שם הקורס: תורת הקומפילציה. שם המרצה: ד"ר רוני סיון.

דואר אלקטרוני: ronny.sivan@gmail.com

הרכב הקורס: תרגילי בית - 25% (תקף)

מבחן סוף - 75%

ספר הקורס: Compilers - Principles, Techniques & Tools

Addison-Wesley, 2007

ספר נוסף: Compiler Design, Addison-Wesley

מבוא

בשנות ה-50 התעוררה בעיית הקומפיילרים, ולכן נכתבה תיאוריה של תורת הקומפילציה והשנות ה-60 מקאן ביתרון שהוא טוב.

שפת P-code = שפת ביניים שהומצאה כחלק מהמאמץ לבניית קומפיילר לפסקל.

מורכבות

בע"ת הקומפילציה קשה ומורכבת ועוסקת באובייקטים רבים של מדעי המחשב.

למשל: שפות תכנות, מבני נתונים, מערכות הפעלה, שפות פורמליות, אלגוריתמים, הנדסת תוכנה וכדומה.

קומפיילר צריך לדעת מה אינה מעמד הוא עובד, וכך הוא מסוגל להיות יעיל יותר.

כיום, קיימים מעבדים, שמסוגלים לבצע על תלויות והריץ במקביל פקודות שאינן תלויות.

כמו כן, הקומפיילר משמש לתכנון מבנה מחשבים.

ניתן לחבר Test cases (תוכניות בדיקה) על מנת לבדוק תקינות של בדיקות.

## מבוא לשפה

תוכניות שמקבלות כקלט טקסט בשפה מסוימת, ומחזירות פלט בשפה מסוימת.

קיימת משמעות רבה לאה שנכנס ומה שיוצא.
לשפה, אלו תוכניות יקרות, שמעבירות סטלקס או סימנים משפה מסוימת.

אולם, מסופו של דבר המשמעות תישמר.

**תרגום** = התלפה של שפה.

## משמעות

**נשאלות מס' שאלות:**

1) מהי משמעות?
2) עד כמה מחשב מסוגל להבין משמעות?
3) האם נתכנה הנה על מנת לשמר משמעות (בתרגום, למשל)?

## קומפילציה

**זמן קומפילציה:**

| קלט | תוכנית | פלט |
|---|---|---|
| Source Program → | compiler | → target program |

**זמן ריצה:**

input → target Program → output

הקומפיילר הראשון היה אסמבלר.

בשפה הראשונה - Formula Translation (Fortran).

היה ניתן לרשום:

```
I = 5
J = 7
K = I+J
A = 1.02
```

קיים הבדל משמעותי בין זמן הקומפילציה לזמן ריצה!

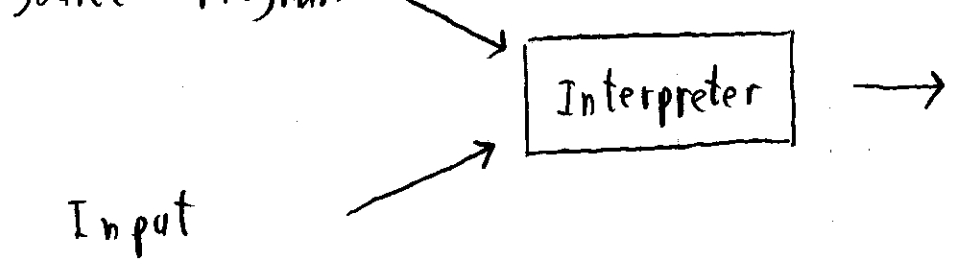
תהליך הקומפילציה נעשה לפני שהתוכנית רצה, ולכן הקומפיילר מייצר תוכנית שלאחר מכן תרוץ בלי סוף המחשב.

שפת תכנות - אינטרפרטיבית שפת מכונה - ספציפית למכונה.

הקומפיילר הינו תוכנית בפני עצמו שיכולה לרוץ על המחשב בו התוכנית תרוץ לאחר מכן, או שלא, כלומר: ניתן להריץ על מחשב קומפיילר המתרגם לתוכנית שתרוץ על מחשב אחר.

אינטרפרטציה (פרשן)

זמן ריצה:




החסרון העיקרי: איטיות!

היתרון: - אינטרפרטציה אינה תלוית מכונה.

**הרצאה:** אין צורך לחדש את האב, ולאחר מכן לקמפל שנית.

קיים מודל מעולה:

**למשל:** Java ו-C#.

Source Program → compiler → byte code
(Main.java) (javac.exe) (Main.class)

virtual machine
(אינטרפרטר)
(java.exe)

למעשה, מריצים קוד ביניים, והאינטרפרטר של המכונה יודע להריץ אותו.

**סוג תוכניות להיות שפה:**

1) **Source to Source** - תרגום לישיר.

C++ Program → translator → C Program

2) **Virtual Machine** - מכונה וירטואלית

Java Program → Compiler → Java byte code

3) **Pre-processors**

Program with embedded pre-processing code (#include, #define) → Pre Processor → "pure" program

אלו למעשה פקודות, שאינן נראות בתוכנית, אלא רק בהתחלה לקומפיילר.

למשל: בגוף הפקודה #include <stdio.h>

תורגם ל"הוראות" הספריה כולה במקום הפקודה.

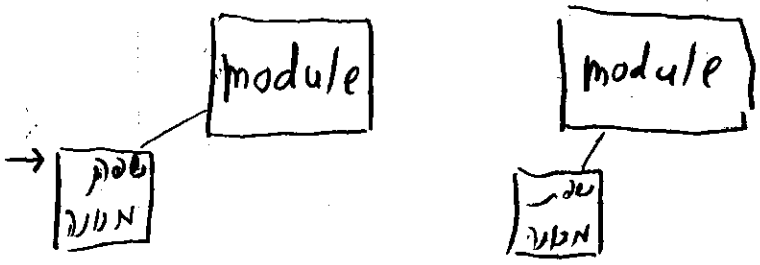
## כלים נוספים לשיפור שפה

קיימים שירותים נוספים להם עזרת הקומפיילר.

| | |
|---|---|
| Preprocessor | איסוף מודולים, פתיחת מקרואים |
| ↓ | |
| Source Program | בתרגום בחסיות, הפניות לספריה |
| ↓ | |
| Compiler | |
| ↓ | |
| Target Assembly Program | בתרגום בחסיות, לכל התוכנית: Link |
| ↓ | בתרגום אוטומטי: Load |
| Assembler | |
| ↓ | |
| Realocateable machine code | |
| ↓ | |
| Loader / Link-editor | |
| ↓ | |
| Absolute machine code | |

**Runtime Library:**

הקומפיילר מתיחס לפונקציות, שאין תוכן מצומן ידוע, אולם אינן תוכן ממה שמתקמפל.




הקומפיילר לוקח את המודולים בלשון ומתרגם אותם למודול אחד גדול.

**תפקיד הLoader**: לארגן את כל התוכנית.

**Dynamic Loading Library** = זיכרון נטענות בזמן ההרצה, ולא בזמן הריצה, את נרדוש, אלו שטעונות. (DLL).

התרגום של קומפל נלס הוא בינארי, ומיודר ע"י הקומפיילר.

**פונקציה לשיפור:**

**מקור:** $a*X**2 + b*x + c$ $(ax^2+bx+c)$

**נדרש:** $(a*X+b)*X + c$ כי נקבל פחות פעולות כפל.

# מהנדס הקומפיילר

קומפיילר הוא מתרגם קבצים שתורה.

```
int a,b;
a=2;
b= a*2+1;
```

קוד מקור

→

למשל:

```
SET R1,2
STORE #0(a), R1
SHIFT R1,1
STORE #1(בינים), R1
ADD R1,1
STORE #2(b), R1
```

קוד מטרה

שימוש בריגיסטר אחד

ניתן להוריד, אולם אולי יעשה שימוש בזיכרון זה לאחר מכן.

## מלאכת קומפיילר התרגום

תוכנית מקור

↓

1) אנליזה (analysis)

↓

יצוג בינים.

↓

2) קוד שניתן לשנות לאופטימיזציה (code optimization) תהליך

↓

יצוג בינים

↓

3) יצור קוד המטרה. (code generation)

↓

תוכנית מטרה

(חלק שני) - Front End

(חלק פלטפורמה) - Back End

→ ניתן לשנות אופטימיזציה שהיא תלויה בפלטפורמה עדיה הולכים לעבוד

## תהליך הקומפילציה

1) Lexical Analyzer = מתבוננים על האותיות,

למשל: F U N C T I O N → Function

2) Syntax Analyzer = האם התוכנית בתחביר סינטקטי נכון.

למשל: חסר ה"נקודה-פסיק".

3) Symantic Analyzer = בדיקת משמעות.

למשל: יהיו הטיפות הבאה: int a; a=1.7;

בעיה: (ב-Symantic Analyzer) מחיקת עד לרגע ה-";", כאמור סוף הפקודה, ואז בודקים.

מופיעים "קישוטים" על ומקבלים:

4) Intermediate Code Generator = קוד ביניים, שניתן לתרגמו תיכף לקוד המכונה.

5) Machine Independent Code Optimization = קוד שאינו תלוי מכונה.

- Lexical Analyzer - קל, טיפול בביטויים רגולריים.
- Syntax Analyzer - יותר מסובך, רקורסיבי, סבוכה עם הקוד תסר הקשר.

## דוגמה

position = initial + rate * 60

↓

**Lexical Analyzer**

↓

<id,1> <=> <id,2> <+> <id,3> <*> <60>

טבלת סימנים

| | | |
|---|---|---|
| 1 | position | .... |
| 2 | initial | ... |
| 3 | rate | ... |

**הערה:** id יכול להיות מחרוזת הבודקת מילה שמורה ובדומה.

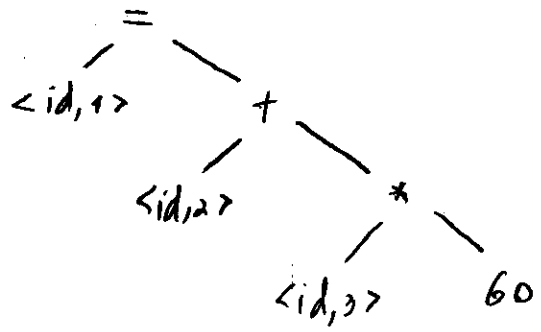
**token** = מה שנמצא בתוך < >.

**Symbol table** = טבלת סמלים המחזיקה את הערך של הקומפילציה. כל id הוא למעשה כניסה לטבלה המימוש נעשה באמצעות Hash Table.

**Syntax Analyzer**

בעל מחסנית, ומקבל מבנה הקומפילציה. (עד הנקודה-פסיק)

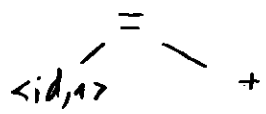
↓




↓

**Symantic Analyzer**

↓



**Intermediate Code Generator**

↓

```
t1 = intTofloat <60>
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

↓

**Code Optimizer**

-האופטימיזר יכול לשנות את סדר הפקודות, למשל- הוצאת פקודה מתוך הלולאה.

↓

```
t1 = id3 * 60.0
id1 = id2 + t1
```

↓

**Code Generator**

↓

```
LDF   R2, id3
MULF  R2, R2, #60.0
LDF   R1, id2
ADDF  R1, R1, R2
STF   id1, R1
```

<u>הערה:</u> הקומפיילר מתייחס אמת לדרגים כרגיסטרים, אולם ייתכן הציבורון שהרגיסטרים הללו מייצגים אזור בזכרון מתעדכן באמצעות הריצה, אבל כמו. למשל- מנה בלולאה.

לשם כך, קיימת ההגדרה volatile על מנת שכן יעדכן הקומפיילר על שינוי בציבורון.

## טבלת הסמלים

מבנה נתונים לכל identifier. שומר את התכונות של (attributes) ומאפשר גישה מהירה.

לדוגמא: מיקום בזיכרון, טיפוס, scope.

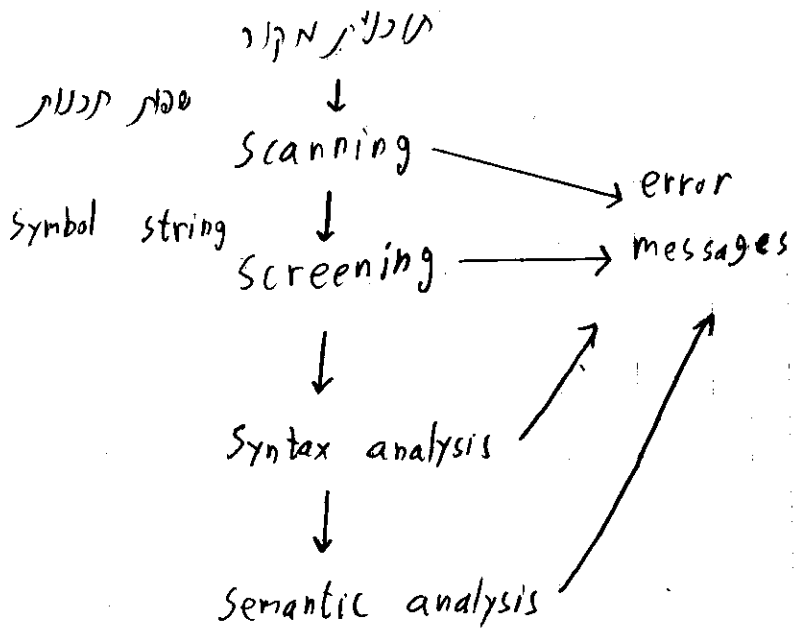
## הודעות שגיאה

מלק תשובה מתהליך הקומפילציה. כל שלב מזהה שגיאות קרמת הניתוח שלו.

**ניתוח תשובה של מלבולרית** = אפשרות לייבר קומפיילרים לפלטפורמות שונות והכנסת תכונות יחודיות שמטפלות בבעיות "חדשים".

**למשל:** תוכנית המייצרת את ה Lexical Analyzer.

# Front end - ניתוח לקסיקלי




– מצוי שפת פיתוחים.

Lexical Analysis תהליכים המבצעים את ה- { Scanning, Screening }

## מדוע מפרידים את הניתוח הלקסיקלי?

1) פישוט הניתוח הלקסיקלי.

למשל: אי יכולת בהגדרת תחילית שפה

begin...end אז { ... }

2) פישוט הניתוח הסינטקטי.

3) מודולריות.

4) שימוש חוזר

5) יעילות – ה הפרדה מקלה למצוא אלגוריתמים יעילים יעילים

# מבוא א"ק

למשל: עבור סימן: 1,2,3,4,5,6,7,8,9, ., x, #

בתרגיל: program, fcount

מבחר אות

**סימן (symbol או lexeme)** = אות או סדרת אותיות המופרדת משאר התרגיל באופן מוסכם (למשל ע"י רווח

**אסימון (token)** = זוג סדר של שם ותכונותיו.

למשל: (identifier, {name=counter, code_location=3})

**תבנית (pattern)** = כלל המגדיר אוסף מחרוזות

## תפקידי ה- scanner

1) **קריאת הקלט**

2) **הפרדת הקלט לסימנים (לקסמות)**

**דוגמאות:** string, boolean, real

3) **לפעמים ה- scanner משתמש גם ב- preprocessor**

4) **ספירת השורות בקלט** - לצורך הודעות שגיאות.

ניתן לבקש מהקומפיילר לעבוד במצב debug ואז הוא יכול לזהות את מיקום השגיאה ע"י הוספת תגית סימנים.

## ג) Scraper

מקבלת מחרוזת תווים
בלי רווחים וכו'.

## Typical Tokens

- Token אחד לכל מילה שמורה משפה, למשל: if, else
- Token אחד לכל אופרטור.

למשל: ! , == , = , <= , < : השוואות

- Token אחד לכל סימן.

משפחות של סמלים (patterns)

| symbol class | תבניות המשפחות |
|---|---|
| ID | foo n_14 last |
| NUM | 73 00 517 |
| IF | if |
| COMMA | , |
| NOT EQ | != |
| L PAREN | ( |
| R PAREN | ) |
| comment | /* ignored */ |
| macro | #define NUM 516 |

**הערה:** נוהגים לקרוא - מספר שלם "מסולקים" מהקלט.

**עמוד הבעיה:** e = m * c ** 2

עמוד ה- tokens הבאים:

| | | |
|---|---|---|
| * | <ID,1> | "e" |
| * | <ASSIGNMENT> | "=" |
| * | <ID,2> | "m" |
| * | <MULTIPLY> | "*" |
| * | <ID,3> | "c" |
| * | <POWER> | "**" |
| * | <NUMBER,4> | "2" |

**טבלת הסמלים**

| שם | סוג | מספר |
|---|---|---|
| e | dbl | 1 |
| m | dbl | 2 |
| c | dbl | 3 |
| 2 | int | 4 |

- **משפט פסקל:** pi = 3.1416; real

tokens-5

- **ב- Fortran הרווחים אינם נחשבים:**

DO 5 I = 1.25 ⇐ (DO5I = 1.25)

tokens-3

שדה שדה משתנה

**אם:**

DO 5 I = 1 , 5

**Lookahead** = הכורח לקרוא סימנים נוספים מהקלט לפני שניתן להחליט מהו סוג סימן.

# אלפבית, מילים, שפות

**אלפבית**, סימון: $\Sigma$, לדוגמה: $\{0,1\}$

**מילה** = מחרוזת סופית מהאל"ב.

**שפה** - יחס קבוצה על $\Sigma^*$, $L \subseteq \Sigma^*$

$\Lambda = \{A, B, C, \dots, a, b, \dots, z, -\}$

$\Delta = \{0, 1, \dots, 9\}$

$(\Delta \cup \Lambda)^* \Lambda$ - שמות חוקיים של משתנים בשפה C.

באמצעות ביטויים רגולריים, ניתן להגדיר קבוצות

**ביטוי רגולרי** = יצוג לשפות רגולריות.

**תרגיל:**

תהי $\Sigma = \{a, b\}$

$L(a|b) = \{a, b\}$

$L((a|b)(a|b)) = \{aa, ab, ba, bb\}$

$L((a|b)^*) = \{\varepsilon, a, b, aa, ab, aaa, \dots, aab, \dots\}$

**הערות:**

$(s|r) \equiv (r|s)$ $\qquad$ $rs \not\equiv sr$

$r|(s|t) = (r|s)|t$ $\qquad$ $r(s|t) = rs|rt$

$r|(st) \equiv \{r, st\}$

binary_number = (0|1)$^+$

decimal = digit$^*$ . ? digit$^+$

? - עוד אופן אחד לכל היותר. 0 או 1.

ה- scanner יכול להשתמש בהגדרות הבאות:

digit = 0-9

digits = digit$^+$

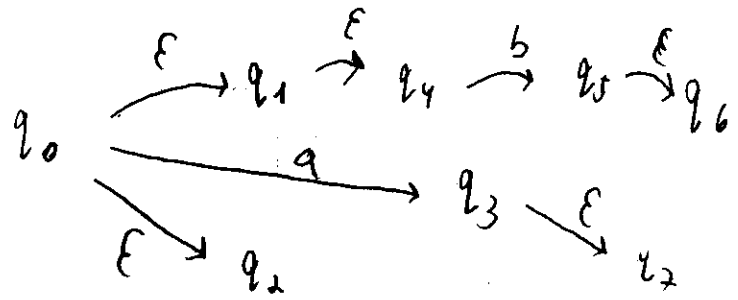
letter = a-z | A-Z

then = then

relop = < | > | <= | >= | = | <>

הגדרת אוטומט זה דטרמיניסטי עם מעברי ε לאוטומט דטרמיניסטי
ללא מעברי אפסילון.

NDFA




$Q \ni$ ε-closure$(q_0)$,

כל עוד יש מצבים לא מסומנים בדגל:

יהי T מצב לא מסומן. מסמנים T.

- לכל סימן באלפבית a.

U = move(T, a)

אם $u \notin Q$

$\delta(T,a)=u;\quad Q=Q \cup \{u\};$

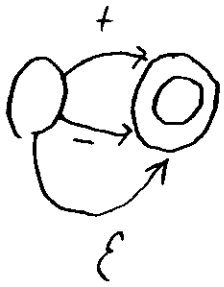
תרגיל בית 1

יש לבנות את הביטוי:

$$\underbrace{(+|-)?}\ digit^{*}\ (.\ digit^{+})?\ (e(+|-)?\ digit^{+})?$$

להפוך אותו לאוטומט סופי דטרמיניסטי,

ולאחר מכן לאוטומט מינימלי דטרמיניסטי.



# תרגיל כתה

1. תארו את השפות הנתונות באמצעות הביטויים הרגולריים הבאים:

א. a(a|b)*a

ב. ((ε|a)b*)*

ג. (a|b)*a(a|b)(a|b)

ד. a*ba*ba*ba*

ה. (aa|bb)*((ab|ba)(aa|bb)*(ab|ba)(aa|bb)*)*

א) $a(a|b)^*a = \{aa, aaa, aba, aaaa, aaba, abaa, abba, \dots\}$

כל המילים שמתחילות ב-a ומסתיימות ב-a.

ב) $((\varepsilon|a)b^*)^* = \{\varepsilon, b, a, bb, aa, ba, \dots\} = (a|b)^*$

כל הצירופים של a ו-b האפשריים

ג) $(a|b)^*a(a|b)(a|b) = \{aaa, abb, aba, aab, \dots$

כל המילים שהסיפא של ה-3 תווים האחרונים שלהם מתחילה ב-a.

ד) $a^*ba^*ba^*ba^* =$ כל המילים המכילות 3 b-ים בדיוק.

ה) $(aa|bb)^*((ab|ba)\cdot(aa|bb)^*\cdot(ab|ba)\cdot(aa|bb)^*)^*$

כל המילים באורך זוגי. אין כלל ab ו-ba,

ולא ab

$\{\varepsilon, aa, bb,$

בזוגות: ab ab

ba ba

# תרגיל 1 – אוטומט לזהוי מספר

ראינו בהרצאה את הביטוי הרגולרי המגדיר את השפה הכוללת את כל המספרים השלמים והשבורים בהצגתם המקובלת בשיח של מחשבים: למשל, 6.022e23 (מספר אבוגדרו), או 1.602e-19 (אלקטרון-וולט).

הבטוי הרגולרי שמתאר מספרים מסוג זה הוא:

$$(+|-)?\ digit^*\ (.\ digit^+)?\ (e(+|-)?\ digit^+)?$$

כאשר ההגדרה של digit היא הברורה: 9|8|7|6|5|4|3|2|1|0

הפכו את הביטוי הרגולרי הנ"ל לאוטומט דטרמיניסטי סופי. זאת תשיגו ע"י בנית אוטומט אי-דטרמיניסטי מהביטוי עצמו והפיכתו לאוטומט דטרמיניסטי. על התוצאה להכיל את המרכיבים הבאים:

1. האלפבית
2. קבוצת המצבים
3. המצב ההתחלתי
4. קבוצת המצבים המקבלים
5. פונקצית המעבר (הדטרמיניסטית)

יהיה יפה אם תצליחו ליצור מצבים מקבלים נפרדים לקטגוריות הבאות:

- מספר שלם
- מספר עשרוני
- מספר עשרוני עם חזקה

בהצלחה!