лексичка анализа 7](#_Toc150079629)

[4.2. Синтаксна анализа 8](#_Toc150079630)

[4.3. Стварање модул фајла 9](#_Toc150079631)

Blaze

# Увод

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

*blazec.exe* – Компајлер (енгл. Compiler)  
*blazei.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 прослеђује параметре инструкцијама.

## 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 | public func main() {  print("Hello World");  } | |

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

|  |
| --- |
| > blazec primer.blz  > blazei primer.blzm  Hello World |

# 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 | 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

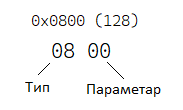
|  |  |  |
| --- | --- | --- |
| Тип | Назив | Опис |
| u16 | name | Индекс константе која представља име функције  *name - 1*  *0 значи да нема име* |
| u8 | num\_of\_args | Број параметара фунцкије |
| u8 (bool) | varargs | Дали функција прима променљив број параметара |
| u8 | num\_of\_locals | Број 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) |

## Формат класе

Тренутно није завршено.

# 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 (18)  
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.

## 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 | JMPA | 1 | → | Апсолутак скок |
| 1E | JMPT | 1 | value → | Релативан скок унапред уколико је објекат на стеку *true* |
| 1F | JMPF | 1 | value → | Релативан скок унапред уколико је објекат на стеку *false* |

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).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  | | --- | --- | |  | primer.blz | | 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 граматичке структуре у односу на дату граматику.

Низ лексичких ознака се претвара у облик који је погодан за даљу обраду, у овом случају *апстрактно синтаксно дрво* (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 дрво 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