nedeljainformatike v6.0

Kratak kurs konstrukcije kompajlera

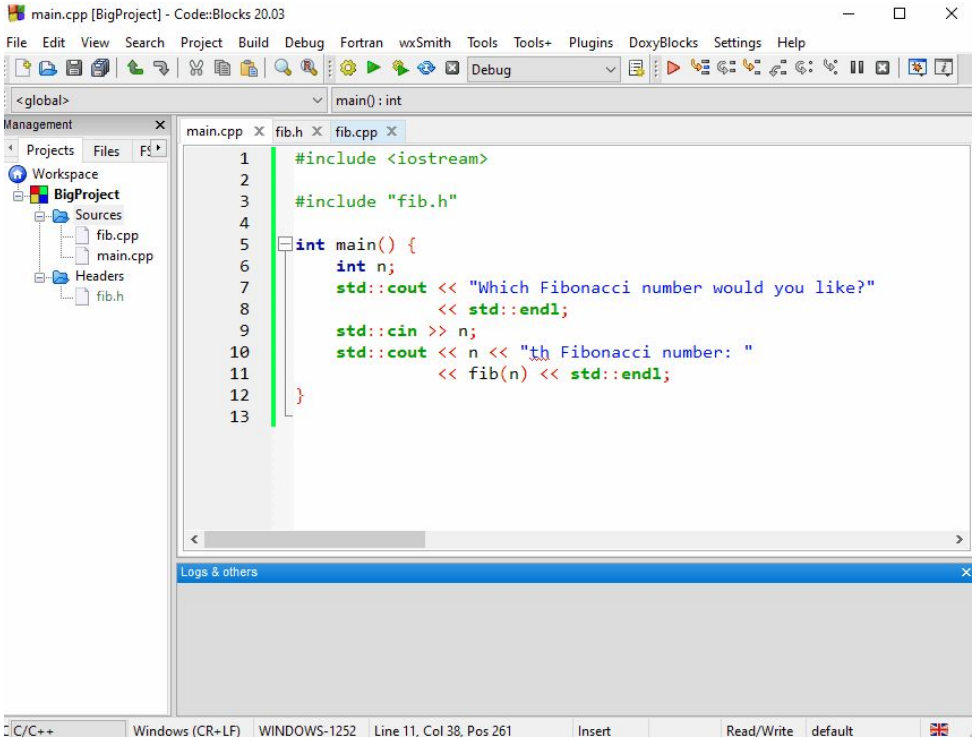
Andrej Ivašković

Matematička gimnazija

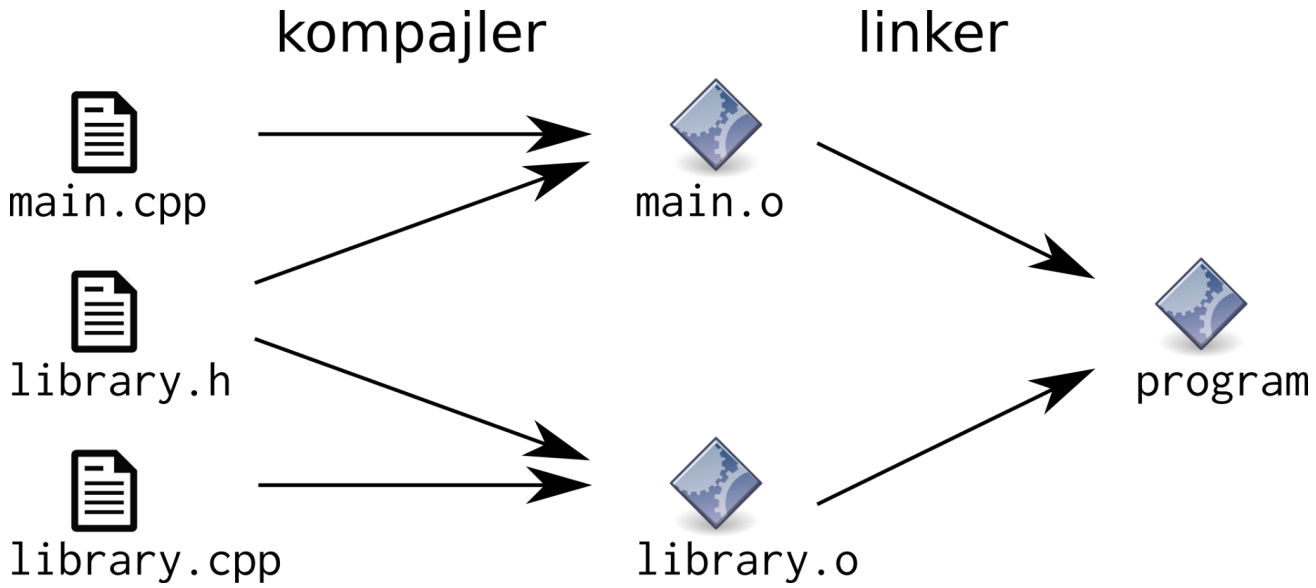
22. 04. 2021.

Magija?

- Put od (velikog) projekta do programa koji može da se izvrši na nekog mašini.
- Prijavljuju nam se greške (sintaksa, tipovi) i upozorenja.

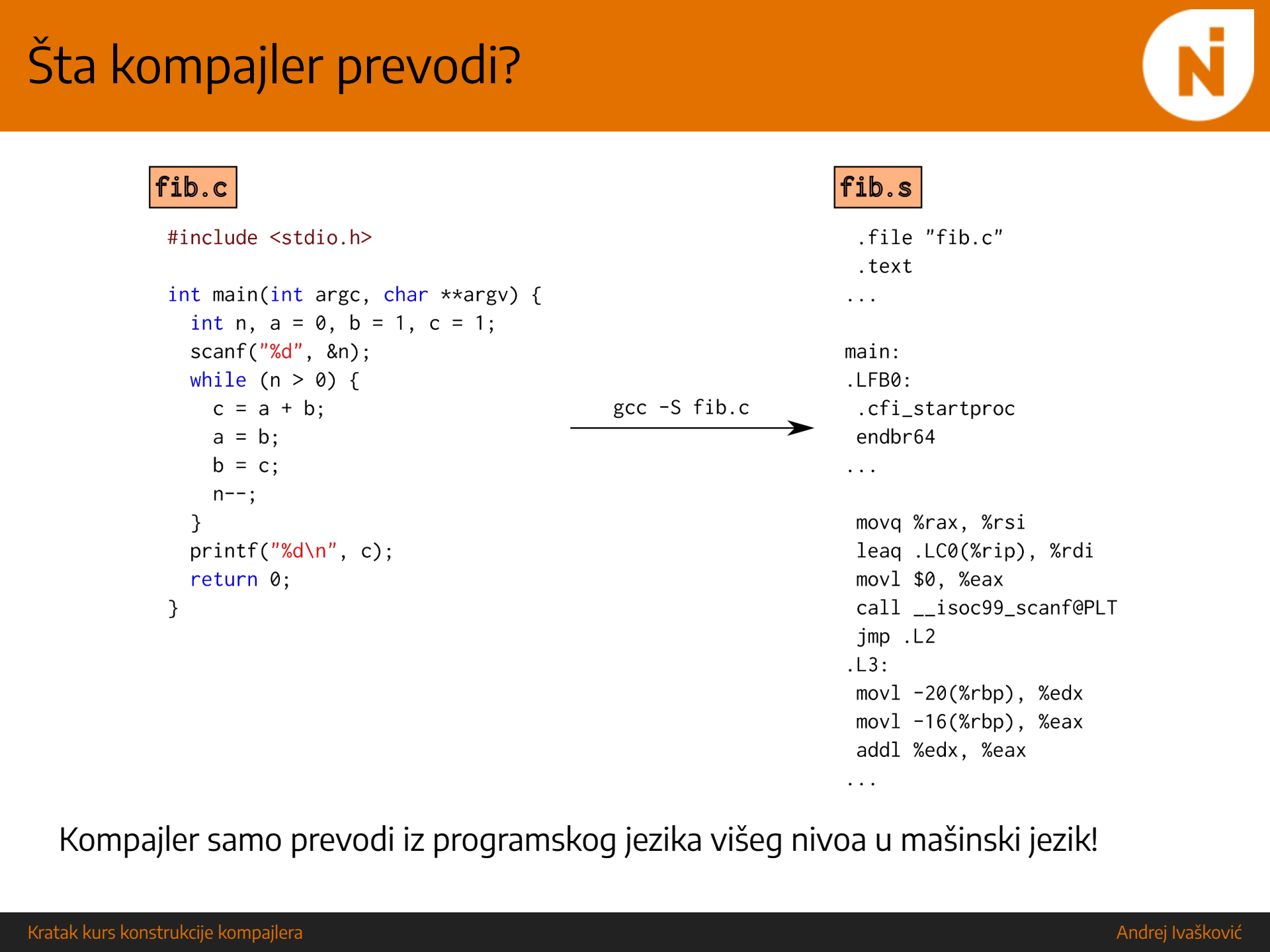


Komponente sa raznim zadacima



- ***Build* sistem** (Makefile, ninja, cmake, ...): koordiniše proces izrade *executable* programa (koraci, opcije kompajlera itd)
- **Kompajler** (ili *programski prevodilac*): proizvodi *objektne fajlove* od *izvornog koda*
- **Linker**: stvara *executable* fajl od objektnih fajlova, popunjava definicije

Šta kompajler prevodi?



fib.c

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int n, a = 0, b = 1, c = 1;
  scanf("%d", &n);
  while (n > 0) {
    c = a + b;
    a = b;
    b = c;
    n--;
  }
  printf("%d\n", c);
  return 0;
}
```

gcc -S fib.c →

fib.s

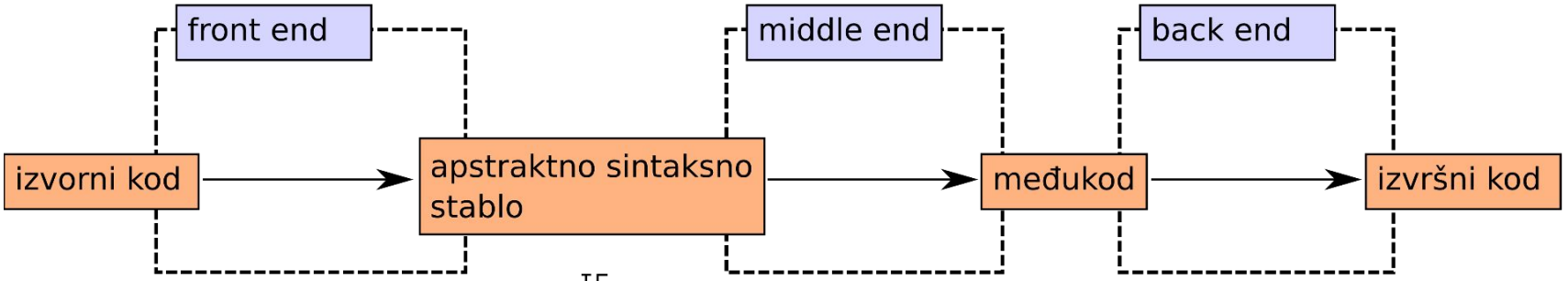
```
 .file "fib.c"
 .text
...

main:
.LFB0:
 .cfi_startproc
 endbr64
...

 movq %rax, %rsi
 leaq .LC0(%rip), %rdi
 movl $0, %eax
 call __isoc99_scanf@PLT
 jmp .L2
.L3:
 movl -20(%rbp), %edx
 movl -16(%rbp), %eax
 addl %edx, %eax
...
```

Kompajler samo prevodi iz programskog jezika višeg nivoa u mašinski jezik!

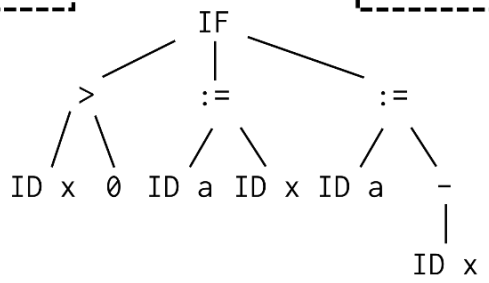
Agenda

1. Struktura kompajlera i termini
2. *Front end* NIlang kompajlera
3. *Middle* i *back end*
4. Implementacija daljih koncepata

Struktura kompajlera




```
IF x > 0
  THEN a := x
  ELSE a := -x
```



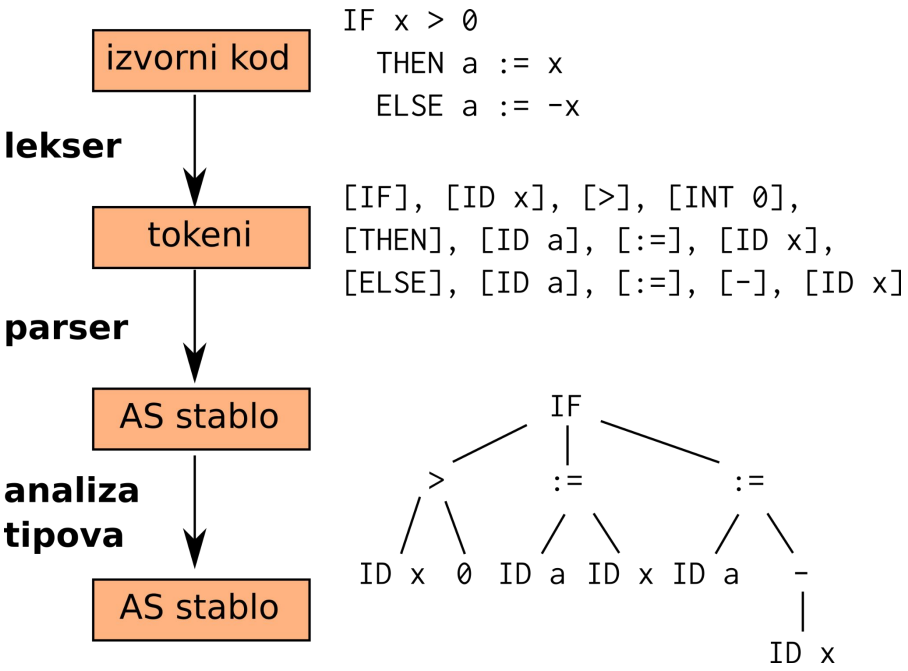

```
  CMPLE 0, x, L2
  STI a x
  JMP L4
.L2:
  NEG t x
  STI a t
.L4:
  NOP
```

```
  .cfi_def_cfa_register 6
  movl %edi, -4(%rbp)
  cmpl $0, -4(%rbp)
  jle .L2
  movl -4(%rbp), %eax
  movl %eax, a(%rip)
  jmp .L4
.L2:
  movl -4(%rbp), %eax
  negl %eax
  movl %eax, a(%rip)
.L4:
  nop
```

Tri dela kompajlera: **front end** (prijavljuje greške), **middle end** (optimizacija) i **back end** (stvara asemblerski/mašinski kod).

Front end struktura

- **Lekser**: kod treba da se predstavi u vidu niza **tokena** izbacivanjem blanko simbola i interpretacijom simbola i reči.
- **Parser**: od tokena formira **apstraktno sintaksno stablo** (*abstract syntax tree*, AST).
- **Analiza tipova**: analizom AST se radi provera tipova (*type checking*) ili određuju tipovi (*type inference*) radi prijave grešaka i pomoći kasnijim fazama kompajlera.
- Prijavljuju se i druge greške (sintaksa).



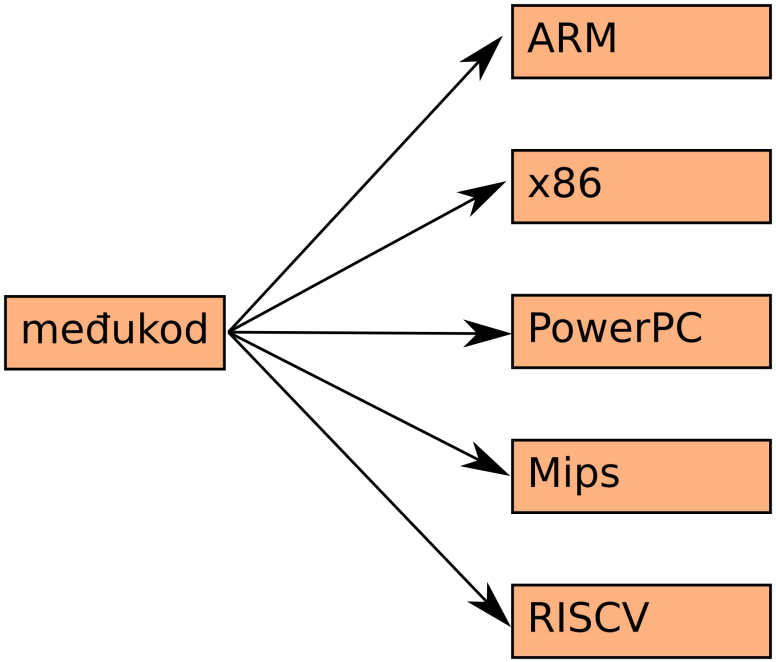
Middle end struktura

- Prevod iz AST u **međukod** (IR, *intermediate representation*) je zasnovan na *post-order* obilasku stabla.
- U međukodu su podaci raspoređeni u **virtuelne registre**.
- Sledi nekoliko faza **optimizacije** koda koje menjaju IR.
  - Odstranjivanje mrtvog koda (*dead code elimination*).
  - Odmotavanje ciklusa (*loop unrolling*).
  - Širenje konstanti (*constant propagation*)
  - Silne druge optimizacije, semantički ekvivalentan rezultujući kod.

```
LDI x, 5
LDI y, 2
LDI z, 0
ADD y, y, 1
MUL z, y, 4
ADD x, x, z
```

```
LDI x, 17
LDI y, 3
LDI z, 12
```

Back end struktura



- Prevod od međukoda do mašinskog koda je relativno jednostavan, ali treba pratiti konvencije *ciljne arhitekture*.
- Moderni kompajleri podržavaju više ciljnih arhitektura.
- Moguće je izvršiti neke optimizacije koje zavisa od ciljne arhitekture (korišćenje efikasnih instrukcija, zamena redosleda instrukcija).

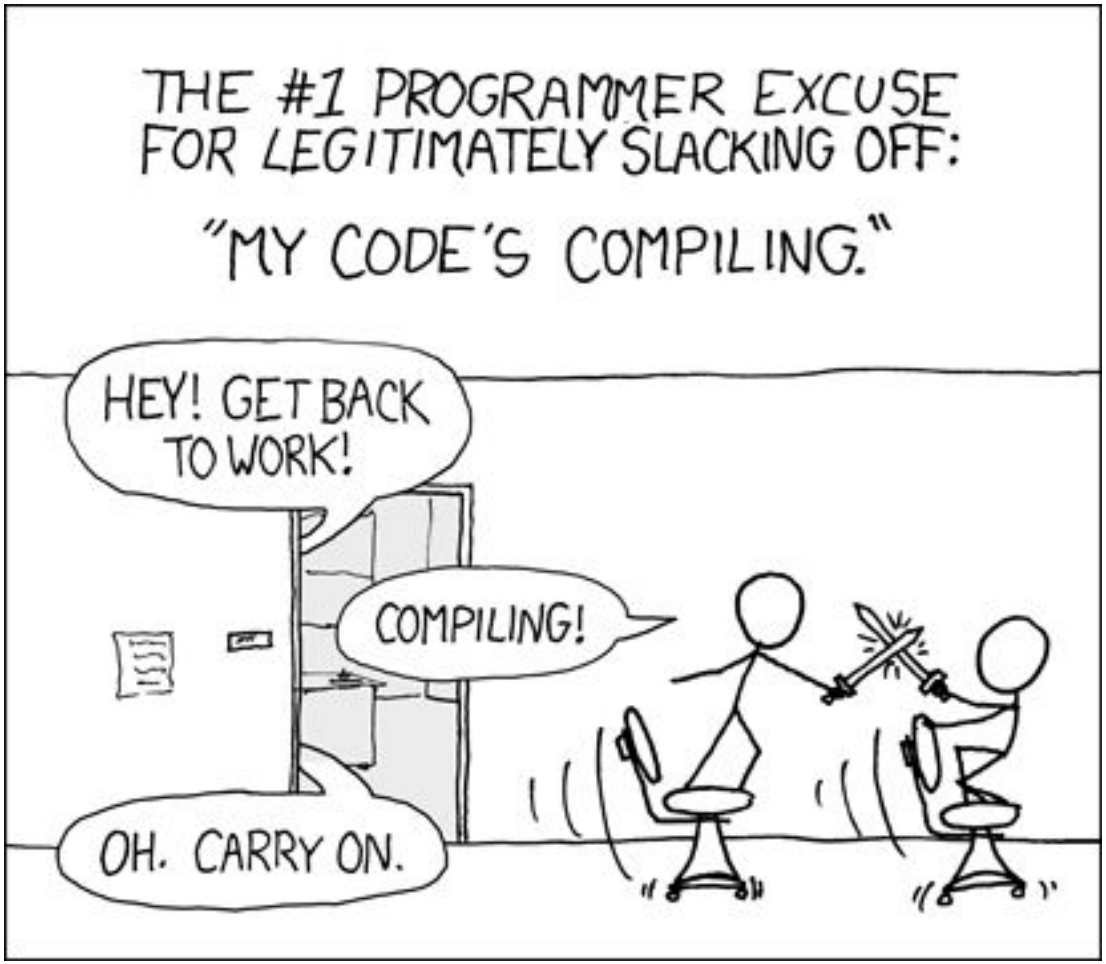
Interpreteri i JIT kompajleri

- **Interpreter** je program koji izvršava program u izvornom jeziku instrukciju po instrukciju (možda u međukodu).
  - Interpretirani jezici: Lisp, Python, MATLAB, Javascript...
  - Interpretirani jezici često podržavaju kompajlere ka ciljnom kodu.
- **JIT (Just in Time) kompajliranje** kombinuje ideje interpretera i kompajlera.
  - Program se prevodi u međukod (koji u ovom slučaju zovemo **bytecode**), ali se blok međukoda prevodi u mašinski kod čim interpreter treba da ga izvrši.
  - Primer: Java Virtual Machine (JVM) *bytecode* jezici poput Java i Scala.

NIlang: jezik Nedelje informatike

- Izradu kompajlera je teško razumeti apstraktno, najbolje je videti na primeru.
- **NIlang** (jezik Nedelje informatike) je jednostavan podskup programskog jezika Pascal i videćemo kako izgleda kompajler za taj jezik (pisan u C++-u).
- Koristimo razne alate da bismo pojednostavili proces konstrukcije kompajlera:
  - *front end* kompajlera koristi **ANTLR** (tačnije ANTLR4) za generisanje leksera i parsera;
  - *middle* i *back end* kompajlera se oslanjaju na **LLVM**, koji je savremena kolekcija alata za implementaciju i eksperimentisanje sa kompajlerima;
  - alati nam omogućavaju da koristimo objektno-orijentisan pristup.
- Implementacija kompajlera se nalazi na mom GitHub profilu: https://github.com/Dreian/NIlang

Kompajliranje kompajlera traje...



Agenda

1. Struktura kompajlera i termini
2. *Front end* NIlang kompajlera
3. *Middle* i *back end*
4. Implementacija daljih koncepata

NIlang sintaksa

- Sintaksa slična Pascal-u (sa istom gnjavažom oko “`;`” simbola).
- Dva tipa promenljivih: celobrojne (`integer`) i logičke (`boolean`).
- Četiri vrste instrukcija:
  - standardni izlaz (`print`);
  - dodela (`x := e`);
  - `while` ciklus;
  - `if` grananje.
- Očekivane aritmetičke i logičke operacije su prisutne.

```
var
  n : integer;
  s : integer;
begin
  n := 1000;
  s := 0;
  while n > 0 do begin
    s := s + n;
    n := n - 1
  end;
  print(s)
end.
```

ANTLR

- ANTLR je **generator leksera** i **generator parsera**, zajedno sa bibliotekama i interfejsima (API) za sintaksnu analizu.
  - Generatori leksera su programi koji za zadatu listu definicija tokena formira funkcije za leksiranje i tokeniziranje koda. Tradicionalno se program **lex** koristi u ove svrhe.
  - Generatori parsera su programi koji za zadatu definiciju sintaksnih konstrukcija (poput ciklusa, grananja i instrukcija) formiraju parser koji od liste tokena kreira stablo. Tradicionalno se koristi program **yacc**.
- ANTLR4 ima API za C++, C#, Javu, Python i par ostalih jezika.

Definicija leksera

- Definicija leksera je kratka i jednostavna, najbitniji delovi su prikazani na ovom slajdu.
- Prepoznavanje tokena je zasnovano na prepoznavanju **regularnih izraza**.
- Specijalno pravilo WS za blanko simbole: ignoriši sve!
- Redosled je bitan: da li je “`if`” ime promenljive ili ključna reč?

```
lexer grammar NILexer;

LT: '<';
LEQ: '<=';
ASSIGN: ':=';
SEMI: ';';
PLUS: '+';

IF: 'if';
THEN: 'then';
WHILE: 'while';
TRUE: 'true';
FALSE: 'false';

Int: DIGIT+;
Bool: TRUE | FALSE;
Ident: LETTER (LETTER | DIGIT |
                UNDER)*;

WS: [ \t\r\n]+ -> skip;
DIGIT: [0-9];
LETTER: [a-zA-Z];
UNDER: '_';
```

Kratko pitanje

Koji od narednih regularnih izraza odgovara opisu
“string nula i jedinica u kom se i jedinica i nula pojavljuju bar jednom”?

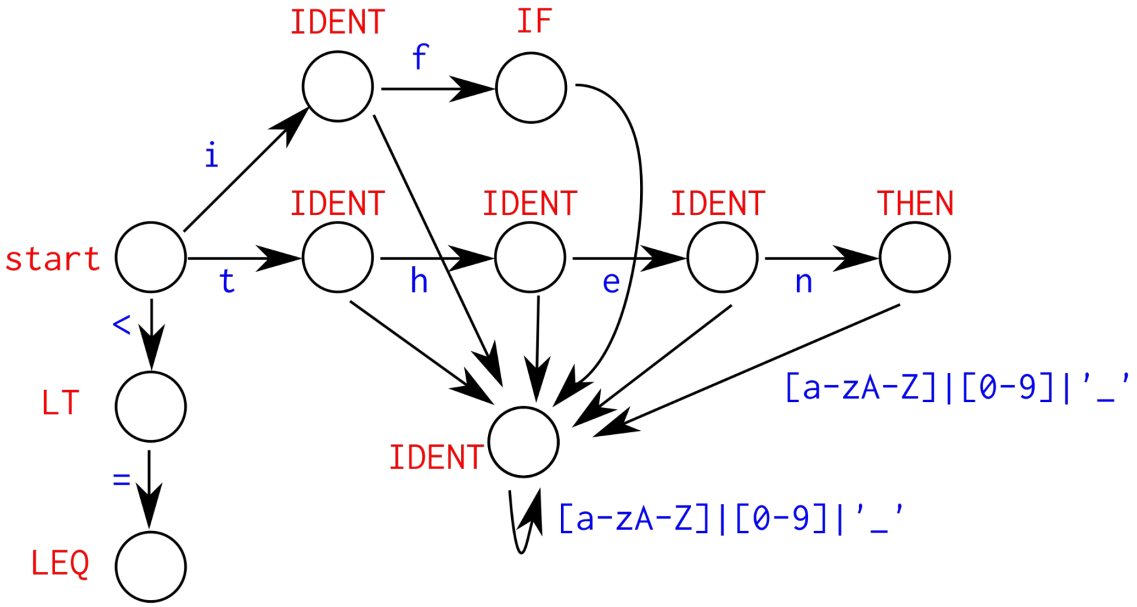
a. (0* 1*)*
b. (0|1)* 0* 1* (0|1)* | (0|1)* 1* 0* (0|1)*
c. (0|1)* (01 | 10) (0|1)*
d. (01)* (10)*

Lekser komponenta u našem kompajleru

- Ova specifikacija može da se pretvori u lekser korišćenjem ANTLR alata i tako se generiše biblioteka sa potpunom implementacijom leksera!
- Sve što sada treba da uradimo je da pozovemo generisane funkcije (važne provere izostavljene):

```
// input from the file
std::ifstream stream;
stream.open(argv[1]);
antlr4::ANTLRInputStream input(stream);
stream.close();

// lexing
NILexer lexer(&input);
antlr4::CommonTokenStream tokens(&lexer);
tokens.fill();
```

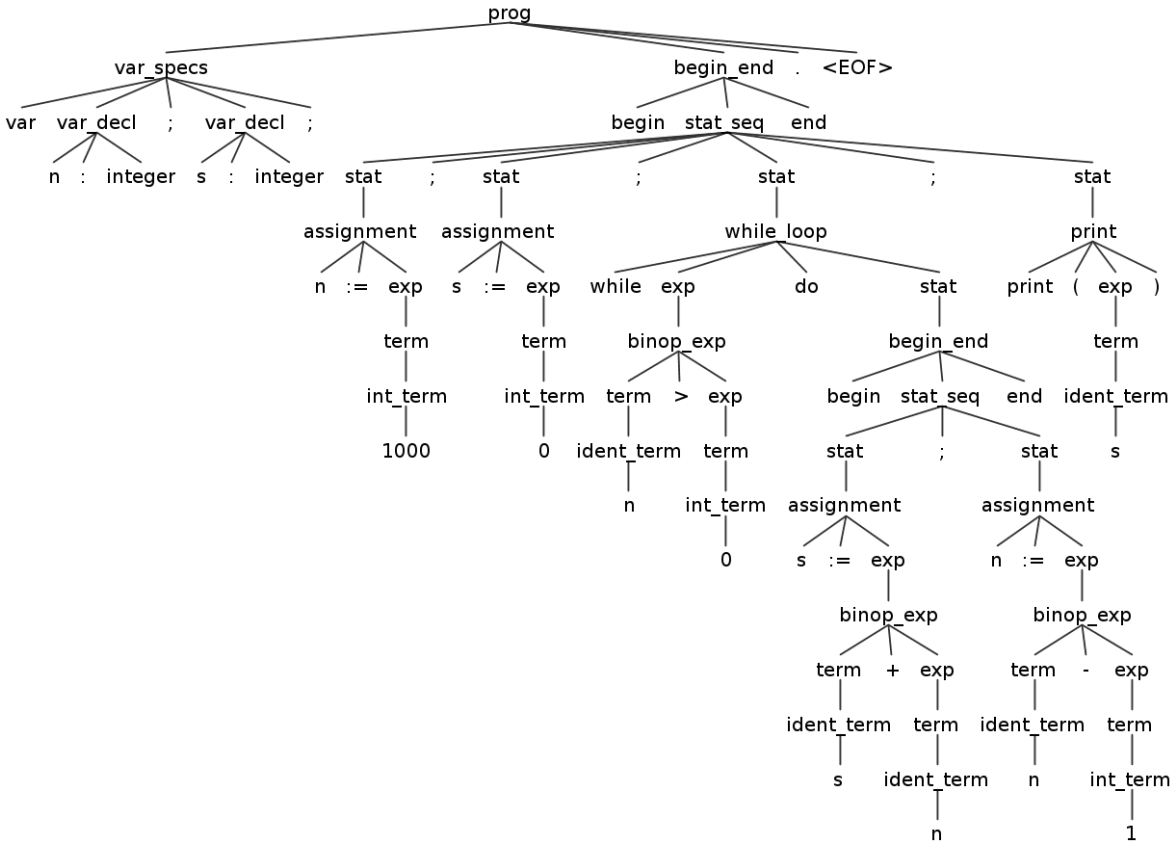
Kako funkcioniše lekser?



- Poznata je veza između regularnih izraza i **konačnih automata**.
  - Automat je graf u kom grane imaju oznake simbola i svaki string odgovara nekom putu u automatu.
- Od definicija svih tokena se formira jedan veliki automat (izuzetno uprošćena slika gore).
- Traži se najduži mogući prefiks koji odgovara nekom tokenu, nakon čega se emituje taj token i počinje leksiranje od prvog narednog simbola.

NIlang parser

- Želimo da definišemo parser koji će da grupiše tokene u smislene konstrukcije.
- Rezultat parsiranja u ANTLR4 nije apstraktno sintaksno stablo, već **stablo parsiranja** (slika).



Definicija parsera: promenljive i instrukcije

- Pratimo **kontekstno-slobodnu gramatiku** jezika (u formalnim specifikacijama se koristi notacija poput EBNF).
- Instrukcije se prepoznaju jednostavno: skoro sve počinju nekom ključnom reči.
  - Logično bi bilo koristiti neki algoritam **rekurzivnog spusta** za parsiranje...
  - (formalno: LL(*) gramatika)

```
prog: (var_specs)?
      begin_end DOT EOF;

var_specs: VAR (var_decl SEMI)+;
var_decl: Ident COLON Typename;

stat: assignment
    | if_then_else
    | if_then
    | while_loop
    | print
    | begin_end;

stat_seq: stat (SEMI stat)*;
begin_end: BEGIN stat_seq END;

assignment: Ident ASSIGN exp;
if_then: IF exp THEN stat;
if_then_else: IF exp THEN stat
                      ELSE stat;
while_loop: WHILE exp DO stat;
print: PRINT LPAR exp RPAR;
```

Definicija parsera: izrazi

- Parseri zasnovani na rekurzivnom spustu (kao što je ANTLR parser) imaju probleme sa **levom rekurzijom**:

  `exp: exp PLUS exp`
- ANTLR razrešava prioritet operacija na osnovu redosleda pravila, a u slučaju istog prioriteta se pridržava pravila “s leva na desno”:

  3−1+2 = (3−1)+2 ≠ 3−(1+2)

```
exp: minus_exp
   | not_exp
   | binop_exp
   | bool_binop_exp
   | term;

term: par_exp
    | int_term
    | bool_term
    | ident_term;

int_term: Int;
bool_term: Bool;
ident_term: Ident;
par_exp: LPAR exp RPAR;
not_exp: NOT exp;
minus_exp: MINUS exp;

binop_exp: term (TIMES | DIV) exp
         | term (PLUS | MINUS) exp
         | term (EQ | NEQ | LEQ | GEQ
                | GT | LT) exp;

...
```

Parser komponenta u našem kompajleru

- ANTLR generiše definicije za stablo parsiranja (definicija zasnovana na nasleđivanju u OOP) i kod koji od niza tokena generiše stablo parsiranja.
- Na nama je da pretvorimo stablo parsiranja u AST (koji moramo sami da definišemo): ANTLR očekuje da sami napišemo `visit` funkciju za stablo parsiranja.

```
// parsing
NIParser parser(&tokens);
auto tree = parser.prog();

// turning the parse tree into an AST
NIVisitor visitor;
auto visit_outcome = visitor.visit(tree);
auto ast = (AST::ProgNode*)visit_outcome;

// type check
```

Agenda

1. Struktura kompajlera i termini
2. *Front end* NIlang kompajlera
3. *Middle* i *back end*
4. Implementacija daljih koncepata

Definicija AST

```
class ASTNode {
public:
  virtual Type typeCheck(TypeEnv&) = 0;
  virtual llvm::Value *codegen(Compiler::Components&) = 0;
};

class VarDeclNode : public ASTNode {
private:
  IdentifierName variable;
  Type type;
public:
  VarDeclNode(IdentifierName, Type);
  virtual Type typeCheck(TypeEnv&) override;
  virtual llvm::Value *codegen(Compiler::Components&)
                                                  override;
};
```

LLVM

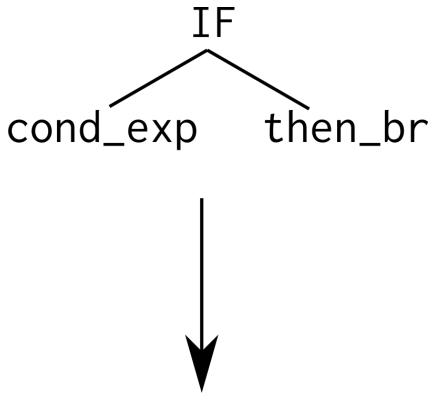
- **LLVM projekat** je skup alata koji mogu da se koriste za izradu *middle end* i *back end* delova kompajlera.
- Ključna komponenta je međukod (**LLVM IR**) i API koji se koristi za manipulaciju istim:
  - u C++ koristimo klasu llvm::IRBuilder za izradu programa:
    ```
    builder.CreateLoad(reg_from, "newreg_name")
    ```
- Implementirane optimizacije koje možemo da dodajemo po volji (a možemo i da eksperimentišemo sa novim, originalnim).

Primer LLVM IR

```
  br label %while.cond

while.cond:                                       ; preds = %while.body, %entry
  %n1 = load i32, i32* %n
  %gttmp = icmp sgt i32 %n1, 0
  %0 = icmp ne i1 %gttmp, false
  br i1 %0, label %while.body, label %while.cont

while.cont:                                       ; preds = %while.cond
  %z7 = load i32, i32* %z
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds
    ([4 x i8], [4 x i8]* @0, i32 0, i32 0), i32 %z7)
  ret i32 0

while.body:                                       ; preds = %while.cond
  %n2 = load i32, i32* %n
  %subtmp = sub i32 %n2, 1
  store i32 %subtmp, i32* %n
```

Generisanje LLVM IR: rekurzija

- Da bi se kompajlirao izraz koji ima glavnu operaciju sabiranje, treba:
  - kompajlirati dva sabirka (deca u AST);
  - onda generisati instrukciju koja te sabirke sabira i smešta u nov registar.
- Kompajliranje grananja se vrši tako što se:
  - kompajlira uslov grananja (dete u AST);
  - kompajlira telo grananja (takođe dete u AST);
  - definišu labele koje odgovaraju telu grananja i prvoj instrukciji nakon grananja;
  - ubace uslov i telo na odgovarajuća mesta;
  - Generiše instrukcija uslovnog skoka.
- Ovo su primeri **rekurzije**!




```
  %c = [cond_exp]
  BR_NOT_0 %c, if.cont
if.then:
  [then_br]
  JMP if.cont
if.cont:
  ...
```

IRBuilder: generisanje Binop koda

```
llvm::Value
*BinopExpressionNode::codegen(Compiler::Components &comp) {
  llvm::Value *left = left_exp->codegen(comp);
  llvm::Value *right = right_exp->codegen(comp);
  switch (op) {
  case PLUS:
    return comp.builder.CreateAdd(left, right, "addtmp");
  case MINUS:
    return comp.builder.CreateSub(left, right, "subtmp");
  case TIMES:
    return comp.builder.CreateMul(left, right, "multtmp");
  …
  }
}
```

IRBuilder: generisanje If koda

```
llvm::Value *IfNode::codegen(Compiler::Components &comp) {
  llvm::Value *condV = condition->codegen(comp);
  llvm::BasicBlock *cont =
    llvm::BasicBlock::Create(comp.llvmCtx, "if.cont",
                             comp.fun);
  llvm::BasicBlock *ifBody =
    llvm::BasicBlock::Create(comp.llvmCtx, "if.then",
                             comp.fun);
  condV = comp.builder.CreateIsNotNull(condV);
  comp.builder.CreateCondBr(condV, ifBody, cont);
  comp.builder.SetInsertPoint(ifBody);
  thenBranch->codegen(comp);
  if (comp.builder.GetInsertBlock() != nullptr) {
    comp.builder.CreateBr(cont);
  }
  comp.builder.SetInsertPoint(cont);
}
```

Tabela simbola

- Šta generišemo kada dođemo do čvora koji predstavlja promenljivu?
- Neophodna nam je **tabela simbola**, odnosno veza između imena i definicije promenljive (gde se nalazi u memoriji).
- U našem kompajleru tabela simbola je **heš tabela** koja slika stringove (imena promenljivih) u pokazivače na instrukcije koje alociraju memoriju za tu promenljivu:

```
std::unordered_map<IdentifierName,
                   llvm::AllocaInst*> symbols
```

Kompajler je gotov!

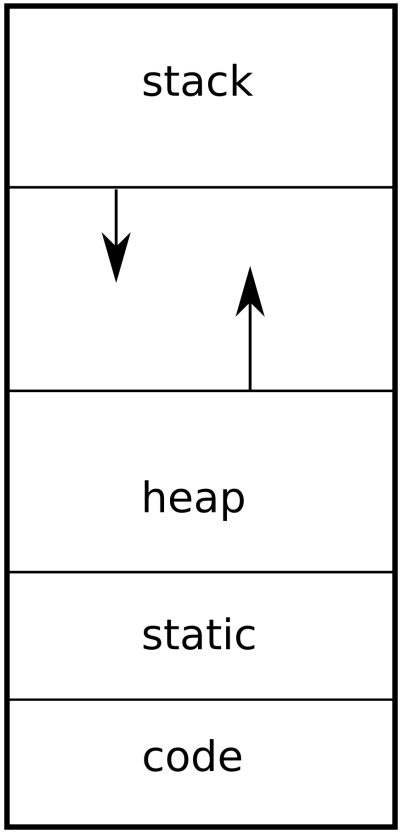
- Na ovaj način `ast->codegen()` prevodi ceo program u LLVM IR.
- Dovoljno je iskoristiti par poziva funkcija da bi se međukod preveo u asemblerski ili mašinski kod.
- Hajde da vidimo kako ovo radi!

Agenda

1. Struktura kompajlera i termini
2. *Front end* NIlang kompajlera
3. *Middle* i *back end*
4. Implementacija daljih koncepata

Raspored memorije

- Bilo šta komplikovanije zahteva da razumemo **raspored memorije** u programima.
- Uglavnom posmatramo četiri **segmenta**:
  - **kod**;
  - **statički segment** (globalne promenljive, definicije);
  - **stek** (pozivi funkcija, lokalne promenljive);
  - **hip** (dinamička memorija kreirana pomoću **new**, `malloc`...).



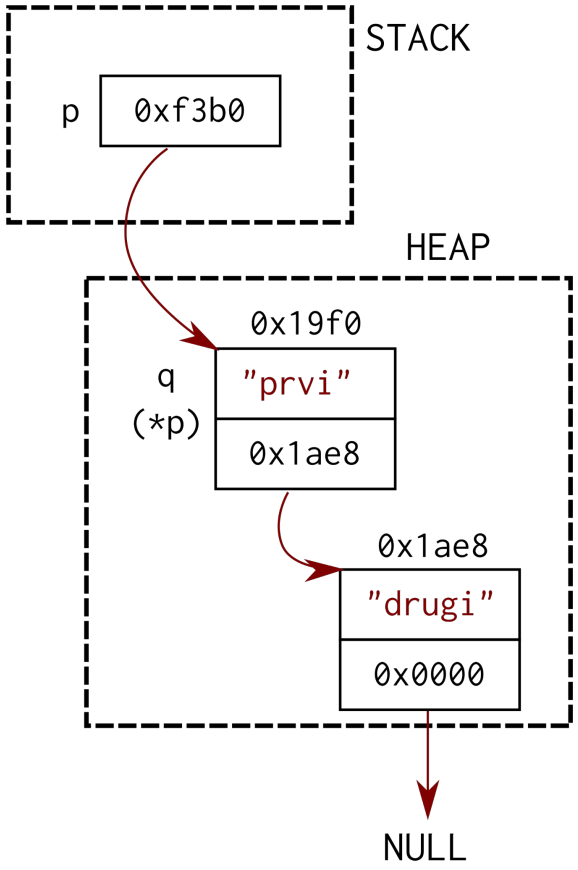
Potprogrami

- Jedan ***stack frame*** koji odgovara pozivu funkcije sadrži bar:
  - adresu povratka (za *program counter*);
  - prostor za rezultat;
  - prostor za argumente;
  - prostor za lokalne promenljive;
- LLVM IR sadrži **`alloca`** instrukciju koja se koristi za alokaciju lokalnih promenjivih.
- Dodatne instrukcije: **`call`** i **`ret`**.


```
define i32 @test(i1 %Condition) {
entry:
  %X = alloca i32
  br i1 %Condition, label %cond_true,
                    label %cond_false
...
cond_next:
  %X.2 = load i32* %X
  ret i32 %X.2
}
```

Dinamička memorija

- Šta ćemo sa dinamičkom memorijom (pokazivači na razne objekte, strukture podataka, ..., kojima može svaki *stack frame* da pristupi)?
- U LLVM IR ne postoji instrukcija koja ovo radi, moramo da zovemo C++ pozive (**new**, `malloc`, `free`...) koji koriste sistemske pozive.
- U samom jeziku biramo da li podržavamo **sakupljanje đubreta** ili ručnu alokaciju memorije.



Rezime

- Kompajleri su veliki i komplikovani!
- Postoje alati koji olakšavaju razvoj kompajlera, među kojima su ANTLR i LLVM.
- Tri dela kompajlera:
  - front end: lekser, parser, analiza tipova i prijava grešaka
  - middle end: rad sa međukodom, optimizacije
  - back end: generisanje mašinskog koda za ciljnu arhitekturu
- Stabla i rekurzija su veoma korisni koncepti u dizajnu kompajlera.

Hvala na pažnji!

Pitanja?