פרויקט הסיכום

היום נדון בפרויקט – איך להתחיל אותו, איך לטפל בקבועים ובמשתנים גלובליים ומה עושים עם פונקציות ספריה.

איך להתחיל

- עלינו לכתוב קובץ C קטן שמבצע אתחול וסיום המכונה, ולבדוק שזה עובד
  - זכרו שהקוד שאנו מייצרים הוא "יעני אסמבלי" – זהו קוד C, שמורכב משימוש ב-macros שמדמים אסמבלי שהמרצה המציא.
- נצטרך להשתמש ב-makefile
  - ניתן להשתמש ב-"makefile לא קיים" [[אני לא יודע מה זה]].
  - אחרת ניתן ליצור makefile עם שורה אחת שקוראת לקומפיילר.
  - שימו לב **לא להשתמש ב-makefile מקורסים קודמים** – אם אתם לא מבינים מה הם עושים ולמה, מאוד ייתכן שתשאירו תלויות לא רלוונטיות ואז הוא לא יעבוד בבדיקה.
  - הדבר שהכי כדאי לעשות זה לכתוב makefile בעצמכם מאפס – זה **באמת** לא מסובך; תמצאו באינטרנט makefile tutorial, תבינו איך זה עובד ואיך כותבים קובץ בסיסי – וזה כבר יספיק לכם.
  - שימו לב שיש להוסיף את הדגלים הבאים בהפעלה של gcc:

```
-Wall -pedantic-errors
```

זה יראה לנו את כל ה-warnings וה-errors האפשריים.

[[**הערה**: בבדיקה שלי מצאתי שהדגל `-pedantic-errors` גורם לקומפיילר לתת שגיאה על כל מה שלא בסטנדרט של C. אמרנו בעבר שבתשתית של פרויקט עושים שימוש נרחב בפונקציונליות של labels וְ-goto ש-gcc מממש שלא נמצאת בסטנדרט של C (ולכן לא ניתן להשתמש בקומפיילר אחר) – לכן שימוש בדגל זה יגרום לכל תכנית המשתמשת בארכיטקטורה שלנו להניב שגיאות קומפילציה!
אני ממתין לקבלת תשובה בפורום לגבי עניין זה, אבל בינתיים מציע שלא תשתמשו בדגל הנ"ל (אני מאמין שזה גם מה שמאיר יגיד לגבי זה).]]

  - פרויקט שיש בו שגיאות או warnings יקבל אוטומטית אפס!
  - שימו לב **לא לשים** את הדגל `-ansi`, אחרת כל הקריאות לפרוצדורות לא תעבודנה!

- קובץ ה-C שנייצר יהיה מחולק ל-3 חלקים:

התחלה (prologue)
הקוד שנוצר ע"י ה-code generator
סיום (epilogue)

- ה-prologue וה-epilogue הם קבועים (כלומר תמיד יכילו את אותו הקוד), לכן ניתן לעשות אחד משני דברים:
  - לייצר את הקוד הזה בשני קבצים, ואז שה-code generator ישרשר את הקובץ הראשון, הקוד שהוא מייצר והקובץ השני.
  - להוסיף את המחרוזות של הקבצים הללו כמחרוזות קבועות שה-code generator שם בהתחלה ובסוף.
- בתחילת ה-epilogue, יש להוסיף קוד שבודק את הערך שנמצא ב- R_0 : אם הוא null (הקבוע, שנגדיר עוד מעט), לא עושים כלום; אחרת, קוראים ל-write_sob ומדפיסים את מה שיש שם.

- עלינו לכתוב פרוצדורה שפותחת קובץ ומחזירה את כל ה-SExprs שיש בו:

```
(define file->sexprs
  (lambda (filename)
    (let ((input (open-input-file filename)))
      (letrec ((run
                 (lambda ()
                   (let ((e (read input)))
                     (if (eof-object? e)
                         (begin (close-input-port input)
                                '())
                         (cons e (run)))))))
        (run)))))
```

כאשר נפעיל את הפרוצדורה הזו על שם קובץ קלט, למשל:

```
(file->sexprs "moshe.scm")
```

הערך שהפרוצדורה תחזיר יהיה רשימת כל ה-sexprs שיש בקובץ.

- המלצה:
  - בשלב הראשון אל תכניסו את האופטימיזציה של tail calls, כי עדיין לא תמכתם בזה בקומפיילר.
    - כלומר, בהתחלה תעשו רק parse, fold ו-annotate-vars [[אני לא בטוח שצריך את fold]].
  - **<u>זכרו כן להוסיף את זה אח"כ!</u>**

- איך נראה ה-code generator עצמו?

```
(define cg
  (lambda (pe env-size param-size)
    (cond
      ((pe-const? pe) _______________)
      ((pe-var? pe) _______________)
      .
      .
      .
      (else (error ...)))))
```

הסבר:

- הערכים `env-size` ו-`param-size` הם גודל הסביבה ומספר הפרמטרים בנקודה הנוכחית בקוד; אנחנו צריכים את המספרים הללו בטיפול ב-lambda (כתבנו שם שאנחנו יודעים בזמן קומפילציה מהו; דיברנו על זה).
  - שאלה: לא ניתן לרשום את הדברים הללו בזמן ריצה?
  - תשובה: כן, אבל חבל, כי אנחנו יכולים לטפל בזה כבר בזמן קומפילציה.
  - ניתן במקום להשתמש בשני המשתנים המספריים האלה לשמור רשימה של רשימות של שמות משתנים שמדמה את הסביבה, וכן רשימה של שמות משתנים שמדמה את הפרמטרים – זה ייתן לנו את האפשרות לכתוב הערות כגון "אנחנו עכשיו מייצרים קוד עבור המשתנה `x`," מה שיעזור לנו לדבג.
- בהתחלה המימוש של כל ה-"_____" צריכים להיות `"not yet implemented"`.
- בהמשך נשים שם קריאות לפרוצדורות שונות שמטפלות בכל אחד מהמקרים, והם יוציאו פלט כפי שתיארנו בשיעורים הקודמים.
- ובסוף else עם error.
- **הערה חשובה**:

  אל תכתבו cond בלי else, וב-else תמיד שימו error!

  אם לא תשימו else, זה מתכון לקבלת void פתאום באמצע הריצה ולא לדעת מאיפה הוא הגיע.

  ואם תשימו ב-else משהו שהוא לא error, זה יכול להוות בעיה אח"כ כשתעשו שינויים.

- **מאוד מאוד מאוד מומלץ לפתח באופן אינקרמנטלי** (הדרגתי)!

כלומר, לסדר תשתית שתאפשר להריץ בדיקות ולהשוות את התוצאות למה שמצופה, ואז לפתח כל פעם חלק קטן מהפרויקט, להוסיף בדיקות עבורו ולהריץ את כל הבדיקות (גם שבודקות את החלקים הקודמים), ע"מ לוודא שהחלק החדש עובד וכן שלא הרסנו משהו אחר שעבד קודם.

קבועים בסיסיים

- הצעדים הבאים:
  - להתחיל את הקוד שמוציאים בשורה:

```
ADD(IND(0), IMM(1000))
```

זה יקצה לנו 1000 תאים בזיכרון שנוכל לשחק איתם.

  - אח"כ לממש את הקבועים של Scheme (#t, #f, void-object, null):
    - Void:

```
        MOV(IND(1), IMM(T_Void))
#define       SOB_VOID      1
```

טיפוס זה לוקח תא אחד בזיכרון (צריך רק את הטיפוס שלו).

    - Null:

```
        MOV(IND(2), IMM(T_NIL))
#define SOB_NIL             2
```

טיפוס זה לוקח תא אחד בזיכרון (צריך רק את הטיפוס שלו).

    - True:

```
        MOV(IND(3), IMM(T_BOOL))
        MOV(IND(4), IMM(0))
#define SOB_FALSE    3
```

טיפוס זה לוקח שני תאים בזיכרון (אחד לטיפוס שלו ואחד לערך).

    - False:

```
        MOV(IND(5), IMM(T_BOOL))
        MOV(IND(6), IMM(1))
#define SOB_TRUE     5
```

  - במידה ונרצה לשנות את ההגדרות הללו אח"כ, נצטרך לשנות אותן רק בהגדרות הנ"ל.
- עכשיו אנו יודעים איך לממש את const:
  - את כל הקבועים הנ"ל מימשנו בתור כתובת ההתחלה של האובייקטים המתאימים שלהם, לכן בטיפול ב-const נצטרך לבדוק את הטיפוס של האובייקט שיושב בכתובת של הקבוע (שזה המספר שנמצא בזיכרון באותה הכתובת), ובמידת הצורך (אם זה טיפוס T_BOOL) לבדוק גם את הערך שנמצא בכתובת הבאה
  - אם נרצה, למשל, ב-if שאין בו else לשים בתוצאה את ה-void-object, הקוד שיעשה זאת ב-C יהיה:

```
MO(R0, IMM(SOB_VOID))
```

המשך הפיתוח האינקרמנטלי

- תתחילו מלממש את sequence – זה יאפשר לבדוק את `begin`.
- אח"כ תממשו את `if` – זה יאפשר לבדוק גם את `if` וגם את `and`.
- עכשיו בואו נוסיף תמיכה ב-`cons`:
  - נייצר את הקוד:

```
        JUMP(L_Cont_1)

L_Prim_cons:
        PUSH(FP)
        MOV(FP, SP)
        PUSH(FPARG(3))        // The cdr
        PUSH(FPARG(2))        // The car
        CALL(make_sob_pair)
        DROP(2)
        POP(FP)
        RETURN

L_Cont_1:
```

כך יצרנו פונקציה `L_Prim_cons` (שלא מתבצעת עדיין לעולם, כי אף אחד לא קורא לה), שמייצרת pair.

- ניתן להוסיף בדיקה גם שמספר הפרמטרים במחסנית הוא 2 כדי לוודא תקינות.

אבל עדיין אין לנו את הפרוצדורה "cons" ממש, כי חסרים דברים נוספים של הטיפוס (מזהה של "closure" וסביבה).
לכן נמשיך את הקוד:

```
L_Cont_1:
        MOV(IND(10), IMM(T_CLOS))
        MOV(IND(11), IMM(your_id_number))
        MOV(IND(12), LABEL(L_prim_cons))
#define        prim_cons      10
```

כעת, כשנכתוב את השם `prim_cons` במקום כלשהו, נקבל את הכתובת של פרוצדורה פרימיטיבית שמממשת את הלוגיקה של `cons` – וזוהי בדיוק הפרוצדורה `cons`.
כעת, ע"מ לטפל בקוד במשתנה חופשי בשם `cons` (עוד לא דיברנו על איך מטפלים במשתנים חופשיים), נוסיף פרוצדורה לטיפול במשתנים חופשיים שתכיל

בדיקה (cond גדול כזה), שבין היתר יבדוק אם שם המשתנה החופשי הוא cond, ואם כן, מחזירה את prim_cons.

- כעת, כשנקמפל קוד ה-Scheme הבא:

```
(cons #f (cons #t '()))
```

נקבל באמת את התוצאה:

```
(#f #t)
```

- עכשיו נוסיף תמיכה ב-lambda-simple וב-applic ונקבל קומפיילר שהוא Turing-complete שיכול כבר לעשות המון דברים.
  - המרצה יפרסם בדיקות שמממשות עצרת ואת פונקציית Ackermann (בלי מספרים, בעזרת closures בלבד).
- אז נממש את tc-applic, את lambda-var ואת lambda-opt.

קבועים מספריים וכו'

- על קבועים להתנהג כמו ב-Scheme רגיל – היזכרו בדוגמה שראינו עם ה-"he said ha ha"; מה שאנו עושים צריך להתנהג באותה צורה.
- אם כתוצאה מחישוב אנו מקבלים את הקבוע 4 (ה-scheme object הקבוע, לא סתם תוצאה בקוד C של חיבור בזמן כלשהו), ובפעם אחרת שוב מקבלים אותו – בשני המקרים הקבוע צריך לשבת **באותה כתובת בזיכרון**, כלומר ש**יש רק מופע אחד שלו**.
  - כנ"ל לגבי מחרוזות ורשימות קבועות.
- זה בעייתי כי ב-Scheme קבועים יכולים להכיל קבועים אחרים:
  - ניקח את הרשימה הקבועה (שמכילה symbols קבועים):

```
'(a b (a b))
```

כמה קבועים יש פה?

- הרשימה הריקה מופיעה פעמיים – בסוף הרשימה הגדולה וכן בסוף הרשימה הקטנה
  - למעשה עלינו למיין את הקבועים טופולוגית [[בגרף בו הקדקודים הם קבועים והקשתות אומרות " $x \to y$ כאשר x מופיע בתוך y "]].

- להלן הפונקציה שמטפלת בקבועים:

```
(define foo
  (lambda (e)
    (cond
      ((or (number? e) (string? e) (void? e) (null? e) (bool? e)) `(,e))
      ((pair? e)
       `(,e ,@(foo (car e)) ,@(foo (cdr e))))
       ((vector? e)
        `(,e ,@(apply append
                      (map foo
                           (vector->list e)))))
       ((symbol? e)
        `(,e ,@(foo (symbol->string e))))
       )))
```

<u>הערות</u>:

- בטיפול ב-symbol יכולנו במקום:

```
,@(foo (symbol->string e))
```

לכתוב ישירות את המחרוזת (בלי קריאה רקורסיבית ל-foo), אבל בצורה שבה רשמנו כאן הטיפול במחרוזות מתבצע במקום אחד בלבד.
זה עדיף כי אם נצטרך לשנות את הטיפול במחרוזת, נצטרך לשנות רק במקום אחד, ואז אין סכנה שנשכח לשנות אותו במקומות נוספים.

- ניתן להשמיט את החלק המסומן, ולהחליט שתמיד הקבועים הבסיסיים יופיעו כבר בטבלת הקבועים (עליה נדבר עוד רגע).

- כעת אנו יכולים לבנות את **טבלת הקבועים**:

```
`((1 #<void> (,t_void))
  (2 () (,t_nil))
  (3 #f (,t_bool 0))
  (5 #t (,t_bool 1))
  (7 "b" (,t string 1 98))
  (10 "abc" (,t tring 3 97 98 99))
  (15 ("abc") (,t pair 10 2))
  ...
  )
```

<u>הסברים</u>:

- זו טבלה בה העמודה הראשונה מכילה את כתובת הקבוע בזיכרון, העמודה השנייה מכילה את הקבוע עצמו והעמודה השלישית מכילה את הערכים המספריים שמייצגים את הקבוע הזה בזיכרון של התוכנה שמקמפלים.
- ניתן לדעת איפה יישב קבוע חדש שמוסיפים בכך שלוקחים את כתובת הקבוע האחרון ולהוסיף לו את אורך הרשימה שמופיעה אצלו בעמודה השלישית.
- כל קבוע מכיל בעמודה השלישית שלו את המידע שצריך בשבילו:
  - כולם מכילים בהתחלה את קוד הטיפוס שלהם.
  - הקבועים הבוליאנים מכילים גם את הערך שלהם.
  - מחרוזות מכילות גם את אורך המחרוזת ואז רשימה של הערכים המספריים של כל אחד מהתווים במחרוזת, לפי הסדר.
  - זוגות מכילים את כתובת הקבוע הראשון בזוג ואת כתובת הקבוע השני בזוג: למשל בדוגמה הנ"ל, הזוג מכיל את הקבוע `t_pair` שאומר שהוא זוג, את הכתובת 10 שהיא הכתובת של המחרוזת `"abc"` ואת הכתובת 2, שהיא הכתובת של הרשימה הריקה, `()`.
- ניתן למחוק את כל הפסיקים לפני הטיפוסים ברשימה:

```
`((1 #<void> (t_void))
  (2 () (t_nil))
  (3 #f (t_bool 0))
  (5 #t (t_bool 1))
  (7 "b" (t_string 1 98))
  (10 "abc" (t_tring 3 97 98 99))
  (15 ("abc") (t_pair 10 2))
  ...
  )
```

ואז מקבלים במקום את הקודים המספריים של הטיפוסים את שמות הטיפוסים, מה שיכול מאוד לעזור ב-debug – רק שימו לב שבסוף לשם יצירת הזיכרון **כן צריך את הפסיקים האלה**.

- ע"מ לחפש קבוע ברשימה, מחפשים לפי העמודה השנייה (היא המפתח).
- ע"מ לקבל את העמודה השלישית ניתן לבצע על הרשימה הנ"ל map עם caddr.
- אז ניתן לקחת את העמודה השלישית הזו, ולהפעיל עליה apply עם הפרוצדורה append, ומקבלים בדיוק את תמונת הזיכרון (זו פשוט רשימה ארוכה של מספרים) החל מכתובת ההתחלה.
- ברגע שיש לנו את הרשימה, מייצרים ממנה מחרוזת של מערך של C:

```
long* mem_init = { t_void, t_nil, t_bool, 0, ... };
```

[[למעשה יופיעו שם רק מספרים, לא t_void וכו', אלא אם נמחק את הפסיקים שהזכרנו מקודם.]]
ואותה שמים מיד לאחר ה-prologue, ואחריה לולאה שעוברת על כל המערך ושמה אותו בזיכרון החל מהכתובת ההתחלתית.

- לא חייבים שהקבוע הראשון יופיע בכתובת מס' 1 – שינוי מספר זה יזיז את כל הרשימה, מה שיכול לפנות לנו מקום להוסיף עוד כל מיני דברים.

- אל תתחיל בכתובת 0, כי היא שמורה לשם הדרך שבה התשתית מקצה זיכרון.
- מומלץ להתחיל בכתובת גדולה יותר (נגיד 200), כדי שיהיה מקום להוסיף מידע אדמיניסטרטיבי (שימושי במיוחד ל-debugging).

- אפשר להחליט שתמיד בתחילת הרשימה יופיעו הקבועים הבסיסיים ואז אין צורך לטפל בהם בפונקציה שכתבנו מקודם [[כתבנו שם שאפשר לעשות זאת (איפה שסימנו חלק בקוד בהדגשה)]].