עבודת גמר

לקבלת תואר טכנאי תוכנה

תמונה שמכילה אומנות קליפיפם, גרפיקה, לוגו, סמל

התיאור נוצר באופן אוטומטי

נושא: The Easy Compiler

מהדר לשפת Easy

המגיש: אביב אש

ת.ז המגיש: 214887556

שם המנחה: מיכאל צ'רנובילסקי

אפריל 2025 תשפ"ה

# תוכן עניינים

[תוכן עניינים 2](#_Toc194622096)

[תקציר 4](#_Toc194622097)

[מושגים 5](#_Toc194622098)

[מושגים כללים 5](#_Toc194622099)

[תיאור הפרויקט 6](#_Toc194622100)

[תכולת השפה 6](#_Toc194622101)

[תיאור אבני השפה 6](#_Toc194622102)

[אופרטורים בינארים: 6](#_Toc194622103)

[משתנים בסיסיים: 7](#_Toc194622104)

[השמת נתונים 8](#_Toc194622105)

[השמה מורכבת 8](#_Toc194622106)

[השוואות 9](#_Toc194622107)

[תנאים לוגים 9](#_Toc194622108)

[תנאים 10](#_Toc194622109)

[פונקציות 11](#_Toc194622110)

[קריאה לפונקציה 11](#_Toc194622111)

[מצביעים 12](#_Toc194622112)

[תוכנית לדוגמא בשפת easy 13](#_Toc194622113)

[דקדוק השפה 14](#_Toc194622114)

[רקע תאורטי בתחום הפרויקט 18](#_Toc194622115)

[קומפיילר/מהדר – הגדרה כללית: 18](#_Toc194622116)

[סוגים שונים של מהדרים: 20](#_Toc194622117)

[סוגים שונים של שפות תכנות: 20](#_Toc194622118)

[שלבי הקומפילציה: 20](#_Toc194622119)

[Front end: 20](#_Toc194622120)

[ניתוח לקסיקלי(Lexical Analysis) 20](#_Toc194622121)

[ניתוח תחבירי (Syntax Analysis) 21](#_Toc194622122)

[דקדוק חסר הקשר (Context free grammar) 21](#_Toc194622123)

[**סוגי מנתחים תחביריים** 23](#_Toc194622124)

[מלמעלה למטה (**top - down**) 23](#_Toc194622125)

[**מנתחי bottom-up** 24](#_Toc194622126)

[מלמטה למעלה (**bottom - up)** 24](#_Toc194622127)

[ניתוח סמנטי (Semantic Analysis) – 25](#_Toc194622128)

[Back end- 25](#_Toc194622129)

[אופטימיזציה – 25](#_Toc194622130)

[יצירת קוד הסף – 25](#_Toc194622131)

[טיפול בשגיאות- 25](#_Toc194622132)

[תיאור הבעיה האלגוריתמית 26](#_Toc194622133)

[אסטרטגיה נבחרת לפתרון 27](#_Toc194622134)

[האסטרטגיה לפיתוח המנתח הסמנטי 27](#_Toc194622135)

[**האוטומט של tec** 28](#_Toc194622136)

[האסטרטגיה לפיתוח המנתח התחבירי 31](#_Toc194622137)

[**הפעולות האפשריות** 31](#_Toc194622138)

[האסטרטגיה לפיתוח המנתח הסמנטי 34](#_Toc194622139)

[האסטרטגיה לפיתוח שלב כתיבת קוד הסף 34](#_Toc194622140)

[ארכיטקטורה של הפתרון המוצע בפורמט של Top-Down Level Design 37](#_Toc194622141)

[תרשים Top-down מפורט 38](#_Toc194622142)

[מבני נתונים 39](#_Toc194622143)

[מטריצה 39](#_Toc194622144)

[מפת hash 40](#_Toc194622145)

[מחסנית 40](#_Toc194622146)

[עץ 41](#_Toc194622147)

[מערך 41](#_Toc194622148)

[תיאור סביבת העבודה ושפות התכנות 42](#_Toc194622149)

# תקציר

בחרתי לעשות את פרויקט הגמר שלי בנושא פיתוח מהדר (compiler)

כבר בתחילת לימודיי במדעי המחשב מצאתי את המהדר כרכיב מסתורי וקסום, איך זה ייתכן שאני כותב קוד באנגלית והמחשב יודע לפרש זאת ולהריץ אותו, זאת אחת הסיבות שבגללן בחרתי את נושא זה כי איזו דרך יותר טובה להבין משהו מללכלך את הידיים ולעשות אותו בעצמך.

במהלך הפרויקט חקרתי מספר כלי הקומפילציה ותרגום (אינטרפטציה) שונים ביניהם (**cpython**,**gcc**) בשביל להבין יותר על הנושא וקריאה יותר עם הנושאים האלו עניינה אותי מאוד ועזרה לי בכתיבת והבנת הפרויקט

הפרויקט פיתח אותי מאוד אני מרגיש שהשתפרתי מאוד כמתכנת. למדתי לכתוב קוד יותר קריא ודינמי וזהו יתרון גדול. במהלך הכתיבה הבנתי שהרבה פעמים ארצה לשנות או להוסיף פונקציונליות ברכיב בקוד שלי ואם הקוד לא כתוב כראוי. למדתי שפת תכנות חדשה, החלטתי לכתוב את הפרויקט שלי בשפת c++ בשל גמישות השפה והיותה גם מונחת עצמים וגם זמן הריצה המהיר שלה וגישה לזיכרון, תהליך למידת השפה היה משמעותי ומקדם עבורי.

וזהו הפרויקט הגדול והמורכב ביותר שלי עד כה, פגשתי אתגרים רבים ולמדתי המון.

לפני שאנחנו קופצים למים נשאל את השאלה – **מהי שפה?**

**שפה פורמלית** (במדעי המחשב ובתורת האוטומטים) מוגדרת כקבוצה של L מחרוזות (strings) מעל אלפבית נתון. Σ כלומר:

L  ⊆ ∗ Σ

**ומה זה קומפיילר?** – תוכנה אשר לוקחת שפה ומתרגמת אותה לקוד אשר רץ על מחשב. זה הכל!

מקווה שתהנו 😁

# מושגים

בפרק זה אציג את המושגים השונים הקשורים ביצירת מהדר

## מושגים כללים

**מהדר** (compiler) - רכיב תוכנה שמטרתו לתרגם קוד מקור (Source Code) לשפת יעד, כמו קובץ הרצה (Executable) או שפת ביניים (Intermediate Representation - IR). מהדרים נפוצים: gcc, clang

**אסימון** (token) – אסימון הוא חלק בעל משמעות בקוד, לכל אסימון יש סוג ושם. לדוגמא בשורת הקוד while(x) נמצאים שלושה אסימונים:

**while** : keyWord] , [**(**:openParen] , [**x**: identifier] ,[**(**,closedParen] ]

**עץ ניתוח תחבירי** (AST) –עץ המורכב מאסימונים המתאר את המבנה הסמנטי של התוכנית

**טבלת הסימנים** (Symbol table) - מבנה נתונים שמנהל מידע על משתנים, פונקציות, וטיפוסים בקוד.

**דקדוק חופשי-הקשר** (CFG) הינו חלק מענף במתמטיקה הנקרא הוא חלק מענף המתמטיקה שנקרא **תורת השפות הפורמליות** (Formal language Theory) בעזרת דקדוק זה נבטא את המבנה של התוכנית (הסבר מפורט יותר בחלק **הרקע התאורטי**)

# תיאור הפרויקט

## תכולת השפה

השפה easy שפה יחסית מינימליסטית והיא נבנתה כשמה, היא אמורה להיות קלה ללמידה ואינטואיטיבית לאנשים עם רקע בתכנות ולוקחת הרבה השראה מC למרות שהיא כוללת גם מאפיינים משפות אחרות כמו **JS**ו **Java**

היא שפה **פרוצדורלית** ולכן מכילה את המבני השפה הנחוצים לה

השפה מבנים בסיסיים לולאות, תנאים, והגדרת ביטויים מתמטיים אך היא מכילה מבנים נוספים כגון: פונקציות, מצביעים ומערכים

## תיאור אבני השפה

### אופרטורים בינארים:

כמו בהרבה שפות גם easy מכילה מגוון אופרטורים בינארים

אופרטור בינארי הינו אופרטור המפעיל פעולה בין שני ביטויים

**<ביטוי 1> <אופרטור> <ביטוי 2>**

**האופרטורים:**

+ **חיבור**

- **חיסור**

\* **כפל**

/**חילוק**

~ פעולת ביטים **not**

| פעולת ביטים **or**

& פעולת ביטים **and**

^ פעולת ביטים **xor**

### משתנים בסיסיים:

השפה כוללת מספר סוגי משתנים בסיסיים:**int**, **char**, **float**

הגדרת משתנה תתבצע בצורה הבאה (בדומה לC):

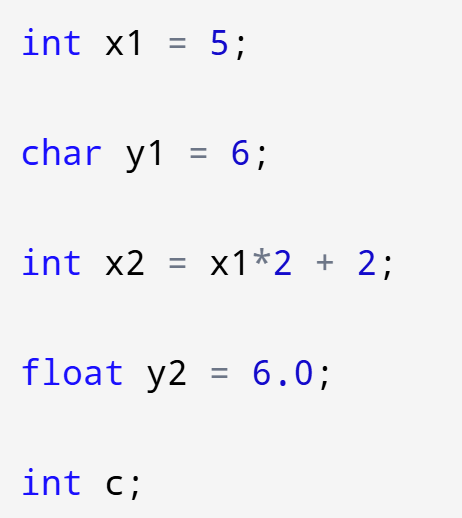
<**ערך אתחול ראשוני**> = <**שם המשתנה**> <**סוג המשתנה**>

**או:**

;<**שם המשתנה**> <**סוג המשתנה**>

יש לשים לבכי בדרך האתחול השנייה המשתנה יאותחל לערך "זבל" ויכלול את תכולת הזכרון הקודם במקום שהוא תפס מבלי לדרוס אותו.

דוגמאות לאתחול נתונים בשפה

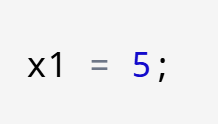


### השמת נתונים

ההשמה מתבצעת בדומה לאתחול רק ללא שם המשתנה בתחילת ההשמה

צד ימין של ההשמה לא יכול להיות ריק

<**ערך אתחול ראשוני**> = <**שם המשתנה**>



דוגמה להשמה תקנית:

תמונה שמכילה גופן, לבן, גרפיקה, עיצוב

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

דוגמה להשמה לא תקנית:

### השוואות

השוואה מחזירה אחד משתי ערכים **0** (שקר) או **1** (אמת)

השוואה מתבצעת בעזרת האופרטורים **==** (שווה) או **=!** (לא שווה)

דוגמא:



תחזיר **שקר** כי **0 הוא לא 3!**

### תנאים לוגים

בין שני ביטויים לניתן לשים אופרטור לוגי בצורה הבאה

**<ביטוי 1> && <ביטוי 2>**

**או**

**<ביטוי 1> || <ביטוי 2>**

&& - מסמן **"ו-" לוגי** והוא יחזיר 1 (אמת) רק **אם שתי הביטויים שונים מאפס** (אמת)

&& - מסמן **"או" לוגי** והוא יחזיר 1 (אמת) **אם אחד או יותר משתי הביטויים שונים מאפס** (אמת)

תמונה שמכילה גופן, טקסט, קו, גרפיקה

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.לדוגמא:

יחזיר האם המספר מתחלק בשלוש ללא שארית **וגם** מתחלק בשתיים ללא שארים

### תנאים

תנאים בשפה יכתבו בצורה הבאה:

**if(<Expression>) <body>**

**אפשר להוסיף**

**Else <body>**

כאשר תוצאת הביטוי מניבה מספר

**<Body>** הינו או ביטוי אחר או מספר ביטויים בתוך סוגריים מסולסלות

**אם המספר לא שווה אפס** התוכנית תמשיך למה שקיים בתוך גוף התנאי

**כשערכו של המספר הוא אפס** התוכנית לא תכנס לגוף התנאי ותכנס לגוף **הElse** (אם יש)

לדוגמא:



עם **x** שונה מ-0 ערכו ישתנה ל5 אחרת - ישאר אותו הדבר

## פונקציות

**מהי פונקציה?** פונקציה היא מבנה בשפת תכנות המאפשר לבצע קטע קוד מסוים באופן חוזר תוך מתן אפשרות להעברת פרמטרים והחזרת ערך.

פונקציות בשפה easy נכתבות כך –

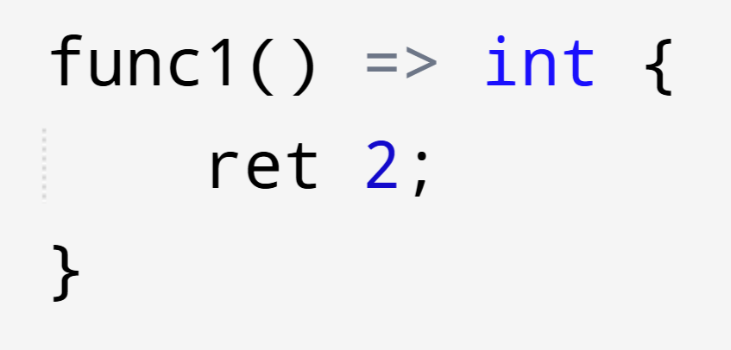
**{<גוף הפונקצייה>} <סוג משתנה> <= (<רשימת פרמטרים>)<שם הפונקצייה>**

גוף הפונקצייה הינו אפס או יותר ביטויים אשר אחד מהביטויים הינו **ביטוי** **ret**

השפה מכילה את מילת המפתח - **ret**כאשר התוכנית מגיעה אל המילה הזאת היא תצא ותחזיר את הערך אשר נמצא אחריה

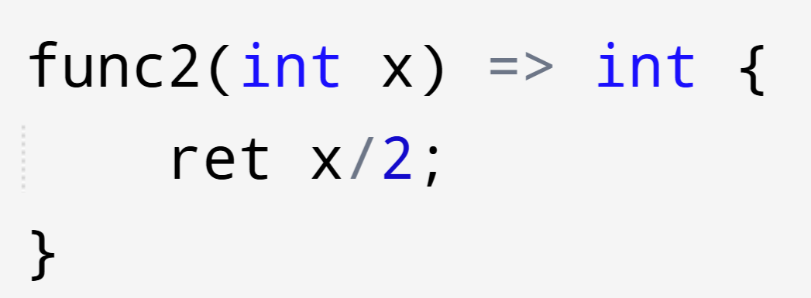
**דוגמאות לפונקציות:**

**דוגמא 1:**

פונקציה שלא מקבלת כלום ומחזירה את הערך 2

**דוגמא 2:**

פונקציה המקבלת מספר ומחזירה את החצי שלו



### קריאה לפונקציה

קריאה לפונקציה תראה כך

**(<פרמטרים>)<שם הפונקציה>**

תמונה שמכילה גופן, גרפיקה, לוגו, טקסט

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.**לדוגמא:**

הפונקצייה func2 תקרא עם הערך **6** ותחזיר את הערך **2**

### מצביעים

עוד בונוס קטן לשפה, **מצביעים**.

בשביל שמירה על הפשטות בשפה easy המצביעים מגיעים עד לרמה אחת של ריחוק

**השפה לא מאפשרת מצביע למצביע** או כל מבציע ברמה גבוהה יותר מהצבעה לערך

מצביעים כתובים כך:

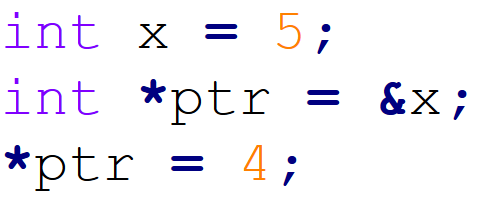
**הגדרת מצביע:**

**< ערך אתחול ראשוני > = <סוג משתנה> \* <שם משתנה>**

**גזירת מצביעה (dereferencing)**

**<שם מצביע>&**

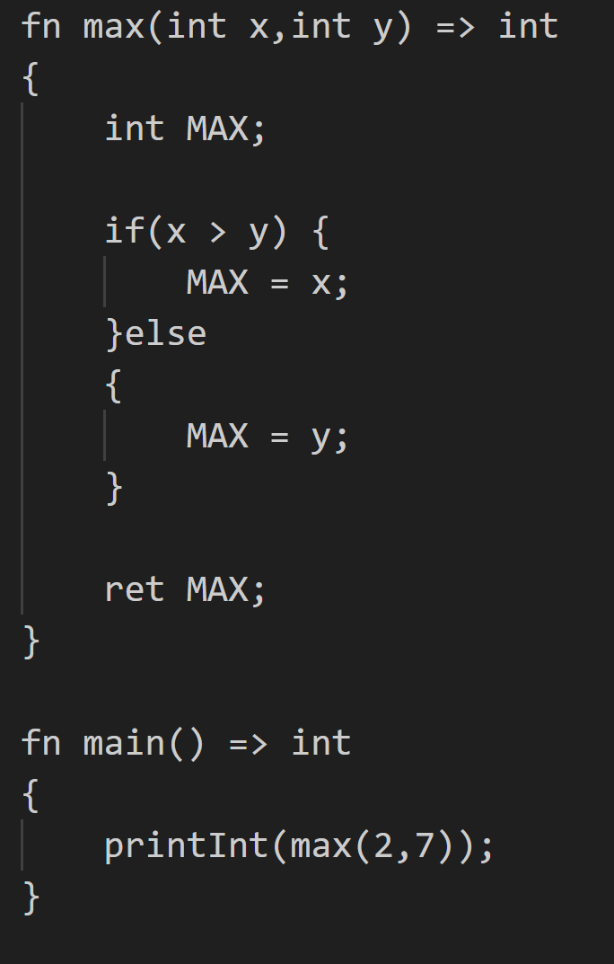
**דוגמא לשימוש:**



בסיום ריצת קטע הקוד הנ"ל ערכו של x יהיה 4

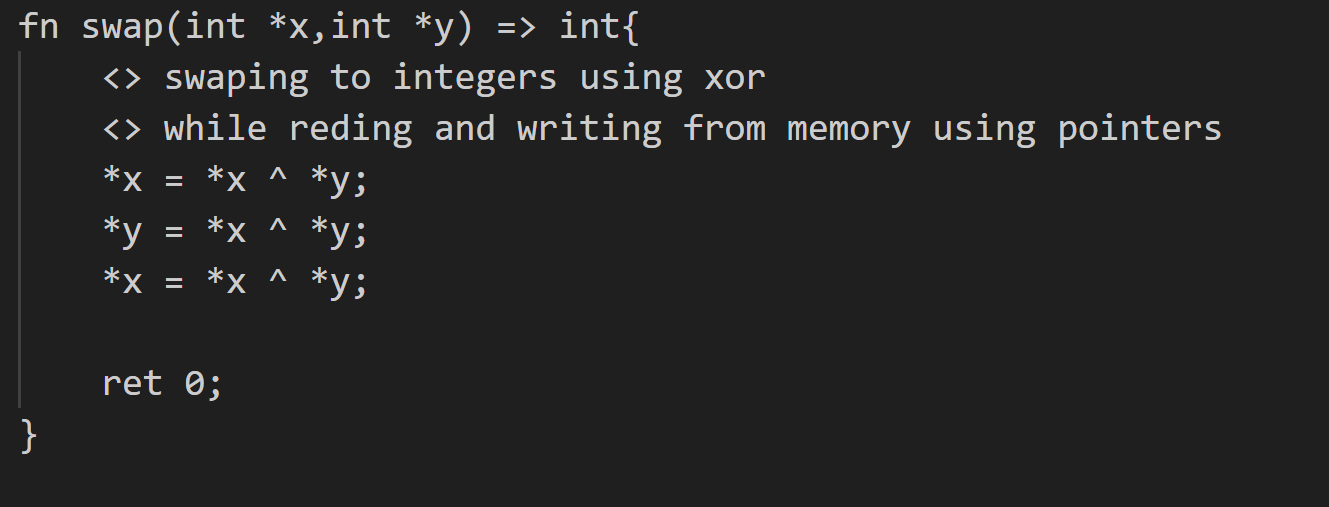
וערכו של ptr הינו הכתובת של x

## תוכניות לדוגמא בשפת easy

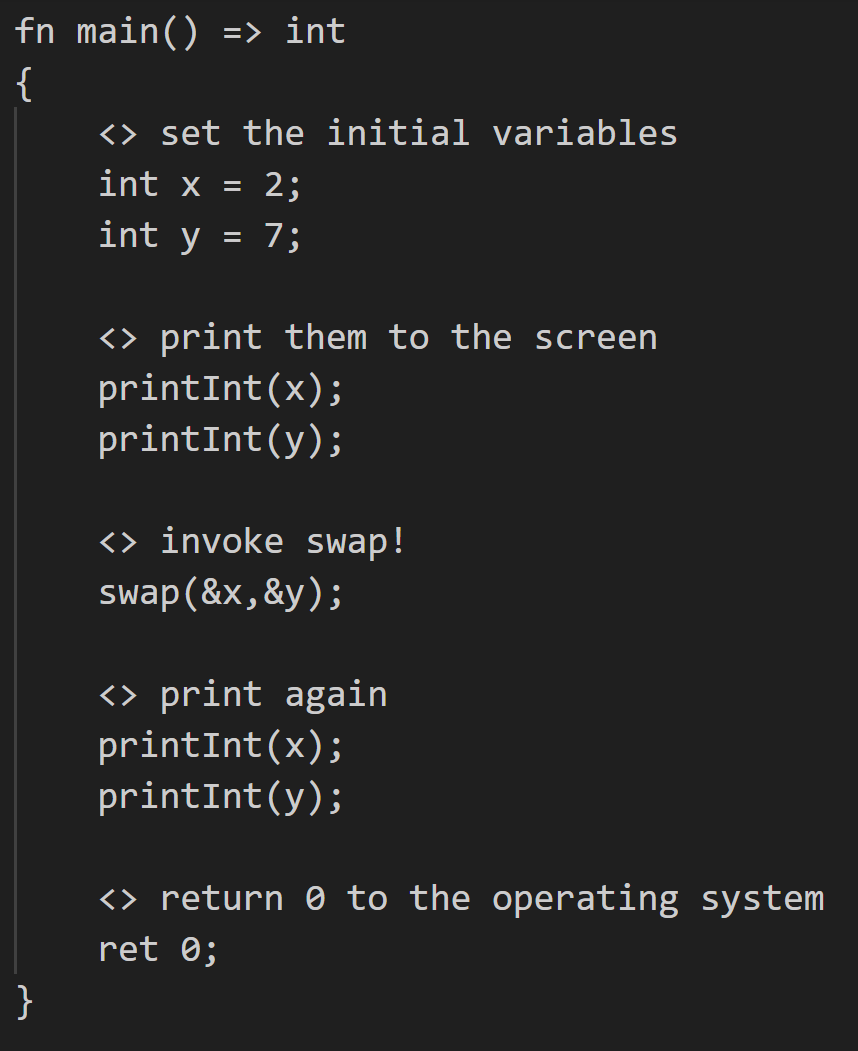


להלן תוכנית המחזירה את המספר הגדול ביותר :

פונקציה המקבלת שתי מצביעים ומחליפה בין הערכים בתוכם:



ופונקציית main אשר קוראת לה :



## דקדוק השפה

מה הוא דקדוק?

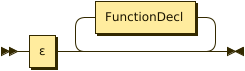
דקדוק הינו אוסף של חוקים המגדירים את השפה.

להלן הדקדוק מוצג בשתי אופנים שונים.

בBNF אשר הינו פורמט סטנדרטי לייצוג שפות

וב **Railroad Diagram**אשר גם הוא דרך סטנדרטית להצגת דקדוק בצורה הממחישה את זרימת השפה

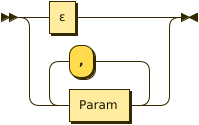
**Program:**



**FunctionDecl:**

FunctionDecl

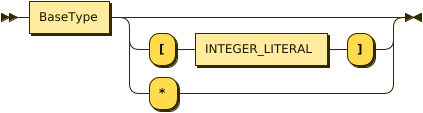
**ParamList:**



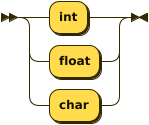
**Param:**

Param

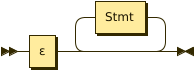
**Type:**



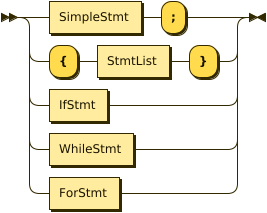
**BaseType:**



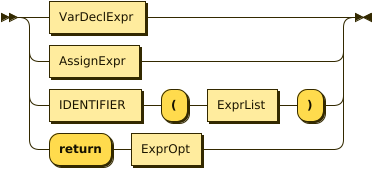
**StmtList:**



**Stmt:**



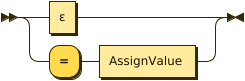
**SimpleStmt:**



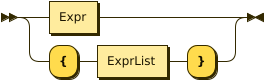
**VarDeclExpr:**

VarDeclExpr

**InitOpt:**



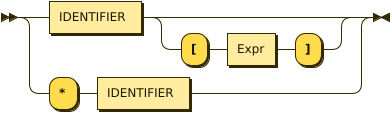
**AssignValue:**



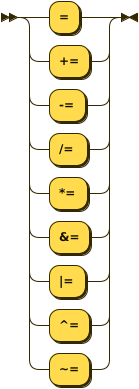
**AssignExpr:**

AssignExpr

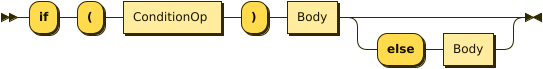
**AssignTarget:**



**AssignOp:**



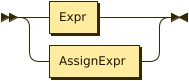
**IfStmt:**



**WhileStmt:**

WhileStmt

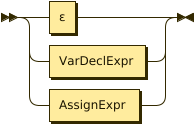
**ConditionOp:**



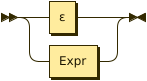
**ForStmt:**

ForStmt

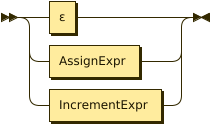
**ForInit:**



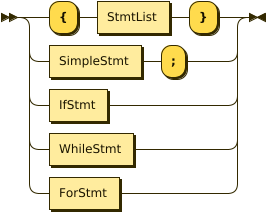
**ExprOpt:**



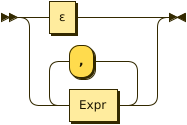
**ForUpdate:**



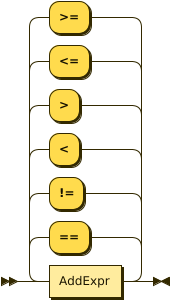
**Body:**

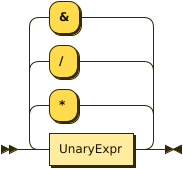


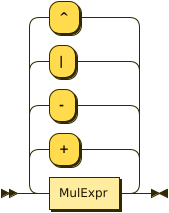
**ExprList:** **Expr:**



**RelationalExpr:**

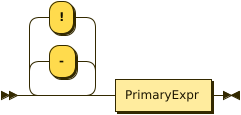


**AddExpr:** **MulExpr:**

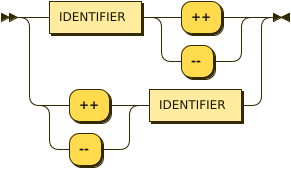


**:**

**UnaryExpr:**



**IncrementExpr:**



**AddressExpr:**

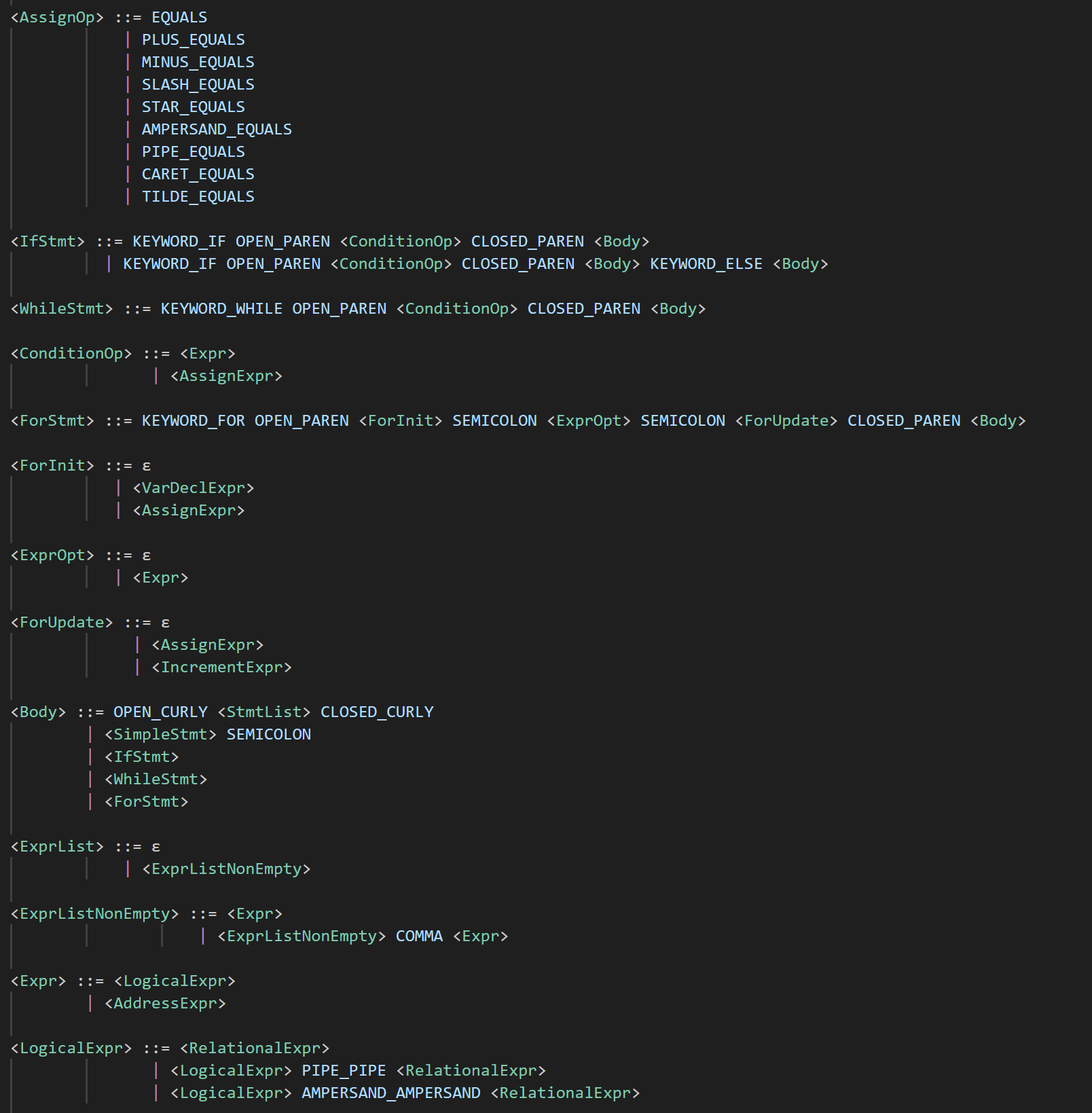
AddressExpr

DereferenceExpr**DereferenceExpr:**

***PrimaryExpr*:**

![PrimaryExpr](data:image/png;base64;base64,)

## **תצוגת bnf**



תמונה שמכילה טקסט, צילום מסך, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

# רקע תאורטי בתחום הפרויקט

### קומפיילר/מהדר – הגדרה כללית:

בעולם המחשבים קומפיילר (או בשמו העברי: מהדר) הוא רכיב תוכנתי שתפקידו הוא להמיר בין שתי שפות תכנה, המהדר הקלאסי ימיר בין שפה עילית לשפת מכונה

**תמונה שמכילה טקסט, קו, צילום מסך, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.**

ניקח לדוגמא את הקוד הבא הנכתב בשפה העילית "C"אשר נרצה להריץ:

**תמונה שמכילה טקסט, צילום מסך, גופן, מספר

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.**

שפות עליות כמו "C" ורבות אחרות עוצבו בצורה הדומה לשפה האנגלית המדוברת ושיהיה למתכנת נוח וקל להשתמש בהן אך המעבד לא בנוי בצורה שהוא יכול להבין בקלות את הוראות אלו. תפקידו של המעבד הוא להמיר את השפה העילית לשפת סף הבנויה בצורה קרובה לארכיטקטורת המעבד שאיתו אנחנו עובדים.

לצורך ההדגמה המהדר יצור לנו את קובץ הassembly הבא:

**תמונה שמכילה טקסט, צילום מסך, מספר, תוכנה

התיאור נוצר באופן אוטומטי**

אז נשתמש בעוד מהדר הממיר קוד סף אל שפת מכונה

תמונה שמכילה טקסט, צילום מסך, גופן, מספר

התיאור נוצר באופן אוטומטי( דוגמאות למהדרים כאלו: NASM לארכיטקטורת x86\_64 TASM לארכיטקטורת x86)

### סוגים שונים של מהדרים:

**Source compiler** – ממיר את הקוד הראשוני (בשפה העילית) היישר לשפת מכונה לדוגמא -רוב המעבדים של השפות C וCPP משתמשים בצורת הידור זאת

**Intermediate Compiler** - ממיר את הקוד הראשוני לשפת ביניים, אז שפת הביניים עוברת מהדר נוסף הממיר אותה לשפת מכונה. לדוגמא – המהדר של C# ממיר את קוד המקור לשפת CIL ומשם עובר עוד מהדר הממיר את הקוד לשפת מכונה

**Transpiler** – ממיר קוד משפה עילית אחת לשנייה לדוגמא – קוד בTypeScript יומר תחילה לJS בשלב הראשוני בהידורו (וזה נעשה בעזרת Transpiler)

### סוגים שונים של שפות תכנות:

שפות תכנות פרוצדורליות - שפות כמו C ופסקל, בשפות אלו, התוכנית מורכבת מרשימת פקודות המבוצעות בסדר שלב אחרי שלב.

שפות תכנות מונחות עצמים -שפות כמו C# וJAVA. בשפות אלו, התוכנית בנויה ממודולים הנקראים אובייקטים, שכל אחד מהם כולל נתונים (תכונות) ופעולות (מתודולוגיות).

שפות תכנות פונקציונליות – שפות כמו Haskell וErlang. בשפות אלו, הפוקנציה היא יחידת התכנות העיקרית. התכנות נעשה דרך קריאה לפונקציות, ולא על ידי שינוי מצב של משתנים.

### שלבי הקומפילציה:

תהליך הקומפילציה מתחלק לשני שלבים מרכזיים,

הfront end- והbeck end-.

### Front end:

שלב זה מתעסק בניתוח הקוד והמרתו לתצורת ביניים (IR) שאיתה השלב השני (beck end) יכול לעבוד. הוא עוסק בניתוח לוגי ובהבנה של המבנה והמשמעות של הקוד, וגם הוא מתחלק לכמה חלקים.

## ניתוח לקסיקלי(Lexical Analysis)

השלב הראשון בfront end עוסק בחילוק הקוד הנתון לטוקנים, אשר כל טוקן הוא יחידה בעלת משמעות כמו מילים שמורות, משתנים, מספרים, אופרטורים וכדומה.

לדוגמה, פיסת הקוד int x = 7; תוכל להיות מתורגמת לטוקנים

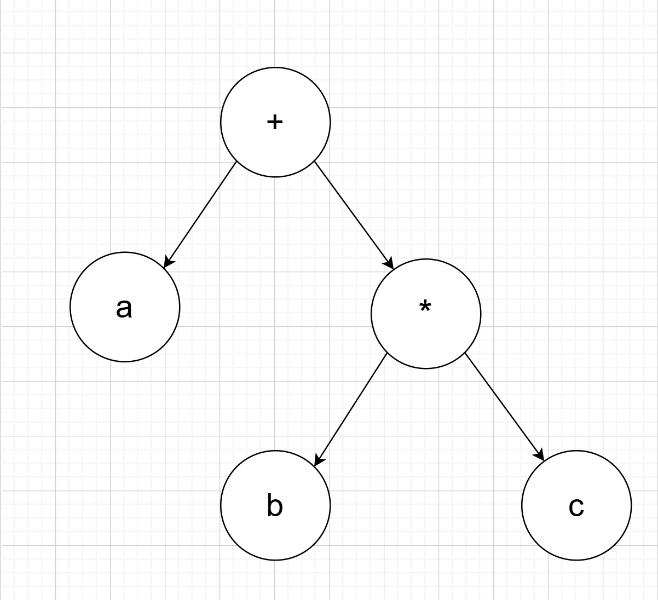
* **Int**  (מילה שמורה)
* **x** (מזהה)
* **=** (אופרטור)
* **7** (מספר)
* **;** (נקודה פסיק)

## ניתוח תחבירי (Syntax Analysis)

מטרה: השלב הבא הוא לבדוק אם רצף הטוקנים שנוצר בתהליך הקודם תואם לחוקי התחביר של השפה (הדקדוק שלה). השלב הזה יוצר את ה-עץ תחבירי או ה-עץ סמנטי (abstract syntax tree - AST), שמייצג את מבנה הקוד לפי כללי השפה.

דוגמה: אם הקוד הוא **a + b \* c**, האנליזר התחבירי ייצור עץ תחבירי שמייצג את ההצהרה על המשתנה וההקצאה.

תמונה שמכילה עיגול, תרשים, קו

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.דוגמא לעץ סמנטי תקין: דוגמא לעץ סמנטי לא תקין:

במהלך השלב הזה, אם יש בעיות תחביריות בקוד (כגון חוסר בסוגריים או סדר לא נכון של רכיבי השפה), יופיעו הודעות שגיאה.

## דקדוק חסר הקשר (Context free grammar)

דקדוק חסר הקשר הינו ענף במתמטיקה אשר עוזר לנו להבין את המבנה של השפה והוא השלד שלפיו נבנה את העץ

**סימני הדקדוק** מתחלקים לשתי קבוצות **סופיים** ו**לא סופיים**

**סופיים**: סימנים אשר ניתן להרחיב אותם לפי כלל דקדוק

**לא סופיים**: סימנים אשר לא ניתנים להרחבה

**כללי דקדוק:** דקדוקים בכללי ודקדוקים חסרי הקשר בפרט מורכבים מאוסף של חוקים אשר מגדירים את הדקדוק

**כלל דקדוק בדקדוק חסר הקשר נראה ככה**:

**<רצף סימנים (סופיים או לא)> → <סימן לא סופי>**

ניקח לדוגמא את הדקדוק הבא:

**הסימנים הסופיים בדקדוק: S**,**E**,**T**,**F**

**הסימנים הלא סופיים בדקדוק: id** , **=** , **;** , **+** , **\***

אם היינו מקבלים את הקלט הבא: **x = a + b \* c;**

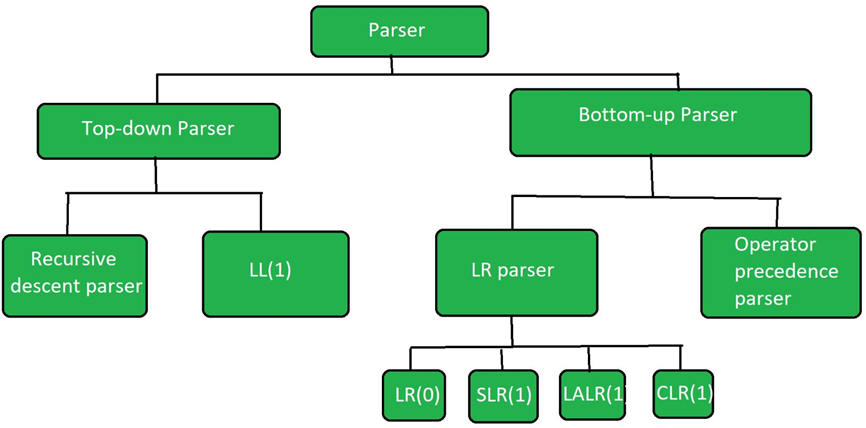
לפי הדקדוק הנתון יתקבל העץ:

תמונה שמכילה קו, תרשים

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

## **סוגי מנתחים תחביריים**

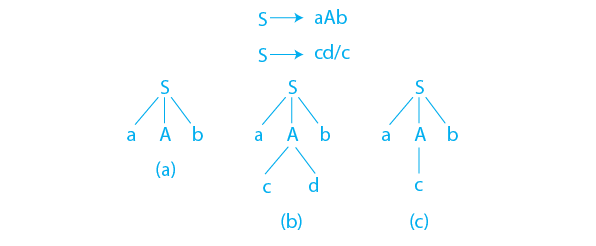
המנתח התחבירי הוא הרכיב המסובך ביותר במהדר. עקב זאת הוא בעל הכי הרבה תאוריה ומחקר בין כל הרכיבים

המנתחים התחביריים המתחלקים לשתי קבוצות עיקריות: **bottom up** ו- **Top down**

## מלמעלה למטה (**top - down**)

ניתוח תחבירי מלמעלה למטה מתחיל עם כלל התחבירי הראשון ומנסה לפתח אותו אל הnon terminals השונים בשפה

אם ניקח לדוגמא את הgrammar הבא:

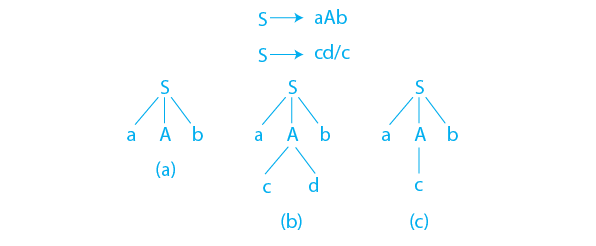


ניתוח top – down יוכל לפתח את העץ בצורה הבאה:

**שלב 1**

**שלב 3**

**שלב 2**



## **מנתחי bottom-up**

מנתחי LR (left-right)

משפחה של מנתחם תחביריים המנתחים מלמטה למעלה ומשתמשים באוטומט מחסנית pda

**(0 LR** – מנתח פשוט המשתמש הלא יודע להתמודד עם דקדוקים מסובכים ומורכבים מידי

**(1)LR**– עובד בדומה ל**(0)LR** אך מסתכל גם על האסימון הנוכחי וגם על האחד שאחריו וכך יכול להתמודד על דקדוקים מסובכים יותר

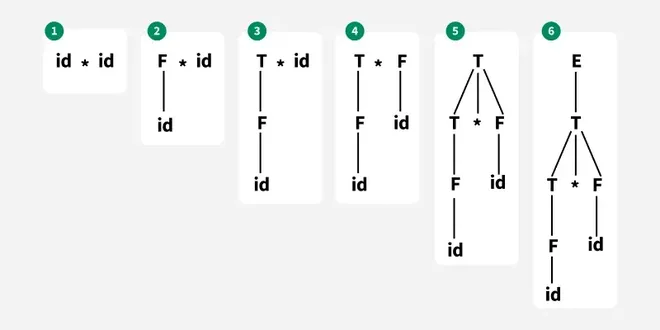
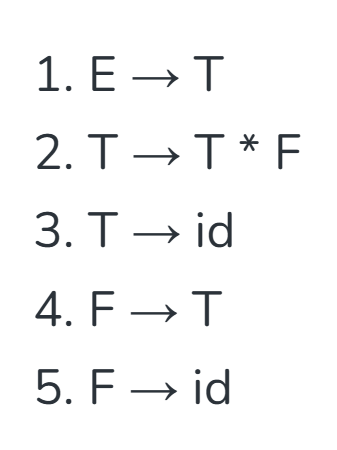
**(1)LALR (look-ahead, left-to-right)** – עובד בדומה ל **(1)LR** אך יוצר טבלה קטנה יותר וכך חוסך מקום זיכרון

**(1)CLR** **(Canonical, left-to-right )** – המנטח הרובסטי יותר מכולם ויודע להתמודד עם כל הדקדוקים חסרי ההקשר שהם

## מלמטה למעלה (**bottom - up)**

ניתוח תחבירי מלמטה למעלה ינסה לקחת את כל האסימונים ולצמצם אותם עד לכדי הגעה לכלל הראשון של הדקדוק

**הgrammar הבא: ינותח בצורה הבאה:**



**Recursive Descent** - מנתח המשתמש בפונקציה לכל כלל דקדוק

– **LL(k) Parsing**משתמש בטבלת ניתוח תחבירי ובמבט קדימה של k סמלים. יכול להיות ממומש עם מחסנית

## ניתוח סמנטי (Semantic Analysis) –

שלב הניתוח הסמנטי הוא שלב שמטרתו לעבור על העץ הסמנטי ולוודא שאין בקוד שגיאות סמנטיות כגון, בדיקת טיפוסים, וידוא התאמה של פרמטרים לפונקציות ועוד....

## Back end-

שלב זה עוסק בניתוח קוד הביניים שהתקבל בשלב הfront end. המטרה בשלב זה היא לא "הבנה" או "ניתוח " של קוד המקור אלה לעבור על קוד הביניים ולהעביר אותו בצורה אופטימלית לשפת הסף

## אופטימיזציה –

שלב האופטימיזציה יעבור על קוד הביניים וייעל אותו בעזרת כמה שיטות, מחיקת שורות ללא משמעות, קיצור הקוד בדרכים שישאירו לו את אותה המשמעות וכו...

## יצירת קוד הסף –

לקיחת קוד הביניים שעבר אופטימיזציה ולהמיר אותו לקוד סף

## טיפול בשגיאות-

בכל שלבי ההידור נרצה לתפוס שגיאות.

לדוגמא, כתיבת שם משתנה לא תקין (שגיאה תחבירית) או פנייה למשנה לא מוצהר (שגיאה סמנטית). טיפול בשגיאות הוא חלק קריטי בכל תהליך פיתוח תוכנה, ובפרט בקומפיילר. המטרה המרכזית היא להבטיח שהתוכנית שכתבת, או במקרה שלנו, הקוד שהקומפיילר מייצר, יתפקד בצורה נכונה, יציבה ובטוחה.נרצה להודיע למתכנת כשהקוד לא תקין ואיפה נמצאת השגיאה לנוחות ולפיתוח מהיר.

# תיאור הבעיה האלגוריתמית

הבעיה האלגוריתמית טמונה במימוש ששת השלבי פיתוח הקומפיילר. כל שלב מציג בעיה אלגוריתמית שונה.

ניתוח לקסיקלי – חלוקת קוד המקור לטוקנים בעזרת **אוטומט דטרמיניסטי סופי**.

ניתוח תחבירי – שלב זה הוא מהעמוסים ביותר מבחינה אלגוריתמית בפיתוח המהדר, הוא כולל את יצירת העץ הסמנטי ועבודה אתו הכוללת מספר אלגוריתמים על עצים והכרעת מצבים בין בניית עצים שונים. שלב זה גם פועל על **אוטומט דטרמיניסטי סופי**.

ניתוח סמנטי – עבירה על העץ הסמנטי ובדיקה של **התקינות הסמנטית** שלו

יצירת קוד הביניים – עבירה על העץ הסמנטי ותרגומו להצגת הביניים.

אופטימיזציה – עבירה על קוד הביניים ובדיקה על איזה מקודות ניתן לעשות לו אופטימיזציה בדרכים שונות.

יצירת קוד הסף – תרגום קוד הביניים אל קוד הסף.

# אסטרטגיה נבחרת לפתרון

האסטרטגיה – לפרק את הפרויקט הגדול לחלקים, כל חלק בעל פונקציונליות משל עצמו.

בסוף פיתוח כל חלק נשתמש בו כקופסא שחורה (black box) כך נוכל להפסיק להתעסק בחלק זה ולהתחיל לפתח את החלק השני בלי להתייחס לתאוריה של החלק הראשון, כך הפיתוח יהיה ממוקד יותר ומוציא קוד דינמי וקריא יותר.

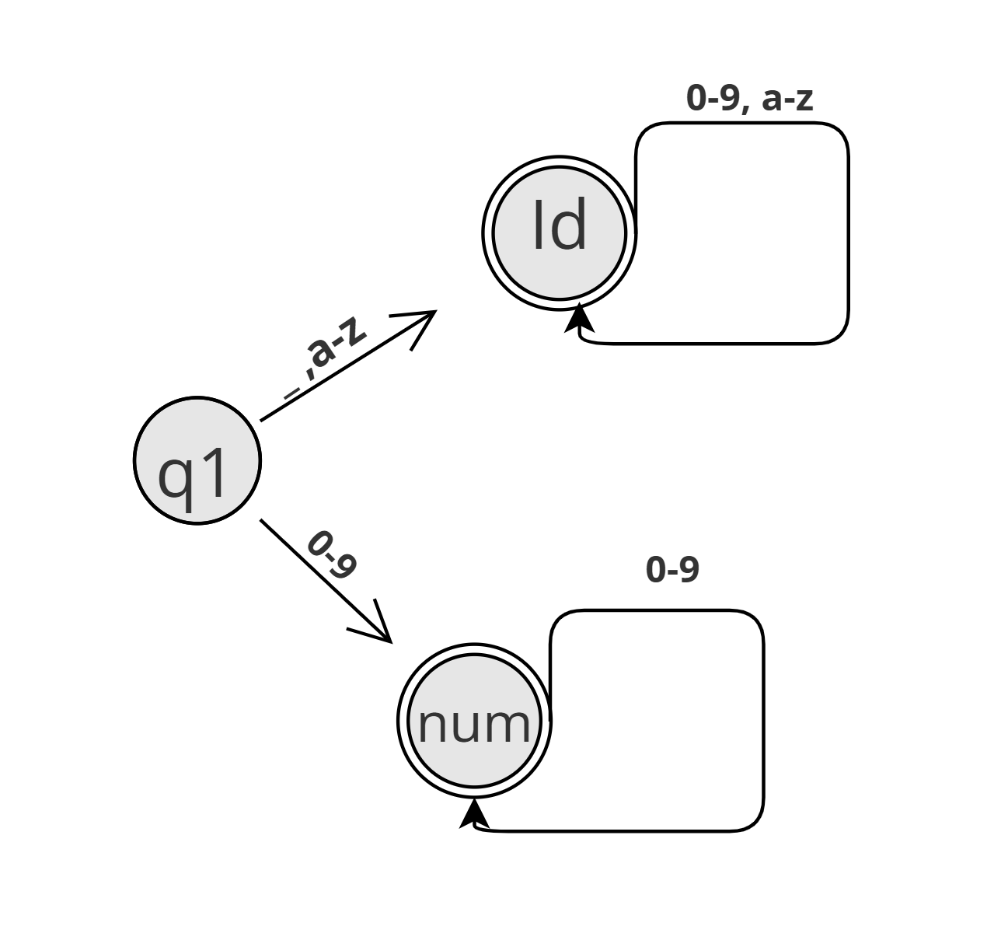
להלן האסטרטגיה לפיתוח כל חלק:

## האסטרטגיה לפיתוח המנתח הסמנטי

בחרתי לפתח את המנתח הסמנטי בעזרת **אוטומט סופי דטרמיניסטי (DFA)**. המנתח עובר תו תו על הקלט ומחלק אותו לאסימונים

ניתח לדוגמא שנראה לחלק את השפה לארבעה סוגים שונים של אסימונים:

**מזהה , מספר, ואסימון לא תקין (לא שייך לשפה)**

נבנה את האוטומט הבא:

כפי שאנו רואים האוטומט למעלה יצליח לזהות מספרים ומזהים . כלומר: אם ניקח לדוגמה את המזהה sum נראה כי האוטומט יסיים בצומתId וכך נדע שהוא מזהה. ציין כי עם קיבלנו קלט שהוא לא אחד מהקלטים האפשריים בצומת הנוכחית, נשלח לצומת מלכודת (Trap state) המציינת אסימון לא תקין

### **האוטומט של tec**

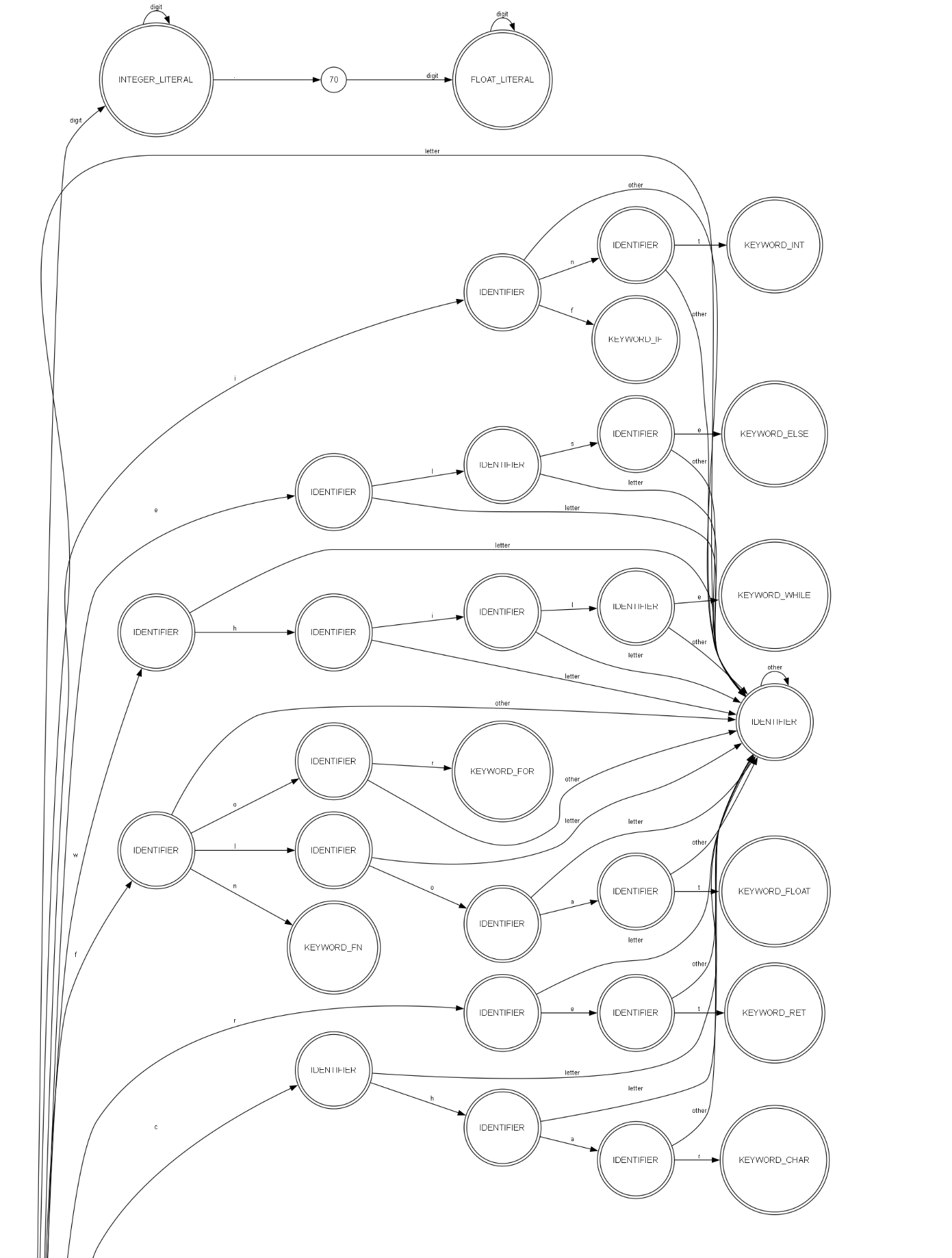
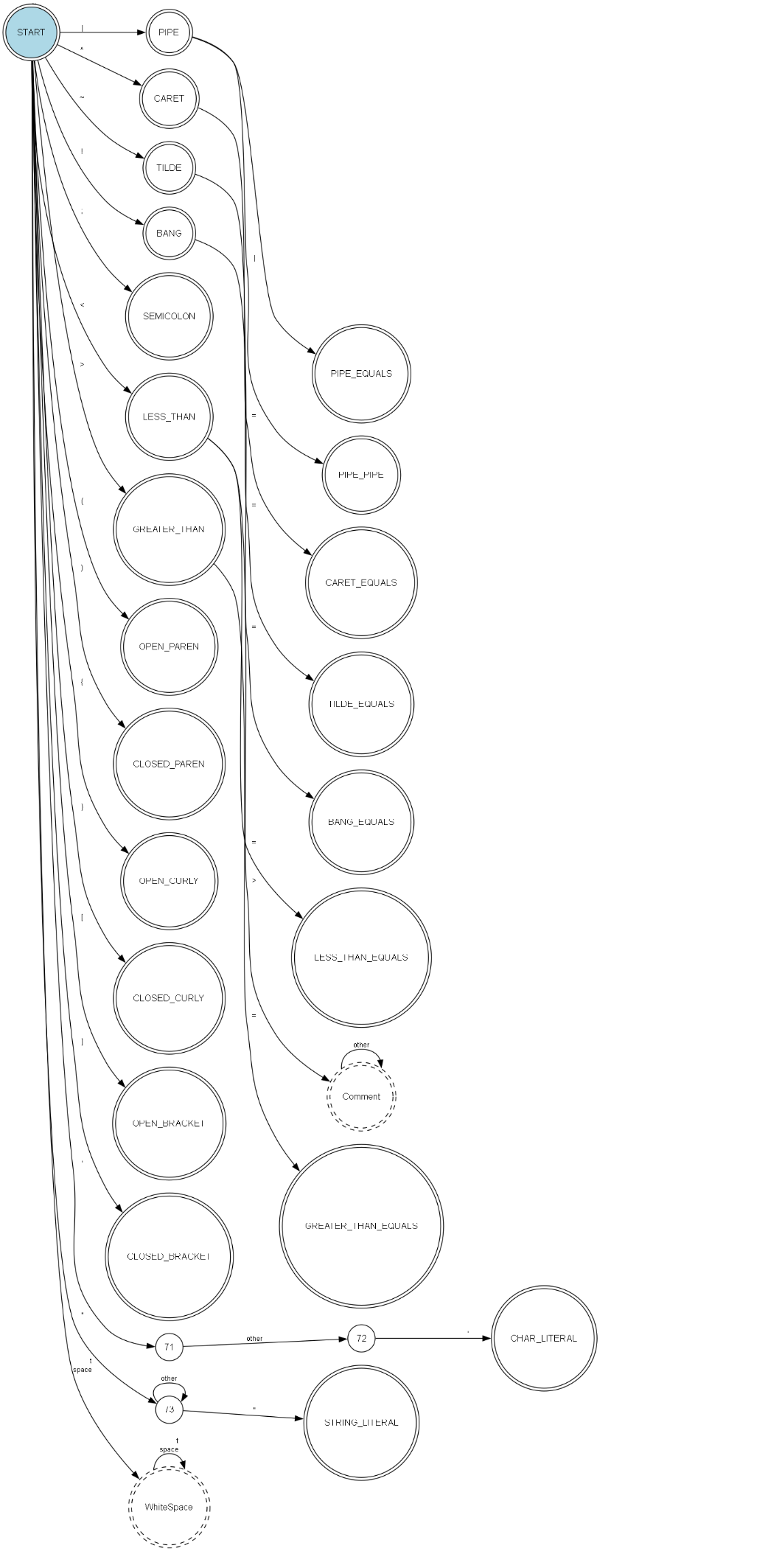
הסבר על התרשים:

מצב התחלתי (מסומן ב**S**-) – מסמל את המצב הראשוני לסריקת אסימון לפני קבלת כל אות

מצבי ביניים (מסומנים ב**מספרים**) – מסמלים מעבריי ביניים באוטומט. סיום הסריקה בהם תסתמן כשגיאה במהדר

מצבי סיום (מסומנים בעיגול כפול) – מסמלים מצביי קבלה/סיום של האוטומט. סיום הסריקה בהם תסתמן כאסימן תקין

מצבי דילוג (מסומנים בעיגול מקווקו) – מסמלים מצביי דילוג של האוטומט. סיום הסריקה בהם לא תחשב ככלום במהדר

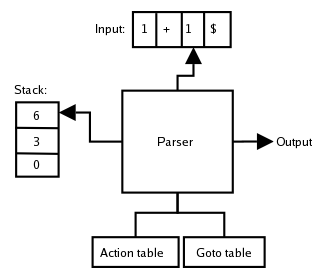
חלק האוטומט הקולט מזהים (שמות משתנים ופונקציות),מחרוזות,מספרים שלמים, מספרים עשרוניים ו חלק האוטומט הקולט את שלל סימני השפה, רווחים והערות:

מילות מפתח:

## האסטרטגיה לפיתוח המנתח התחבירי

בחרתי להשתמש ב **LR parser**לקומפיילר שלי

איך הוא בנוי?



המנתח התחבירי מסתמך על שתי טבלאות – **Goto**, **Action**

ומבנה נתונים – **מחסנית**.

הוא בנוי על מכונת מצבים גם כן

המחסנית – **מאחסנת את סימני הדקדוק ומספרי המצבים**

הטבלאות:

**Goto** – מסמנת לאיזה מצב לעבור על פי מצב הקלט והמחסנית

**Action** – מסמן איזה פעולה לבצע לפי המצב הנוכחי

### **הפעולות האפשריות**

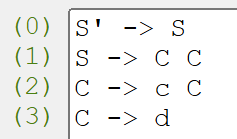
**Shift –** הכנס את האסימון הבא מהקלט אל המחסנית ביחד עם מספר המצב המצויין

**Reduce – צמצם** את הקלטים בראש המחסנית לפי חוק דקדוק מצוין

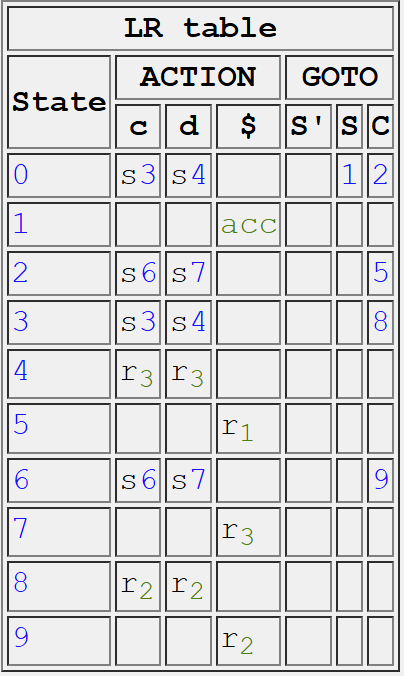
**לפי מה נבנה את הטבלאות ?**

ישנם כמה שיטות לבנות את הטבלאות אך אני בחרתי לבנות את הטבלה לפי אלגוריתם **IELR**

**למה בחרתי באלגוריתם זה?** כי הוא בונה את הטבלה יחסית מצומצמת ויכול לקבל הרבה סוגים שונים של חוקי דקדוק

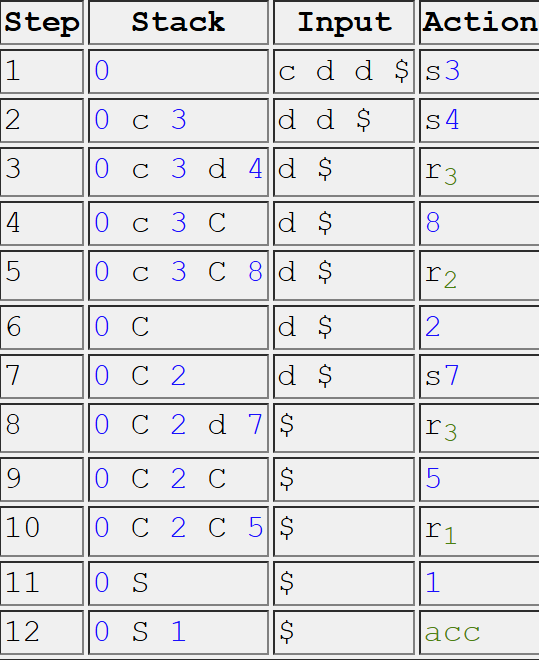
ניקח לדוגמא את **הCFG** הבא:

בעזרתו נבנה את טבלאות **הGOTOו** **ACTION**



ואם נרצה לנתח לדוגמא את הקלט: **c d d**

המעקב יראה ככה:



תמונה שמכילה מלבן, עיצוב

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.והעץ הסופי יראה ככה:

## 

## האסטרטגיה לפיתוח המנתח הסמנטי

בשביל לממש שלב זה נצטרך:

**לעבור על טבלת הסימנים** ולראות ש-

* אין כפילויות.
* כל משתנה משומש לפי הscope שלו

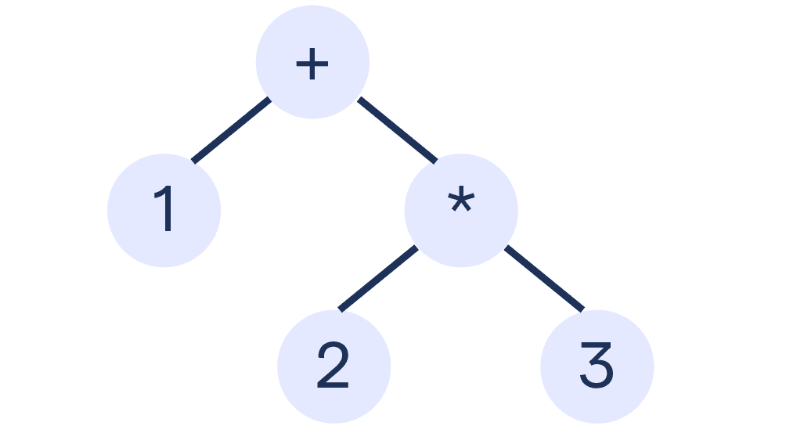
לעבור על העץ בצורה רקורסיבית, ולבדוק את התקינות של דברים שונים כגון:

**האם בהכרזת וייסום משתנים סוגי המשתנים תואמים?**

**האם הפונקציה מחזירה את מה שנמצא בהכרזה שלה?**

**האם קריאה לפונקציה תואמת את הפרמטרים בהכרזה שלה?**

גם כן יש ליזכור שבשפה ישנם טיפוסים שונים ונרצה לבדוק אם הם תואמים בכדי שנוכל לבצע עליהם פעולת. בשביל זה ניתן לכל חולייה בעץ טיפוס, החל מלמטה במעלה העץ ניתן לכל החוליה טיפוס



**int**

**int**

**int**

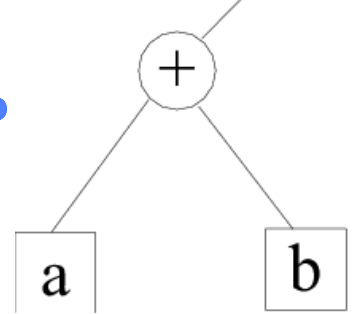
## 

**int**

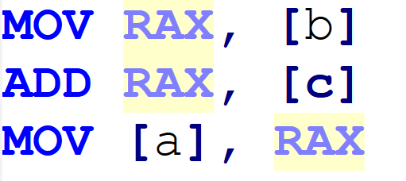
**int**

## האסטרטגיה לפיתוח שלב כתיבת קוד הסף

בשלב זה גם נעבור על העץ אך הפעם נסתכל על כל צומת בעץ ולפי סוגה נמיר לקוד סף.

**פעולות אריתמטיות**

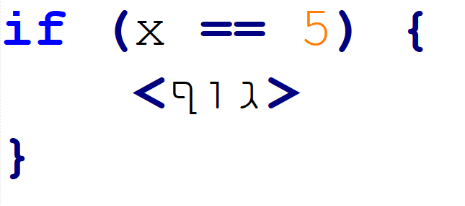
**החלק בעץ:**



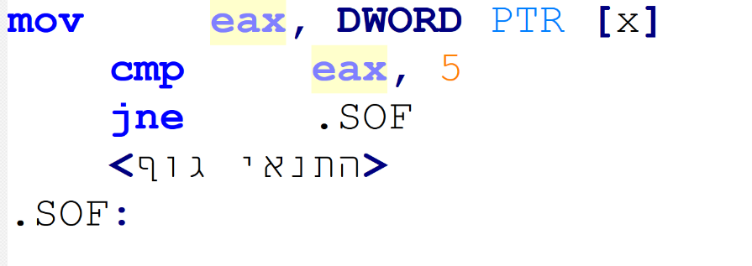
**יהפוך ל:**

**תנאים**

נשתמש במבנה בסיסי של שפת סף לתנאים

**הביטוי:**

**יהפוך ל:**



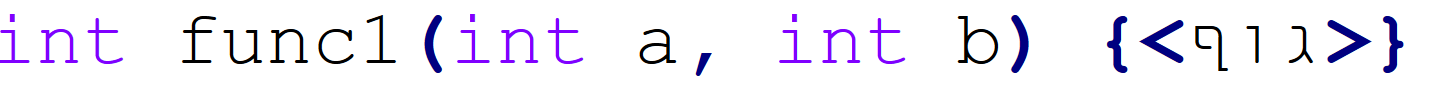
## פונקציות:

**יצירה:**

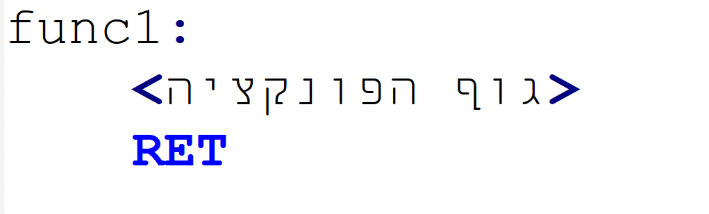
נדחוף למחסנית את כל המשתנים של הScope הנוכחי (כולל הפרמטרים)

נאתחל את הפרמטרים עם הערכים שנתנו להם בזימון

ניקח לדוגמא פונקציה.

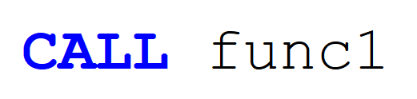


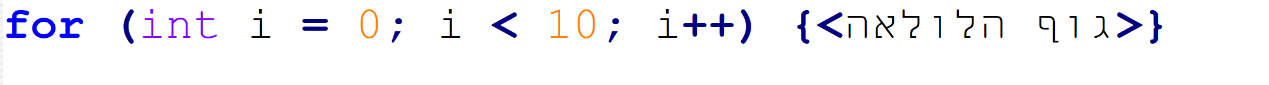
נוכל ליצור אותה באמצעות יצירת label חדש וסיום בשימוש הפקודה ret



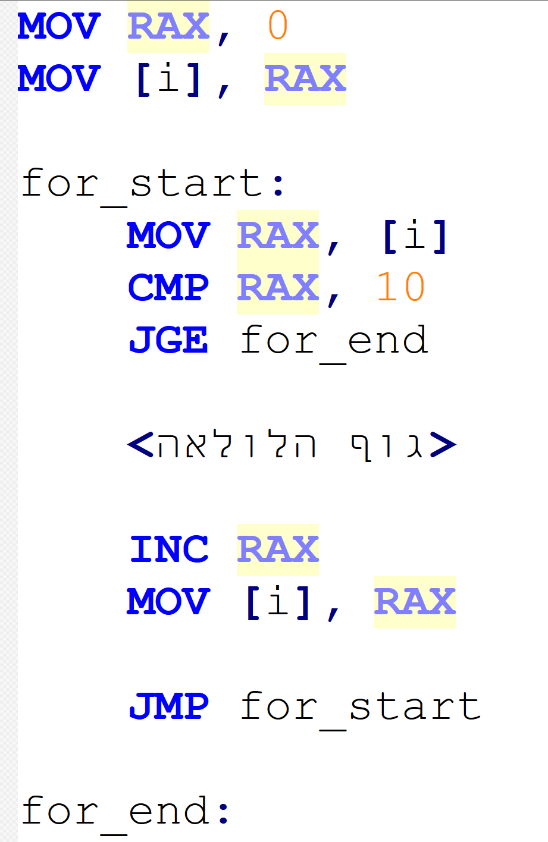
כאשר **a**ו**b**- שמורים ב**[bp - 8]**ו **[bp - 16]-**במחסנית

**קריאה לפונקציה:**



**לולאת for**

**יתורגם ל:**



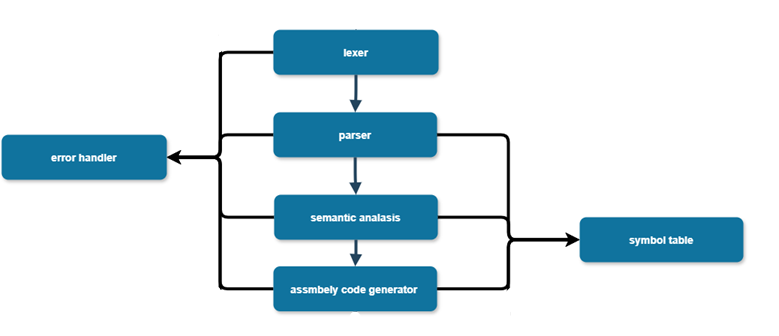
**לולאות while**



**יתורגם ל:**

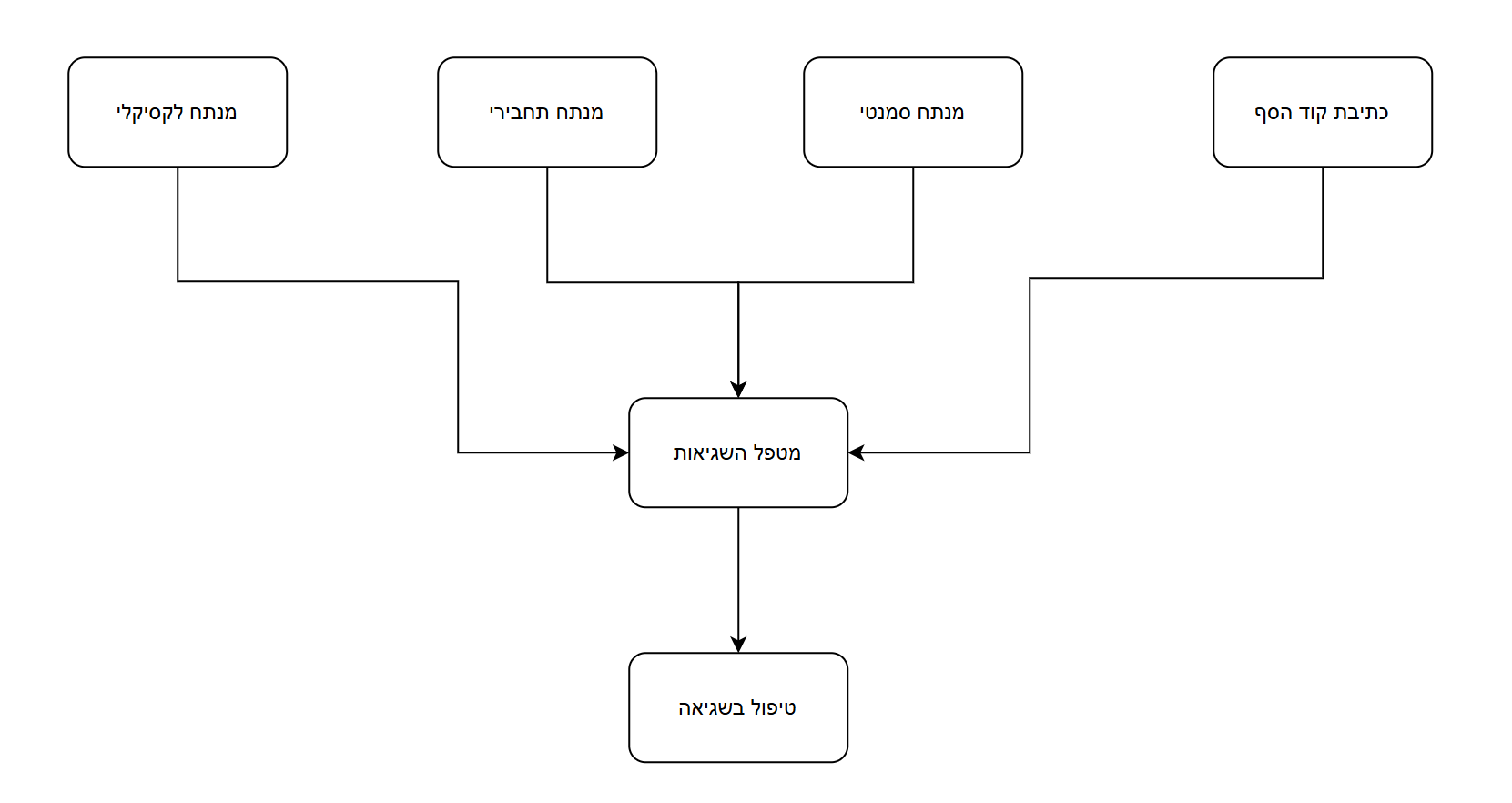


# ארכיטקטורה של הפתרון המוצע בפורמט של Top-Down Level Design

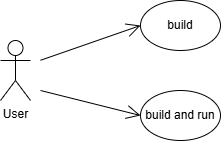
להלן תרשים זרימה המתאר את החלקים השונים ואת התקשורת ביניהם בפורמט השקפת על (High level):

## תמונה שמכילה תרשים, טקסט, תוכנית, שרטוט טכני תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.תרשים Top-down מפורט

כל חלק אשר גזור ישירות מהקומפוננטה הראשית מקושר ישירות אל מטפל השגיאות



# תרשים מקרי שימוש UML Use cases



# מבני נתונים

## מטריצה

פיתוח מהדר כולל עבודה עם כלל מבני נתונים שונים ומגוונים. הרי אנחנו עובדים עם קלט אשר אנו רוצים לפרק ולעבד.

ובשביל לעשות זאת נשתמש **במבנה נתונים מתאים** לכל בעיה

כפי שראינו בפרקים הקודמים כתיבת המנתח הלקסיקלי כרוכה בעבודה עם **אוטומט סופי דטרמיניסטי (dfa)**. אך איך נמיר אותו לקוד.

ישנם שתי דכים מקובלות לעשות זאת.

**מפת hash** – אשר שומרת כמפתח את ה**מצב ההתחלתי** ובערך את הצמד של **תו המעבר** **והמצב הסופי**

**מטריצה –** אשר שומרת את אותם שלושה ערכים בפורמט שבו **השורות** מייצגות את ה**מצב ההתחלתי העמודות** את **תו המעבר** **והחיתוך בינהן** הינו **המצב הסופי**

בחרתי להשתמש ב**מטריצה**

מטריצה הינו מבנה נתונים הבנוי כמערך דו ממדי של ערכים (בעל עמודות ושורות)

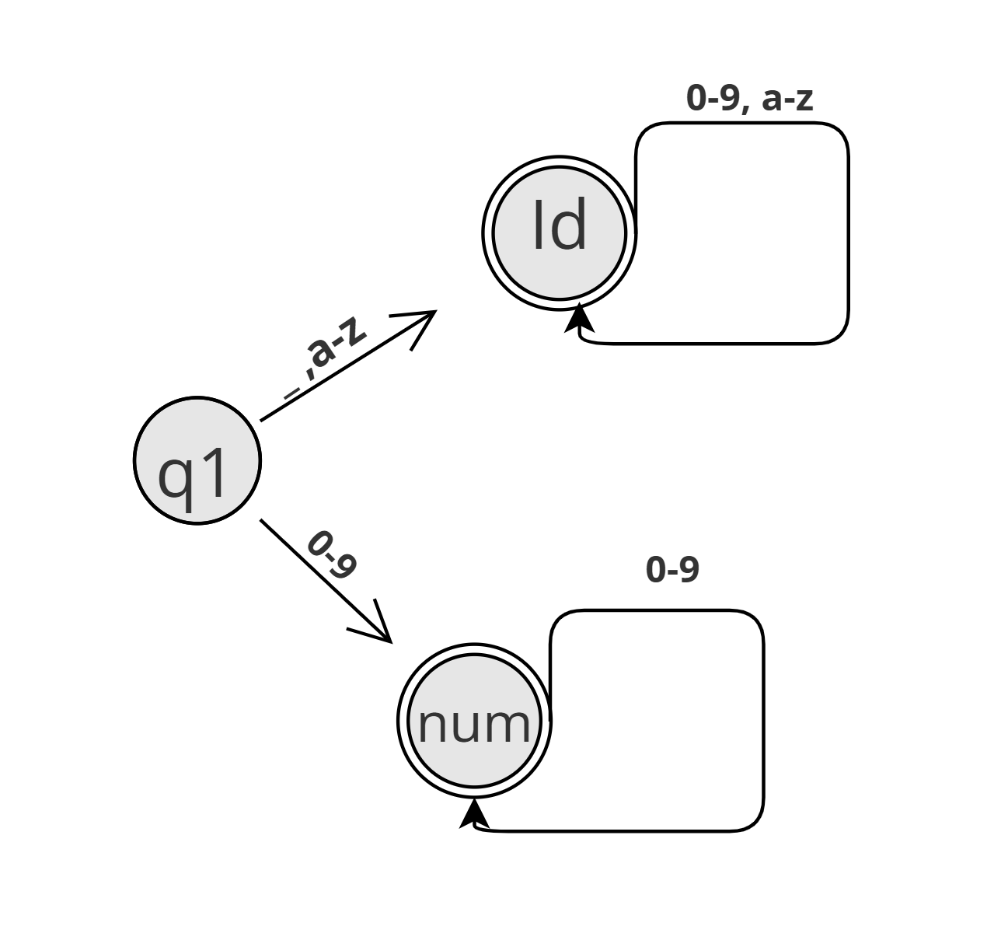
למה? – זהו מבנה נתונים אשר קל לעבוד איתו ולהכניס אילו נתונים.

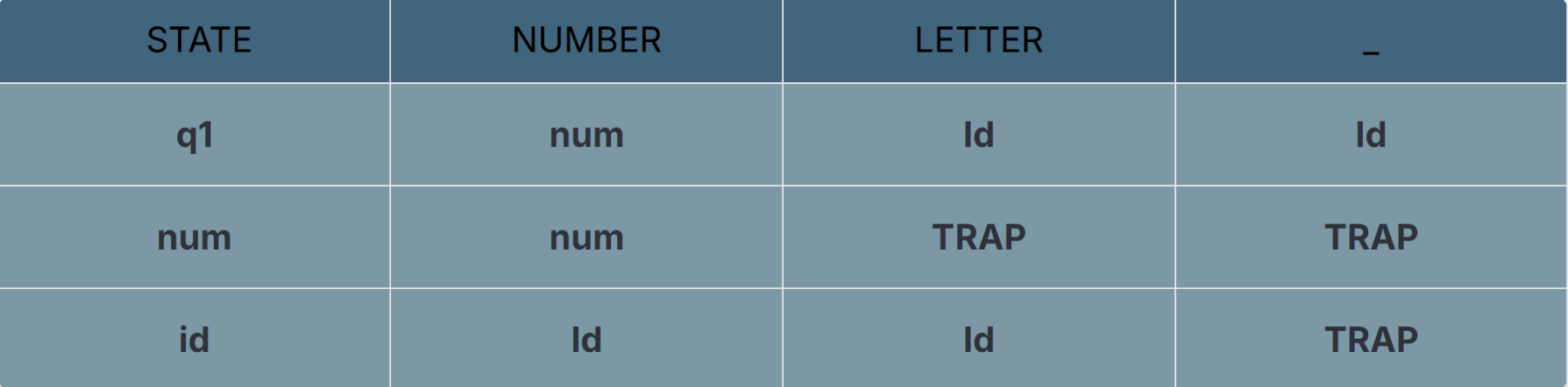
שליפת **מצב הבא** בהינתן מצב נוכחי ותו מעבר הינו **O(1)** לעומת **מפת הhash** ששם היא **O(N)** אך יש לשים לב שהמטריצה לא יותר טובה בצורה אבסולוטית מהמפה מכיוון שיעילות המקום של שהמטריצה תופסת בהתייחס לכמות המצבים שיש הינה **O(N^2)** לעומת המפה אשר תופסת **O(N)** מקום.

אך למרות זאת בחרתי במטריצה מכיוון שאעדיף להקריב **מקום** מאשר **זמן ריצה**

**דוגמא להמרת אוטומט למטריצה**

ניקח לדוגמא את האוטומט הבא:



נוכל להמירו למטריצה הזאת:

קל מאוד לראות שאם לדוגמא מצבו הנוכחי של האוטומט הוא q1 והקלט הוא מספר נעבור למצב **num**

אנו נשתמש במטריצה בעוד מספר מקומות בקוד שבהם השימוש בא יותר טריוויאלי. כמו טבלאות (**symbol table**, **goto** **table**, **action table**)

## מפת hash

Hash map הינו מבנה נתונים חשוב בתכנות אשר מאפשר לשמור צמדים של נתונים key,value בצורה כזאת שניתנן אפשרות לשלוף את ערך הvalue בעזרת המפתח ביעילות זמן ריצה **O(1)**

השתמשתי במבנה נתונים זה במקומות רבים בקוד שלי בעיקר לנוחות.

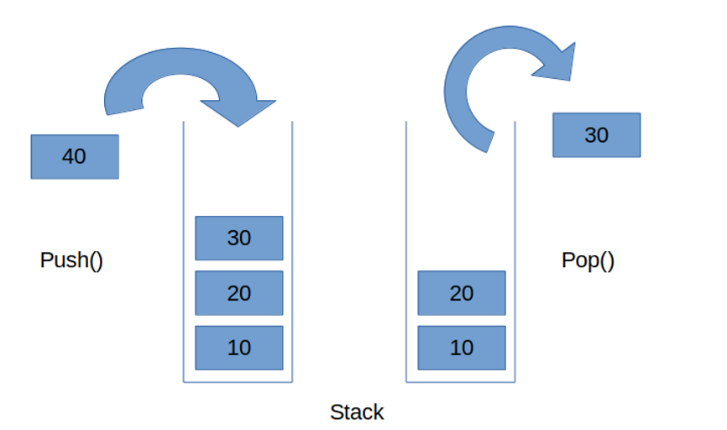
לדוגמא:

**בטבלת הסימנים –** שם משתנה\פונקציה (**key**) רשומה בטבלה (**value**)

**באוטומטDFA -** סימן בשפה (**key**) מיקומו במערך הסימנים (**value**)

## מחסנית

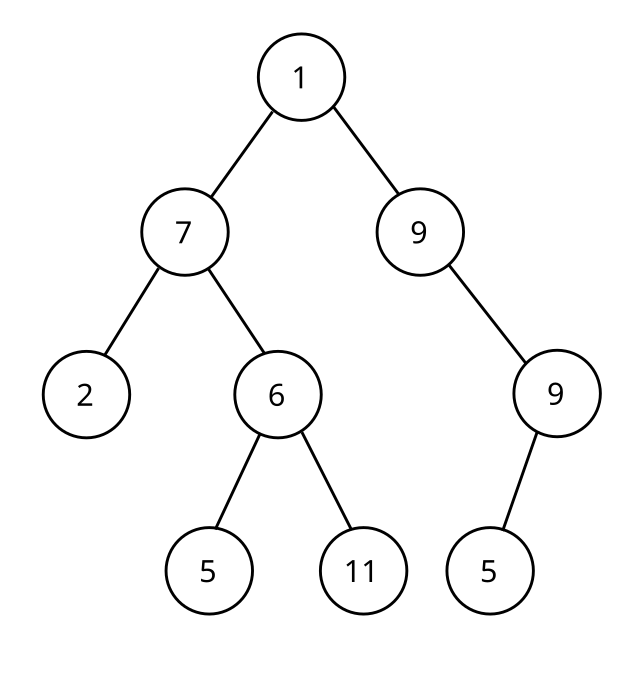
מחסנית הינו מבנה נתונים העובד בשיטת LIFO – Last in first out (הראשון בפנים האחרון בחוץ)

הוא מכיל שתי פעולות עיקריות push ו pop אשר משמשות להוצאה והכנסה של איברים מהמחסנית

**נשתמש מחסנית כחלק ממימוש המנתח המילוני**

## עץ

עץ בינארי הוא מבנה נתונים הבנוי מצמתים, לכל צומת יש אפס "בנים" או יותר המחוברים אילו בקשת כך שלא יכולה להיות קשת המובילה מהבן בחזרה לאב (**או לדרגה קדומה יותר**)



**דוגמא לעץ:**

**איך זה שימושי לנו?**

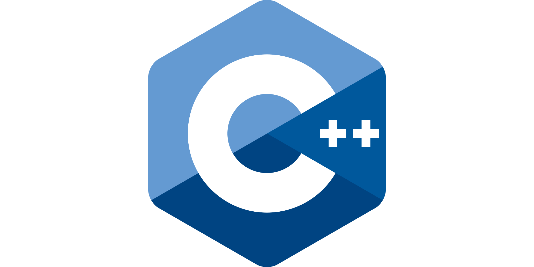
החלק המרכזי של המנתח התחבירי הוא ליצור עץ המתאר את המבנה התחבירי של הקוד. הקוד בשפה בנוי בצורה היררכית ועץ זאת דרך מעולה לשמור את הנתונים בצורה בעלת ב=המשמעות שאנחנו רוצים

## מערך

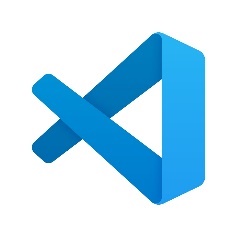
מערך הינו מבנה הנתונים הנפוץ ביותר. ובכללי בתכנות, נשתמש בו בלי סוף

מדובר באיחסון הנתונים בצורה רציפה בזיכרון, דבר המאפשר התעסקות קלה ושליפת איברים ב**O(1)**

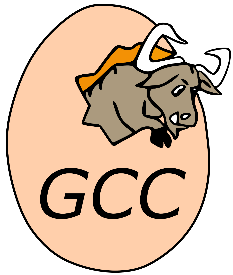
# תיאור סביבת העבודה ושפות התכנות



לפרויקט זה, החלטתי להשתמש בשפת ++C מכיוון שהיא מאפשרת גם עבודת Low **Level והתעסקות ישירה עם הזיכרון** וגם שימוש **בתכנות מונחה עצמים (OOP)** אשר נותן לכל ישות בקוד להיות עצם משל עצמו בעל תכונות ופונקציות.

ככה אוכל לשמור על ביצועים גבוהים בעוד שימוש בתכנות מונחה עצמים וכלים מודרניים יותר

את הקוד עצמו כתבתי בתוכנת **Visual Studio Code Version: 1.95.2**



הידרתי אותו באמצעות **GCC** (gnu C compiler) אשר באופן אירוני בהכנת הפרויקט יצא לי לחקור עליו בעצמי וללמוד עליו

# אלגוריתם ראשי

בפרק זה אתאר **בפסיאודו קוד** את האלגוריתם

## **אלגוריתם העל**

1. **אתחל את המנתח הלקסיקלי**
2. **אתחל את מדווח השגיאות**
3. **ייצר רשימת אסימונים מהקובץ הנתון**
4. **אם יש אסימון שגיאה דווח.**
5. **אתחל את המנתח המילוני**
6. **ייצר את העץ התחבירי**
7. **אתחל את המנתח סמנטי**
8. **עבירה על העץ הסמנטי**
9. **אתחל את מכולל קוד הסף**
10. **ייצר את קוד הסף**

## **אלגוריתם המנתח הלקסיקלי**

1. **שמור את שם הקובץ**
2. **אתחל את טבלת האוטומט**
3. **עד שלא קיבלנו אסימון עם סוג סוף קובץ** 
   1. **הוסף לרשימת האסימונים את האסימון הבא**

## מציאת אסימון הבא

1. **פתח את הקובץ**
2. **אתחל את המצב ההתחלתי**
3. **אם הגענו לסוף הקובץ** 
   1. **החזר אסימון סוף קובץ**
4. **עשה** 
   1. **קרא את התו הבא בקובץ**
   2. **קבל את המצב הבא**
   3. **אם המצב שקיבלנו לא תקין \ הגענו לסוף הקובץ**
      1. **צא מהלולאה**
   4. **אחרת**
      1. **עדכן את הפוזיציה של האסימון ואת תכולתו**
5. **חזור ל 5.2.**
6. **אם המצב שהגענו אילו הינו מצב דילוג** 
   1. **קרא את האסימון הבא**
7. **אחרת**
   1. **אם זהו מצב סופי תקין של האוטומט** 
      1. **החזר אסימון חדש אם סוג מתאים למצב הסיום ועם התכולה שנצברה**
   2. **אחרת** 
      1. **דווח שגיאה לקסיקלית למדווח השגיאות והחזר אסימון חדש עם סוג שגיאה**

## **אלגוריתם המנתח התחבירי**

1. **אתחל את הטבלה**
2. **אתחל את חוקי הדקדוק**
3. **אתחל את קבוצת הFOLLOW**
4. **אתחל את המחסנית**
5. **הוסף את מצב 1 לראש המחסנית**
6. **עשה**
   1. **הסתכל על המצב שבראש המחסנית**
   2. **קבל את הפעולה לפי האסימון הנוכחי והמצב שבראש המחסנית בטבלת הACTION**
   3. **אם פעולה הינה SHIFT**
      1. **בצע פעולת SHIFT**
   4. **אחרת אם הפעולה הינה REDUCE**
      1. **בצע פעולת REDUCE**
7. **כל עוד הפעולה היא לא ACCEPT חזור ל 6**
8. **הוצא מהמחסנית את האיבר הראשון (שורש העץ)**
9. **החזר את השורש**

### Shift

1. **דחוף את האסימון למחסנית**
2. **שנה את המצב הנוכחי לפי המספר בפעולה**
3. **עבור לאסימון הבא**

### Reduce

1. **טעינת חוק הדקדוק התואם לפעולה**
2. **הוצאת מספר איברים לפי גודל חוק הדקדוק**
3. **ייצר צומת חדשה שכל ילדיה הם האיברים שהוצאנו והערך שלו הוא הסימן בצד שמאל של החוק**
4. **אם הצומת החדשה מסוג יצירת משתנה הוסף רשומה לטבלת הסימנים**
5. **דחוף את הצומת החדשה למחסנית**
6. **חשב את מצב החדש לפי סוג הצומת והצומת הקודמת עם טבלת הGOTO**
7. **דחוף את המספר המתקבל**

## אלגוריתם המנתח הסמנטי

1. **לכל צומת בעץ**
   1. **אם הצומת הינה מסוג השמה**
      1. **בדוק האם הסוגים בהשמה מתאימים**
      2. **בדוק בטבלת הסימנים האם הסימן קיים בScope הנוכחי**
   2. **אם הצומת מסוג קריאה לפונקציה**
      1. **בדוק כי הפונקציה קיימת**
      2. **בדוק כי הפרמטרים טועמים**
   3. **אם הצומת מסוג הכרזת פונקציה** 
      1. **בדוק שהפונקציה לא קיימת**
   4. **אם הצומת מסוג return statement**
      1. **בדוק שסוג הצומת תואם להחזרה של הפונקצייה**
   5. **לכל שאר הצמתים**
      1. **התאם להם טיפוס בהתאם**
      2. **אם ישנה שגיה - דווח**

## אלגוריתם מכולל קוד הסף

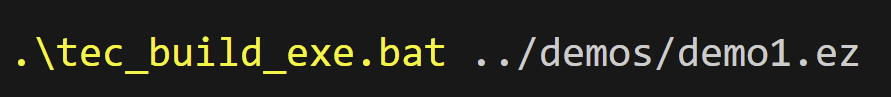
1. **לכל צומת בעץ**
   1. **ייצר קוד בהתאם**

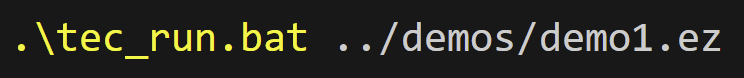
# תיאור ממשקים חיצוניים

קיימות שתי פקודות למשתמש:

**tec\_build\_exe.bat –** אשר יוצר קובץ exe מקובץ המקור הנתון לו

**tec\_run.bat** – אשר מריץ את קובץ המקור ומדפיס אותו למסך







# PlantUML Diagram

# תרשים מחלקות - UML

# מחלקות\פונקציות מרכזיות בפרויקט

## המחלקה DFA

מחלקה המייצגת אוטומט דטרמיניסטי סופי

|  |  |  |  |
| --- | --- | --- | --- |
| כותרת הפונקציה | הסבר | סיבוכיות | פסאודו קוד |
| DFA(int stateCount, vector<char> \*alphabet) | בנאי האוטומט, המאתחל אותו בעזרת מספר המצבים והשפה של האוטומט | O(1) | אתחל את מספר המצבים, ואת האלף-בית |
| DFA(string DFAConfigFile) | בנאי המאתחל את האוטומט לפי קובץ | O(n) | פירוק קובץ הconfig לחלקים ולהכניסן למטריצה |
| void insertTransition(int from, char alpha, int to) | מכניס מעבר למטריצה | O | **mat[from][char] = to** |

## המחלקה Lexer

מחלקה אשר ייעודה לחלק את קטע המקור אל אסימונים בעלי משמעות

|  |  |  |  |
| --- | --- | --- | --- |
| כותרת הפונקציה | הסבר | סיבוכיות | פסאודו קוד |
| Lexer(string srcFile, string DFAConfigFile, ErrorHandler \*handler) | בנאי הרכיב, הוא מקבל את קובץ המקור, קובץ הconfig של האוטומט ומצביע אל מדווח השגיאות | O(n) | שמור את שם קובץ המקור  אתחל את האוטומט  שמור את המצביע את מדווח השגיאות |
| SyntaxToken \*getNextToken() | מחזיר את האסימון הבא מהקלט | O(n) | פסואודו קוד מלא נכתב בפרק "אלגוריתם העל"  [כאן](#_אלגוריתם_המנתח_הלקסיקלי) |
| vector<SyntaxToken \*> getTokens(); | מחזיר את כל האסימונים מהקלט הנתון | O(n) | כל עוד לא הגענו לסוף הקובץ  הוסף לרשימה את האסימון הבא |

## המחלקה Parser

מחלקה אשר ייעודה לחלק את קטע המקור אל אסימונים בעלי משמעות

|  |  |  |  |
| --- | --- | --- | --- |
| כותרת הפונקציה | הסבר | סיבוכיות | פסאודו קוד |
| void shift(action currAction) | דוחף את האסימון הבא למחסנית | O(1) | צור חולייה בעץ בשביל האסימון  מצא את המספר של המצב שמתאים לאסימון לפי המספר הנתון מהטבלאה  אתחל את החוליה אם הנתונים האלו ודחוף למחסנית |
| void reduce(action currAction) | מצמצם את החוליות בראש המחסנית לחוליה אחת לפי כלל ייצור נתון | O(n) | מצא את הכלל לפי הפעולה הנתונה  הוצא מהמחסנית כמספר האיברים בצד ימין של הכלל  הפוך אותם לילים של החוליה החדשה  דחוף אותה למחסנית |
| ASTNode \*parse() | מחזיר את העץ ניתוח מופשט AST מהאסימונים הנוכחים | O(n) | פסואודו קוד מלא נכתב בפרק "אלגוריתם העל"  [כאן](#_אלגוריתם_המנתח_התחבירי) |

## המחלקה CodeGen

מחלקה אשר ייעודה לייצר את קוד הסף

|  |  |  |  |
| --- | --- | --- | --- |
| כותרת הפונקציה | הסבר | סיבוכיות | פסאודו קוד |
| void genCode() | ייצר את הקוד של התוכנית | O(n) | עבור כל חולייה יצר את הקוד שלה |
| int layoutLocals(scope \*s, int runningOffset) | מקצה זיכרון במחסנית למשתנים שבScope הנוכחי | O(n) | חשב לכל משתנה את הכתובת שלו  הוסף לoffset הכללי  עשה זאת לscopes הילדים |
| string getVarAddr(const &name) const; | מחזירה את הכתובת של המשתנה בזיכרון | O(n) | חפש בScope הנוכחי. אם המשתנה לא שם, חפש בScope האב |

## המחלקה Semantic

מחלקה אשר ייעודה הוא לבדוק את תקינות העץ התחבירי ולעדכן את טבלאת הסימנים

|  |  |  |  |
| --- | --- | --- | --- |
| כותרת הפונקציה | הסבר | סיבוכיות | פסאודו קוד |
| void updateSybolTable(ASTNode \*node) | מעדכן את טבלאת הסימנים ובודק שהסימן לא כבר קיים. אם כן ,ידווח על שגיאה | O(n) | לפי כל חולייה עדכן את הטבלאה כראוי |
| void updateScope(SyntaxToken\*currToken) | מעדכן את הscope הנוכחי לפי החולייה הנתונה | O(1) | אם הסימון הינו {   צור scope חדר  אם הסימן הינו }  צא מהscope הנוכחי |
| void assignNodeType(ASTNode\*node) | נותן לחולייה סוג ערך | O(1) | לכל חולייה בנפרד בחן אותה והבן איזה ערך היא צריכה לקבל |
| valType checkCompatibilityBinaryOp(valType leftOp, valType rightOp, SyntaxToken \*opToken) | בדוק האם שני סוגי הערכים הנתונים יכולים לתפקד ביחד עם הפעולה הנתונה | O(1) | זהה את אופרטור ההשוואה ואתה האופרנדים ובדוק את תיאומם |
| valType checkCompatibilityAssignExp(valType leftOp, valType rightOp, SyntaxToken \*opToken) | בדוק אם יכולה להתבצע השמה של צד ימיד לשמאל | O(1) | בדוק האם ההשמה יכולה להתבצע באזרת שני האופרנדים |
| void checkForMainFunction() | בודק את תקינות פונקציית הmain | O(1) | בדוק אם פונקציית הmain קיימת ומחזירה Int |
| void checkReturnStatements(functionEntry \*funcEntry) | בודק את תקינות משפטי ההחזרה מפונקציה | O(1) | בודק את תאימות מה שהמשפט מחזיר למה שהפונקציה מחזירה |

## המחלקה ErrorHandler

מחלקה אשר ייעודה לאגור רשימת שגיאות שיצטברו במהלך הקומפילציה

|  |  |  |  |
| --- | --- | --- | --- |
| כותרת הפונקציה | הסבר | סיבוכיות | פסאודו קוד |
| void addError(Error \*error) | הוספת שגיאה | O(1) | הוספת שגיאה למערך השגיאות |

## מחלקה Error

מחלקה אשר מייצגת שגיאה בקוד

למחלקה זאת משמשת כמחלקת אב להרבה מחלקות שגיאה אחרות אשר כל אחת מתייחסת לשגיאה מסוימת

|  |  |  |  |
| --- | --- | --- | --- |
| כותרת הפונקציה | הסבר | סיבוכיות | פסאודו קוד |
| Error(string body) | ייצר שגיאה חדשה | O(1) | אתחל את גוף השגיאה למה שניתן בקלט |
| virtual string toString() const = 0; | פונקציה אבסטרקטית אשר מחזירה את הייצוג המחרוזתי של השגיאה | O(1) | הפוך את השגיאה למחרוזת |

# מדריך למשתמש

אז איך משתמשים?

דבר ראשון צריך להתחיל לכתוב קוד! וכשהקוד מוכן, מצוין כל מה שיש לעשות הוא רק לקרוא לScript **tec\_run.bat** ופלט תוכנית תודפס למסך!



יש

תמונה שמכילה לוגו, גרפיקה, סמל, אומנות קליפיפם

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

# רפלקציה

אני יכול להגיד בלב שלב שהפרויקט הזה שיפר אותי כמתכנת וכתלמיד. הגעתי לפרויקט הזה מסקרנות מוחלטת ולא ידעתי למה אני מכניס את עצמי.

במסגרתי למדתי שלל דברים חדשים -

שפת תכנות חדשה, הבנתי שאני רוצה לבנות את הפרויקט בשפה שתיתן לי שליטת low-level אך הבנתי גם שאני רוצה גישה לעצמים. אז התשובה הייתה ברורה c++ אך ישנה בעיה אחת. לא ידעתי את השפה. למידת השפה לא רק הביאה לי כלים בפיתוח הפרויקט אך כלים בפיתוח ובעולם התוכנה בכללי. ++C הינה שפה אהובה מאוד בקרב מתכנתים ואנשי פיתוח ועקב זה יוצא לי להיתקל בא הרבה בין עם זה בהרצאות או סרטוני הסבר ועוד ועכשיו אני סוף סוף יכול להבין על מה הם מדברים. בנוסף לכך שנוספה עוד שפה לארגז הכלים שלי שזה תמיד דבר טוב

תאוריה, כמות התאוריה שאפשר ללמוד בתחום פיתוח הקומפיילרים הוא בלתי נגמר. זה נושא אחר נושא אחר נושא. הלמידה של דברים אלו פיתחה אותי כתלמיד, למדתי לספוג חומר מהר יותר וליישם אותו בצורה חלקה יותר והתעשרתי בידע רב בתחום הזה

אך נתקלתי גם בהרבה בעיות, היו הרבה פעמים שנתקעתי לפעמים אפילו לשבועות, אך תמיד הצלחתי למצוא את הטעות ולהמשיך אלאה עם זה בכוחות עצמי או בעזרה עם חברים. חוויה זו בהחלט נתנה לי כלים להמשיך גם אם יש כישלון ולא הכל נראה טוב

לסיכום. אני שמח שבחרתי בפרויקט זה והחוויה בפיתוחו הייתה מהנה ומעשירה.

# ביבליוגרפיה

**Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D.** (2006). *Compilers: Principles, techniques, and tools* (2nd ed.). Addison-Wesley.

**GeeksforGeeks.** (n.d.). *Compiler design tutorials*. <https://www.geeksforgeeks.org/compiler-design-tutorials/>

**Landwerth, I.** [terrajobst]. (n.d.). *Building a compiler* [Video playlist]. YouTube. <https://www.youtube.com/watch?v=wgHIkdUQbp0&list=PLRAdsfhKI4OWNOSfS7EUu5GRAVmze1t2y>

**Neso Academy.** (n.d.). *Compiler design* [Video playlist, 42 lessons]. YouTube. <https://www.youtube.com/playlist?list=PLBlnK6fEyqRjT3oJxFXRgjPNzeS-LFY-q>

**Thain, D.** (n.d.). *Compilers: Code generation* [Video]. YouTube. <https://www.youtube.com/watch?v=-ti07Z0xKKg&t=2778s&pp=ygUZY29tcGlsZXJzIGNvZGUgZ2VuYXJhdGlvbg%3D%3D>