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](#_Toc194264751)

[תקציר 3](#_Toc194264752)

[מושגים 4](#_Toc194264753)

[מושגים כללים 4](#_Toc194264754)

[תיאור הפרויקט 5](#_Toc194264755)

[תכולת השפה 5](#_Toc194264756)

# 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.

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

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):

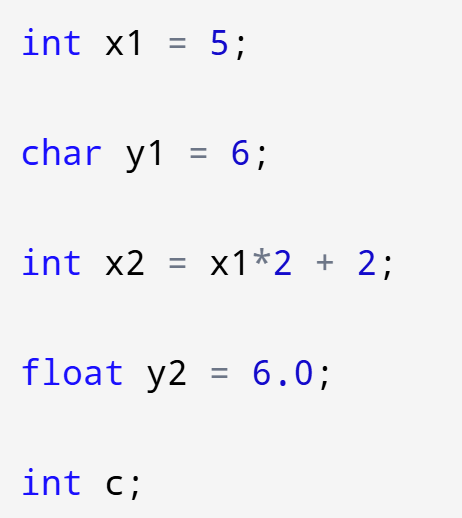
<**ערך אתחול ראשוני**> = <**שם המשתנה**> <**סוג המשתנה**>

**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

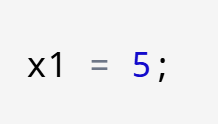


### השמת נתונים

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:

### השמה מורכבת

השמה מורכבת הינה השמה שבנוסף להשמה מתבצעת גם פעולה נוספת

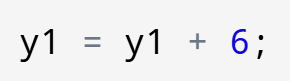
<**ערך אתחול ראשוני**> <**אופרטור השמה מורכב**> <**שם המשתנה**>

לדוגמא:

תמונה שמכילה גופן, גרפיקה, לבן, לוגו

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

הביטוי לא ישים את הערך 6 במשתנה y1 אלה יוסיף אילו 6 המהדר יפרש ביטוי זה כ –



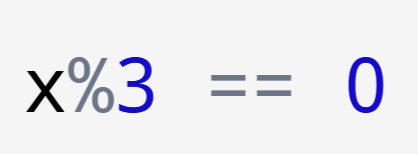
להלן כל האופרטורים להשמה מורכבת שהשפה כוללת:

* **=+** **מוסיף** למשתנה בצד ימין את הביטוי בצד שמאל
* **=-** **מחסר** למשתנה בצד ימין את הביטוי בצד שמאל
* **=/** **מחלק** למשתנה בצד ימין את הביטוי בצד שמאל
* **=\*** **כופל** למשתנה בצד ימין את הביטוי בצד שמאל
* **=~** **מבצע פעולת סיביות** **NOT** למשתנה בצד ימין את הביטוי בצד שמאל
* **=|** **מבצע פעולת סיביות** **OR** למשתנה בצד ימי את הביטוי בצד שמאל
* **=&** **מבצע פעולת סיביות AND** למשתנה בצד ימין את הביטוי בצד שמאל
* **=^** **מבצע פעולת סיביות XOR** למשתנה בצד ימין את הביטוי בצד שמאל

### השוואות

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

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



דוגמא 1:

תחזיר **אמת** אם המספר מתחלק בשלוש ללא שארית אחרת **שקר**

דוגמא 2:

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>**

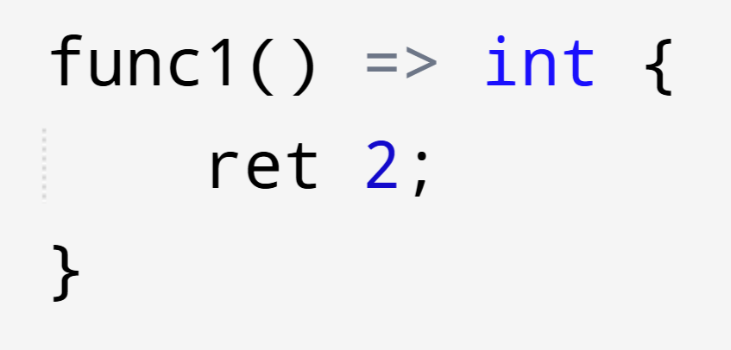
גוף הפונקצייה הינו אפס או יותר ביטויים אשר אחד מהביטויים הינו **ביטוי** **ret**

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

**דוגמאות לפונקציות:**

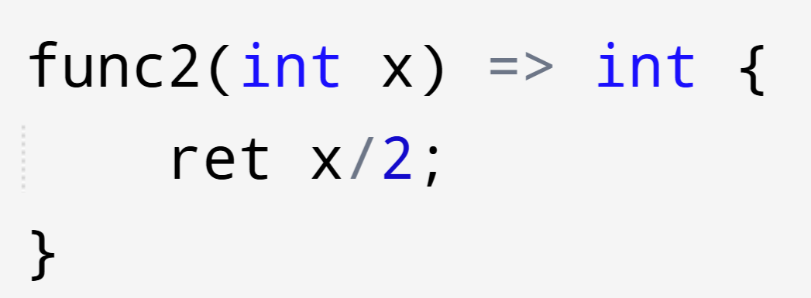
**דוגמא 1:**

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 func2תקרא עם הערך **6** ותחזיר את הערך **2**

### מצביעים

עוד בונוס קטן לשפה, **מצביעים** **ומערכים**.

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>**

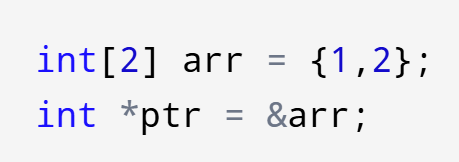
**הגדרת מערך:**

**< רשימת אתחול > = <סוג משתנה> [<מספר איברים במערך>] <שם משתנה>**

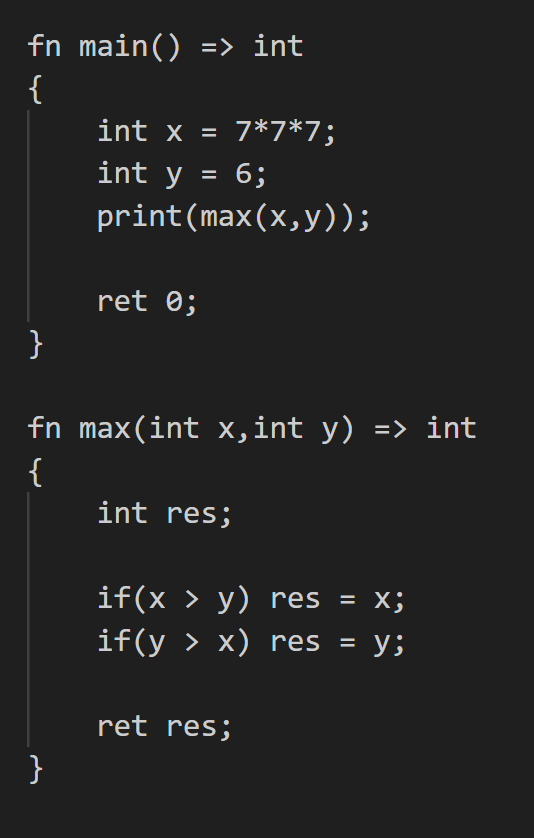
**גזירת מצביעה (dereferencing)**

**<שם מצביע>&**

**דוגמא לשימוש:**



## תוכנית לדוגמא בשפת easy

להלן תוכנית המחזירה את המספר הגדול ביותר :

## דקדוק השפה

מה הוא דקדוק?

דקדוק הינו אוסף של חוקים המגדירים את השפה.

להלן הדקדוק מוצג בBNF אשר הינו פורמט סטנדרטי לייצוג שפות

**<Program> ::= <FunctionDecl> | ε**

**<FunctionDecl> ::= "fn" IDENTIFIER "(" <ParamList> ")" "->" <Type> "{" <StmtList> "}"**

**<ParamList> ::= <Param> <ParamTail> | ε**

**<ParamTail> ::= "," <Param> <ParamTail> | ε**

**<Param> ::= <Type> IDENTIFIER**

**<Type> ::= <BaseType> <TypeTail>**

**<BaseType> ::= "int" | "float" | "char"**

**<TypeTail> ::= "[" INTEGER\_LITERAL "]" | "\*" | ε**

**<StmtList> ::= <Stmt> <StmtList> | ε**

**<Stmt> ::= <VarDeclStmt> | <AssignStmt> | <IfStmt> | <WhileStmt> | <ForStmt> | <ReturnStmt> | <ExprStmt>**

**<Body> ::= "{" <StmtList> "}" | <Stmt>**

**<VarDeclStmt> ::= <VarDeclExpr> ";"**

**<VarDeclExpr> ::= <Type> IDENTIFIER <InitOpt>**

**<InitOpt> ::= "=" <AssignValue> | ε**

**<AssignValue> ::= <Expr> | "{" <ExprList> "}"**

**<AssignStmt> ::= <AssignExpr> ";"**

**<AssignExpr> ::= <AssignTarget> <AssignOp> <Expr>**

**<AssignTarget> ::= IDENTIFIER | IDENTIFIER "[" <Expr> "]" | "\*" IDENTIFIER**

**<AssignOp> ::= "=" | "+=" | "-=" | "/=" | "\*=" | "&=" | "|=" | "^=" | "~="**

**<IfStmt> ::= "if" "(" <ConditionOp> ")" <Body> | "if" "(" <ConditionOp> ")" <Body> "else**

**<WhileStmt> ::= "while" "(" <ConditionOp> ")" <Body>**

**<ConditionOp> ::= <Expr> | <AssignExpr>**

**<ForStmt> ::= "for" "(" <ForInit> ";" <ExprOpt> ";" <ForUpdate> ")" <Body>**

**<ForInit> ::= <VarDeclExpr> | <AssignExpr> | ε**

**<ExprOpt> ::= <Expr> | ε**

**<ForUpdate> ::= <AssignExpr> | ε**

**<ReturnStmt> ::= "return" <ExprOpt> ";"**

**<ExprStmt> ::= <Expr> ";"**

**<ExprList> ::= <Expr> <ExprTail> | ε**

**<ExprTail> ::= "," <Expr> <ExprTail> | ε**

**<Expr> ::= <UnaryExpr> <RelOpTail> | <PointerRefExpr>**

**<RelOpTail> ::= <RelOp> <UnaryExpr> <RelOpTail> | ε**

**<RelOp> ::= "<" | ">" | "<=" | ">=" | "==" | "!="**

**<UnaryExpr> ::= <UnaryOp> <UnaryExpr> | <PostIncrement> | <PreIncrement> | <SimpleExpr>**

**<PointerRefExpr> ::= "&" IDENTIFIER | "\*" IDENTIFIER**

**<PostIncrement> ::= IDENTIFIER "++" | IDENTIFIER "--"**

**<PreIncrement> ::= "++" IDENTIFIER | "--" IDENTIFIER**

**<UnaryOp> ::= "-" | "!"**

**<SimpleExpr> ::= <Term> <AddOpTail>**

**<AddOpTail> ::= <AddOp> <Term> <AddOpTail> | ε**

**<AddOp> ::= "+" | "-" | "|" | "^" | "||"**

**<Term> ::= <Factor> <MulOpTail>**

**<MulOpTail> ::= <MulOp> <Factor> <MulOpTail> | ε**

**<MulOp> ::= "\*" | "/" | "&" | "&&"**

**<Factor> ::= "(" <Expr> ")" | IDENTIFIER | INTEGER\_LITERAL | FLOAT\_LITERAL | CHAR\_LITERAL | IDENTIFIER "(" <ExprList> ")"**

# 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 וTSEM 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

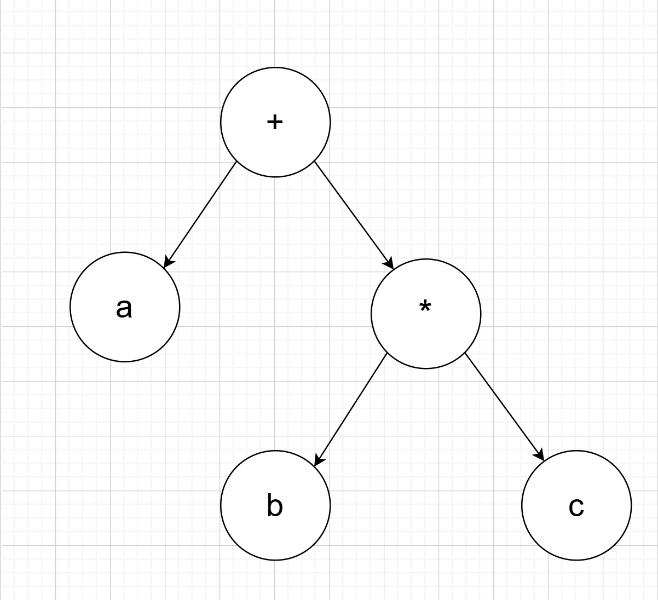
* Int (מילה שמורה)
* x (מזהה)
* = (אופרטור)
* 7 (מספר)
* ; (נקודה פסיק)

## 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**

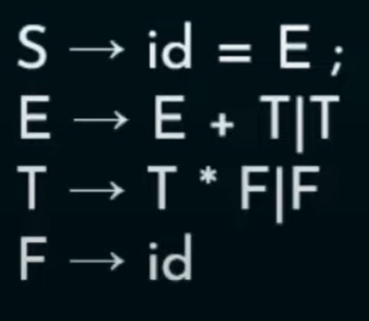
**סופיים**: סימנים אשר ניתן להרחיב אותם לפי כלל דקדוק

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

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

**חוק 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;**

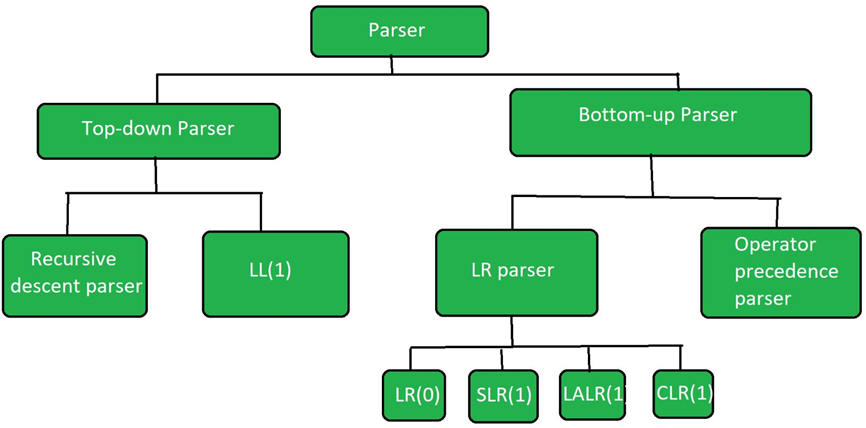
לפי הדקדוק הנתון יתקבל העץ:

תמונה שמכילה קו, תרשים

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

## **סוגי מנתחים תחביריים**

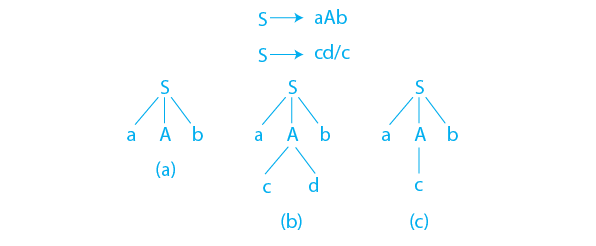
המנתח התחבירי הוא הרכיב המסובך ביותר במהדר. עקב זאת הוא בעל הכי הרבה תאוריה ומחקר בין כל הרכיבים

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:

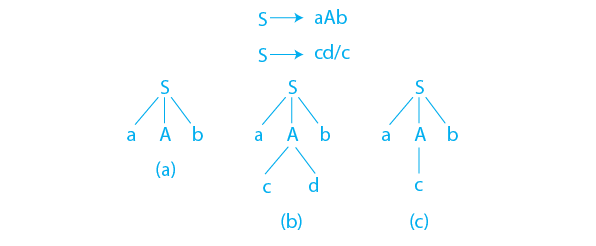


ניתוח top – down יוכל לפתח את העץ בצורה הבאה:

**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

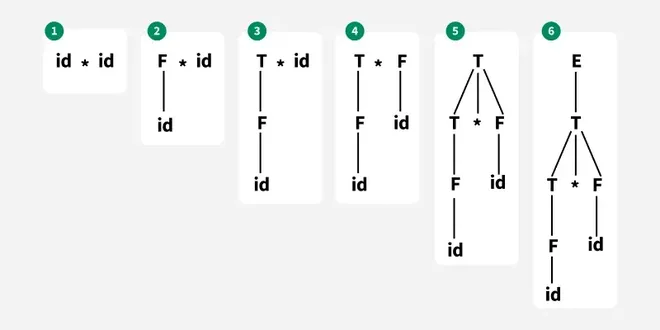
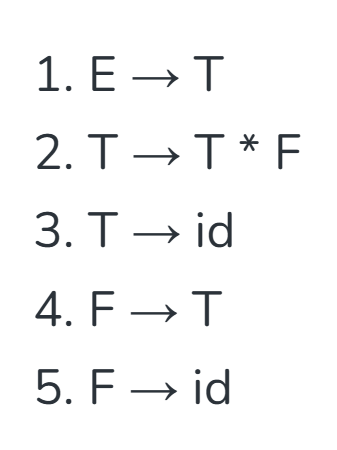
**(1)LALR (look-ahead, left-to-right)** – עובד בדומה ל **(1)LR** אך יוצר טבלה קטנה יותר וכך חוסך מקום זיכרון

**(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

## 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.