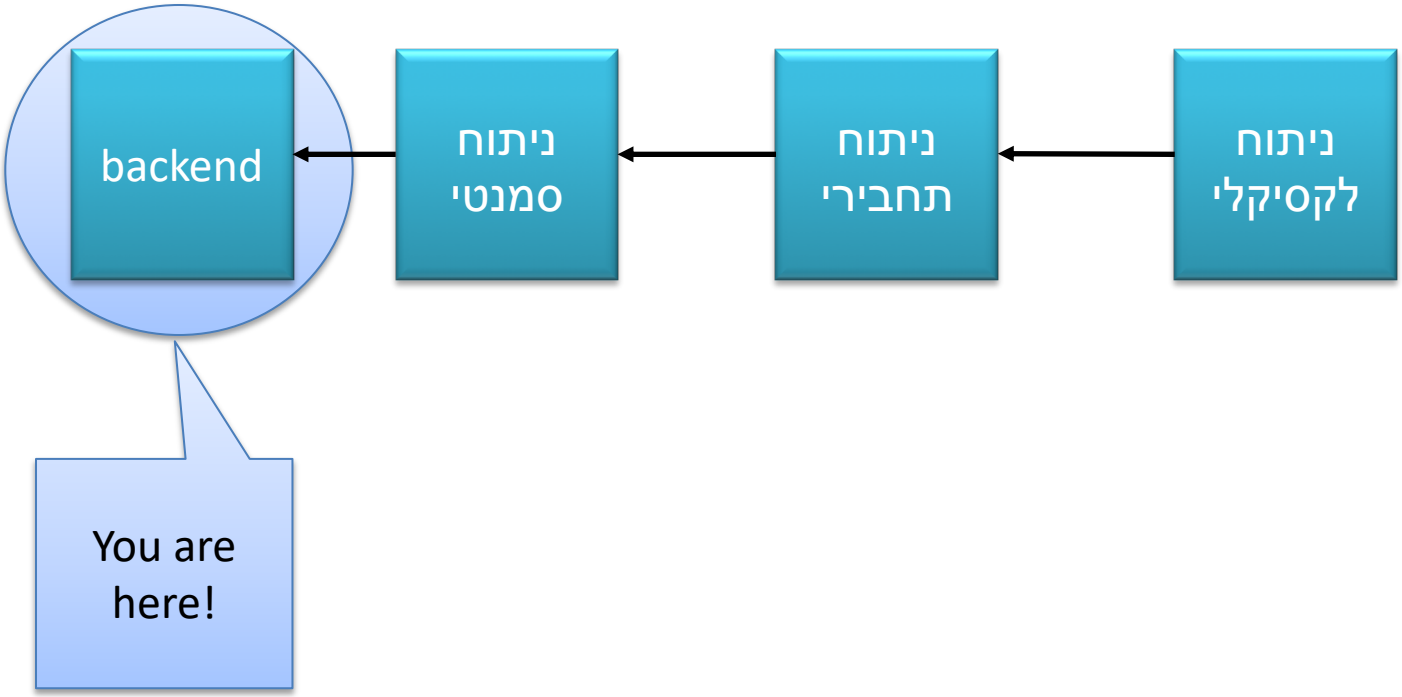
רשומות הפעלה,
LLVM IR

תזכורת מהתרגולים הקודמים

- מבנה סכמתי של קומפיילר



תמיכה בפרוצדורות

- תמיכה בפרוצדורות יוצרת אתגרים שעוד לא ראינו
- כל פרוצדורה צריכה
  - גישה למשתנים הפנימיים שלה
  - גישה לארגומנטים
  - לדעת לאן לקפוץ בסיום הפרוצדורה

```
int add(int x, int y)
{
   int inc = x;
   inc = inc + y;
   return inc;
}
```

קופצים למים העמוקים – רקורסיות!

```
int frac(int n)
{
   if (n == 1)
      return 1;
   return n*frac(n-1);
}

void f()
{
   frac(5);
}
```

- איך נבדיל בין המשתנים של כל אחת מהקריאות?
- איך נדע אם בסוף frac צריכים לחזור לקריאה קודמת לfrac או לf?

הפתרון – Activation Records

- בכל קריאה לפרוצדורה נפתח scope חדש עבור הקריאה
- נייצג את הscope של הפרוצדורה באמצעות activation record (aka frame)

- נחזיק activation records במחסנית
  - הactivation record העליון שייך לפרוצדורה הנוכחית
  - מתבצע בזמן ריצה
  - מחסנית גדלה כלפי מטה (מתחילים מכתובות גבוהות)

```
void f()
{
    frac(5);
}
```

```
int frac(int n)
{
    int frac(int n)
```

```
int frac(int n)
{
    int frac(int n)
    {
        if (n == 1)
            return 1;
        return n*frac(n-1);
    }
```

מה קורה אם נגמר המקום במחסנית?

המחסנית

f()

frac(5)

frac(4)

frac(3)

frac(2)

frac(1)

Stacktrace

Activation record structure

- הActivation record מכיל:
  - ארגומנטים לפרוצדורה
  - משתנים מקומיים של הפרוצדורה
  - מידע מנהלתי
    - גודל קבוע

החלק המנהלתי

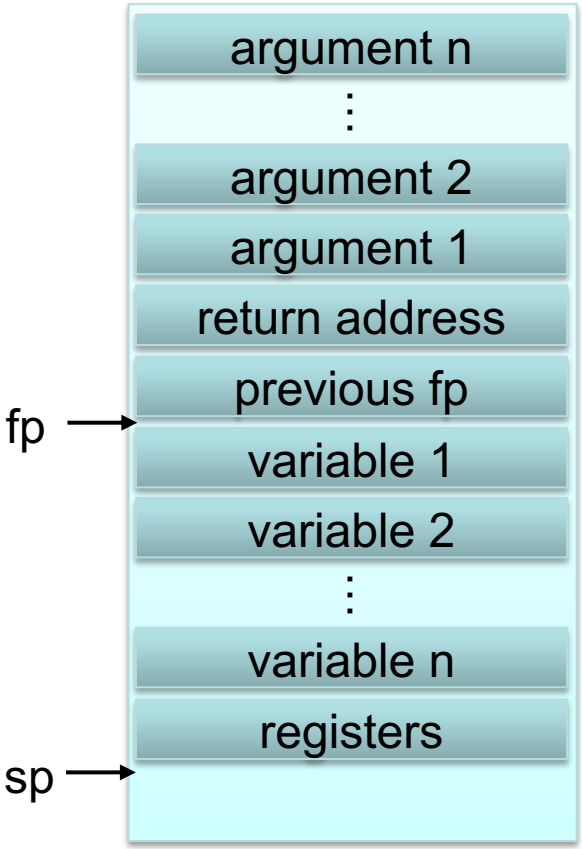
- מבטיח המשך ריצה תקין לאחר סיום הפרוצדורה

- **בעיה:** לא יודעים לאן לקפוץ בסוף הפרוצדורה
- **פתרון:** נשמור את כתובת החזרה

- **בעיה:** כמות מוגבלת של רגיסטרים, ערכים חשובים עלולים להידרס, לא יעיל לכתוב לזכרון בכל קריאה
- **פתרון:** נגבה את ערכי הרגיסטרים

- וכו'

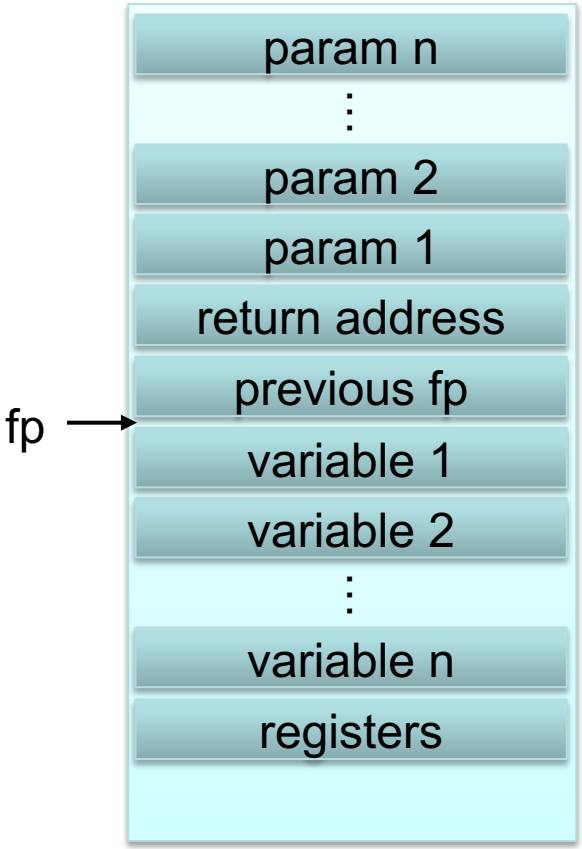
שימוש בactivation records



- נשתמש בשני רגיסטרים מיוחדים
- Frame Pointer (fp)
  - לעיתים נקרא גם Base Pointer (bp)
  - מצביע לframe של הפרוצדורה הנוכחית
- Stack Pointer (sp)
  - מצביע לכתובת הפנויה הבאה במחסנית
- שניהם מגובים בחלק המנהלתי

* הסדר בין החלקים והמידע המנהלתי עלולים להשתנות בין ארכיטקטורות/מערכות הפעלה

גישה למשתנים



- לכל משתנה offset ידוע מראש מFP
  - בהתאם לטבלת סמלים שבנינו בתרגול 6
- משתנים -> offset שלילי
- ארגומנטים -> offset חיובי

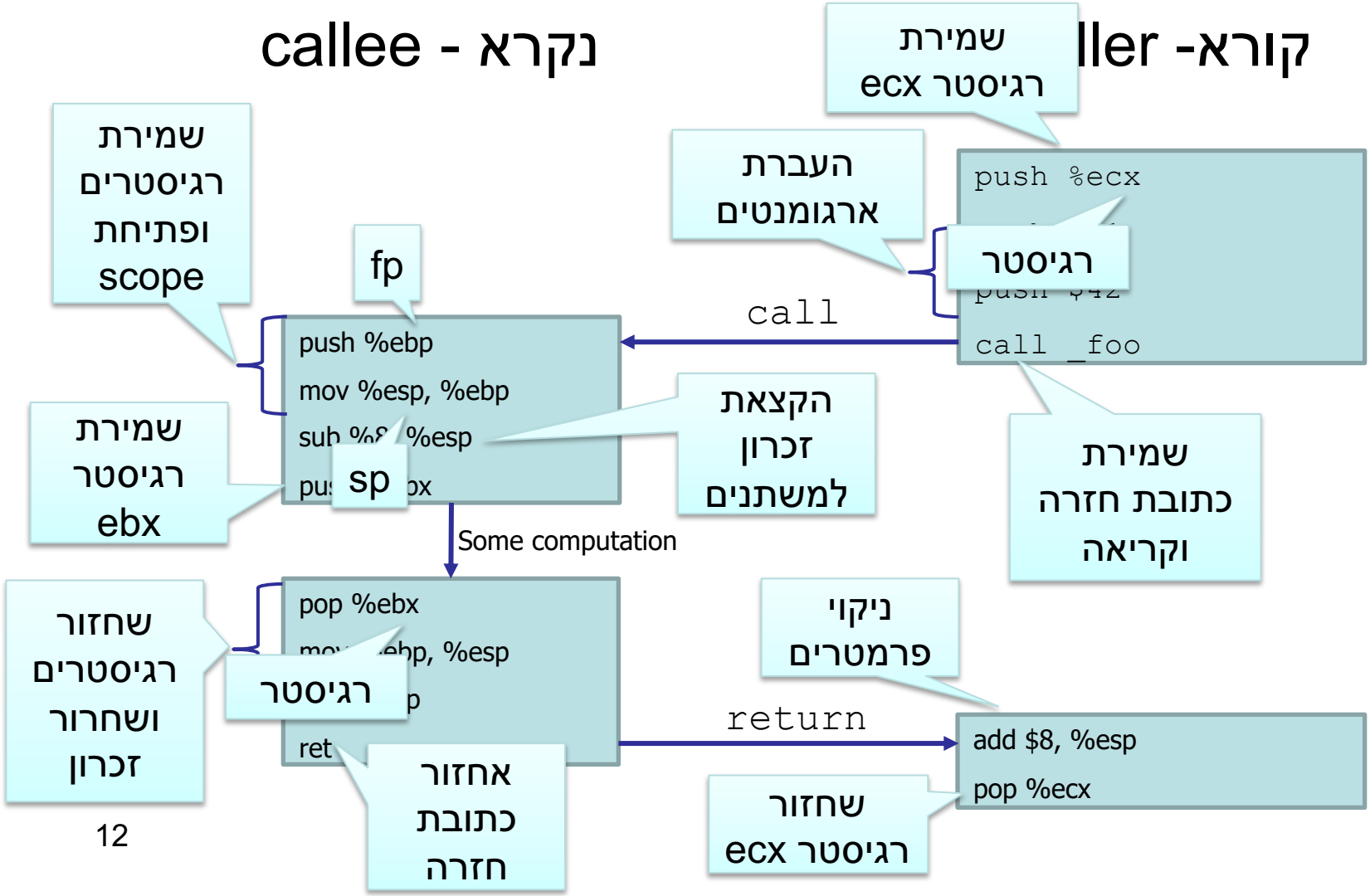
- למשתנים גלובליים ומשתנים בheap ניגש על פי כתובת אבסולוטית

טיפול בקריאות

- בכניסה לפונקציה:
  1. העברת ארגומנטים
  2. שמירת כתובת חזרה
  3. הקצאת זכרון למשתנים
  4. גיבוי ערכי רגיסטרים
- ביציאה מפונקציה:
  1. שחזור ערכי רגיסטרים
  2. שחרור זכרון מוקצה
  3. "מחיקת" ארגומנטים
  4. שמירת ערך חזרה
  5. אחזור כתובת חזרה

מי אחראי לכל שלב?

דוגמה (x86)


קורא- ller
נקרא - callee
שמירת רגיסטר ecx
העברת ארגומנטים
רגיסטר
push %ecx
call _foo
call
שמירת כתובת חזרה וקריאה
fp
שמירת רגיסטרים ופתיחת scope
push %ebp
mov %esp, %ebp
sp
%esp
הקצאת זכרון למשתנים
שמירת רגיסטר ebx
Some computation
pop %ebx
רגיסטר
שחזור רגיסטרים ושחרור זכרון
ניקוי פרמטרים
return
add $8, %esp
pop %ecx
אחזור כתובת חזרה
שחזור רגיסטר ecx

חלוקת אחריות

- באחריות הקורא (caller):
  1. העברת ארגומנטים
  2. שמירת כתובת חזרה
- אחריות הנקרא (callee):
  1. הקצאת זכרון למשתנים
  2. שחרור זכרון מוקצה
  3. שמירת ערך חזרה
  4. אחזור כתובת חזרה
- שאר ההחלטות לא כל כך ברורות
  - מי אחראי למחוק את הארגומנטים?
  - מי אחראי לגבות ערכי רגיסטרים?

העברת ארגומנטים

- איך מעבירים?
  - שמירה במחסנית
  - שמירה ברגיסטרים
    - במקרה הזה אין מה לנקות
  - חצי-חצי?
- באיזה סדר להעביר?
- מי מנקה?
  - שני הצדדים יודעים כמה ארגומנטים צריך לנקות... האמנם?

העברת ארגומנטים

- שמירה במחסנית


```
push 5
push 37
call f
```

- שמירה ברגיסטרים


```
mov eax, 5
mov ebx,37
call f
```

```
int f(int a, int b)
{
    ...
}
```

```
void g()
{
    f(5,37);
}
```

העברת ארגומנטים

- שמירה במחסנית
  - חסרונות:
    - גישה יותר איטית לארגומנטים
    - צריכים לנקות את המחסנית
- שמירה ברגיסטרים
  - חסרונות:
    - מגביל את כמות הארגומנטים
    - צריכים לגבות את ערכי הרגיסטרים

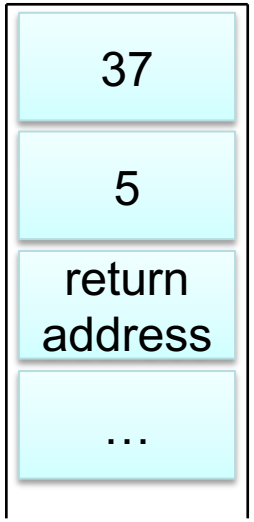
```
int f(int a, int b)
{
   …
}
```

```
void g()
{
   f(5,37);
}
```

העברת ארגומנטים דרך המחסנית

- סדר שמירת הארגומנטים
  - לפי סדר ההופעה
  - בסדר הפוך


```
push 5
push 37
call f
```




```
push 37
push 5
call f
```

```
void g()
{
    f(5,37);
}
```

```
int f(int a, int b)
{
    ...
}
```

העברת ארגומנטים דרך רגיסטרים

- באיזה רגיסטרים להשתמש?
  - כולם? חלק?
- באיזה סדר להקצות את הרגיסטרים?
  - באיזה רגיסטר שמור הארגומנט הראשון? באיזה שמור השני, השלישי וכו'...

```
int f(int a, int b)
{
   …
}
```

```
void g()
{
   f(5,37);
}
```

החזרת ערכים

- דומה לדילמת הפרמטרים
- איך מחזירים ערך?
  - שמירה במחסנית
    - הקורא צריך למחוק את הערך מהמחסנית
  - שמירה ברגיסטר מיוחד
- מה נעשה אם נרצה להחזיר אובייקט לא כמצביע?

גיבוי ערכי רגיסטרים

- מי אחראי לגבות ערכים של רגיסטרים חשובים?
  - הקורא יודע איזה רגיסטרים חשוב לשמור
  - הנקרא יודע איזה רגיסטרים ידרסו
- חלוקת הרגיסטרים לקבוצה באחריות הקורא וקבוצה באחריות הנקרא?
  - משפיע על הקצאת רגיסטרים על ידי הקומפיילר

שמירת כתובת חזרה

- שמירה במחסנית או שמירה ברגיסטר מיוחד?
- איזה כתובת לשמור?
  - כתובת נוכחית או כתובת הפקודה הבאה?

- מעשית, ההחלטה הזאת לא נתונה להחלטת הקומפיילר אלא נקבעת על ידי המעבד!

דילמות בחלוקת האחריות

- איזה גישה עדיפה?

- אין תשובה יחידה נכונה!
  - תלוי ביכולות המעבד
  - תלוי במוסכמות הנהוגות

Application Binary Interface

- מגדיר את האינטרקציה בין הפונקציה הקוראת והפונקציה הנקראת
- מגדיר חלוקת אחריות בין הצדדים
- מגדיר את תכולת ומבנה החלק המנהלתי
- Calling conventions

* משתנה בין מערכות הפעלה / ארכיטקטורות / קומפיילרים

למה ABI חשוב?

- למה מעניין אותי ABI אם אני מקמפל את הקוד?

- עבודה עם ספריות
  - התממשקות עם ספרייה קיימת
    - איך לקרוא לפונקציות של הספרייה
  - בניית ספרייה חדשה
    - שאחרים ידעו איך לקרוא לפונקציות בספרייה

דוגמה - X86 ABI

- העברת ארגומנטים במחסנית (בסדר הפוך)
- החזרת ערכי חזרה ברגיסטר ייעודי
- גיבוי חלק מהרגיסטרים באחריות הcallee
  - וגיבוי שאר הרגיסטרים באחריות הcaller

- אבל עדיין יש הבדלים...

דוגמת X86 ABI

- בקוד מונחה עצמים, איך מעבירים את *this*?
  - בlinux דרך המחסנית, בwindows רגיסטר ייעודי

```
class Test {
private:
    int x;
public:
    void setX(int x) {
        this->x = x;
    }
};
```

- מי מנקה את המחסנית?
  - בlinux הcaller, בwindows הcallee
    - אלא אם יש מספר לא ידוע של ארגומנטים

דוגמת X86 ABI

```
printf("%d",1);
printf("%d,%d",1,2);
```

- כמה ארגומנטים printf צריך לנקות מהמחסנית?
  - הקוד של printf לא משתנה עבור כל קריאה
  - printf לא יכול לדעת כמה ארגומנטים יועברו לו

- בקריאות לפרוצדורות שמשתמשות במספר לא ידוע של פרמטרים (ellipsis), ניקוי המחסנית באחריות הcaller (גם אם הABI אמר אחרת)

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.

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 , 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
```

פקודות

- icmp:

מבצעת השוואה בין האופרנדים לפי תנאי מסוים שהוא חלק מהפקודה ומחזירה ערך בוליאני (מטיפוס i1) בהתאם לתוצאת ההשוואה.

- 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.
- חיבת 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
```

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
```

שאלה ממבחן

סעיף א':

- נתונה תכנית C שקומפלה עם הצהרה על הפרוצדורה הבאה:

```
void printParams(int x, int y);
```

- בהכנה לקריאה לפרוצדורה סדר רשומת ההפעלה הוא: רגיסטרים, fp ישן, כתובת חזרה, ארגומנטים לפרוצדורה בסדר הפוך

- ציירו את רשומת ההפעלה עבור הקריאה:

```
printParams(2,3)
```

שאלה ממבחן


...
רשומת הפעלה
של הcaller
saved registers
old fp
return address
3
2
כיוון
גדילת
המחסנית

שאלה ממבחן

סעיף ב':

- נפלה טעות ובתור מימוש לפונקציה קומפל המימוש הבא:

```
void printParam(int x, int y, char* z) {
        printf("%s",z);
}
```

- בהנחה שהקומפילציה עברה בשלום, תארו מה יקרה בעת ביצוע הקריאה מסעיף א'

שאלה ממבחן

- הפונקציה תחפש ארגומנט שלישי שלא קיים
  - במקום תמצא את כתובת החזרה
- כתובת החזרה מצביעה לאזור הקוד
- printf ידפיס את תוכן הזיכרון
  - יתייחס לקוד כמחרוזת ascii
  - ידפיס עד שיראה terminating null
  - מה יקרה אם לא ימצא null?

בשבוע הבא

- אופטימיזציות