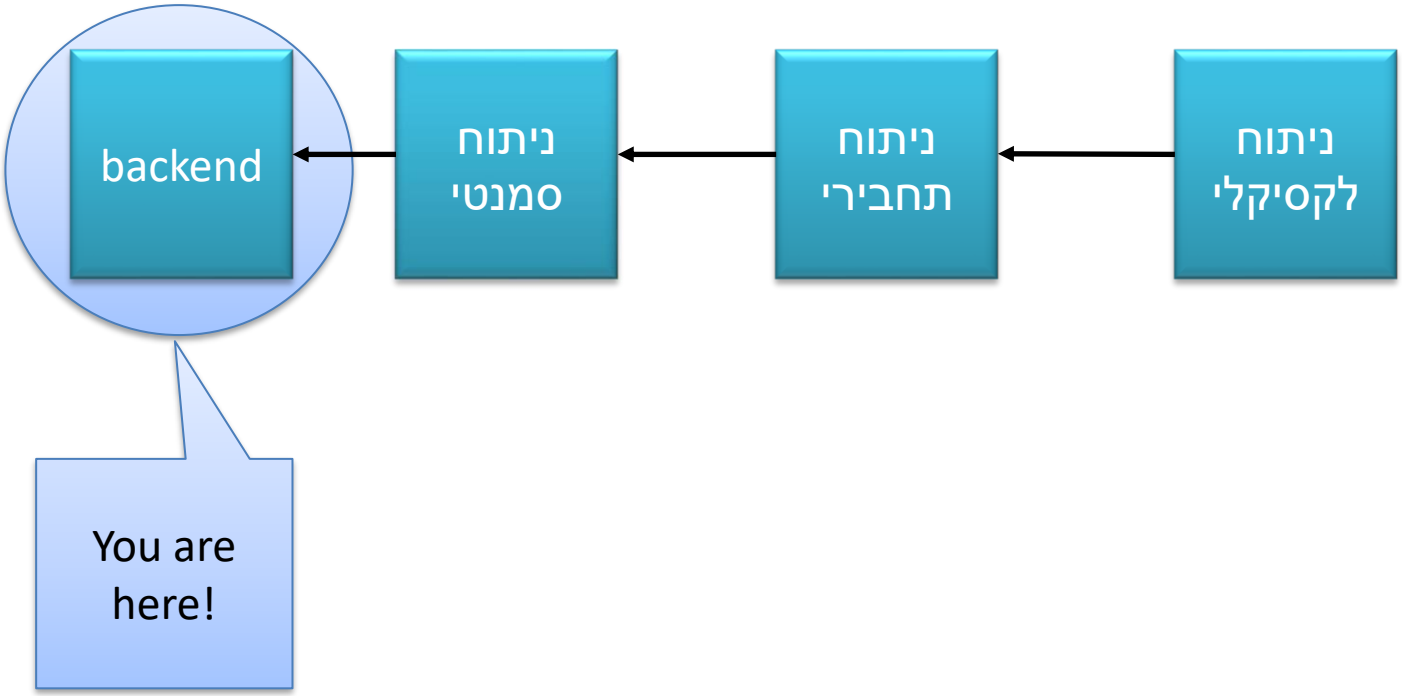
LLVM IR-
Low-Level Virtual Machine
Intermediate Representation

תזכורת מהתרגולים הקודמים

- מבנה סכמתי של קומפיילר



LLVM IR

- LLVM IR הינה שפת ביניים נפוצה הדומה לשפת 3AC (3 address code).
- כל ערך בשפה הוא בעל טיפוס שצריך לציינו.
- אין הגבלה על מספר הרגיסטרים שבהם ניתן להשתמש.
- Static Single Assignment (SSA) - לכל רגיסטר יש השמה יחידה.

```
%num = add i32 %inp, 48
```

target opcode type operands

LLVM IR

- נתמקד רק בחלק מאוד קטן מהשפה.
- המדריך המלא נמצא כאן:
https://llvm.org/docs/LangRef.html

טיפוסים

- שלמים:
  יכול להיות בעל כל מספר ביטים.
  לדוגמא: i1, i8, i32, i64
- label:
  כתובת של קוד.

אוסף (Aggregate):

- מערכים:
  בעל מימד וטיפוס בסיסי קבוע.
  לדוגמא: [4 x i8], [10 x i32]

טיפוסים

- מצביע:
  יכול להיות לכל טיפוס.
  לדוגמא: i8*, [4 x i32]*

פקודות

- add:
  מבצעת חיבור בין האופרנדים השלמים (signed או unsigned).
  לדוגמא:

```
%var1 = add i32 4, %var0
```

- sub:
  מבצעת חיסור בין האופרנדים השלמים.
  לדוגמא:

```
%var1 = sub i32 4, %var0
```

פקודות

- mul:
  מבצעת כפל בין שלמים.
  לדוגמא:

```
%var1 = mul i32 4, %var0
```

פקודות

- udiv:
  מבצעת חילוק בין האופרנדים השלמים (unsigned).

```
%var1 = udiv i32 4, %var0
```

- sdiv:

מבצעת חילוק בין האופרנדים השלמים (signed).

```
%var1 = sdiv i32 4, %var0
```

פקודות

- alloca:
מקצה מקום על המחסנית ברשומת ההפעלה של הפונקציה הנוכחית, שמפונה אוטומטית בעת סיום הפונקציה. הפקודה מחזירה מצביע מהטיפוס המתאים.
- לדוגמא:

```
%ptr = alloca i32
%ptr = alloca i32, i32 4
```

פקודות

- getelementptr:
מחשבת כתובת של אלמנט מתוך מצביע למשתנה שהוא מטיפוס שהוא אוסף (aggregate type). הפקודה מחזירה מצביע לאלמנט מהטיפוס המתאים.
- לדוגמא:

```
%MyArr = alloca [10 x i32]
%first = getelementptr [10 x i32],
        [10 x i32]* %MyArr, i32 0, i32 0

%last = getelementptr [10 x i32],
        [10 x i32]* %MyArr, i32 0, i32 9
```

הטיפוס עליו פועלים

המצביע

טיפוס המצביע

האינדקס של איבר המטרה במערך

האינדקס של המערך

פקודות

- store:
  כותבת ערך לזיכרון כאשר הכתובת היא מצביע לטיפוס הערך הנכתב.
- load:
  טוענת ערך מהזיכרון
- לדוגמא:

```
%ptr = alloca i32              ; yields i32*:ptr
store i32 3, i32* %ptr         ; yields void
%val = load i32, i32* %ptr ; yields i32:val = i32 3
```

Terminator Instruction

- כל בלוק LLVM חייב להתחיל עם label ולהסתיים ב- terminator instruction.
- Terminator Instruction שאנחנו משתמשים בהם בקורס הם: ret, br.
- אחרי terminator instruction חייב להתחיל בלוק חדש שנפתח בlabel.

פקודות

- icmp:
מבצעת השוואה בין האופרנדים לפי תנאי מסוים שהוא חלק מהפקודה ומחזירה ערך בוליאני (מטיפוס i1) בהתאם לתוצאת ההשואה.
- לדוגמא:

```
%var0 = icmp eq i32 4, 5  ; yields: result=false
%var1 = icmp ult i16 4, 5 ; yields: result=true
```

פקודות

- br:

מבצעת קפיצה לבלוק בסיסי אחר בתוך הפונקציה הנוכחית.

- ישנה פקודה לקפיצה מותנית ופקודה לקפיצה לא מותנית.
- לדוגמא:

```
br label %CondBr        ; Unconditional branch
CondBr:
  %cond = icmp eq i32 %a, %b
  br i1 %cond, label %IfEqual, label %IfUnequa
IfEqual:
  ret i32 1
IfUnequal:
  ret i32 0
```

פקודות

- call:
  מבצעת קריאה לפונקציה.
- חייבת להכיל את טיפוס החזרה מהפונקציה או את חתימת הפונקציה לפני שם הפונקציה.
- לדוגמא:

```
%retval = call i32 @test(i32 2)

%retval = call i32 (i32) @test(i32 2)
```

פקודות

- ret:

מבצעת חזרה מהפונקציה הנוכחית אל הפונקציה הקוראת וממשיכה בביצוע הפקודות שלאחר הקריאה.

- לדוגמא:

```
ret i32 5      ; Return an integer value of 5

ret void       ; Return from a void function
```

פקודות

- phi:

מחזירה את הערך של הרגיסטר בהתאם למסלול ממנו הגענו.

```
cond:
        %b = icmp slt i32 %i, %j
        br i1 %b, label %then,
                  label %else
then:
        %max = or i32 0, %j
        br label %exit
else:
        %max = or i32 0, %i
        br label %exit
exit:
        ret i32 %max
```

```
cond:
        %b = icmp slt i32 %i, %j
        br i1 %b, label %then,
                  label %else
then:
        %max1 = or i32 0, %j
        br label %exit
else:
        %max2 = or i32 0, %i
        br label %exit
exit:
                                %j
        %max = phi i32 [%max1, %then],
                       [%max2, %else]
        ret i32 %max            %i
```

הצהרה על פונקציה

```
define return_type @function_name(arg1_type, arg2_type,…) {…}
```

לדוגמא

```
define i32 @fn(i32) {…}
```

טיפול בקריאה לפונקציה

- בשפת הביניים של LLVM הקריאה לפונקציה נעשית באמצעות הפקודה call.
- הטיפול בקריאה ובחזרה מהפונקציה נעשית ע״י ה-backend של LLVM.
- ניתן להקצות מקום על המחסנית בתוך המסגרת של הפונקציה באמצעות alloca כך שהם מפונים ע״י ה-backend של LLVM בעת היציאה מהפונקציה.

דוגמא – חישוב פיבונצ'י

```
@.intFormat = internal constant [4 x i8] c"%d\0A\00”
define i32 @fn_fib(i32) {
fn_fib_entry:
  %1 = icmp sle i32 %0, 1
  br i1 %1, label %fn_fib_entry.if, label %fn_fib_entry.endif
 fn_fib_entry.if:
  ret i32 %0
fn_fib_entry.endif:
  %2 = sub i32 %0, 1
  %3 = sub i32 %0, 2
  %4 = call i32 @fn_fib(i32 %2)
  %5 = call i32 @fn_fib(i32 %3)
  %6 = add i32 %4, %5 ret i32 %6
}
define i32 @main() {
entry:
  %0 = call i32 @fn_fib(i32 10)
  %1 = call i32 (i8*, ...) @printf(i8* getelementptr ([4 x i8],
      [4 x i8]* @.intFormat, i32 0, i32 0), i32 %0)
  ret i32 0
}
declare i32 @printf(i8*, ...)
```

הרצת התוכנית

- lli הינה תוכנית המקמפלת ומריצה באמצעות JIT compilation (just-in-time) תוכניות הכתובות בשפת LLVM.
- נשתמש בתוכנית lli בכדי להריץ את התוכנית שייצרנו ב-LLVM IR.
- הפקודה הינה:

```
lli example.ll
```