

Jezički invarijantna provera semantičke  
ekvivalentnosti strukturno sličnih segmenata  
imperativnog koda

Ivan Ristović

# Teme

- ▶ Motivacija i uvod
- ▶ AST
- ▶ Dobijanje AST - ANTLR
- ▶ Opšti AST
- ▶ Poređenje opštih AST
- ▶ LICC — Language Invariant Code Comparer

# Motivacija i uvod

```
1 void array_sum(int[] arr, int n) {  
2     int sum = 0, i = 0;  
3     while (i < n) {  
4         int v = arr[i]  
5         sum += v;  
6         i++;  
7     }  
8     return sum;  
9 }
```

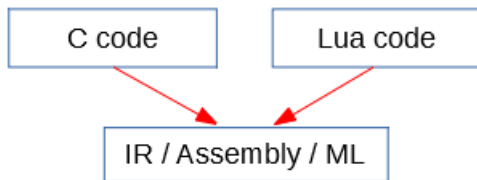
```
1 function array_sum(arr, n)  
2     local sum = 0  
3     for i,v in ipairs(arr) do  
4         sum = sum + v  
5     end  
6     return sum  
7 end
```

# Uvod i motivacija

- ▶ Pristup?
  - ▶ "Niski" pristup
  - ▶ "Visoki" pristup
- ▶ Razlika je u reprezentaciji na koju se dovode segmenti koda pre procesa poređenja

# Uvod i motivacija

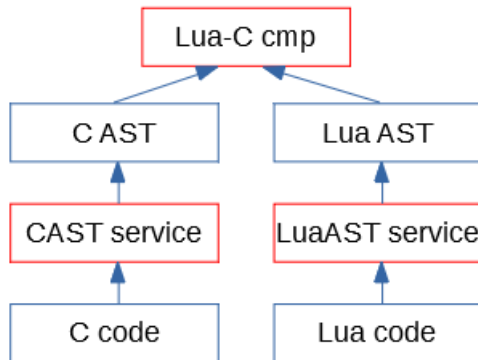
- ▶ Niski pristup



- ▶ Prednosti: jedinstvena reprezentacija (?)
- ▶ Mane: vezanost sa specifičnom arhitekturom procesora, potrebno prevoditi kod, JVM/CLR, različiti programski jezici?

# Uvod i motivacija

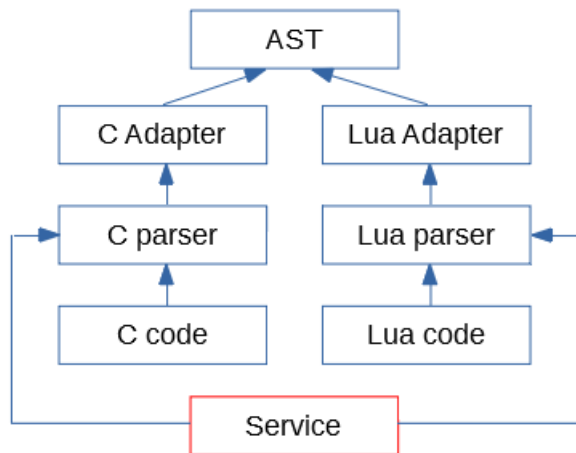
- Visoki pristup (varijanta 1)



- Prednosti: jedinstvena reprezentacija (?), nije potrebno prevoditi kod, kompatibilno sa bilo kojim programskim jezikom, moguće koristiti algoritme za poređenje stabala
- Mane: zavisnost od eksternih servisa, skaliranje

# Uvod i motivacija

- Visoki pristup (varijanta 2)



- Prednosti: jedinstvena reprezentacija (!), nema prevođenja, proizvoljan programski jezik, moguće koristiti algoritme za poređenje stabala, skalabilno, samo jedan servis

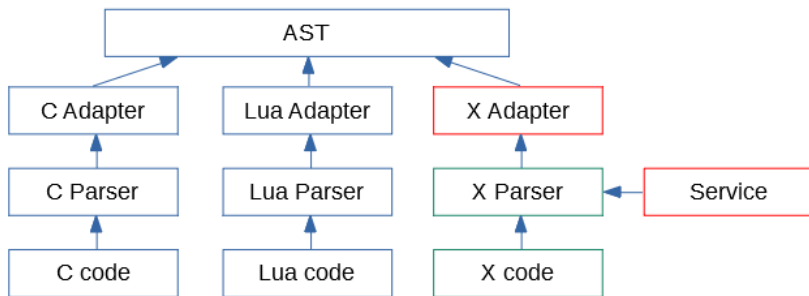
# Uvod i motivacija

- ▶ Pošto nema prevođenja, može se analizirati kod samo na osnovu gramatike njegovog jezika
- ▶ AST se dobija od stabla parsiranja izvornog koda korišćenjem adaptera
- ▶ Adapteri se moraju razlikovati zbog razlika u AST



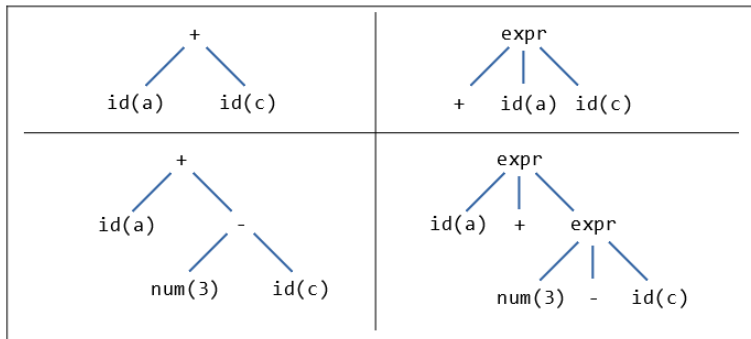
# Uvod i motivacija

- ▶ Kako proširiti?



# AST

## ► AST - Abstract Syntax Tree



# AST

## ► Go AST

```
package main

import "fmt"

func fib() func() int {
    a, b := 0, 1
    return func() int {
        a, b = b, a+b
        return a
    }
}
```

```
- File {
    Comments: [ ]
    - Decls: [
        + GenDecl {Loc, Lparen, Rparen, Specs, Tok}
        - FuncDecl {
            - Body: BlockStmt {
                Lbrace: 55
            - List: [
                + AssignStmt {Lhs, Loc, Rhs, Tok}
                - ReturnStmt {
                    + Loc: {End, Start}
                    - Results: [
```

# AST

## ► Lua AST

```
function Fibonacci.naive(n)
  local function inner(m)
    if m < 2 then
      return m
    end
    return inner(m-1) + inner(m-2)
  end
  return inner(n)
end
```

```
- Chunk {
  type: "Chunk"
- body: [
  - FunctionDeclaration {
    type: "FunctionDeclaration"
+ identifier: MemberExpression {type, .
    isLocal: false
+ parameters: [1 element]
- body: [
  - FunctionDeclaration {
    type: "FunctionDeclaration"
+ identifier: Identifier {type,
```

# Dobijanje AST - ANTLR

- ▶ Neophodan je parser!
- ▶ AST nastaje apstrahovanjem stabla parsiranja
- ▶ Dosta alata: Yacc, BYACC, GNU Bison, ANTLR
- ▶ Svi ovi alati mogu generisati parsere za proizvoljne gramatike

# Dobijanje AST - ANTLR

- ▶ *ANother Tool for Language Recognition*
- ▶ ANTLR v4 izabran zbog:
  - ▶ Mogućnosti generisanja parsera u raznim jezicima (uključujući C#)
  - ▶ Trivijalno definisati gramatike (dosta poznatih jezika već podržano)
  - ▶ Mogu se generisati i klase koje pružaju interfejs za obilazak stabla parsiranja

# Dobijanje AST - ANTLR

- ▶ Prvi korak: definicija gramatike

```
1  grammar Lua;
2  chunk : block EOF ;
3  block : stat* retstat? ;
4  stat
5      : ';'
6      | varlist '=' explist
7      | functioncall
8      | label
9      | 'break'
10     | 'do' block 'end'
11     | 'while' exp 'do' block 'end'
12     | 'if' exp 'then' block ('elseif' exp 'then'
13                               block)* ('else' block)? 'end'
14     | 'for' NAME '=' exp ',' exp (',' exp)? 'do'
15                               block 'end'
16     | 'function' funcname funcbody
17     ...
```

# Dobijanje AST - ANTLR

- ▶ Prvi korak: definicija gramatike

```
1  NAME
2      : [a-zA-Z_][a-zA-Z_0-9]*
3      ;
4
5  NORMALSTRING
6      : '"' ( EscapeSequence | ~('\\"'|'\"') ) * '"'
7      ;
8
9  WS
10     : [ \t\u000C\r\n]+ -> skip
11     ;
```



# Dobijanje AST - ANTLR

- ▶ Drugi korak: generisanje parsera

```
1 $ antlr4 Lua.g4 -Dlanguage=CSharp --visitor
```

- ▶ Generisane LuaLexer i LuaParser klase
- ▶ Generisani interfejsi LuaListener i LuaVisitor

# Dobijanje AST - ANTLR

- ▶ Treći korak: obići stablo parsiranja i kreirati AST

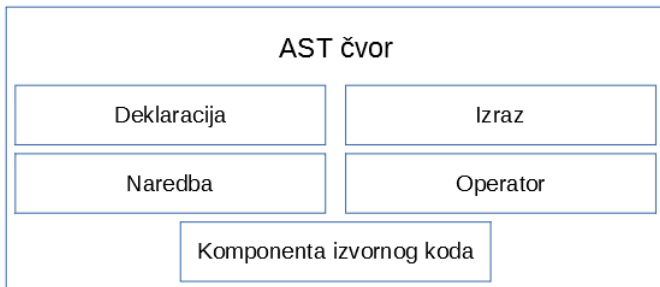
```
1  public interface ILuaVisitor<T> :  
    IParseTreeVisitor<T>  
2  {  
3      T VisitChunk([NotNull]  
        LuaParser.ChunkContext context);  
4      T VisitBlock([NotNull]  
        LuaParser.BlockContext context);  
5      T VisitStat([NotNull] LuaParser.StatContext  
        context);  
6  
7      ...  
8  }
```

# Opšti AST

- ▶ Želimo opšti AST, koji će podržavati koncepte raznih imperativnih jezika
- ▶ Koncepti: literali, izrazi, naredbe, ...
- ▶ Kreirati dovoljno (ali ne previše) apstraktne tipove čvora za ove koncepte
- ▶ Specifičnosti svesti na "već viđeno"
- ▶ Ako svođenje nema smisla, uvesti novi tip AST čvora
- ▶ Izgubiti što manje informacija!!!

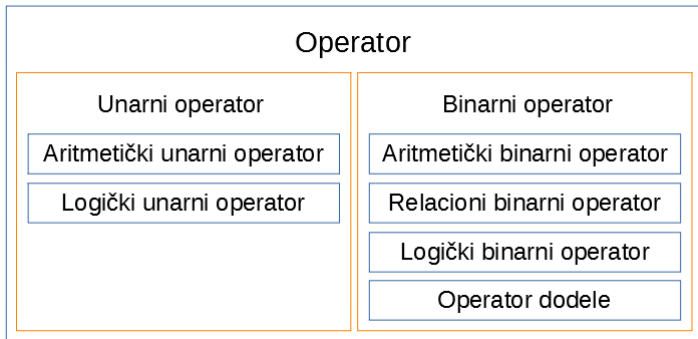
# Opšti AST

## ► Bazna hijerarhija



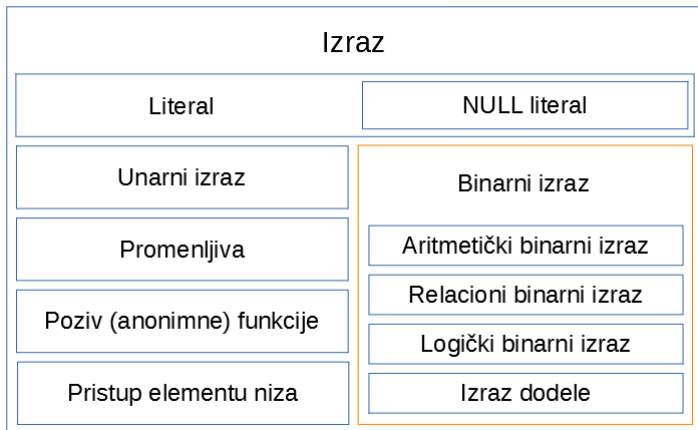
# Opšti AST

## ► Operatori



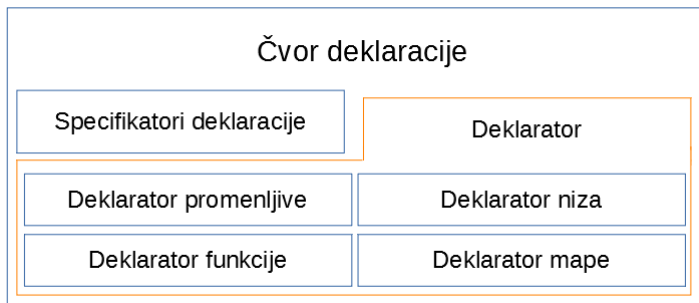
# Opšti AST

## ► Izrazi



# Opšti AST

## ► Deklaracije



# Opšti AST

## ► Naredbe





# Opšti AST

## ► Primer - *swap*

```
1  int tmp = x;  
2  x = y;  
3  y = tmp;
```

```
1  x, y = y, x
```

- Paziti na nove konstrukte
- U slučaju skript jezika, deklarirati promenljive pre korišćenja

# Gde smo sada?

- x Motivacija i uvod
- x AST
- x Dobijanje AST - ANTLR
- x Opšti AST

Poređenje opštih AST

LICC — Language Invariant Code Comparer

# Pitanja

???