עבודת גמר

לקבלת תואר טכנאי תוכנה

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

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):

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

<u>או:</u>

<סוג המשתנה> <שם המשתנה>;

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

דוגמאות לאתחול נתונים בשפה

```
int x1 = 5;

char y1 = 6;

int x2 = x1*2 + 2;

float y2 = 6.0;

int c;
```

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:

```
x1 = 5;
```

דוגמה להשמה לא תקנית:

```
x1;
```

השמה מורכבת

השמה מורכבת הינה השמה שבנוסף להשמה מתבצעת גם פעולה נוספת

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

לדוגמא:

```
y1 += 6;
```

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

```
y1 = y1 + 6;
```

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:

```
x%3 == 0
```

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

דוגמא 2:

```
3 != 0
```

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)

לדוגמא:

```
x%3 == 0 && x%2 == 0
```

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:

```
if(x) {x = 5;}
```

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

```
func1() => int {
    ret 2;
}
```

דוגמא 2:

פונקציה המקבלת מספר ומחזירה את החצי שלו

```
func2(int x) => int {
    ret x/2;
}
```

קריאה לפונקציה

קריאה לפונקציה תראה כך

(<פרמטרים>)<שם הפונקציה>

לדוגמא:

```
func2(6)
```

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:

```
int[2] arr = {1,2};
int *ptr = &arr;
```

תוכנית לדוגמא בשפת easy

להלן תוכנית המחזירה את המספר הגדול ביותר :

```
fn main() => int
{
    int x = 7*7*7;
    int y = 6;
    print(max(x,y));

    ret 0;
}

fn max(int x,int y) => int
{
    int res;

    if(x > y) res = x;
    if(y > x) res = y;

    ret res;
}
```

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>

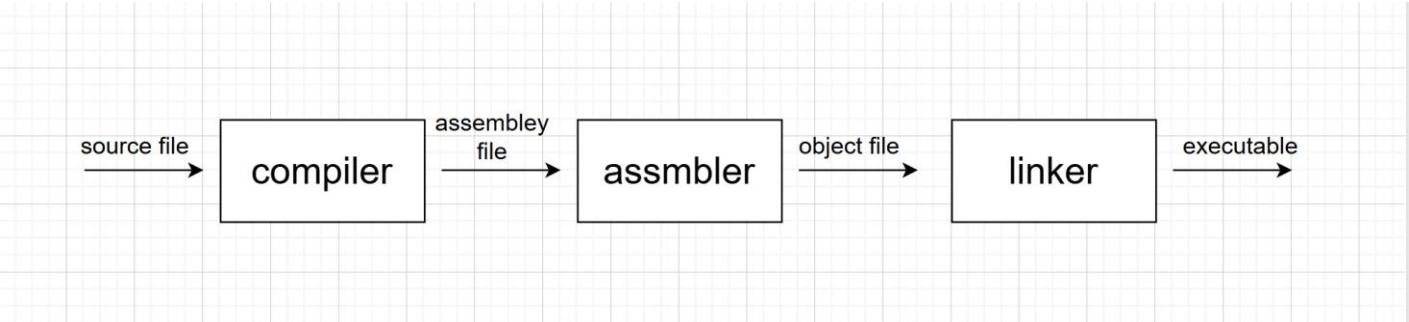
"&&" | "&" | "/" | "*" =:: <MulOp>

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



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

```
1  int main()
2  {
3      int x = 10;
4      int y = 20;
5
6      x = x+y;
7
8      return 0;
9  }
```

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:

```
        .file   "dugma.c"
        .def    ___main;    .scl    2;  .type   32; .endef
        .text
        .globl  _main
        .def    _main;  .scl    2;  .type   32; .endef
_main:
LFB0:
        .cfi_startproc
        pushl   %ebp
        .cfi_def_cfa_offset 8
        .cfi_offset 5, -8
        movl    %esp, %ebp
        .cfi_def_cfa_register 5
        andl    $-16, %esp
        subl    $16, %esp
        call    ___main
        movl    $10, 12(%esp)
        movl    $20, 8(%esp)
        movl    8(%esp), %eax
        addl    %eax, 12(%esp)
        movl    $0, %eax
        leave
        .cfi_restore 5
        .cfi_def_cfa 4, 4
        ret
        .cfi_endproc
LFE0:
        .ident  "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"
```

אז נשתמש בעוד מהדר הממיר קוד סף אל שפת מכונה

(דוגמאות למהדרים כאלו: NASM לארכיטקטורת x86_64 וTSEM לארכיטקטורת x86)

```
4C 01 05 00 00 00 00 00 7C 01 00 00 0F 00 00 00 00 00 04 01 2E 74 65 78 74 00 00 00 00 00 00 00 00 00 00 00
30 00 00 00 DC 00 00 00 68 01 00 00 00 00 00 00 01 00 00 00 20 00 30 60 2E 64 61 74 61 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 30 C0 2E 62 73 73 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 80 00 30 C0 2F 34 00 00
00 00 00 00 00 00 00 00 00 00 00 00 24 00 00 00 0C 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 40 00 30 40
2F 31 35 00 00 00 00 00 00 00 00 00 00 00 00 00 38 00 00 00 30 01 00 00 72 01 00 00 00 00 00 00 01 00 00 00
40 00 30 40 55 89 E5 83 E4 F0 83 EC 10 E8 00 00 00 00 C7 44 24 0C 0A 00 00 00 C7 44 24 08 14 00 00 00 8B 44
24 08 01 44 24 0C B8 00 00 00 00 C9 C3 90 90 90 47 43 43 3A 20 28 4D 69 6E 47 57 2E 6F 72 67 20 47 43 43 2D
36 2E 33 2E 30 2D 31 29 20 36 2E 33 2E 30 00 00 14 00 00 00 00 00 00 00 01 7A 52 00 01 7C 08 01 1B 0C 04 04
88 01 00 00 1C 00 00 00 1C 00 00 00 04 00 00 00 2D 00 00 00 00 41 0E 08 85 02 42 0D 05 69 C5 0C 04 04 00 00
0A 00 00 00 0E 00 00 00 14 00 20 00 00 00 04 00 00 00 14 00 2E 66 69 6C 65 00 00 00 00 00 00 00 FE FF 00 00
67 01 64 75 67 6D 61 2E 63 00 00 00 00 00 00 00 00 00 00 00 5F 6D 61 69 6E 00 00 00 00 00 00 00 01 00 20 00
02 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00 00 00 00 00 01 00 00 00
03 01 2D 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 64 61 74 61 00 00 00 00 00 00 00 02 00 00 00
03 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2E 62 73 73 00 00 00 00 00 00 00 00 03 00 00 00
03 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 19 00 00 00 00 00 00 00 04 00 00 00
03 01 23 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 24 00 00 00 00 00 00 00 05 00 00 00
03 01 38 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00 5F 5F 5F 6D 61 69 6E 00 00 00 00 00 00 00 20 00
02 00 2E 00 00 00 2E 72 64 61 74 61 24 7A 7A 7A 00 2E 65 68 5F 66 72 61 6D 65 00 2E 72 64 61 74 61 24 7A 7A
7A 00 2E 65 68 5F 66 72 61 6D 65 00
```

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,

ה-front end וה-beck 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

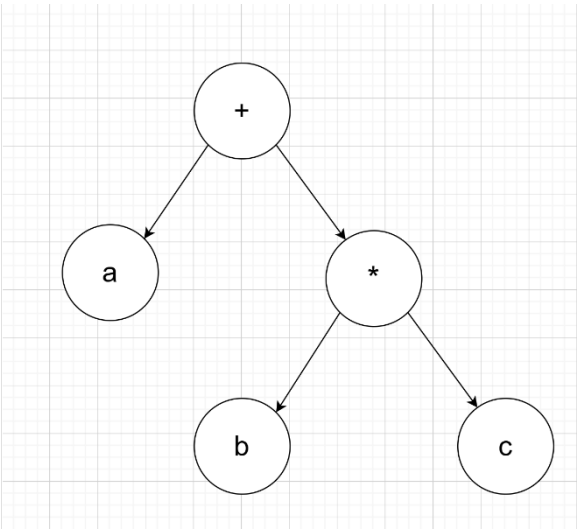
- **Int** (מילה שמורה)
- **x** (מזהה)
- **=** (אופרטור)
- **7** (מספר)
- **;** (נקודה פסיק)

ניתוח תחבירי (Syntax Analysis)

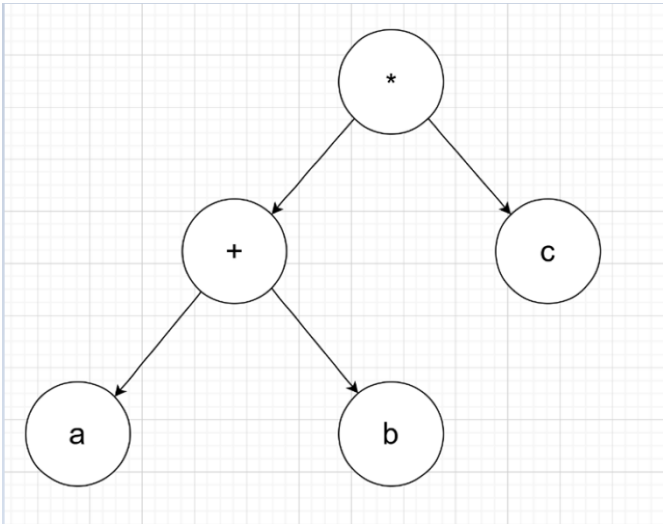
מטרה: השלב הבא הוא לבדוק אם רצף הטוקנים שנוצר בתהליך הקודם תואם לחוקי התחביר של השפה (הדקדוק שלה). השלב הזה יוצר את ה-עץ תחבירי או ה-עץ סמנטי (abstract syntax tree - AST), שמייצג את מבנה הקוד לפי כללי השפה.

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

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:

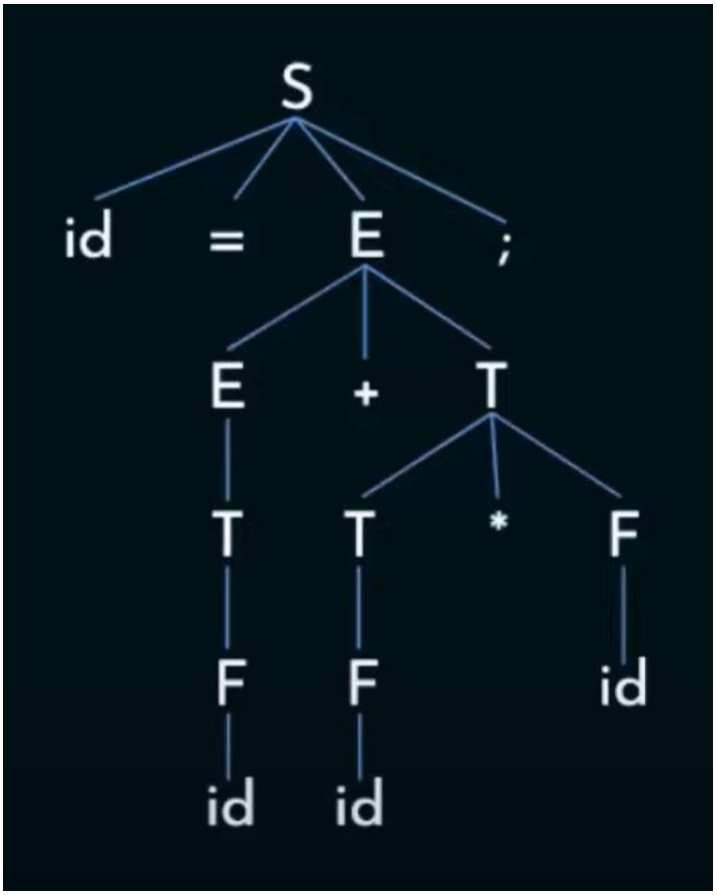
$$S \rightarrow id = E\ ;$$
$$E \rightarrow E + T|T$$
$$T \rightarrow T * F|F$$
$$F \rightarrow id$$

הסימנים הסופיים בדקדוק: S,E,T,F

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

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

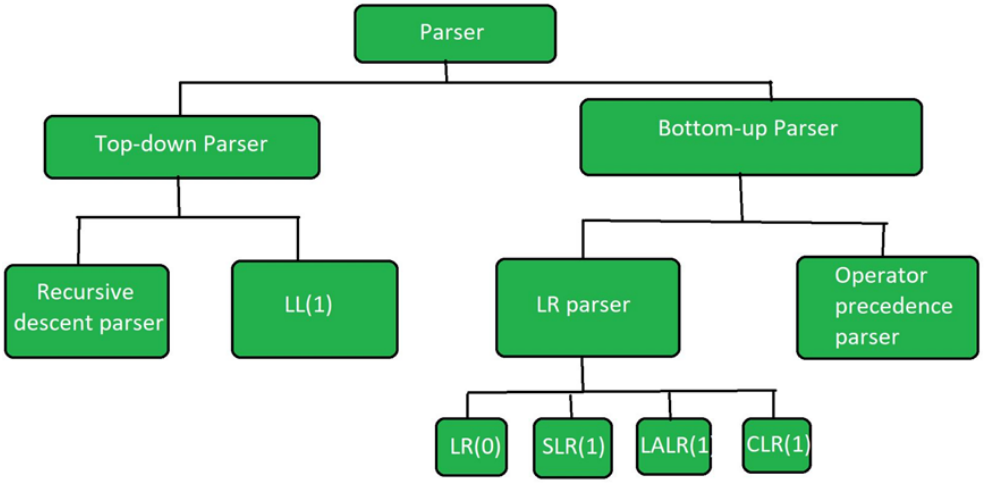
לפי הדקדוק הנתון יתקבל העץ:



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)

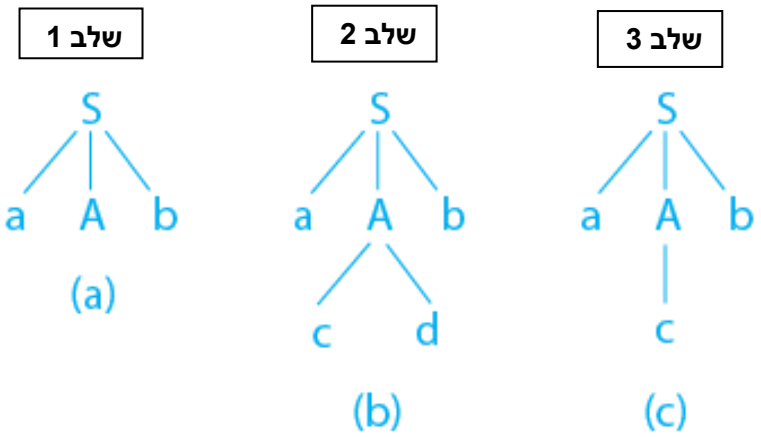
ניתוח תחבירי מלמעלה למטה מתחיל עם כלל התחבירי הראשון ומנסה לפתח אותו אל ה non terminals השונים בשפה

אם ניקח לדוגמא את הgrammar הבא:

S ⟶ aAb

S ⟶ cd/c

ניתוח top – down יוכל לפתח את העץ בצורה הבאה:



מנתחי bottom-up

<u>מנתחי LR (left-right)</u>

משפחה של מנתחם תחביריים המנתחים מלמטה למעלה ומשתמשים באוטומט מחסנית pda

LR0) – מנתח פשוט המשתמש הלא יודע להתמודד עם דקדוקים מסובכים ומורכבים מידי

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

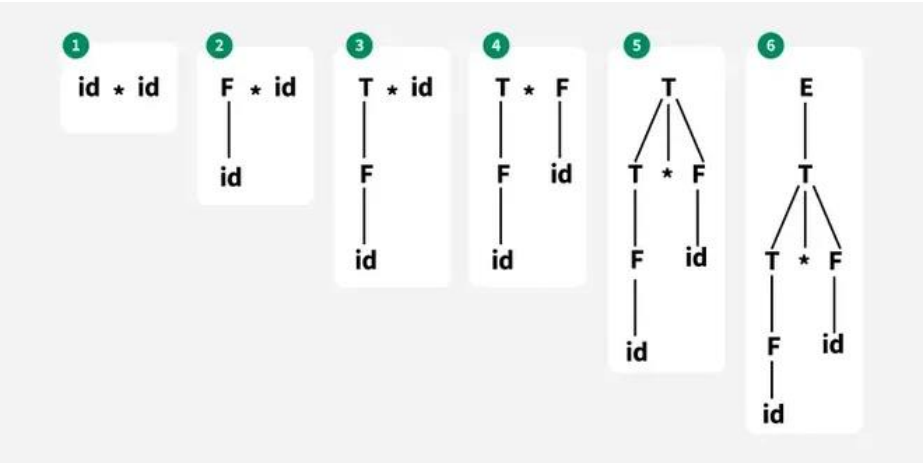
LALR(1) (look-ahead, left-to-right) – עובד בדומה ל **LR(1)** אך יוצר טבלה קטנה יותר וכך חוסך מקום זיכרון

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

<u>מלמטה למעלה (bottom- up)</u>

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

הgrammar הבא: **ינותח בצורה הבאה:**



1. E → T
2. T → T * F
3. T → id
4. F → T
5. F → id

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:

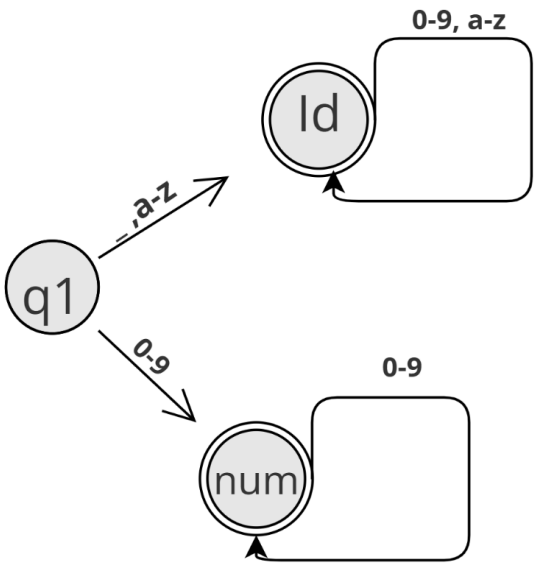
האסטרטגיה לפיתוח המנתח הסמנטי

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 בצומת Id 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:

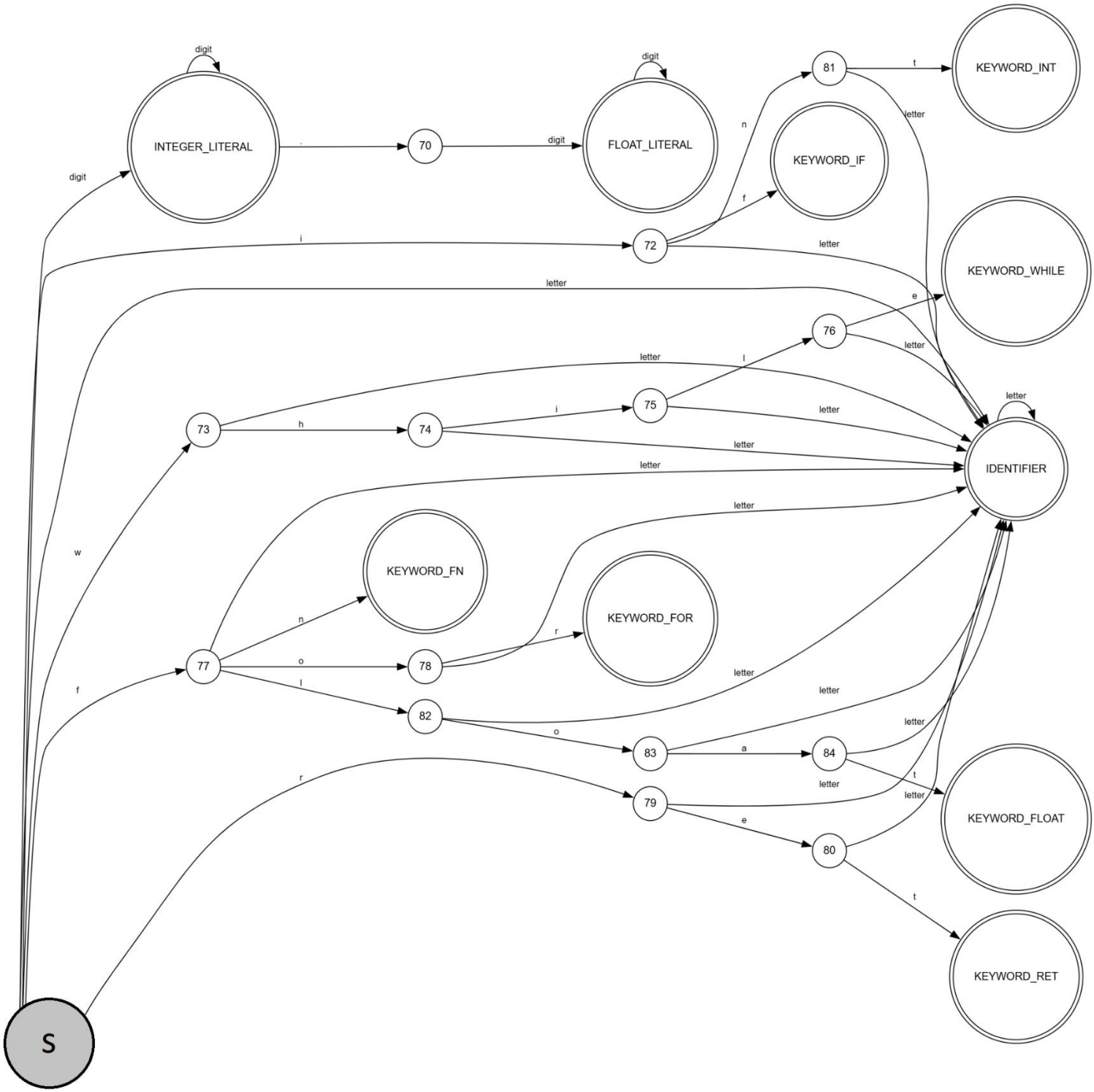
מצב התחלתי (מסומן ב-**S**) – מסמל את המצב הראשוני לסריקת אסימון לפני קבלת כל אות

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

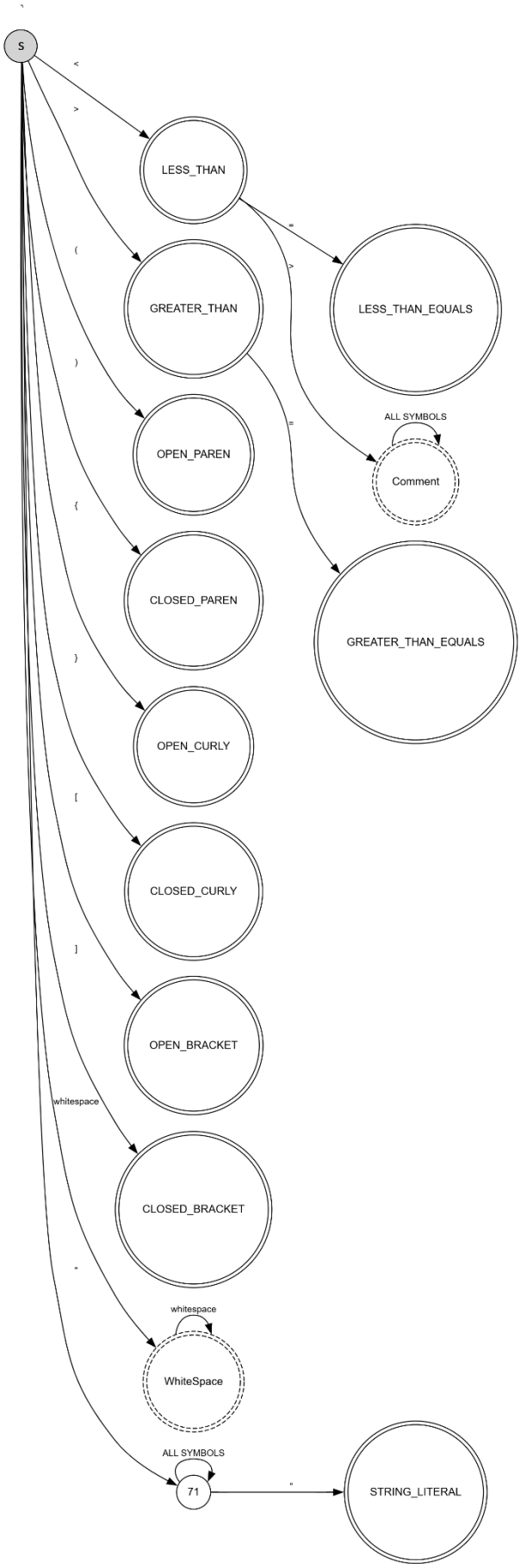
מצבי סיום (מסומנים בעיגול כפול) – מסמלים מצביי קבלה/סיום של האוטומט. סיום הסריקה בהם תסתמן כאסימן תקין

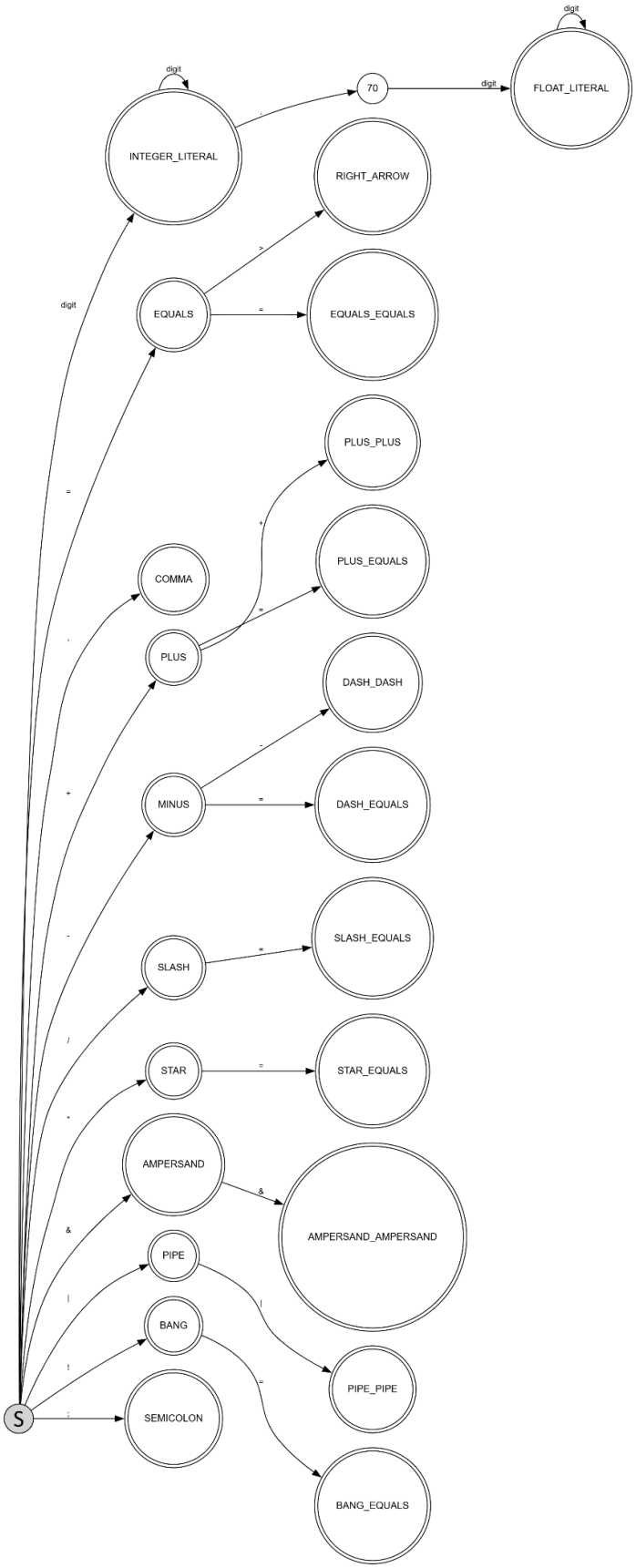
מצבי דילוג (מסומנים בעיגול מקווקו) – מסמלים מצביי דילוג של האוטומט. סיום הסריקה בהם לא תחשב ככלום במהדר

חלק האוטומט הקולט מזהים (שמות משתנים ופונקציות),מחרוזות,מספרים שלמים, מספרים עשרוניים ומילות מפתח:



חלק האוטומט הקולט את שלל סימני השפה

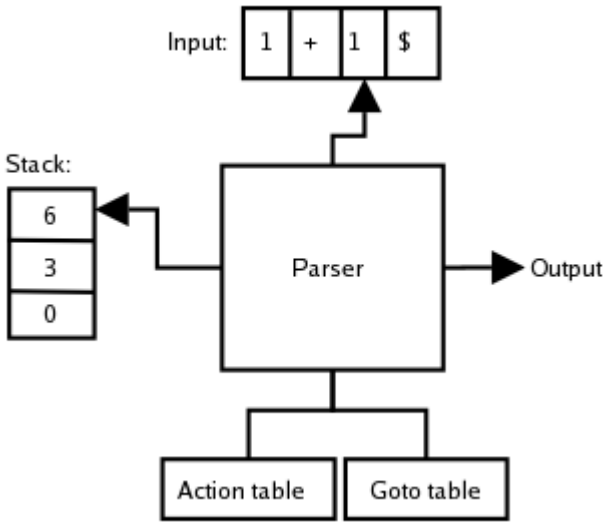




האסטרטגיה לפיתוח המנתח התחבירי

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:

```
(0) S' -> S
(1) S -> C C
(2) C -> c C
(3) C -> d
```

בעזרתו נבנה את טבלאות הGOTO וACTION

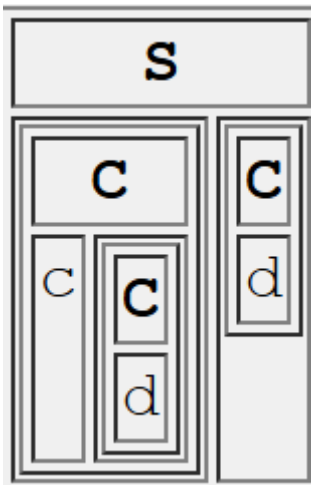
LR table						
State	ACTION			GOTO		
	c	d	$	S'	S	C
0	s3	s4			1	2
1			acc			
2	s6	s7				5
3	s3	s4				8
4	r_3	r_3				
5			r_1			
6	s6	s7				9
7			r_3			
8	r_2	r_2				
9			r_2			

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

המעקב יראה ככה:

Step	Stack	Input	Action
1	0	c d d $	s3
2	0 c 3	d d $	s4
3	0 c 3 d 4	d $	r_3
4	0 c 3 C	d $	8
5	0 c 3 C 8	d $	r_2
6	0 C	d $	2
7	0 C 2	d $	s7
8	0 C 2 d 7	$	r_3
9	0 C 2 C	$	5
10	0 C 2 C 5	$	r_1
11	0 S	$	1
12	0 S 1	$	acc

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?

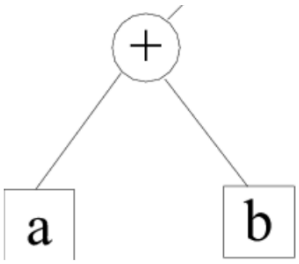
ועוד...

האסטרטגיה לפיתוח שלב כתיבת קוד הסף

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

<u>פעולות אריתמטיות</u>

החלק בעץ:



יהפוך ל:

```
MOV RAX, [b]
ADD RAX, [c]
MOV [a], RAX
```

תנאים

נשתמש במבנה בסיסי של שפת סף לתנאים

הביטוי:

```
if (x == 5) {
    <קוד>
}
```

יהפוך ל:

```
mov     eax, DWORD PTR [x]
    cmp     eax, 5
    jne     .SOF
    <קוד התנאי>
.SOF:
```

פונקציות:

יצירה:

האוגרים RDI, RSI, RDX, RCX, R8, R9– משמשים להעברת ששת הארגומנטים הראשונים של הפונקציה.

ניקח לדוגמא פונקציה.

```
int func1(int a, int b) {<קוד>}
```

נוכל ליצור אותה באמצעות יצירת label חדש וסיום בשימוש הפקודה ret

```
func1:
    <קוד הפונקציה>
    RET
```

כאשר **a**-ו**b** שמורים ב**RDI** ו-**RSI**

קריאה לפונקציה:

```
CALL func1
```

לולאת for

```
for (int i = 0; i < 10; i++) {<גוף הלולאה>}
```

יתורגם ל:

```
MOV RAX, 0
MOV [i], RAX

for_start:
    MOV RAX, [i]
    CMP RAX, 10
    JGE for_end

    <גוף הלולאה>

    INC RAX
    MOV [i], RAX

    JMP for_start

for_end:
```

לולאות while

```
while (x > 0) {<גוף הלולאה>}
```

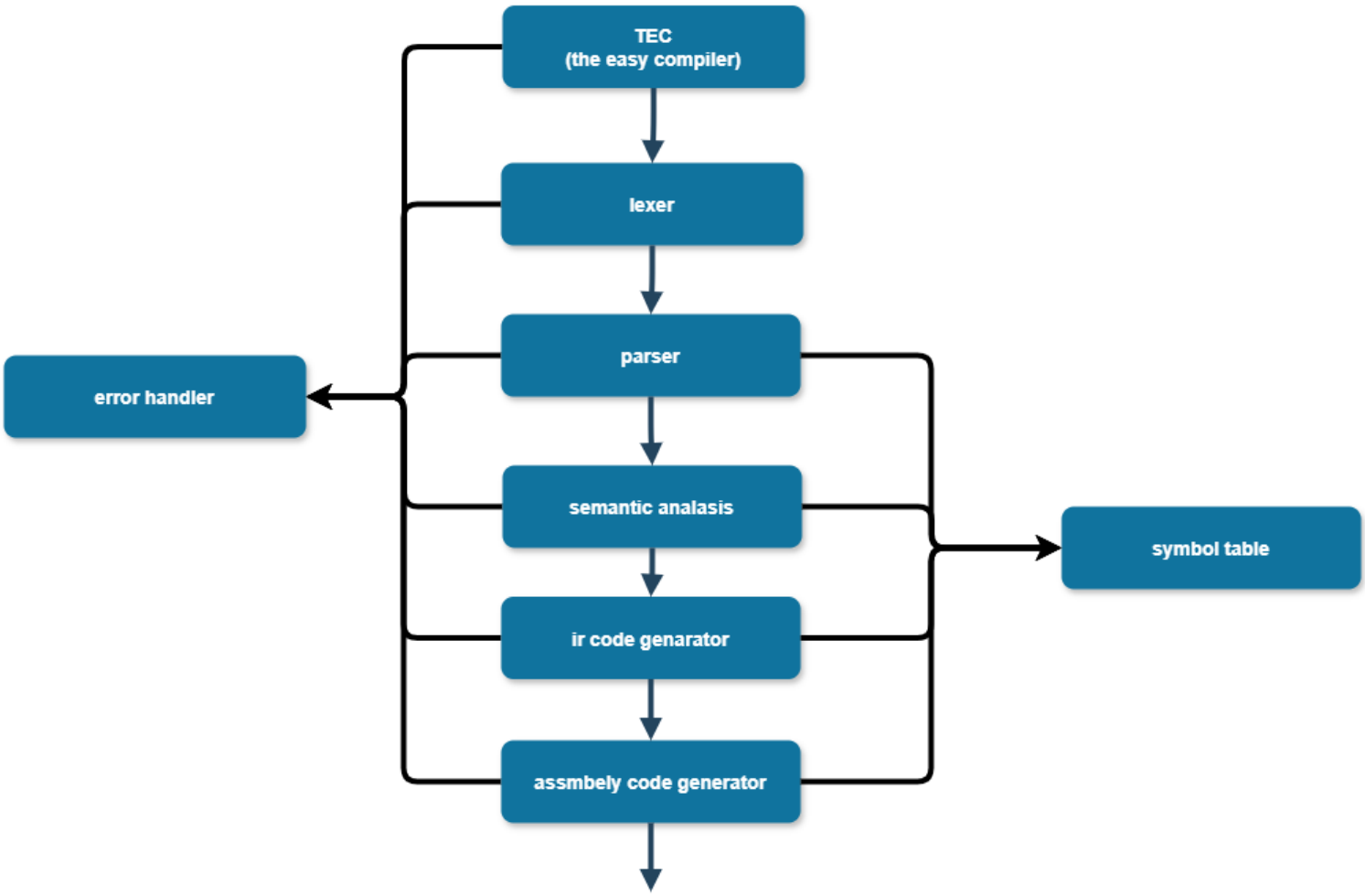
יתורגם ל:

```
MOV RAX, [x]

loop_start:
    CMP RAX, 0
    JLE loop_end

    <גוף הלולאה>

    MOV [x], RAX

    JMP loop_start
```

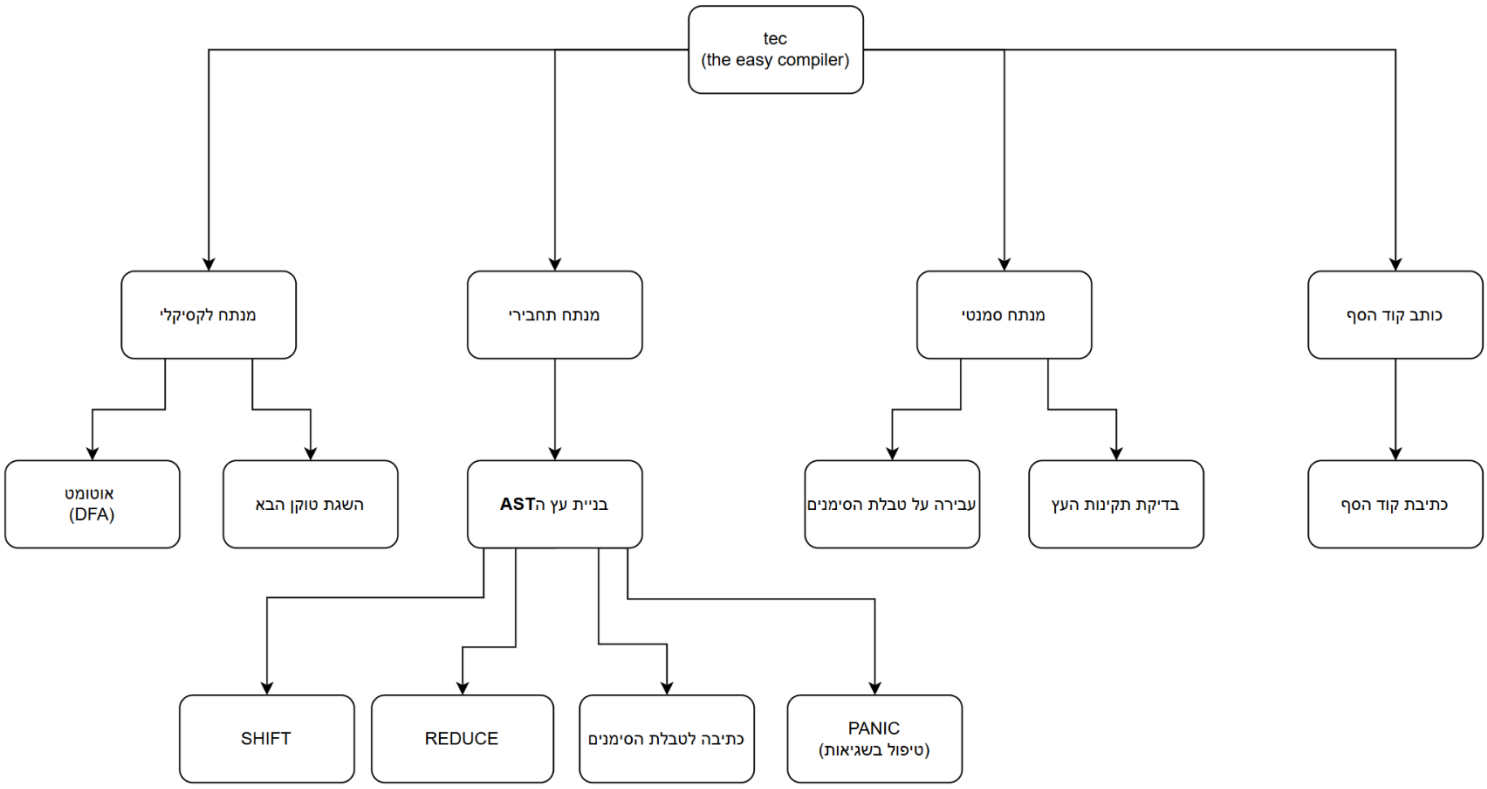
ארכיטקטורה של הפתרון המוצע בפורמט של

Top-Down Level Design

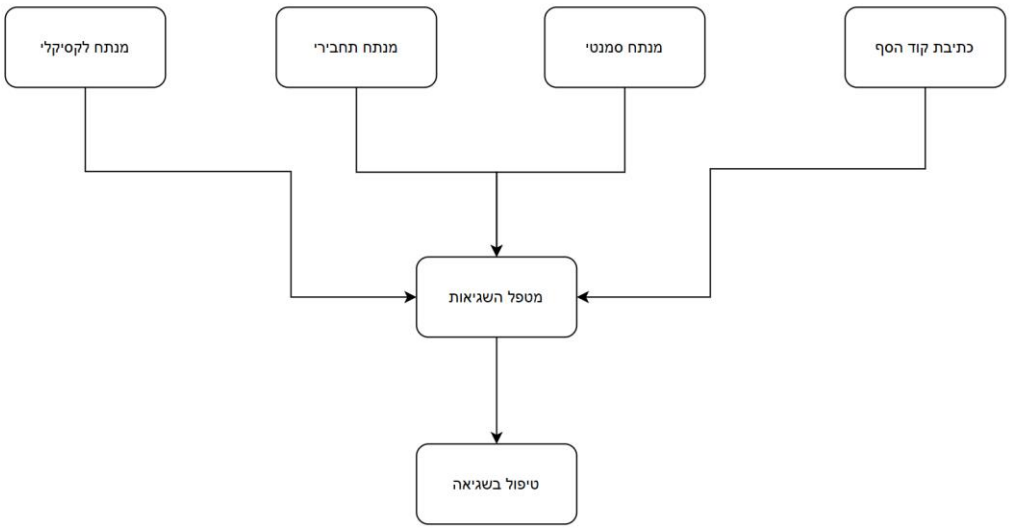
להלן תרשים זרימה המתאר את החלקים השונים ואת התקשורת ביניהם בפורמט השקפת על (High level):



תרשים Top-down מפורט



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



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)

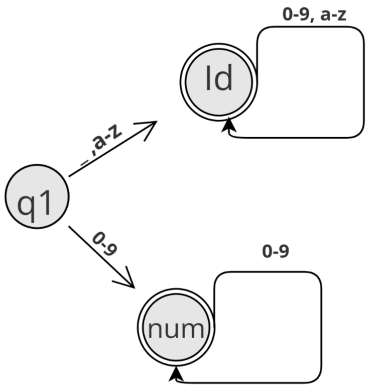
למה? – זהו מבנה נתונים אשר קל לעבוד איתו ולהכניס אילו נתונים.

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:

STATE	NUMBER	LETTER	_
q1	num	Id	Id
num	num	TRAP	TRAP
id	Id	Id	TRAP

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**)

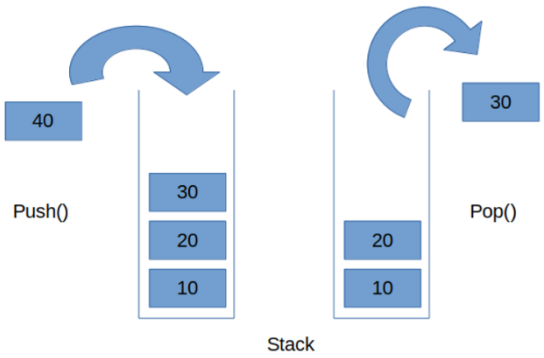
באוטומט DFA - סימן בשפה (**key**) מיקומו במערך הסימנים (**value**)

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

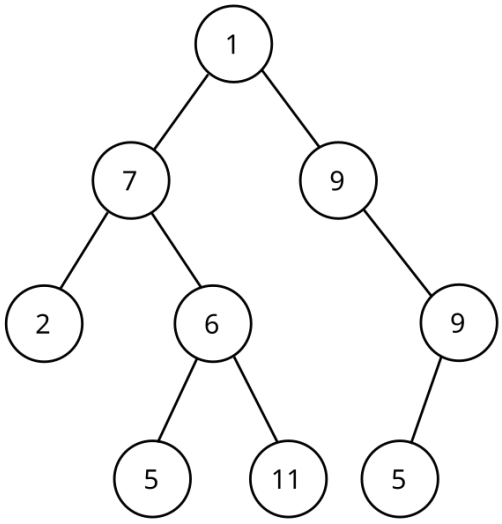
נשתמש מחסנית כחלק ממימוש המנתח המילוני



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**
   3.1 **הוסף לרשימת האסימונים את האסימון הבא**

מציאת אסימון הבא

1. **פתח את הקובץ**
2. **אתחל את המצב ההתחלתי**
3. **אם הגענו לסוף הקובץ**
   3.1 **החזר אסימון סוף קובץ**
4. **עשה**
   4.1 **קרא את התו הבא בקובץ**
   4.2 **קבל את המצב הבא**
   4.3 **אם המצב שקיבלנו לא תקין \ הגענו לסוף הקובץ**
      4.3.1 **צא מהלולאה**
   4.4 **אחרת**
      4.4.1 **עדכן את הפוזיציה של האסימון ואת תכולתו**
5. **חזור ל 5.2.**
6. **אם המצב שהגענו אילו הינו מצב דילוג**
   6.1 **קרא את האסימון הבא**
7. **אחרת**
   7.1 **אם זהו מצב סופי תקין של האוטומט**
      7.1.1 **החזר אסימון חדש אם סוג מתאים למצב הסיום ועם התכולה שנצברה**
   7.2 **אחרת**
      7.2.1 **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**
   - **6.1 הסתכל על המצב שבראש המחסנית**
   - **6.2 קבל את הפעולה לפי האסימון הנוכחי והמצב שבראש המחסנית בטבלת הACTION**
   - **6.3 אם פעולה הינה <u>SHIFT</u>**
     - **6.3.1 בצע פעולת SHIFT**
   - **6.4 אחרת אם הפעולה הינה <u>REDUCE</u>**
     - **6.4.1 בצע פעולת REDUCE**
7. **כל עוד הפעולה היא לא <u>ACCEPT</u> חזור ל <u>6</u>**
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**
   - **1.1 אם הצומת הינה מסוג השמה**
     - **1.1.1 בדוק האם הסוגים בהשמה מתאימים**
     - **1.1.2 בדוק בטבלת הסימנים האם הסימן קיים בScope הנוכחי**
   - **1.2 אם הצומת מסוג קריאה לפונקציה**
     - **1.2.1 בדוק כי הפונקציה קיימת**
     - **1.2.2 בדוק כי הפרמטרים טועמים**
   - **1.3 אם הצומת מסוג הכרזת פונקציה**
     - **1.3.1 בדוק שהפונקציה לא קיימת**
   - **1.4 אם הצומת מסוג return statement**
     - **בדוק שסוג ההחזרה תואם לפונקציה**