**ספר פרוייקט - י"ג תוכנה  
ACE compiler**



**שם תלמיד: ליחי סויסה**

**תעודת זהות: 214907651**

**שם מרצה: מיכאל צ'רנובילסקי**

**תאריך: מאי 2024 תשפ"ד**

**תוכן עניינים**

[**מבוא 5**](#_kxpu5npmb0if)

[תיאור הפרויקט 6](#_acv0coelcgyg)

[מטרת הפרויקט 6](#_43et1kmlemli)

[**הקדמה 7**](#_oahibs4wmll4)

[מונחי יסוד 7](#_mjt9dewcv5jt)

[קצת על שפת ACE 7](#_3fyo0orr0qay)

[אבני השפה 8](#_im60ydiftyh1)

[קבועים - literals 8](#_r499n0pofs6q)

[משתנים –Identifier 8](#_3ikqg5dinuch)

[Expressions & Statements – ביטויים 9](#_q7uodsbevpj1)

[**תכולת השפה 11**](#_bt9bq3d7lrj2)

[השמה 11](#_i3njb16nofnp)

[תנאים ולולאות 11](#_j0byr3z4prhg)

[הערות – Comments 12](#_s4skksj1r32l)

[פונקציות - Functions 12](#_tuvtnykmzllj)

[דקדוק השפה 13](#_cxt61c3knqmx)

[תחביר השפה 13](#_m3dnd3g2l37m)

[רשימת האסימונים בשפה: 13](#_8vjllnm0ozty)

[BNF 14](#_3oem4kv88pri)

[דוגמא לקוד בשפת ACE 14](#_qsps6f2e8dp5)

[**רקע תיאורטי 15**](#_3hfav88hl2nt)

[מהו קומפיילר 15](#_2fm5n33vac30)

[Compiler vs. Interpreter 15](#_s7ms803s9u6y)

[למה נדרש קומפיילר 15](#_wol389e9ghia)

[כיצד עובד קומפיילר 16](#_lz66x6jxtw8c)

[הסבר קצר על שלבי תהליך הקומפילציה 16](#_jiairh4c7uy7)

[● לקסר 16](#_oq7thftmqafv)

[● פרסר 16](#_fik9wyk8f46j)

[● עץ בינארי אבסטרקטי 17](#_ps9foqcty7yq)

[intermediate representation ● 17](#_boykl3bnlivc)

[● קמפול לשפת מכונה 17](#_i19nb38ckpma)

[מודולים נוספים 18](#_o0voe98efj9u)

[Register allocation 18](#_402n18eweb81)

[Assembly, linking and loading 18](#_br1k641azyno)

[Symbol table 18](#_3dtf8amcs2s)

[Error handler 18](#_xa7iwsfbztoq)

[**מכונות מצבים 19**](#_g2yhclgz7it0)

[הגדרה 19](#_at7c7n6aw2vw)

[רכיבים עיקריים 19](#_ol0fmds1yb5d)

[פעולה 19](#_lqidbq5cwcog)

[סוגים של מכונות מצבים 19](#_q3pbvf74r6ug)

[מכונות מצבים דטרמיניסטיות (DFSM): 19](#_998a06qf9a6t)

[מכונות מצבים לא דטרמיניסטיות (NFSM): 19](#_v41kzbhdrzg7)

[יישומים 19](#_ac7qe6yv4s8f)

[יתרונות 20](#_b1xefwe4hldv)

[חסרונות 20](#_x55lv4k9tsdd)

[שפה פורמלית 20](#_c9iv94hkx0zj)

[מאפיינים עיקריים 20](#_f679d3vvev5m)

[**הגדרת מטרות ומשימות 21**](#_h3fo75i4b34l)

[מטרת הפרויקט: 21](#_a41emqb4mwcp)

[אפיון שפת התכנות: 21](#_i4b9p9ggu2ec)

[מגבלות הפרויקט: 21](#_o8rxllcuq6g4)

[אתגרים בפרויקט: 21](#_i0ic7vbun2ll)

[**הגדרת הבעיה האלגוריתמית 21**](#_vi63zulgdd14)

[אתגרים ספציפיים 22](#_n964kldgjif6)

[פתרונות 22](#_vr4a5l6xfx2x)

[ניתוח הבעיות בכל שלב 22](#_djlu5yh7t0p8)

[ניתוח לקסיקלי (Lexical Analysis) 22](#_bsfxkiudbzap)

[ניתוח תחבירי (Parsing) 22](#_wmfhei8c5try)

[ניתוח סמנטי (Semantic Analysis) 23](#_5l46ugsrqof8)

[ייצור קוד (Code Generation) 23](#_4aodu5xhd6za)

[**סקירת אלגוריתמים בתחום הבעיה 23**](#_g0u10hdg8xue)

[מונחים 23](#_8cz1nfqg8usr)

[Left Factoring 24](#_8ncvlsc56kd6)

[אלגוריתמי פרסר 25](#_s2xmitww3h9a)

[Top Down Parsing (TDP) 25](#_hfgvr2dk7v1n)

[LL Parser 26](#_e6cnq5dj3jj1)

[Recursive Decent Parsing 26](#_jxn5zfuznee8)

[Back tracking 26](#_p4p0kwygkb1t)

[Bottom Up Parsing (BUP) 26](#_gjj8mhbnjsqc)

[Reduce Shift 26](#_8yx5odej22j4)

[LR Parser 27](#_tle2t4yyx7n6)

[Operator Precedence Parser 27](#_e34bxrjx82bo)

[LALR parser 27](#_brw3x6rz2qs8)

[דוגמא לטבלאות הניתוח של LALR parser: 28](#_s3wn53ueewr2)

[**ניתוח שלבי אלגוריתמיקה 29**](#_fnmjskexvbjs)

[ניתוח מילוני - Analysis Lexical 29](#_bvkdq2iwgxqv)

[דוגמא לתרשים האוטומט הסופי 30](#_rclsdodqgll4)

[ניתוח תחבירי – Analysis Syntax 31](#_r68t9f6j93c2)

[תרשים פעולת LALR parser 32](#_wuyy40ffqsj0)

[Parsing table & Stack 32](#_ud9z5i28znxh)

[Action table 32](#_btou5rpqdvw6)

[Goto table 32](#_koaph2jos8sf)

[עץ ניתוח 33](#_wiugzh1qqf38)

[מימוש LALR parser 34](#_49if0j8vc9i)

[דוגמאות משפת ACE 34](#_3kptj5aiveut)

[טבלאת goto & action 34](#_ix7tpiaj1rky)

[ניתוח סמנטי 35](#_ldc68g20widq)

[מהו ניתוח סמנטי? 35](#_4yo6eb2w0ppw)

[ההבדל בין ניתוח תחבירי לניתוח סמנטי: 35](#_5iefgxdw6l28)

[מטרות ניתוח סמנטי: 35](#_mq8uovk8ms30)

[יתרונות ניתוח סמנטי: 35](#_xlynauw7535a)

[טכניקות לניתוח סמנטי: 35](#_245pbfcw9ddg)

[בדיקת ייחודיות שמות המשתנים 36](#_ogs8y98r1aej)

[אסטרטגיה לפתרון 36](#_c5a745zth9m)

[הפקת קוד 36](#_kwx5n4y6nzb3)

[מטרות יצירת קוד: 36](#_59ofeus9l0v0)

[אסטרטגיות ליצירת קוד: 36](#_udqyyd1md3xb)

[שיקולים ביצירת קוד: 36](#_sc63lahbwrf0)

[דוגמה: 37](#_fiafmnshten9)

[טבלת סמלים 37](#_3by5m6coyh61)

[מטרות טבלת הסמלים: 37](#_hj2gq1eiwvd2)

[ייצוג טבלת הסמלים: 37](#_ph3e1x6pz0rc)

[יתרונות שימוש במילון: 38](#_bjymggmw6zma)

[ניהול תחומי גישה: 38](#_boce3dlp6loz)

[טיפול בשגיאות 38](#_aagoi01f36wu)

[גישות שונות לטיפול בשגיאות 38](#_xhu795yoiv2n)

[**Top down level design 39**](#_xrllr378umnn)

[Compiler 39](#_w1lpighwqpks)

[Lexer 39](#_17ux01y168fa)

[Parser 40](#_49neejln6gkq)

[Semantic analyzer 40](#_dawctm191oqa)

[Code generation 40](#_vhi5ljmkeb19)

[**מודולים 41**](#_gjf7n0hp9gcn)

[Token 41](#_3vc6t62t13wi)

[Lexer 42](#_uaex66po8kf3)

[Parser 43](#_rvtlkccnal5c)

[Stack 45](#_8c1o061gvz0y)

[semanticAnalysis 47](#_sd5p20swib90)

[Code generation 49](#_ze1j92owckcm)

[**אלגוריתם ראשי 52**](#_x7fs5eehjr06)

[**מבני נתונים 54**](#_ybbfphtygi7s)

[מנתח מילוני – Analysis Lexical 54](#_kbsip25wlozu)

[מנתח תחבירי – Analysis Syntax 54](#_edndapp4gear)

[Parse Table 55](#_35ltltfpgecy)

[Parse Stack 56](#_qxggqhcn14j)

[דקדוק השפה – Rules Production 56](#_jkfgly6k0su)

[עץ הניתוח – Tree Parse 56](#_g0yrk7cxboli)

[מנתח סמנטי – Analysis Semantic 57](#_ew8cigqkqyuo)

[Hash table 58](#_bisyaimm0fud)

[**סביבת העבודה 59**](#_7n3ttni2e1aw)

[עורך קוד 59](#_5iu2t7ju1los)

[מערכת הפעלה 59](#_lfinvuvt9gt8)

[שפת תכנות 59](#_ixpg6klnzsqo)

[קומפיילר 59](#_23xeclqrboqy)

[**מדריך למשתמש 60**](#_xwbv0wjw4e2)

[דרישות 60](#_q78agsupt0uk)

[**סיכום אישי 61**](#_rnadzpxj5ber)

[מסע מרתק בעולם קומפיילרים: סיפורו של סטודנט צעיר 61](#_1utk9w2dhzsc)

[במידה והיה לי זמן נוסף בפרויקט הייתי רוצה לבצע מספר דברים 62](#_kvu016h6bkj5)

[סיכום: 62](#_rbzg6oujku85)

[**ביבליוגרפיה 63**](#_7pqewd8cp6cv)

[**קוד הפרויקט 64**](#_2gzvvwf34k9i)

[Dfa 64](#_69pycgfbo9wa)

[Lexer 69](#_sdmjojm3hpul)

[Parser 73](#_d5v4k5a0zk07)

[Production 79](#_qncty63amw93)

[semanticAnalysis 80](#_9m29j63384sd)

[Stack.hpp 84](#_u7wziv1s7j3m)

[Symbol 85](#_xtsdpnq1tmkp)

[Token.hpp 86](#_efsrtloju8wb)

[Types.hpp 87](#_ywr58h1wkrr4)

[Dfa 87](#_3oextjo4rxzd)

[Code generation 91](#_ji9m2bc6zbky)

# 

# מבוא

## תיאור הפרויקט

פרויקט זה עוסק בפיתוח מהדר(compiler) אשר תפקידו לתרגם קובץ טקסט כלשהו אשר רשום בשפת תכנות עם תבניות מסוימות לשפת אסמבלי 64 ביט אשר ניתן להריץ על מחשב סטנדרטי, לצורך כך בניתי את שפת התכנות ACE.

שפת תכנות הינה סט של חוקים תחביריים וסמנטים המגדירים תהליכים מסויימים שנרצה שיבוצעו על ידי המחשב, לצורך בניית המהדר נצטרך להגדיר שפת תכנות קודם לכן.

כאשר מגדירים שפת תכנות, מתייחסים כאמור לשלושה מישורים: מילוני, תחבירי ו לשוני .

המישור המילוני מגדיר אילו מילים שייכות לשפה, ואילו לא.

לדוגמא, המילה if היא מילה המקובלת בשפת C בעוד שהמילה 0l@#Hel איננה.

המישור התחבירי מגדיר אילו רצפי מילים של השפה הם חוקיים, ואילו הם לא.

לדוגמא, הרצף ;3 = x int הוא רצף חוקי בשפת C, בעוד שהרצף then 5 is x if איננו.

המישור הלשוני מתייחס למשמעות רצפי המילים, והוא מגדיר חוקים כלליים שחייבים להתקיים בכל רצף מילים בשפה.

לדוגמא, חובת ההצהרה – לפני שימוש במשתנה, קיימת חובה להצהיר עליו.

הקומפיילר עושה שימוש בהגדרת השפה על מנת לנתח את קטע הקוד שנקלט, וכדי לייצר את תוכנית היעד.

## מטרת הפרויקט

מטרתי בבחירה בפרויקט זה היא להעמיק את ידיעותי ויכולותיי ב"אחורי הקלעים" של עולם התכנות, על ידי בניית קומפיילר אני מבין יותר טוב את תהליך בניית שפת תכנות ומתנסה במגוון דברים אשר יעמיקו את הבנתי בעולם התכנות בכלל ובעולם הלואו לבל בפרט

* בניית כל חלקי המהדר בצורה יעילה יחסית
* למידה מעמיקה על תהליך ההידור
* יצירת שפה חדשה מאפס
* פיתוח יכולות למידה אישיות

מכיוון שבעבר כבר בניתי מהדר אני מודע לתהליך ולרמת הקושי, אך בניגוד לשפה הראשונה שכתבתי הפעם לא איעזר בטכנולוגיות חיצוניות (בעבר השתמשתי ב flex & bison) אלא אבנה הכל בעצמי ללא עזרה חיצונית (למעט ספריות של מבני נתונים ב CPP)

# הקדמה

## מונחי יסוד

לפני שאסביר על תהליך בניית המהדר ארצה להציג בפני מי שקורא את ספר הפרויקט מונחים בסיסיים שעליהם אבנה במהלך כתיבת הספר וחובה על הקורא להבין ולדעת על מנת לא לפגוע ברצף הקריאה והבנת הפרויקט.

שפת תכנות - כאשר אנחנו מדברים על שפת תכנות אנחנו בעצם מדברים על אוסף הגדרות שכאשר נרכיב אותן ביחד ניצור תוכנית בשפת התכנות, מטרת שפת התכנות היא להוות מעין גשר בין האדם המתכנת לבין המכשיר הטכנולוגי שאותו הוא רוצה להפעיל, מבחינת המחשב לדוגמא, אין משמעות לשפת התכנות עצמה אלא למה שהיא הופכת אליה לאחר תהליך ההידור, אך מבחינת האדם היא קריטית ביותר שכן קשה מאוד לתכנת בשפת מכונה המורכבת מ2 ספרות שהן 0 ו1.

הידור - לשם כך נוצרו המהדרים (קומפיילרים) אשר כל מטרתם היא להמיר בין שפת תכנות שהומצאה על ידי אנשים, לשפת מכונה (כאמור 0 ו1) אשר מחשב יכול להבין ולהפעיל פקודות בהתאם

תהליך ההידור כולל 5 שלבים מרכזיים שאפרט בהרחבה בהמשך הספר, כעת נסתפק בשמם

לקסר, פארסר, עץ סינטקס אבסטרקטי, ייצוג אמצעי, והפקת קוד הינם החלקים המרכיבים קומפיילרים מודרניים.

בפרויקט זה אבנה כל חלק בעצמי ואגדיר שפת תכנות חדשה שתיקרא ACE (ראשי תיבות רקורסיביים של Ace Compiling Environment)

בפרויקט זה ארצה להעמיק את הידע וההבנה שלי בתחום הLow level ולהבין מה קורה מאחורי הקלעים של דבר שנדמה לנו כל כך פשוט ומובן מאליו אך בעצם חבויים בו אתגרים רבים שגם עליהם אפרט בהמשך

קוד - הינו רצף של פקודות בשפת תכנות מסוימת אשר לאחר תהליך ההידור הופך לרצף תווים המורכבים מ0 ומ1, מעבד המחשב יודע להפוך תווים אלה לפקודות פיזיות בפועל בעזרת חשמל כאשר 0 מסמל "אין חשמל" ו1 מסמל "יש חשמל"

## קצת על שפת ACE

שמה של שפת התכנות מגיע מראשי התיבות

A - Ace

C- Compiling

E- Environment

שהם ראשי תיבות רקורסיביים, ההשראה לשם נשאבת מGNU

השפה דומה בסינטקס לשפת C סטנדרטית

## אבני השפה

אלו הם אבני היסוד הבסיסיים ביותר של השפה אשר לא ניתן לפרק אותם לביטויים פשוטים יותר והם מרכיבים את כל הביטויים המורכבים בשפה

### קבועים - literals

קבוע הוא ערך המופיע ישירות בקוד התוכנית

קבוע יכול להיות מטיפוסים שונים – מספר שלם, תו.

טיפוסו של הקבוע נקבע על ידי המהדר בהתאם לערכו.

int x = 81 :דוגמא

בדוגמא שלעיל 81 הוא קבוע, אשר יובן על ידי המהדר כקבוע מטיפוס מספר שלם.

**טיפוסי קבועים**

קבוע מטיפוס מספר שלם - int.

על מנת להגדיר קבוע מסוג int נכתוב את ערכו ישירות בקוד התוכנית:

o דוגמא:

a / 2

בדוגמא זו 2 הוא קבוע ו – a הוא שם משתנה כלשהו.

קבוע מטיפוס תו - char.

הגדרת קבוע מסוג char תהיה בתוך שני גרשים בודדים:

o דוגמאות להגדרה:

char ch := 'b'

הערך של התו b ייכנס אל תוך המשתנה ששמו ch.

קבוע מטיפוס בוליאני - bool

הגדרת קבוע מסוג בוליאני תתאפשר לפי ערך מילולי true\false בצורה הבאה:

Bool flag := true

ערך בוליאני מייצג אפשרות אחת, דלוק או כבוי וחוסך בזיכרון

### משתנים –Identifier

משתנה מייצג מקום בזיכרון בו אפשר לשמור ערכים.

מקום זה בזיכרון מיוצג על ידי שם המשתנה, שנקרא גם מזהה

**שמות משתנים**

.1 שם משתנה הוא רצף של אותיות בשפה האנגלית, ספרות, והתו '\_'

רצף זה חייב להתחיל באות בשפה האנגלית או בתו '\_'.

.2 אין להשתמש במילים שמורות כמזהים.

.3 קיימת הבחנה בין אותיות גדולות וקטנות

4. שם משתנה הוא ייחודי ואין להשתמש באותו שם יותר מפעם אחת

**טיפוסי משתנים**

לכל משתנה בשפה יש גם טיפוס אשר מציין את סוג הערכים שהוא יכול להכיל.

ישנם שלושה סוגים של טיפוסי משתנים:

* int – משתנה מטיפוס מספר שלם.
  + מכיל ערכים מסוג מספרים שלמים. ,1 ,-15 ,79 0 וכו'.
* char – משתנה מטיפוס תו
  + מכיל ערכים מסוג תו. f 7, 0, ,g ,a וכו'.
* Bool - משתנה מסוג בוליאני
  + מכיל ערכים מסוג true או false

**הגדרת משתנים**

הגדרה כללית של משתנה:

<Data-type> <Identifier>;

דוגמאות:

int x

Bool f

char c

### Expressions & Statements – ביטויים

Expression

יחידה תחבירית בשפת תכנות שניתן להעריכה על מנת לקבוע את ערכה. שילוב של אחד או יותר קבועים, משתנים,

פונקציות, אופרטורים, ו – Expression נוספים, שהשפה מפרשת )לפי הכללים של קדימות ושיוך(, ומחשבת כדי לייצר ערך. תהליך זה, עבור ביטויים מתמטיים, נקרא הערכה

בפשטות, הערך המתקבל הוא בדרך כלל אחד מהסוגים הפרימיטיביים השונים, כמו ערך מספרי, ערך בוליאני וכו'.

דוגמאות ל – Expressions:

3 + 15 •

4 •

(x – 5) / y •

7 <= 22 •

X % 2 == 0 •

(x + 15) < 3 \* (y – 4) •

Statement

יחידה תחבירית בשפת תכנות המבטאת פעולה כלשהי שיש לבצע. תכנית הנכתבת בשפה כזו נוצרת על ידי רצף של אחד או יותר Statements. בשונה מ – Expression, Statement לא מוערכת לכדי ערך. ל - Statement יכולים להיות רכיבים פנימיים

דוגמאות ל – Statements:

if, else – תנאים•

• לולאות – while

• הצהרה על משתנה – ;x int

• השמת ערך למשתנה – ;4 = x set

אופרטורים – Operators

חשבוניים

• חיבור - +

• חיסור - –

• כפל - \*

• חילוק - /

• שארית - %

לוגיים

• שווה ל - ==

• לא שווה ל - =!

• גדול מ - >

• קטן מ - <

• גדול או שווה ל - =>

• קטן או שווה ל - =<

! - not •

קשרים לוגיים

|| - or •

&& - and •

# תכולת השפה

בפרק זה אעסוק בתכולת השפה (כלומר היכולות שהיא מסוגלת לבצע) ובכללי הדקדוק והכתיבה הנכונה בשפה.

תכולת השפה מוגדרת היטב גם לפי תרשים הbnf שאותו ניתן למצוא בנספחים של הספר בנוסף אצרף קטעים רלוונטים במידת הצורך

## השמה

סימול של השמה מבוצע באמצעות אחת מהמילים השמורות לטיפוסים וסימן השווה עם נקודותיים =:

• הגדרה כללית להשמה:

<type> <Identifier> = <Expression>;

על מנת שההשמה תהיה חוקית, טיפוס המשתנה אליו עושים את ההשמה, של ה - <Identifier> צריך לתאום לטיפוס הערך המושם, .

• דוגמאות:

Int x := 80

Char y := ‘v’

Bool flag := true

## תנאים ולולאות

תנאים ולולאות הם חלקי קוד המתבצעים כתלות באם ביטוי מסוים הוא אמת או שקר.

**תנאים – Conditions**

לתנאי יכולים להיות שני חלקים:

if •

else •

אם הביטוי נותן תוצאת אמת, הקוד שבחלק של ה – if יתבצע.

ואם נותן תוצאת שקר, הקוד שבחלק של ה – if לא יתבצע.

אם יש חלק של else, הוא יתבצע כאשר הביטוי נותן תוצאה שקרית.

חלקים אלו של ה – if וה – else יתבצעו 0 או 1 פעמים.

בכל מקרה, לאחר ביצוע התנאי התוכנית תמשיך לקוד שנמצא אחריו.

• דוגמא להגדרת תנאי בעזרת שימוש ב – if בלבד:

if (<Expression>):

}

Do if <Expression> is True

{

דוגמא להגדרת תנאי בעזרת שימוש ב – if ו - else:

if (<Expression>):

{

Do if <Expression> is True

}

Else:

Do if <Expression> is false

**לולאות – Loops**

לולאה דומה מאוד במבנה שלה לתנאי, if, אך ההבדל היחיד הוא שחלק הקוד שבתוך הלולאה מתבצע כל עוד הביטוי נותן תוצאת אמת, ולאו דווקא 0 או 1 פעמים. כלומר לולאה יכולה להתבצע מספר רב של פעמים.

while •

דוגמא כללית להגדרת לולאה בעזרת שימוש ב - while:

while (<Expression>):

Do while <Expression> is True

## הערות – Comments

הערות הן קטעי קוד אשר הקומפיילר מתעלם מהם .

לרוב מתכנתים משתמשים בהערות על מנת להסביר למה כתבו שורת קוד מסוימת, למה עשו זאת דווקא בצורה הספציפית הזאת ועל מנת להשאיר תזכורות לעצמם ולמתכנתים אחרים שיקראו את הקוד שלהם בעתיד. כל הקוד מתחילת ההערה ייחשב הערה, עד אשר נרד שורה. הערות בשפה יתחילו עם הסימן ~

• דוגמא להערה:

~ This is a comment

## פונקציות - Functions

פונקציות הן קטעי קוד קומפקטיים אשר נועדו לבצע קטעי קוד מסויימים וניתן לקרוא להן ממקומות שונים בקוד, פונקציה יכולה לקבל משתנים ולהחזיר משתנים ויכולה שלא, על המשתנים שהיא מחזירה יש לתאום את סוג המשתנה בהצהרת הפונקציה

דוגמא:

Give type identifier ( input\_var\_list )

{

statement\_list

return\_statement

}

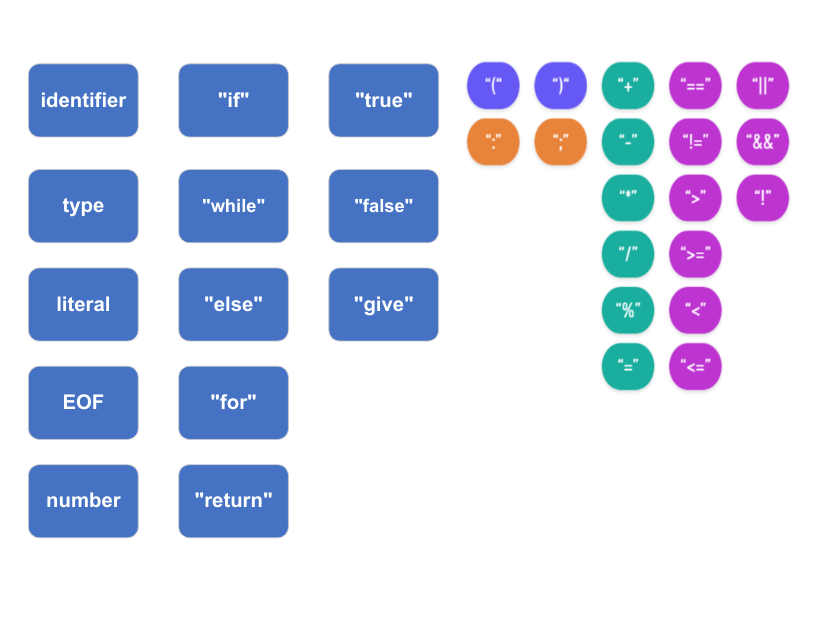
## דקדוק השפה

לאחר שהגדרתי את אבני השפה ותכולת השפה, כעת אגדיר את תחביר / דקדוק השפה. ה – Grammar של השפה.

### תחביר השפה

הדקדוק מורכב מ – Terminals ו – Terminals-Non. הסימנים )Terminals )הם המילים )Tokens )שנקלטו כקלט מקטע הקוד, בעוד שהמשתנים )Terminals-Non )הם רצפי סימנים ומשתנים. תחביר השפה מוגדר באמצעות שילוב הסימנים והמשתנים, בכללים שנקראים כללי יצירה . כללי היצירה בעצם מגדירים את המשתנים, באמצעות הסימנים המוגדרים בשפה ומשתנים אחרים.

### רשימת האסימונים בשפה:

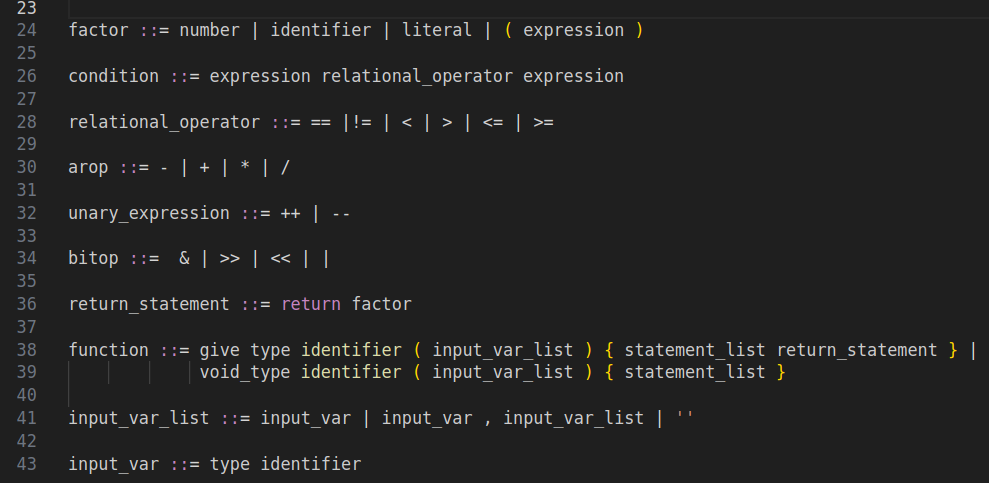
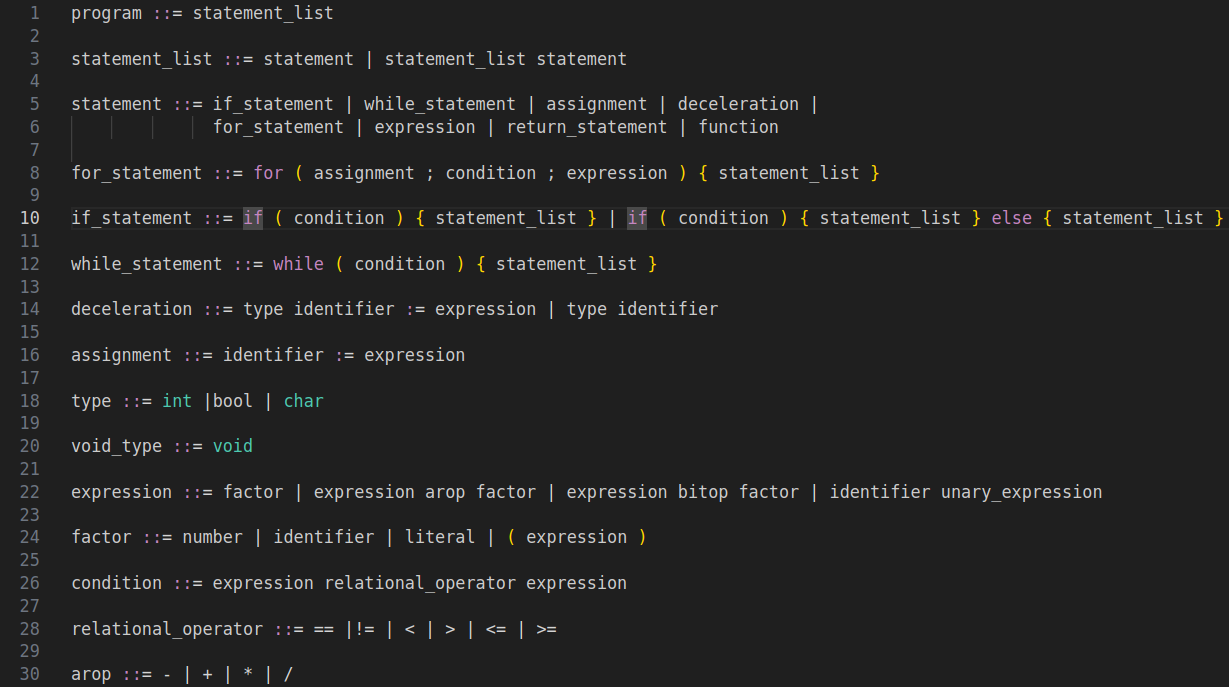


## BNF

Form Naur-Backus היא צורת כתיבה פורמלית עבור תיאור Grammars של שפות נטולות הקשר . צורת כתיבה זו משמשת לעיתים קרובות לתיאור שפות תכנות

BNF עוזר לכתוב בצורה חד-חד משמעית את כללי ה – Grammar של שפה מסוימת, באופן יחסית קל וקריא.

תרשים bnf של שפת ACE:



## דוגמא לקוד בשפת ACE

# רקע תיאורטי

## מהו קומפיילר

תוכנת מחשב אשר מתרגמת קוד מקור הכתוב בשפת תכנות אחת לקוד הכתוב בשפת תכנות אחרת, ללא שינוי המשמעות של קוד המקור. לרוב מתרגם משפה עילית )Java ++,C ,C), לשפת מכונה. הקומפיילר גם מייעל ומשפר את קוד המקור כמה שניתן. כמו כן, מתריע על השגיאות / אזהרות שמצא, ומציע הצעות לפתרונות שלדעתו יפתרו שגיאות / אזהרות אלו.

### Compiler vs. Interpreter

ישנם שני אופני עבודה עיקריים של קומפיילר: תרגום כל קוד המקור לכדי יחידת הרצה אחת או תרגום כל פקודה בנפרד בקוד המקור תוך כדי ריצת התוכנית.

כפי שצוין לעיל, Compiler, עובר על כל קוד המקור לפני הריצה, בודק את תקינותו, ומתרגם וממיר אותו ליחידת הרצה אחת בשפת מכונה. שפות שמתורגמות על ידי Compiler נקראות שפות מקומפלות . דוגמאות לשפות מקומפלות הן ,C, C++, Java בניגוד ל – Compiler, ה – Interpreter מתרגם וממיר כל פקודה בנפרד בקוד המקור לפקודות בשפת מכונה , תוך כדי

ריצת התוכנית, ללא בדיקת תקינות הקוד לפני הריצה. שפות שמתורגמות על Interpreter נקראות שפות סקריפטים. דוגמאות לשפות אלו הן JavaScript ,Python ועוד.

## למה נדרש קומפיילר

מכיוון שלבני האדם קל יותר לכתוב קוד בשפות תכנות עיליות אשר יותר קרובות אליהם )יותר קרובות לאנגלית(, מאשר

לכתוב קוד בשפת מכונה, אנו משתמשים בשפות עיליות אלו לכתיבת קוד. אך המחשב אינו מבין שפות עיליות אלו, הוא

מבין רק קוד בשפת מכונה.

בשביל לגשר על הפער בין בני האדם לבין המכונה, צריך קומפיילר. שיתרגם את מה שאנו מתכוונים כאשר אנו כותבים קוד

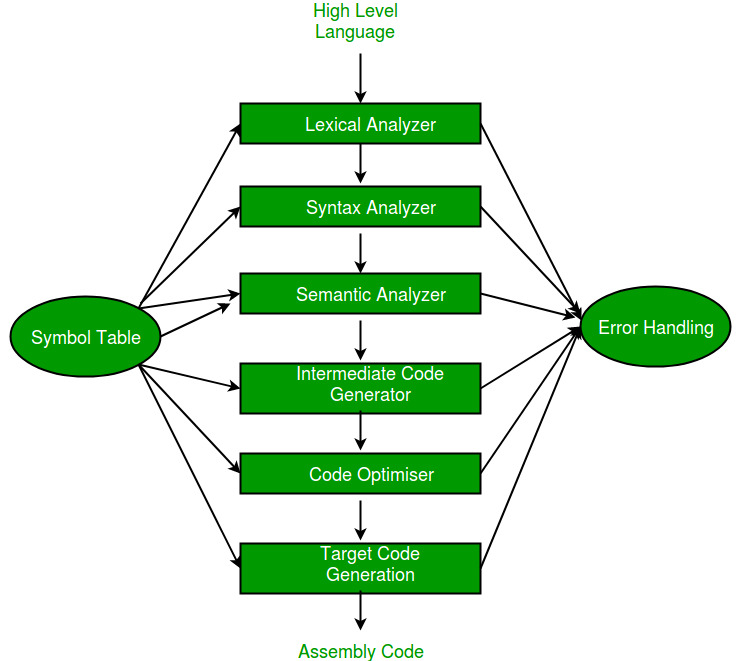
בשפה עילית לשפה שהמחשב יבין, שפת מכונה.

### כיצד עובד קומפיילר

תהליך הקומפילציה הוא תהליך מורכב, ולכן מוטב לחלק אותו לשלבים. הקומפיילר עובד כך שכל שלב מקבל כקלט את

התוצאה של השלב הקודם.

להלן דיאגרמה של השלבים:



## הסבר קצר על שלבי תהליך הקומפילציה

### ● לקסר

○ הקלט הוא הקוד אשר אנו רוצים לקמפל לשפת מכונה, בדר"כ, ייקרא תחילה כקובץ ואז

נקרא כל מילה בנפרד )התעלמות מרווחים(.

○ הפלט הוא Token, מספר או תו אשר מזהה מה הסוג של המילה שנקראה)שם/מזהה, מילת

מפתח, ערך, פעולה בינארית וכו'(.

○ הוא קורא את המילה/מספר הנוכחי/ת, מזהה איזה סוג הם ומחזיר את ה-token התואם תוך

שמירה של המילה/תו לשימוש מאוחר יותר ב-oarser.

○ ע"י קריאת התו הראשוני של המילה/מספר. לאחר מכן הוא בודק לאיזה קריטריון בסיסי היא

תואמת )מזההים, ערכים, פעולות בינריות וכו'(. לאחר מכן הוא קורא את הערך בהתאם ומזהה

איזה token שייך לו)למשל למזההים, האם זה מזהה כללי, או מילת מפתח ספציפית וכו'(.

### ● פרסר

○ הקלט הוא קוד מחולק לTOKENS.

○ הפלט הוא הבסיס לעץ בינארי אבסטרקטי. )יוסבר בהמשך(

○ מחלק TOKENS לקבוצות ומשפחות על מנת להפוך אותם לבסיס עבור השלב הבא שהוא

עץ בינארי אבסטרקטי.

○ הפרסר מזהה מילות מפתח אותן הוא ממיין לקטגוריות שונות שנקבעות על פי סוג מילת

המפתח והקשרה.

### ● עץ בינארי אבסטרקטי

○ הקלט הוא קוד ששעבר תהליך פרסור על ידי הפרסר

○ הפלט הוא קוד שמחולק לתתי משפחות ומוכן לעבור לשלב התרגום לקוד המיוצג על ידי שפת

אמצע

○ התפקיד של עץ בינארי אבסטרקטי הוא לחלק את הקוד המפורסר לתתי משפחות ומחלקות

על פי הגדרות שניתנו מראש

○ העץ הבינארי מזהה משפחות שונות של מילות מפתח ומחלק אותן לקבוצות קטנות

### intermediate representation ●

○ הקלט הוא קוד ממויין ומסודר לקטגוריות בעץ בינארי מופשט.

○ הפלט הוא קוד בשפה אמצעית שדומה לאסמבלי אשר מיועל ומוכן להפוך לקוד

בשפת מכונה.

○ מייעל ומסדר את הקוד וממיר את שפת הקוד לשפה אמצעית אשר נראית דומה

לאסמבלי בדרך כלל וקל יותר להמיר אותה לשפת מכונה ולהריץ.

○ לוקח את הקוד הממוין ומסדר אותו בדרך היעילה ביותר בעזרת מספר אלגוריטמים של

ספריית LLVM, למשל: יצירת מבנה קבוע של ייצוג הביניים, דבר המאפשר לעבור על

פונקציות מספר פעמים בשביל לוודא את ההרצה הכי יעילה של הפונקציה בקומפיילר,

או זיהוי והתאמה בין קבצי קוד וייבוא רק של קטעי הקוד בשימוש, דבר המוביל

ליצירה של קבצי הרצה קטנים יותר ויעילים יותר )optimization Time-Link).

שחוסכת הכי הרבה זיכרון ונותנת זמן הרצה מינימלי על ידי מנגנון מכונה הגיוני.

### ● קמפול לשפת מכונה

○ ייצוג ביניים של הקוד שקרוב יותר לאסמבלי אשר מיועל ומוכן להפוך לשפת מכונה

○ קוד הרצה אשר מכיל את הקוד המומר לשפת מכונה ויכול לרוץ על המחשב.

○ קורא את ייצוג הביניים וממיר אותו לשפת מכונה ויוצר ממנו object

.file/executable file

○ הוא ממיר את ייצוג הביניים הקרוב לאסמבלי לפקודות אסמבלי אשר מיוצגות

בייצוג הביניים ומשם הקוד מומר ישירות לשפת מכונה ומתווסף לקובץ ההרצה

הסופי.

## מודולים נוספים

### Register allocation

לתוכנית יש מספר ערכים שהיא צריכה לשמור במהלך הריצה שלה. ייתכן שארכיטקטורת מכונת היעד לא תאפשר לכל

הערכים להישמר בזיכרון המעבד, או ה – registers. השלב של ה – generator code depended machine, מחליט אילו

ערכים לשמור ב – registers, ואילו registers ישמרו ערכים אלו.

### Assembly, linking and loading

אסמבלר מתרגם שפת אסמבלי לשפת מכונה. הוא יוצר מקובץ asm שמכיל שפת אסמבלי, קובץ object. קובץ object

מכיל הוראות בשפת מכונה, כמו גם המידע הדרוש על איפה צריך לשים את ההוראות האלה בזיכרון.

לינקר מחבר כמה קבצי object לקובץ exe אחד.

כל הקבצים שהוא מחבר יכולים להיות מקומפלים על ידי אסמבלרים שונים.

משימתו העיקרית של הלינקר היא לקבוע את המיקום בזיכרון של כל אחד מהקבצים בעת הטעינה שלהם לזיכרון )על ידי ה

– Loader), כך שההוראות מקבצי ה – obj השונים יתבצעו בסדר הגיוני בעת הריצה.

ה – loader הוא חלק ממערכת ההפעלה שאחראי על הטעינה של קבצי הרצה )exe )לזיכרון, והביצוע שלהם.

הוא מחשב את גודל התוכנית ומקצה לה מספיק מקום בזיכרון. הוא גם מאתחל מספר רגיסטרים שונים שיתחילו את תהליך

הביצוע/הרצה של התוכנית.

### Symbol table

ה – table Symbol או טבלת הסמלים, מכילה רשומה עבור כל Identifier( מזהה( עם שדות עבור התכונות של אותו

המזהה.

טבלה זו עוזרת לקומפיילר למצוא רשומה של מזהה כלשהו בתוכנית ולקבל את הפרטים עליו באופן מהיר יחסית.

ה – table Symbol עוזרת גם ב – managment Scope.

טבלה זו לוקחת חלק בכל אחד מהשלבים שצוינו לעיל, ומתעדכנת בהתאם.

### Error handler

כפי שכתבתי בתיאור השלבים של הקומפיילר, בכל שלב ושלב בתהליך הקומפילציה יכולות להיווצר שגיאות.

בשביל כך יש את ה – handler Error.

השגיאות שמתגלות מדווחות ל – handler Error והוא מדווח, ומציג אותן חזרה למתכנת בתצורה של הודעה. אם

לקומפיילר יש הצעה מסוימת לפתרון הבעיה, גם היא תוצג בהודעה.

# מכונות מצבים

## הגדרה

מכונת מצבים (Finite State Machine - FSM) היא מודל חישובי פשוט אך רב עוצמה המשמש לייצוג מערכות דינמיות בעלות מספר סופי של מצבים אפשריים. ניתן להשתמש בהן כדי לדמות התנהגות של מגוון רחב של מערכות, החל ממעגלים אלקטרוניים פשוטים ועד לתהליכים עסקיים מורכבים.

## רכיבים עיקריים

מכונת מצבים מורכבת משלושה רכיבים עיקריים:

**מצבים:** מערך סופי של מצבים אפשריים עבור המערכת.

**מעברים:** קבוצות של כללים הקובעים כיצד המערכת עוברת ממצב אחד למצב אחר. כל מעבר מותנה בתנאי קלט ספציפי.

**פעולות:** קבוצות של פעולות שהמערכת יכולה לבצע בכל מצב.

## פעולה

מכונת מצבים פועלת באופן הבא:

המערכת מתחילה במצב התחלתי מוגדר.

קלט נותן למערכת. בהתבסס על הקלט הנוכחי והמצב הנוכחי, מופעל מעבר מתאים.

המעבר גורם למערכת לעבור למצב חדש. ייתכן שביצוע המעבר יגרום גם לביצוע פעולה אחת או יותר. שלבים 2-5 חוזרים על עצמם עד שהמערכת מגיעה למצב סופי או עד שהיא מקבלת פקודה לעצור.

## סוגים של מכונות מצבים

קיימים שני סוגים עיקריים של מכונות מצבים:

### מכונות מצבים דטרמיניסטיות (DFSM):

בכל מצב נתון, קיים מעבר אחד לכל קלט אפשרי.

### מכונות מצבים לא דטרמיניסטיות (NFSM):

במצב נתון, ייתכנו מספר מעברים אפשריים עבור אותו קלט, או שאף לא מעבר כלל

## יישומים

מכונות מצבים משמשות במגוון רחב של יישומים, ביניהם:

עיצוב מעגלים אלקטרוניים: ניתן להשתמש במכונות מצבים כדי לתאר את התנהגותם של מעגלים לוגיים ודיגיטליים.

פיתוח תוכנה: ניתן להשתמש במכונות מצבים כדי לתכנת מערכות תוכנה פשוטות, כגון מפרשים ותוכנות בקרה.

עיבוד שפה טבעית: ניתן להשתמש במכונות מצבים כדי לנתח טקסט ולזהות דפוסים לשוניים.

תכנון משחקים: ניתן להשתמש במכונות מצבים כדי ליצור משחקים פשוטים, כגון משחקי מחשב ופאזלים.

### יתרונות

למכונות מצבים יש מספר יתרונות, ביניהם:

פשטות: הן קלות להבנה ולתכנות.

יעילות: הן יכולות לפעול בצורה יעילה גם במערכות מורכבות.

רב-תכליתיות: ניתן להשתמש בהן במגוון רחב של יישומים.

כוח ביטוי: הן מסוגלות לייצג התנהגות מורכבת של מערכות.

### חסרונות

למכונות מצבים יש גם מספר חסרונות, ביניהם:

גודל: ייתכן שהן יהיו גדולות ומסורבלות לייצוג מערכות מורכבות מאוד.

קושי בניהול: קשה לנהל ולתחזק מכונות מצבים גדולות.

קושי בתכנון: תכנון נכון של מכונת מצבים מורכבת יכול להיות מאתגר.

## שפה פורמלית

הגדרה: שפה פורמלית היא מערכת פורמלית המוגדרת על ידי מערכת חוקים וכללים תחביריים שתפקידה לייצג מושגים מתמטיים, לוגיים, או מידע טכני בצורה מדויקת וברורה.

### מאפיינים עיקריים

סינטקסט: מערכת חוקים וכללים תחביריים המגדירים את המבנה התקין של משפטים וחוקי יצירת ביטויים.

סמנטיקה: מערכת משמעויות המיוחסת לסימבולים ולביטויים בשפה.

כוח ביטוי: היכולת של השפה לייצג מושגים מורכבים ומגוונים.

יתרונות:

דיוק: מבטיחים העברה מדויקת וחד משמעית של מידע.

בהירות: מקלים על הבנה וניתוח של מידע מורכב.

פורמליות: מאפשרים ניתוח מתמטי ולוגי קפדני.

יעילות: חוסכים זמן ומאמץ בתקשורת ובהעברת מידע.

# הגדרת מטרות ומשימות

## מטרת הפרויקט:

בניית מהדר עבור שפת תכנות דמוית C, שתתאים למתכנתים מתחילים.

למידה מעמיקה על תהליך ההידור.

יצירת שפה חדשה מאפס.

פיתוח יכולות למידה אישיות.

## אפיון שפת התכנות:

תחביר דומה ל-C, תוך פשטות ונוחות שימוש.

פירוט שגיאות מעמיק.

מיועדת ללימוד עקרונות התכנות הבסיסיים.

מהווה גשר והכנה לקראת שפת C.

## מגבלות הפרויקט:

בניית כל חלקי המהדר בצורה יעילה יחסית.

שימוש בספריות מבני נתונים ב-C++ בלבד, ללא טכנולוגיות חיצוניות (flex & bison).

## אתגרים בפרויקט:

מורכבות אלגוריתמית גבוהה.

ניהול מורכבות קוד.

בדיקות מקיפות.

תחזוקה לאורך זמן.

# הגדרת הבעיה האלגוריתמית

בניית מהדר היא משימה מורכבת מבחינה אלגוריתמית. היא דורשת טיפול במספר רב של בעיות, כגון:

**ניתוח תחבירי:** ניתוח קוד המקור של התוכנית וזיהוי המבנה שלו, תוך יצירת עץ תחביר.

**ניתוח סמנטי:** הבנת משמעות קוד המקור, תוך זיהוי משתנים, פונקציות, טיפוסים וביטויים.

**יצירת קוד ביניים:** הפיכת קוד המקור לייצוג ביניים יעיל, כגון קוד P-code או LLVM.

**מיטוב:** שיפור יעילות הקוד הביניים, תוך שימוש בטכניקות כמו הסרת קוד מת, ניתוח זרימה נתונים ועוד.

**יצירת קוד מכונה:** הפיכת הקוד הביניים לקוד מכונה ספציפי למעבד.

כל אחת מהבעיות הללו היא בעיה אלגוריתמית קשה בפני עצמה. בנוסף, יש צורך לתאם בין כל השלבים הללו כדי ליצור מהדר יעיל וקורקטי.

## אתגרים ספציפיים

בניית מהדר כפרויקט טומנת בחובה מספר אתגרים ספציפיים:

**ניהול מורכבות:** פרויקט מהדר מורכב בדרך כלל ממספר רב של קבצים ומודולים. ניהול המורכבות הזו הוא אתגר משמעותי.

**בדיקות:** בדיקת מהדר היא משימה קשה, כיוון שיש צורך לוודא שהוא עובד בצורה נכונה עבור מגוון רחב של קודי מקור.

**תחזוקה:** מהדרים מתעדכנים באופן קבוע כדי לתמוך בשפות תכנות חדשות, תכונות חדשות וטכנולוגיות חדשות. תחזוקה של מהדר לאורך זמן היא משימה אתגרית.

## פתרונות

קיימות מספר גישות לפתרון הבעיות האלגוריתמיות בבניית מהדר:

**שימוש בטכניקות קיימות:** ניתן להשתמש בטכניקות קיימות מתחום מדעי המחשב, כגון אלגוריתמי ניתוח תחבירי, אלגוריתמי ניתוח סמנטי וטכניקות אופטימיזציה.

**שימוש בכלי תוכנה:** קיימים מספר כלי תוכנה שיכולים לסייע בבניית מהדר, כגון גנרטורים של עצי תחביר ומפרשים (למשל flex & bison).

## ניתוח הבעיות בכל שלב

בניית קומפיילר היא תהליך מורכב הכולל מספר שלבים עיקריים, כאשר בכל שלב ישנן בעיות אלגוריתמיות ספציפיות שיש להתמודד איתן:

### ניתוח לקסיקלי (Lexical Analysis)

* זיהוי וקידוד של יחידות לקסיקליות (מילים שמורות, מזהים, מספרים וכו') מתוך הקלט הטקסטואלי של התוכנית.
* טיפול בתווים מיוחדים ובסימני פיסוק.

### ניתוח תחבירי (Parsing)

* בניית עץ חלוקה תחבירי (Parse Tree) המייצג את המבנה התחבירי של התוכנית.
* זיהוי ופתרון דו-משמעויות תחביריות.
* טיפול בשגיאות תחביריות.

### ניתוח סמנטי (Semantic Analysis)

* בדיקת סוג הנתונים של ביטויים ומשתנים והתאמתם לפעולות.
* איתור שגיאות סמנטיות כגון שימוש לא חוקי במשתנים.
* ניהול טבלת סמלים לאיחזור מידע על משתנים, פונקציות ומבני נתונים.

### ייצור קוד (Code Generation)

* תרגום התיאור המופשט של התוכנית לקוד מכונה ספציפי.
* בחירת רגיסטרים ומיקום זיכרון לאחסון משתנים.
* אופטימיזציית הקוד המיוצר לביצועים טובים יותר.

# סקירת אלגוריתמים בתחום הבעיה

תהליך הניתוח התחבירי, ה – Parsing, הוא התהליך המשמעותי והמורכב ביותר מבחינה אלגוריתמית ורעיונית בתהליך

הקומפילציה. כעת אציג שיטות שונות ואלגוריתמים שונים הנפוצים בשלב זה.

## מונחים

Derivation

בעברית, גִזרָה, היא בעצם רצף של rules Production, על מנת לקבל את מחרוזת הקלט.

במהלך תהליך ה – Parsing אנו בעצם מקבלים שתי החלטות עבור קלט מסוים:

.1 החלטה על ה – terminal-Non אשר יוחלף.

.2 ההחלטה על כלל הייצור, שבאמצעותו יוחלף ה – terminal-Non.

על מנת להחליט על איזה terminal-Non יוחלף בכלל הייצור, יכולות להיות לנו שתי אפשרויות:

**Left-most Derivation**

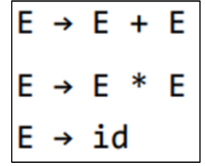
אפשרות זו קובעת כי תמיד ה – terminal-Non השמאלי ביותר הוא זה שיוחלף.

**Right-most Derivation**

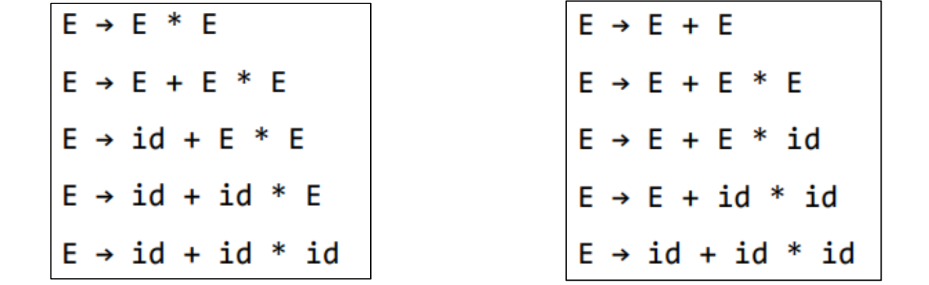
אפשרות זו קובעת כי תמיד ה – terminal-Non הימני ביותר הוא זה שיוחלף.

דוגמא

נתון ה - Grammar הבא:



עבור מחרוזת הקלט " id \* id + id "כך יראו שני סוגי ה – Derivation:

Right-most derivation Left-most derivation

## Left Factoring

אם יותר מ – rule Production אחד מתחיל באותה קידומת, אז ה – Parser לא יכול לבצע הכרעה באיזה מהחוקים הוא

צריך לבחור בשביל לנתח את הקלט הנוכחי.

דוגמא

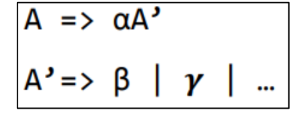
אם כלל ייצור מסוים נראה כך:



המנתח לא יודע להחליט אחרי איזה חוק לעקוב, כיוון ששני החוקים מתחילים באותו Terminal( או terminal-Non). על מנת להסיר בעיה זאת משתמשים בטכניקה שנקראת factoring Left.

factoring Left ממירה את ה – Grammar כך שלא יהיו חוסר הוודאויות האלו. היא עובדת כך שעבור כל קידומת שמשומשת יותר מפעם אחת יוצרים כלל חדש וההמשך של הכלל הישן משורשר לכלל החדש.

לדוגמא הכלל הקודם יוכל כעת להיראות כך:



עכשיו למנתח יש רק כלל אחד עבור הקידומת המסוימת הזאת, מה שמקל עליו לקבל החלטות.

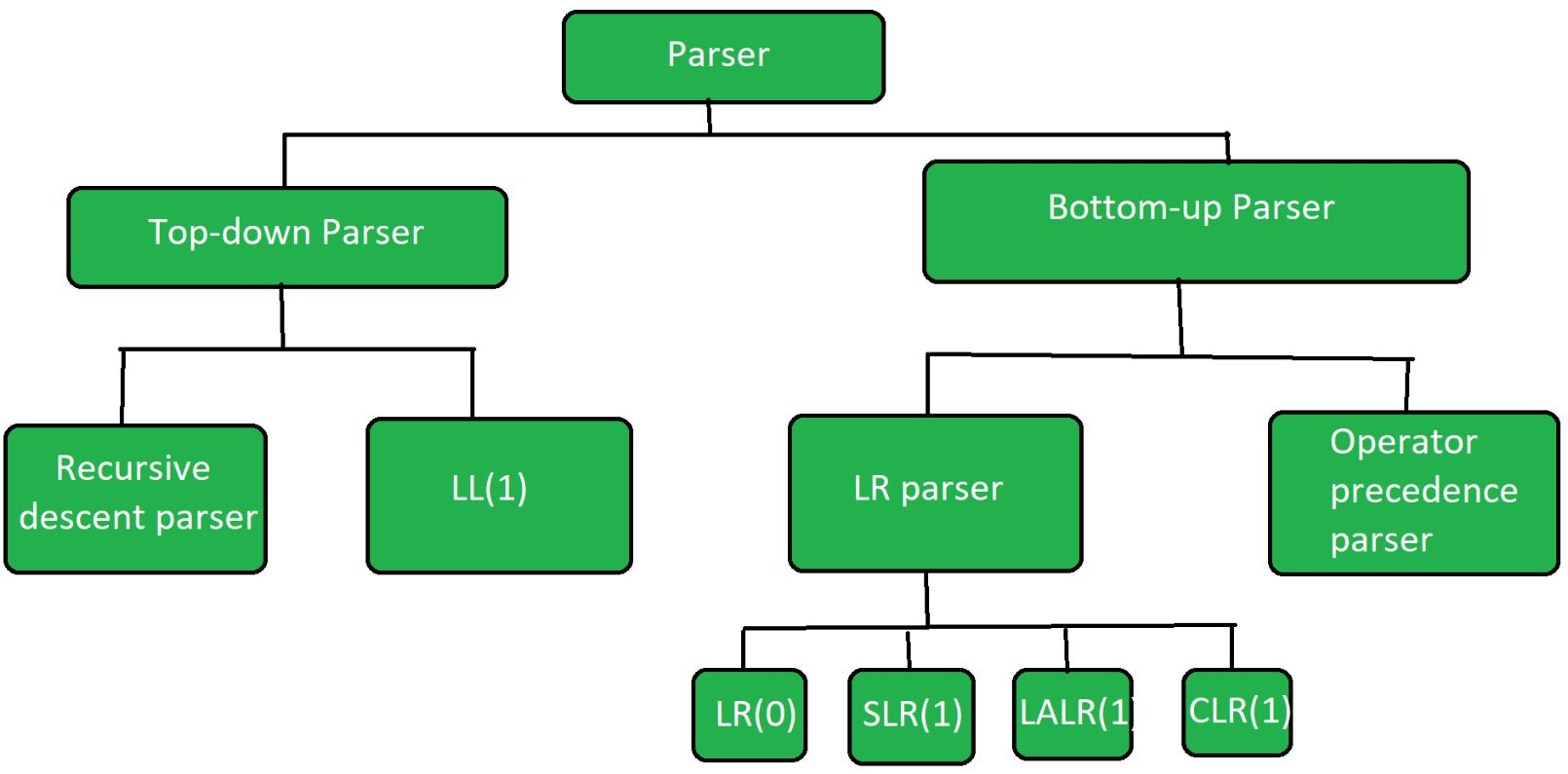
## אלגוריתמי פרסר

על מנת ליצור Tree Parse עליו יתבסס תהליך הקומפילציה, ישנם כמה אלגוריתמים הנקראים Algorithms Parsing. אלגוריתמים אלה מתחלקים לשני סוגים עיקריים.

Top Down Parsing (TDP) .1

Bottom Up Parsing (BUP) .2

שפות תכנות הן בדרך כלל languages free-Context. נהוג לפרש CFL באמצעות מכונות מצבים, ובאופן יותר ספציפי מכונות מצבים המשתמשות במחסנית )machines Pushdown). לכן האלגוריתמים שכעת אציג ישתמשו באוטומט מחסנית לרוב, על מנת לבצע את פעולת ה – Parsing.



מכיוון שבחרתי להשתמש באלגוריתם LALR אסביר רק במגיעה על שאר האלגוריתמים ולא אתעכב עליהם יותר מדי

## Top Down Parsing (TDP)

טכניקה בה עוברים מהחלקים העליונים לחלקים התחתונים של העץ התחבירי, על ידי שימוש בכללי השכתוב של Grammar השפה. עוברים מה – Grammar לקלט.

### LL Parser

אלגוריתם top-down המתבסס על ניתוח מהשמאל לימין (Left-to-right) ובנייה מלמעלה למטה (Top-down). מתאים לשפות תחביר LL(k), כלומר שפות שניתן לנתח אותן על ידי הסתכלות קדימה של k טוקנים. האלגוריתם מתחיל מסמל השורש ומנתח את המבנה התחבירי באופן רקורסיבי תוך כדי קידום בטוקנים.

### Recursive Decent Parsing

צורה נפוצה של TDP. בשיטה זו, כיוון שהיא שיטה שמתבססת על הגישה של Down Top, עץ הניתוח נוצר מלמעלה למטה, והקלט נקרא משמאל לימין. שיטה זו משתמשת בפונקציות עבור כל Terminal ו – terminal-Non שנמצא ב – Grammar השפה. parser descent Recursive יוצר את עץ הניתוח תוך מעבר רקורסיבי על הקלט, מה שיכול לגרום לו לסבול מ – tracking Back בגלל ה – tracking Back יעילותו של האלגוריתם יכולה להיות אף אקספוננציאלית

### Back tracking

כאשר ה – Parser משתמש בשיטה של parsing decent Recursive( וה – Grammar הוא לא factored Left), ייוצרו מצבים במהלך הניתוח של הקלט בהם המנתח יגיע למבוי סתום, וזאת כנראה בגלל שעשה בחירה לא נכונה של כלל מסוים בדרך. לכן, המנתח חוזר חזרה למקום האחרון בו ביצע הכרעה, ושם בוחר באופציה האחרת. החזרה הזאת למקום האחרון בו ביצע הכרעה, על מנת לבצע הכרעה שונה, נקראת tracking-Back. רק כאשר ניסה את כל האפשרויות ולא הצליח להתאים את הקלט לכללי השפה, ניתן להבין שהקלט הוא לא תקין מבחינת השפה.

## 

## Bottom Up Parsing (BUP)

טכניקה בה עוברים מהחלקים התחתונים לחלקים העליונים של העץ התחבירי, על ידי שימוש בכללי השכתוב של Grammar השפה. עוברים מהקלט אל ה - Grammar.

### Reduce Shift

הוא התהליך של הפחתת מחרוזת הקלט ל – terminal-non Starting של ה – Grammar.

שיטה זו משתמשת בשני שלבים הייחודיים ל – parsing up Bottom. שלבים אלו נקראים step Shift ו – step Reduce.

**Shift step**

שלב זה מבצע מעבר לסמל, Symbol, הבא שמגיע מהקלט, והוא נקרא גם symbol Shifted. סמל זה נדחף )PUSH )

למחסנית. ה – symbol Shifted מטופל כצומת אחת של עץ הניתוח.

**Reduce step**

כאשר המנתח מוצא כלל ייצור שלם, כלומר הסמלים שעל המחסנית תואמים לאחד מה – RHS ( side Hand Right )של כללי הייצור של ה – Grammar, ומחליף אותו ב – terminal-Non שנמצא ב – LHS של אותו כלל ייצור, שלב זה נקרא step Reduce. שלב זה בעצם עושה POP למחסנית עבור כל הסמלים שתואמים ל – RHS שמצא, ודוחף למחסנית את ה – LHS התואם.

יעילות האלגוריתם היא (n(O

### LR Parser

אלגוריתם bottom-up המתבסס על ניתוח משמאל לימין ובנייה מלמטה למעלה. מתאים לשפות תחביר LR(k) - שפות שניתן לנתח אותן על ידי הסתכלות אחורה של k טוקנים. האלגוריתם מתחיל מהטוקנים ובונה את עץ החלוקה התחבירי מלמטה למעלה תוך שימוש בטבלת מעברים. קיימים סוגים שונים של LR כמו SLR (Simple LR), LALR (Look-Ahead LR) ועוד.

### Operator Precedence Parser

אלגוריתם bottom-up המנתח ביטויים אריתמטיים על בסיס קדימויות הפעולות. מתאים במיוחד לשפות תכנות עם ביטויים אריתמטיים. האלגוריתם סורק את הטוקנים ומחליט איך לבנות את עץ החלוקה על בסיס טבלת קדימויות של האופרטורים.

### 

### LALR parser

פועל על ידי בניית עץ ניתוח מתוך רצף של סמלים (tokens) המייצגים את הטקסט. עץ הניתוח מתאר את המבנה התחבירי של הטקסט, ומאפשר לבצע עליו פעולות נוספות, כמו בדיקת תקינות, תרגום, ביצוע, ועוד.

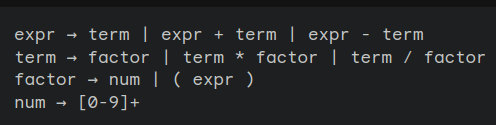
היתרונות של LALR Parser:

* יעילות: LALR Parser הוא יעיל יחסית בביצועים, הן מבחינת זמן והן מבחינת זיכרון.
* קלות שימוש: יחסית לסוגים אחרים של Parsers, LALR Parser קל יותר להבנה וליישום.
* כוח ביטוי: LALR Parser מסוגל לנתח תחביר של מגוון רחב של שפות תכנות ופורמטים טקסטואליים.

החסרונות של LALR Parser:

* מגבלות: לא ניתן להשתמש ב-LALR Parser עבור כל דקדוק LL (Left-to-Left).
* מורכבות: יישום LALR Parser יכול להיות מורכב יותר מאשר סוגים אחרים של Parsers.
* דוגמה לניתוח תחביר באמצעות LALR Parser:

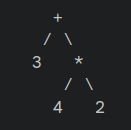
נניח שיש לנו את דקדוק LL הבא, המגדיר ביטוי אריתמטי פשוט:



הנה דוגמה לרצף של סמלים המייצגים ביטוי אריתמטי:



LALR Parser יבנה את עץ הניתוח הבא מתוך רצף הסמלים הזה:



עץ הניתוח הזה מתאר את המבנה התחבירי של הביטוי האריתמטי "3 + 4 \* 2". ניתן להשתמש בעץ הניתוח הזה כדי לבצע פעולות נוספות על הביטוי, כמו חישוב ערכו.

אופן פעולה של LALR Parser:

LALR Parser פועל על ידי שימוש במטריצת פעולות (Action Table) וטבלת מעברים (Goto Table). מטריצת הפעולות מגדירה איזו פעולה לבצע עבור כל זוג של מצב וטוקן. טבלת המעברים מגדירה את מצב היעד עבור כל זוג של מצב וסימן טרמינלי.

במהלך ניתוח תחביר, LALR Parser מתחיל במצב ההתחלה ומעבד את רצף הסמלים אחד לאחד. עבור כל טוקן, LALR Parser משתמש במטריצת הפעולות כדי לקבוע איזו פעולה לבצע. הפעולות האפשריות הן:

Shift: LALR Parser מזיז את ה-pointer על פני רצף הסמלים ומעבר למצב חדש.

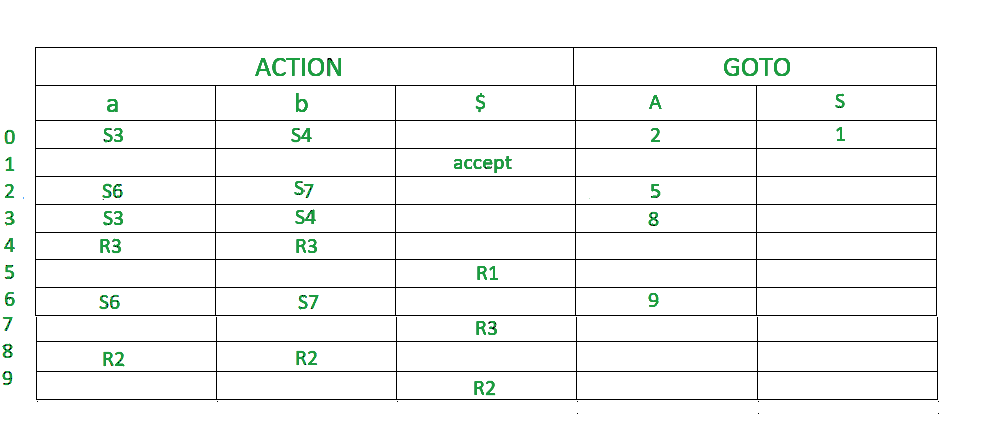
Reduce: LALR Parser מפחית את עץ הניתוח על ידי יישום כלל הפחתה מתאים.

Accept: LALR Parser מקבל את הטקסט.

Error: LALR Parser מזהה שגיאה בתחביר.

LALR Parser משתמש בטבלת המעברים כדי לקבוע את מצב היעד לאחר ביצוע פעולה.

### דוגמא לטבלאות הניתוח של LALR parser:



# ניתוח שלבי אלגוריתמיקה

בפרק זה אציג את האסטרטגיה והגישה לפתרון הבעיה האלגוריתמית הנקראת Compiler.

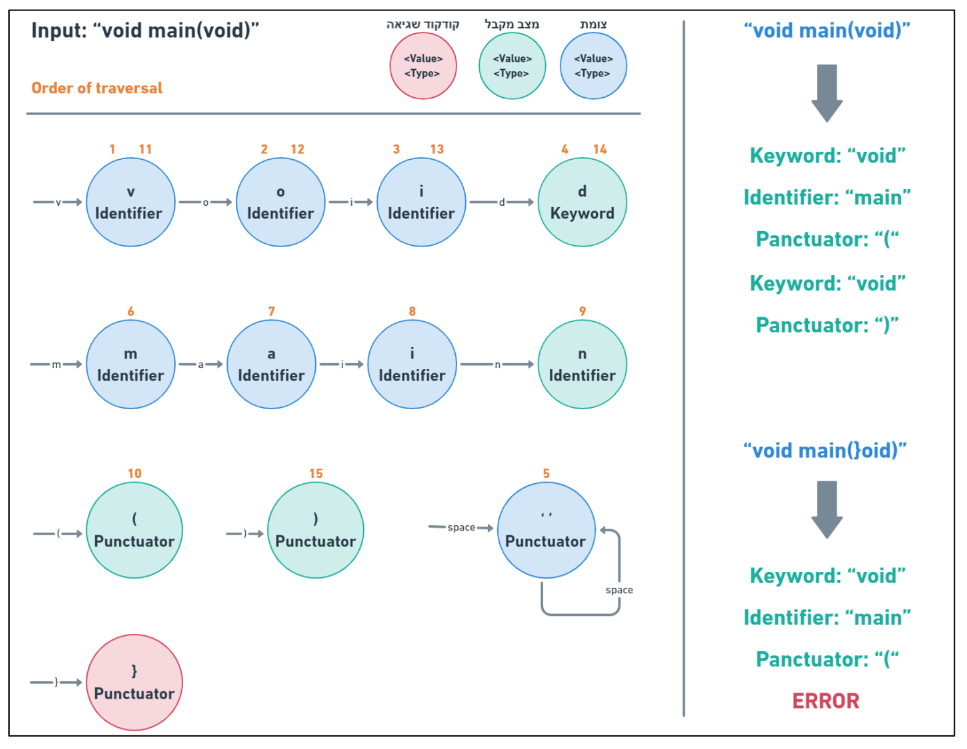
אייצג את ה – Compiler באמצעות מכונת מצבים. על מנת לעשות זאת ועל מנת לתרגם את קוד המקור, תוך שאיפה ליעילות זמן ריצה לינארית, אשתמש במבנה הנתונים גרף שייצג את מכונת המצבים של ה - Compiler. כעת אציג כיצד אסטרטגיה זו באה לידי ביטוי בשלבים השונים של תהליך הקומפילציה.

## ניתוח מילוני - Analysis Lexical

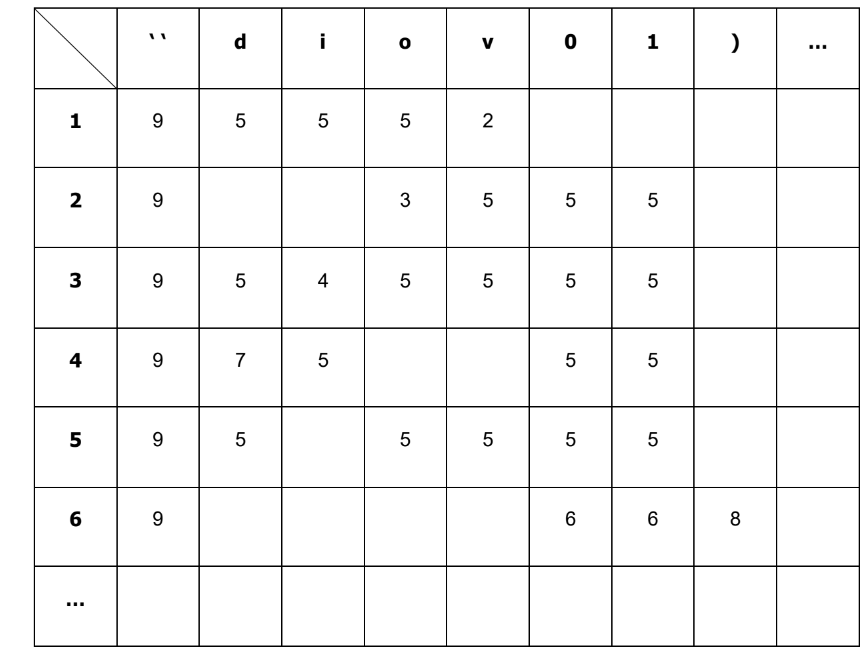
שוב, מטרתנו לשאוף ליעילות זמן ריצה לינארית. עקב כך אנו לא רוצים להתייחס לקוד המקור אותו אנחנו מתרגמים כטקסט. לכן אנו מתרגמים את קוד המקור לאסימונים בעלי משמעות בקוד, כפי שמתואר בחלק של הסקירה התיאורטית. המסלול שייווצר יתבסס על התווים מקוד המקור. המסלול יסתיים כאשר נגיע לקודקוד שהוא עלה, אשר יחזיר סוג אסימון

)Keyword ,Punctuator ,Operator ,Literal ,Identifier )ששמור אצלו. במקרה וקוד המקור יחיל רצף תווים לא חוקי, המסלול יוביל לקודקוד שגיאה.

דוגמא:



דוגמא עבור מטריצת הסמיכויות של הגרף המייצג את מכונת המצבים של המנתח המילוני.

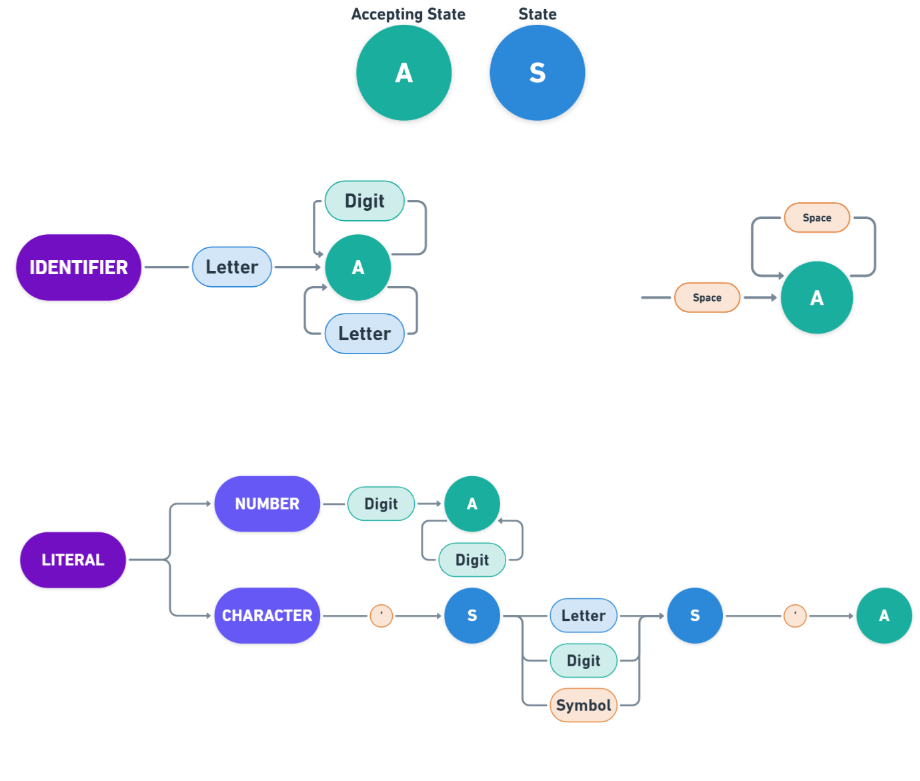
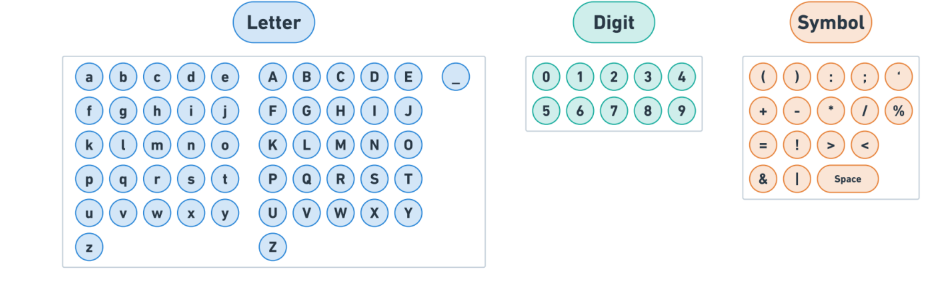


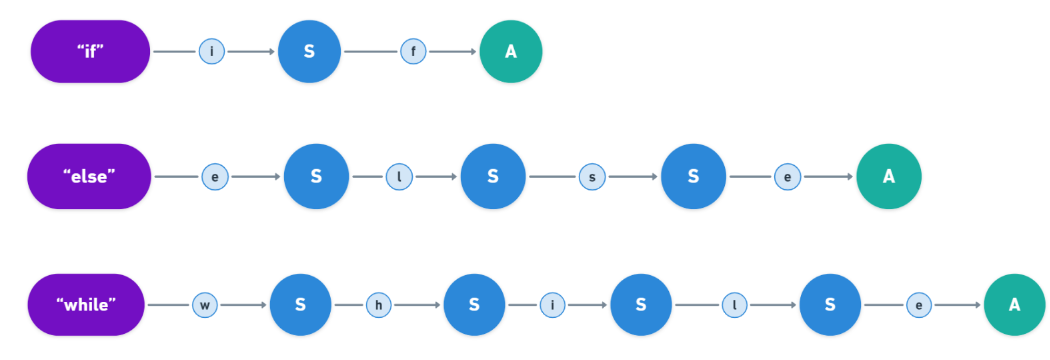
במטריצה שלהלן כל שורה מייצגת קודקוד. כל עמודה מייצגת שכנות של קודקוד. השכנות מסודרת לפי סוגי התווים. עבור

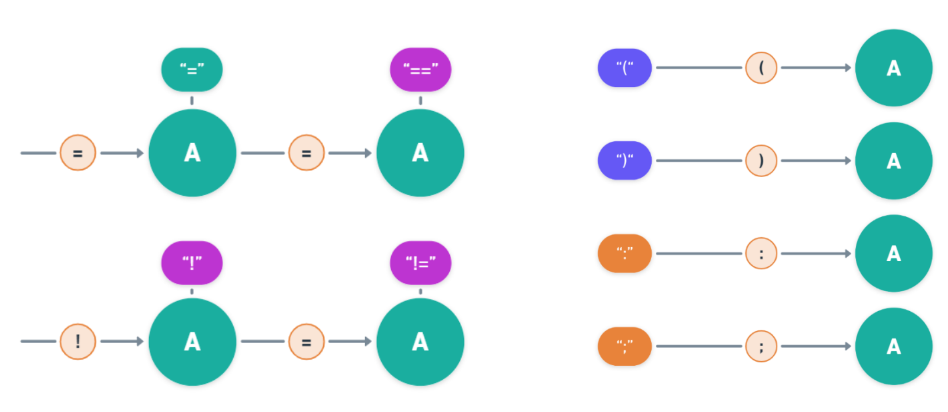
מצב מסוים ותו נוכחי מקוד המקור, נדע לאיזה מצב אנחנו צריכים לעבור. תא ריק מציין שאין שכנות בין המצב הנוכחי ובין

התו הנוכחי מקוד המקור, כלומר הגענו לעלה וזהו סוף האסימון.

## דוגמא לתרשים האוטומט הסופי





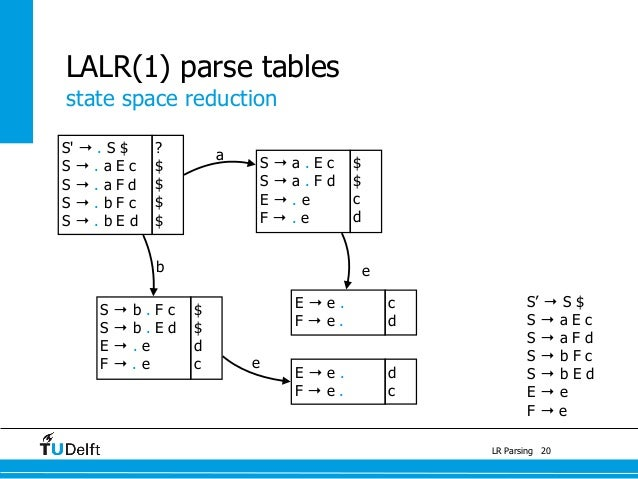


## ניתוח תחבירי – Analysis Syntax

בשלב זה נרצה לבדוק האם רצף האסימונים שקיבלנו מהשלב הקודם הוא רצף אסימונים תקין על פי דקדוק שפת המקור אותה אנחנו מתרגמים, ובנוסף נרצה ליצור עץ ניתוח שייצג את התוכנית התקינה. בשלב הקודם, שלב הניתוח המילוני, השתמשנו בטבלה עבור מימוש הגרף שמייצג את האוטומט הסופי של המנתח המילוני, בה עבור כל מצב ותו נוכחי ידענו לאיזה מצב עלינו לעבור. שלב הניתוח התחבירי מורכב יותר. התחביר של שלב זה הוא תחביר חופשי הקשר ולכן אינו יכול להתקבל על ידי אוטומט סופי. על מנת לממשו נשתמש באוטומט מחסנית.

כעת שהבנו זאת, טבלה אחת של מצב ותו נוכחי לא תספיק לנו לביצוע המטלה. אנו עדיין שואפים ליעילות זמן ריצה לינארית ולכן נשתמש באלגוריתם ניתוח מסוג Up Bottom, אלגוריתם ה - Reduce-Shift, ובאופן כללי, Parser LALR. עליו פירטתי בפרק סקירת אלגוריתמים בתחום הבעיה.

### תרשים פעולת LALR parser



### Parsing table & Stack

על מנת לממש את ה – Parser LR נשתמש במחסנית, ובשתי טבלאות שיחד ייקראו table Parsing. שתי הטבלאות הן .Goto table – ו Action table

### Action table

מציינת למנתח איזו פעולה - Shift, Reduce, Accept או הודעה על שגיאה, Error - הוא צריך לבצע, בהתאם למצב הנוכחי

והאסימון הבא מהקלט.

כל שורה תייצג מצב, וכל עמודה תייצג אסימון )Terminal / Token )שנמצא בשפה. נגיע לתא בטבלה על ידי המצב הנוכחי והאסימון הבא מקוד המקור. כל תא בטבלה יגיד לנו איזו פעולה עלינו לעשות, Shift או Reduce, ולאיזה מצב עלינו לעבור לאחר מכן. תא ריק מציין שהגענו למצב שלא יכול להתקיים על פי תחביר השפה, כלומר שגיאה. בטבלת ה – Action ישנו רק תא אחד שמציין את פעולת ה – Accept. תא זה הוא התא אליו נגיע לאחר שסיימנו את החוק הראשון בדקדוק במלואו, מה שמציין תוכנית תקינה.

### Goto table

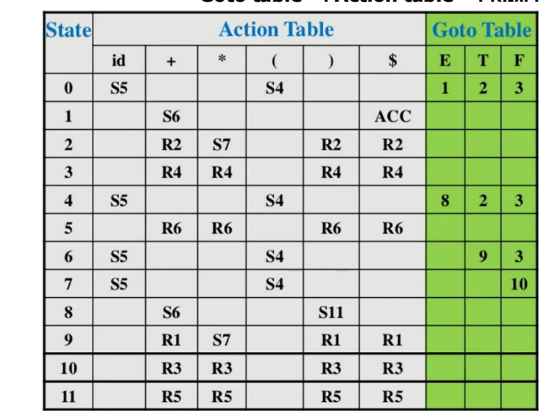
מציינת למנתח לאיזה מצב הוא צריך לעבור לאחר ביצוע פעולת ה – Reduce.

בטבלת ה – Goto כל שורה תייצג מצב, וכל עמודה תייצג משתנה )terminal-Non )שנמצא בתחביר השפה. נגיע לתא

בטבלה על ידי המצב הנוכחי והמשתנה ) terminal-Non )אשר נמצא בראש המחסנית. כל תא בטבלה יגיד לנו לאיזה מצב

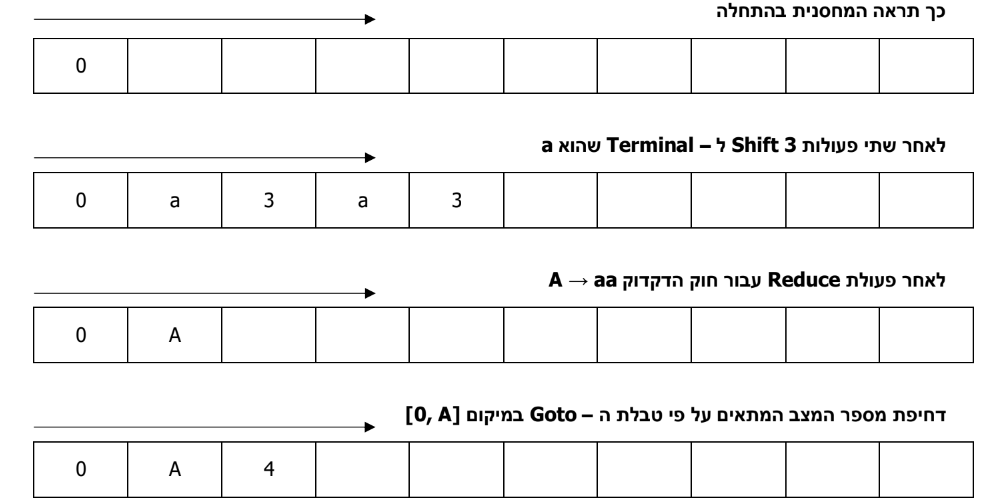
עלינו לעבור בהתאם למצב הנוכחי ולמשתנה שנמצא בראש המחסנית לאחר ביצוע פעולת ה – Shift או ה – Reduce אשר

ה – table Action אמרה לנו לעשות.



המחסנית תחזיק בתוכה זוגות של Terminal או terminal-Non ומספר מצב. בראש המחסנית תמיד יימצא מספר המצב הבא אליו צריך לעבור. המחסנית תאותחל עם איבר אחד שהוא מספר המצב הראשון, .0 עבור כל פעולת Shift נדחוף למחסנית שני איברים: האחד הוא ה Terminal הנוכחי מקוד המקור, והשני הוא מספר המצב כפי שצוין ב – table Action.

עבור כל פעולת Reduce נוציא מהמחסנית את כמות האיברים שנמצאת ב - RHS של חוק הדקדוק שצוין ליד פעולת ה – Reduce בטבלת ה – Action כפול 2 )משום שעבור כל איבר שנדחף למחסנית אנו גם דוחפים את מספר המצב אליו צריכים לעבור(, ונדחוף למחסנית את ה – terminal-Non שנמצא ב - LHS של אותו חוק דקדוק. כעת כל שנותר הוא לדחוף את מספר המצב הבא לראש המחסנית. ניתן לדעת מספר זה על ידי התא בטבלת ה – Goto שנמצא בעמודה של ה - LHS של חלק הדקדוק שכעת דחפנו, ובשורה של מספר המצב שנמצא תחתיו במחסנית.

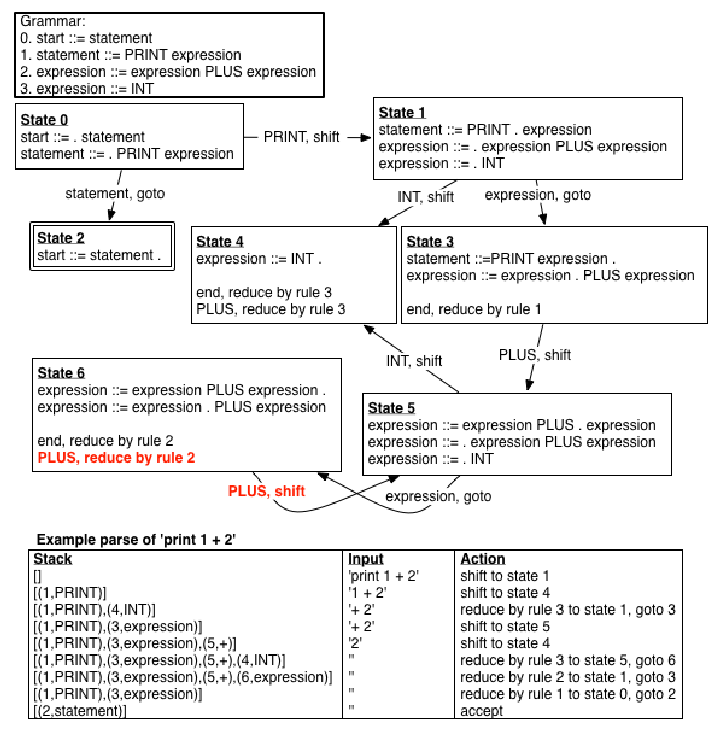


### עץ ניתוח

במהלך תהליך הניתוח התחבירי של תוכנית, נבנה במקביל עץ ניתוח המייצג את המבנה התחבירי של התוכנית. בנייה זו מתבצעת בשלב ה-Reduce של הפרסר, כאשר מיושם חוק דקדוק מסוים. בכל פעולת Reduce, נוסיף לעץ הניתוח צומת חדשה המייצגת את הסמל הלא-טרמינלי (Non-Terminal) שבצד שמאל (LHS) של חוק הדקדוק שיושם. הבנים של צומת זו יהיו הסמלים הטרמינליים (Terminals) והלא-טרמינליים שבצד ימין (RHS) של חוק הדקדוק, כפי שהם נמצאים במחסנית הפרסר. הבנים הללו שמוצאים מהמחסנית הם בעצמם עצים קטנים יותר שנבנו בשלבים קודמים של הפרסינג. כך, אט אט נבנה את עץ הניתוח המלא בתוך המחסנית של הפרסר. בסיום תהליך הניתוח התחבירי התקין, תישאר במחסנית יחידה אחת בלבד - עץ הניתוח המלא של התוכנית. כל צומת בעץ תכיל את הסוג התחבירי שלה (טרמינלי או לא-טרמינלי) ורשימה של הבנים שלה. בנייה מקבילה של עץ הניתוח במהלך הפרסינג מאפשרת ייצוג מבני של התוכנית, שישמש לשלבים הבאים בבניית הקומפיילר כגון ניתוח סמנטי וייצור קוד

### מימוש LALR parser

כפי שהסברתי, תהליך המימוש מורכב מכמה שלבים וכעת אציג אותם באמצעות שרטוט



כפי שניתן לראות התהליך מתחיל במצב 0 כאשר בכל שלב על הפרסר להחליט האם לעשות פעולת צמצום או פעולת שיפט או האם לעבור למצב מקבל כלומר האם נגמר הקלט והכל תקין

## דוגמאות משפת ACE

עד כה דיברתי על מצבים תיאורטיים ונתתי דוגמאות כלליות, כעת ארצה להציג דוגמאות אמיתיות מהקומפיילר שלי

### טבלאת goto & action

מפאת גודל הטבלה לא יכולתי למצוא דרך לצלם אותה ולהדביק כאן תצלום, לכן אצרף לינק לצפייה בקובץ האקסל שסידרתי לנוחיות הקוראים  
[לינק לטבלאה](https://docs.google.com/spreadsheets/d/1z9vxDL1jacOtAiXzdjEitfanu9dF5guJ4q3_Dt9odLA/edit?usp=sharing)

## ניתוח סמנטי

### מהו ניתוח סמנטי?

ניתוח סמנטי הוא שלב חיוני בתהליך הקומפילציה של שפות תכנות. לאחר שהקוד עבר בהצלחה את שלבי הניתוח המילוני (Lexer) והתחבירי (Parser), מגיע תורו של הניתוח הסמנטי. בשלב זה, מנתח הסמנטי בודק האם לתוכנית יש משמעות הגיונית בהתבסס על כללי השפה.

### ההבדל בין ניתוח תחבירי לניתוח סמנטי:

ניתוח תחבירי: בודק האם מבנה התוכנית תקין בהתאם לדקדוק השפה. כלומר, האם רצף הסמלים (tokens) מרכיב משפטים חוקיים בשפה.

ניתוח סמנטי: בודק האם לתוכנית יש משמעות הגיונית בהתבסס על כללי השפה. כלומר, האם השימוש במשתנים, פונקציות וביטויים תקין בהקשר נתון.

דוגמה:

המשפט הבא תקין מבחינה תחבירית, אך לא מבחינה סמנטית:

Int x := "hello" + 10

הסיבה לכך היא שמשתנה מסוג int (מספר שלם) לא יכול לקבל ערך מסוג string (מחרוזת).

### מטרות ניתוח סמנטי:

בדיקת תקינות סוגי המשתנים:

ודא שכל משתנה מקבל ערכים תואמים לסוגו.

ודא שביטויים מורכבים מורכבים מסוגים תואמים.

זיהוי שימוש במשתנים לא מוגדרים:

ודא שכל משתנה מוגדר לפני השימוש בו.

זיהוי ביטויים לא חוקיים:

ודא שפעולות מתבצעות בין סוגים תואמים.

ודא שאין ניסיונות לבצע פעולות לא חוקיות (לדוגמה, חילוק באפס).

זיהוי קוד מיותר:

זיהוי קוד שאינו משפיע על תוצאות התוכנית.

זיהוי קוד חשוד:

זיהוי קוד שעלול להוביל לשגיאות או התנהגות לא רצויה.

### יתרונות ניתוח סמנטי:

זיהוי שגיאות סמנטיות בשלב מוקדם של תהליך הפיתוח חוסך זמן ומאמץ בהשוואה לתיקון שגיאות בשלבים מאוחרים יותר.

זיהוי קוד מיותר וקוד חשוד מסייע בכתיבת קוד נקי, יעיל וקריא יותר.

זיהוי ביטויים לא חוקיים וקוד חשוד מסייע במניעת פגיעות אבטחה בתוכנה.

### טכניקות לניתוח סמנטי:

ניתוח טבלאות סמלים - שמירה על מידע אודות כל משתנה, כולל סוגו, ערכו והקשרים שלו.

ניתוח זרימת נתונים - מעקב אחר האופן שבו ערכים זורמים דרך התוכנית.

ניתוח סטטי - ניתוח התוכנית ללא ביצועה בפועל.

ניתוח דינמי - ניתוח התוכנית תוך כדי ביצועה.

### בדיקת ייחודיות שמות המשתנים

שם משתנה לא יכול להיות מוגדר מספר פעמים באותו ה – Scope. Scope, או תחום, הוא מושג חשוב בתכנות. הוא מגדיר את האזור שבו משתנה זמין.

### אסטרטגיה לפתרון

Grammar Attribute הוא צורה מיוחדת של Grammar free-Context בו אחד או יותר terminal-Non מקבל מידע נוסף, תכונה נוספת, Attribute, על מנת לספק לנו מידע תלוי הקשר, sensitive-Context. לכל תכונה כזאת יש תחום מסוים של ערכים, לדוגמא, מספר, מחרוזת, ביטוי ועוד. כל Attribute היא בעצם צמד של <value Attribute ,name Attribute<

Grammar Attribute הוא דרך לספק סמנטיקה ל – CFG, והוא יכול לעזור לנו להגדיר בד בבד את התחביר והסמנטיקה של שפה. אם נסתכל על ה – Grammar Attribute כעץ, בעצם נראה שהוא מסוגל להעביר מידע בין הצמתים השונים בעץ, מה ש – CFG לבדו לא מסוגל לעשות.

## הפקת קוד

יצירת קוד הוא השלב האחרון בתהליך הקומפילציה של שפות תכנות. בשלב זה, מתרגם הקומפילר את עץ הניתוח המופשט (AST) והטבלת סמלים, המייצגים תוכנית תקינה לאחר שעברה את שלבי הניתוח המילוני (Lexer), התחבירי (Parser) והסמנטי (Semantic Analyzer), להוראות בשפת אסמבלי. הוראות אלו הן הוראות מכונה שניתן לבצע על ידי המעבד.

### מטרות יצירת קוד:

תרגום AST להוראות בשפת אסמבלי: יצירת קוד לוקחת את AST המייצג את התוכנית ברמת הפשטה גבוהה ומפשטת אותה להוראות מכונה ברמת הפשטה נמוכה.

ייעול הקוד: יצירת קוד יכולה לייעל את הקוד שנוצר על ידי שימוש בטכניקות שונות, כגון הקצאת רישומים, הסרת הוראות מיותרות וסידור מחדש של הוראות.

יצירת קובץ אובייקט: יצירת קוד מייצרת קובץ אובייקט המכיל את הוראות המכונה של התוכנית. קובץ אובייקט זה יכול להיקשר עם קבצי אובייקט אחרים על ידי linker כדי ליצור קובץ הפעלה.

### אסטרטגיות ליצירת קוד:

תרגום מבוסס טבלאות: גישת תרגום מבוססת טבלאות משתמשת בטבלאות כדי להגדיר את התרגום של כל צומת ב-AST להוראות בשפת אסמבלי.

תרגום מבוסס דקדוק: גישת תרגום מבוססת דקדוק משתמשת בדקדוק מיוחד כדי להגדיר את התרגום של כל צומת ב-AST להוראות בשפת אסמבלי.

תרגום מבוסס עץ: גישת תרגום מבוססת עץ משתמשת בעץ הניתוח המופשט (AST) עצמו כדי להנחות את תהליך יצירת הקוד.

### שיקולים ביצירת קוד:

נכונות: הקוד שנוצר צריך להיות שקול מבחינה סמנטית לתוכנית המקורית.

יעילות: הקוד שנוצר צריך להיות יעיל מבחינת זמן וזיכרון.

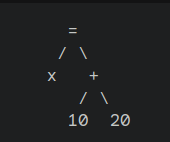
ניידות: הקוד שנוצר צריך להיות נייד בין פלטפורמות שונות.

### דוגמה:

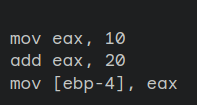
נניח שיש לנו את המשפט הבא בשפת C:

int x = 10 + 20;

ה-AST עבור משפט זה עשוי להיראות כך:



יצירת קוד תתרגם את ה-AST הזה להוראות בשפת אסמבלי הבאות:



הוראות אלו יטעינו את המספר 10 לרישום eax, את המספר 20 ל-eax, וישמרו את התוצאה במשתנה x.

## טבלת סמלים

טבלת סמלים היא מבנה נתונים חשוב המשמש לאורך כל תהליך הקומפילציה של שפות תכנות. היא מאחסנת מידע על כל משתנה בתוכנית, כגון שמו, סוגו, ערכו ותחום הגישתו (Scope).

### מטרות טבלת הסמלים:

שמירה ואחזור מידע על משתנים: טבלת הסמלים מאפשרת לנו לשמור ולמצוא מידע על כל משתנה בתוכנית בצורה יעילה.

ניהול תחומי גישה (Scope Management): טבלת הסמלים עוזרת לנו לעקוב אחר תחומי הגישה של משתנים, כלומר באיזה חלקים בתוכנית כל משתנה זמין.

בדיקות סמנטיות: טבלת הסמלים משמשת לבדיקות סמנטיות שונות, כגון בדיקת ייחודיות שמות משתנים ובדיקת סוגי נתונים.

### ייצוג טבלת הסמלים:

ניתן לייצג את טבלת הסמלים באמצעות מבני נתונים שונים, כגון מערך, רשימה או מילון. ייצוג נפוץ הוא שימוש במילון (Dictionary/Hash Table). כיוון שכל משתנה בתוכנית חייב להיות בעל שם ייחודי, ניתן למפות כל משתנה למיקום ספציפי במילון באמצעות פונקציית גיבוב (Hash Function).

### יתרונות שימוש במילון:

גישה יעילה: שימוש במילון מאפשר גישה מהירה למידע על משתנים בסיבוכיות זמן ריצה קבועה, O(1).

יעילות זיכרון: שימוש במילון מאפשר ניצול יעיל של זיכרון, כיוון שהמילון מוקצה דינמית בהתאם לצורך.

### ניהול תחומי גישה:

ניהול תחומי גישה (Scope Management) הוא חלק חשוב משימוש בטבלת הסמלים. תחום גישה מגדיר את האזור בתוכנית שבו משתנה זמין. טכניקות נפוצות לניהול תחומי גישה:

אחסון מילונים מקוננים: ניתן לאחסן מילונים מקוננים בטבלת הסמלים, כאשר כל מילון מייצג תחום גישה אחר.

שימוש בסימון טוקן מיוחד: ניתן להשתמש בסימון טוקן מיוחד (לדוגמה, #) כדי לציין את תחילת ותום תחום גישה.

טבלת סמלים היא כלי חשוב בתהליך הקומפילציה, המאפשרת ניהול יעיל של מידע על משתנים, ניהול תחומי גישה וביצוע בדיקות סמנטיות. שימוש במילונים מאפשר גישה מהירה ויעילה למידע על משתנים, תוך ניצול יעיל של זיכרון.

## טיפול בשגיאות

טיפול בשגיאות הוא חלק חשוב בתהליך הקומפילציה של שפות תכנות. מטרתו לזהות ולדווח על טעויות בקוד המקור שעלולות למנוע מהתוכנית להתבצע בצורה נכונה.

### גישות שונות לטיפול בשגיאות

1. יציאה מיידית (Exit on Error):

יתרונות:

פשוטה ליישום.

מונעת ביצוע קוד שגוי.

חסרונות:

מציגה רק את השגיאה הראשונה שנמצאה.

מחייבת את המשתמש לתקן שגיאות אחת אחת ולקמפל שוב ושוב.

2. מצב חירום (Panic Mode):

יתרונות:

ממשיכה בקומפילציה לאחר גילוי שגיאה.

עשויה לזהות שגיאות נוספות.

חסרונות:

עשויה לדלג על חלקים משמעותיים בקוד.

קשה יותר לאבחן את מקור השגיאה.

3. מצב משפט (Statement Mode):

יתרונות:

מנסה לתקן את הקוד באופן אוטומטי.

עשויה לאפשר המשך קומפילציה מוצלחת.

חסרונות:

מסובכת יותר ליישום.

עלולה ליצור לולאות אינסופיות או שגיאות נוספות.

4. גישות נוספות:

בחירת גישת הטיפול בשגיאות תלויה במספר גורמים, כגון:

מורכבות השפה.

סגנון התכנות הרצוי.

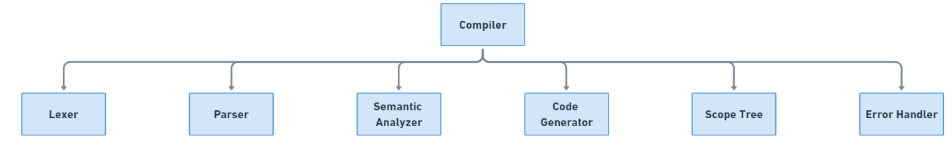
קהל היעד של הכלי.

טיפול בשגיאות תחביר הוא נושא חשוב בתכנות. קיימות גישות שונות לטיפול בשגיאות, לכל אחת מהן יתרונות וחסרונות משלה. בחירת הגישה המתאימה תלויה במספר גורמים, כגון מורכבות השפה, סגנון התכנות הרצוי וקהל היעד של הכלי.

במסגרת פרויקט זה, בחרתי בגישת "יציאה מיידית" (Exit on Error) בשל פשטותה ויעילותה.

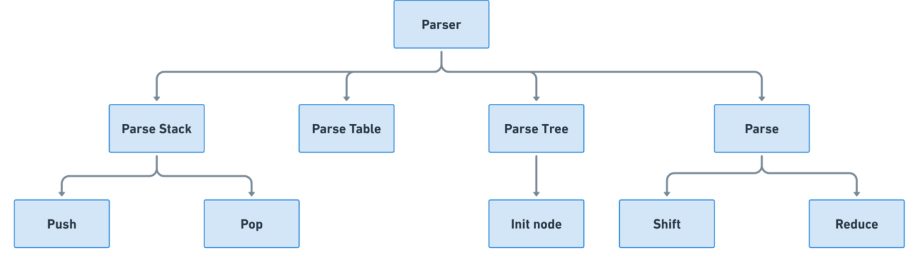
# Top down level design

## Compiler

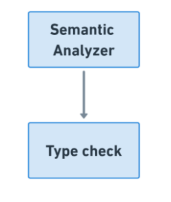


## Lexer

## Parser



## Semantic analyzer



## Code generation



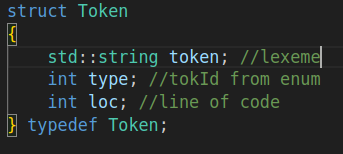
# מודולים

כעת ארחיב על כל מודול ואפרט מה תפקידו, מהם ה – Structs שמכיל, אילו קבצים שייכים לאותו מודול, מהם המודולים

הנמצאים תחתיו, מהן הפונקציות שמכיל ומהי היעילות האלגוריתמית של כל אחת מהן.

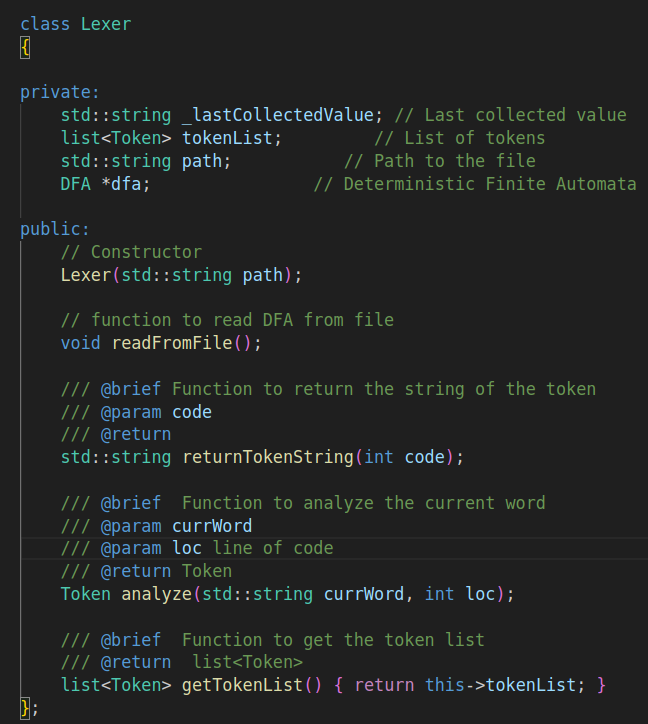
## Token

מודל המייצג אסימון שהוא אבן יסוד ללקסר



## Lexer

‫מודל‬‫ המכיל‬ ‫את‬ ‫כל‬ ‫הפונקציות‬ ‫הקשורות‬ ‫לשלב‬ ‫ה‬ ‫‪CodeGeneration‬‬ ‫בפרסר‪.‬‬



## 

| יעילות | תיאור | פלט | קלט | פונקציה |
| --- | --- | --- | --- | --- |
| O(1) | אתחול |  |  | lexer |
| O(1) | קריאה מקובץ |  |  | readFromFile |
| O(N) | מנתחת מילה ומתייגת אותה כטוקן | טוקן | מילה נוכחית ושורת קוד נוכחית | analyze |
| O(1) | החזרת רשימת הטוקנים | רשימת טוקנים |  | getTokenList |
| O(1) | מחזיר מילה עבור טוקן לדיבאגינג | מחרוזת | מספר | returnTokenString |

## 

## 

## Parser

מודל האחראי על המרת רשימת הטוקנים לעץ סינטקס אבסטרקטי



## 

## 

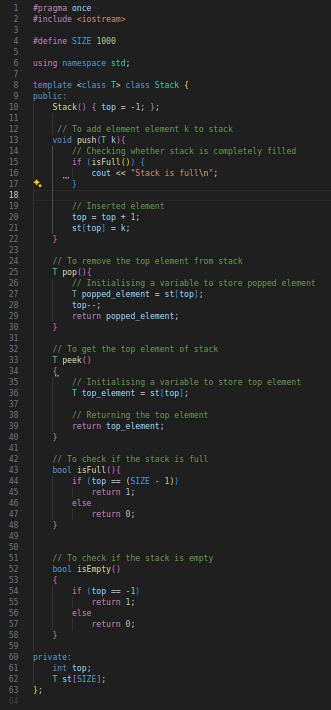
## 

| יעילות | תיאור | פלט | קלט | פונקציה |
| --- | --- | --- | --- | --- |
| O(1) | אתחול |  |  | parser |
| O(1) | הכרזת חוקי השפה |  |  | setRuleList |
| O(1) | קבלת רשימת הטוקנים |  | רשימת טוקנים | setInputList |
| O(1) | קריאת גוטו ואקשיין טייבל מהזיכרון (קובץ) |  |  | generateParseTable |
| O(1) | פעולה מאתחלת |  |  | Parse |
| O(1) | ממפה סוגי טוקנים לסוגי אקשנים |  |  | makeMap |
| O(N) | מדפיס את העץ |  | שורש העץ, האם סופי, וקידומת | printAST |

## 

## Stack

מודל האחראי לשמור את הנתונים בצורת ראשון פנימה אחרון החוצה



## 

## 

| יעילות | תיאור | פלט | קלט | פונקציה |
| --- | --- | --- | --- | --- |
| O(1) | דחיפת איבר לראש המחסנית |  | איבר לדחוף למחסנית | push |
| O(1) | הסרה והחזרה של האיבר שבראש המחסנית | האיבר שבראש המחסנית |  | pop |
| O(1) | הצצה לראש המחסנית | הערך שבראש המחסנית |  | peek |
| O(1) | בדיקת מחסנית | ערך בוליאני שמתאר האם המחסנית ריקה |  | isEmpty |
| O(1) | פעולה מאתחלת |  |  | Stack |

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## 

## semanticAnalysis

מודל האחראי על זיהוי שגיאות מסוג שגיאות התאמת טיפוסים וכדומה

# 

# 

# 

# 

# 

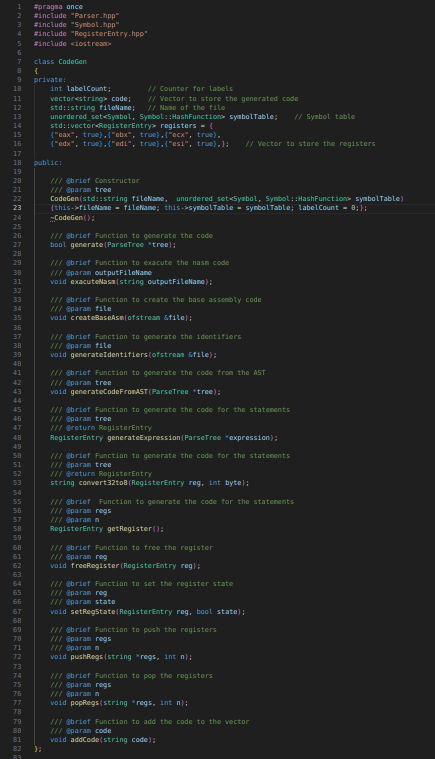
## 

| יעילות | תיאור | פלט | קלט | פונקציה |
| --- | --- | --- | --- | --- |
| O(1) | אתחול |  | עץ פרסר | semanticAnalysis |
| O(N) | יצירת טבלאת הסימנים |  | עץ פרסר וטווח | createSymbolTable |
| O(N) | מבצע ניתוח סמנטי | עץ פרסר |  | semantic |
| O(N) | פה מתבצע הניתוח בפועל | עץ פרסר | עץ פרסר וטווח | semanticHelper |
| O(1) | החזרת טבלאת סימנים | טבלת סימנים |  | getSymbolTable |
| O(1) | בודק האם טוקן מסויים הוא פעולת השוואה | אמת או שקר | טוקן | isCmpOp |
| O(1) | בודק טיפוס מסויים של טוקן | מספר | טוקן | identifyType |
| O(N) | הדפסת טבלה לדיבוג |  |  | printSymbolTable |

## 

## Code generation

מודל האחראי על הפקת הקוד מהעץ המוגמר



## 

| יעילות | תיאור | פלט | קלט | פונקציה |
| --- | --- | --- | --- | --- |
| O(1) | מוסיף שורת קוד לרשימת השורות | אמת או שקר | מחרוזת | addCode |
| O(N) | קורא לכל פונקציות יצירת הקוד |  | עץ פרסר | generate |
| O(1) | מפעיל פקודות להרצת קוד אסמבלי |  | שם קובץ | exacuteNasm |
| O(1) | יוצר בסיס לקוד אסמבלי |  | קובץ | create BaseAsm |
| O(N) | יצירת כל המשתנים המוכרזים לזיכרון |  | קובץ | generate Identifiers |
| O(1) | כאן מתבצעת עיקר היצירה של הקוד מאייס לאסמבלי |  | עץ פרסר | generateCodeFromAst |
| O(N) | מחשבת את הביטוי שבעץ | רגיסטר עם תוצאת הביטוי | עץ פרסר | generateExpression |
| O(1) | ההמרת ביטוי ברגיסטר 32 ביט לרגיסטר 8 ביט בהתאם לדגל | מחרוזת | רגיסטר, דגל | convers32to8 |
| O(1) | מחזיר רגיסטר חופשי | רגיסטר |  | getRegister |
| O(1) | קובע האם רגיסטר בשימוש או לא |  | רגיסטר, מצב | setRegState |
| O(1) | דחיפת כל הרגיסטרים למחסנית |  | וקטור רגיסטרים וכמות | pushRegs |
| O(1) | שליפת כל הרגיסטרים מהמחסנית |  | וקטור רגיסטרים וכמות | popRegs |
| O(1) | שחרור זיכרון הרגיסטר |  | רגיסטר | freeRegister |

## 

# 

# 

# אלגוריתם ראשי

Compiler

.1 קבל מהמשתמש את קובץ הקלט .gal וקובץ הפלט .asm.

.2 אתחל את Handler Error

.3 אתחל את Lexer

.4 הכנס לתוך הרשימה tokens את פלט פונקציית הLexer

a. התחל

b. אתחל את גרף האוטומט הסופי של המנתח המילוני

c. עבור על התווים בקובץ הקלט:

i. עבור אל המצב ההתחלתי עבור התו הנוכחי בקוד המקור

ii. כל עוד לא הגעת למצב ,-1 בצע:

. 1 עבור למצב הבא על פי המצב הנוכחי והתו הנוכחי מקוד המקור

. 2 התקדם תו אחד בקוד המקור

iii. צור אסימון על פי העלה שהגעת אליו

iv. הוסף את האסימון לרשימת האסימונים

d. סיים

.5 אתחל את Syntax

.6 הכנס לתוך העץ AST את פלט פונקציית הParser

a. התחל

b. אתחל את המחסניות ואת הטבלאות Action & Goto

c. הכנס למשתנה a את האסימון הראשון מקוד המקור

d. כל עוד לא עברת על כל קוד המקור בצע:

i. . הכנס למשתנה s את המצב שנמצא בראש המחסנית:

:בצע ,shift == Action[s, a[ אם .ii

.1 דחוף את a למחסנית העצים

.2 דחוף את ]a ,s[Action למחסנית המצבים

. 3 הכנס למשתנה a את האסימון הבא מקוד המקור

:בצע ,Action[s, a] == reduce u אם אחרת .4

.5 קבע את R להיות ה terminal-Non שנמצא בצד השמאלי של הu rule.

.6 הוצא ממחסנית העצים n איברים ובנה מהם עץ שהאב שלו הוא R, כש-n הוא

מספר האיברים הימיני בכלל u.

.7 הוצא ממחסנית המצבים n איברים )n זהה(.

.8 הכנס למשתנה v את המצב הנמצא כעת בראש מחסנית המצבים.

.9 דחוף את העץ החדש למחסנית העצים.

.10 דחוף את ]Goto[v, A למחסנית המצבים.

:בצע ,Action[s, a] == accept אם אחרת .iii

.1 סיים והחזר את עץ הניתוח שנמצא בראש המחסנית

iv. אחרת, בצע:

.1 דווח על השגיאה שנמצאה

e. סיים

.7 אתחל את Symantic

.8 הכנס לתוך AST את העץ החדש שנוצר מפונקציית הsymantic

a. עבור על כל התתי עצים בAST בסריקה תוכית:

i. הכנס לתוך root את הערך בעלה

Ii. אם ערך הroot הוא identifier או litearal או term או expression או assignment:

.1 בצע בדיקת תקינות סמנטית בהתאם ל table symbol.

.2 הגדר את type העלה לפני הבדיקה הסמנטית.

b. החזר

.9 אתחל את CodeGeneration

.10 בצע את פעולת generate

a. סרוק את העץ בסריקה תוכית:

i. עבור כל statemnt שלח את העלה לפונקציה המטפלת בסוג הstatement.

b. רשום את כל הקוד לקובץ

c. סיים

.11 סיים

# 

# 

# 

# 

# 

# 

# 

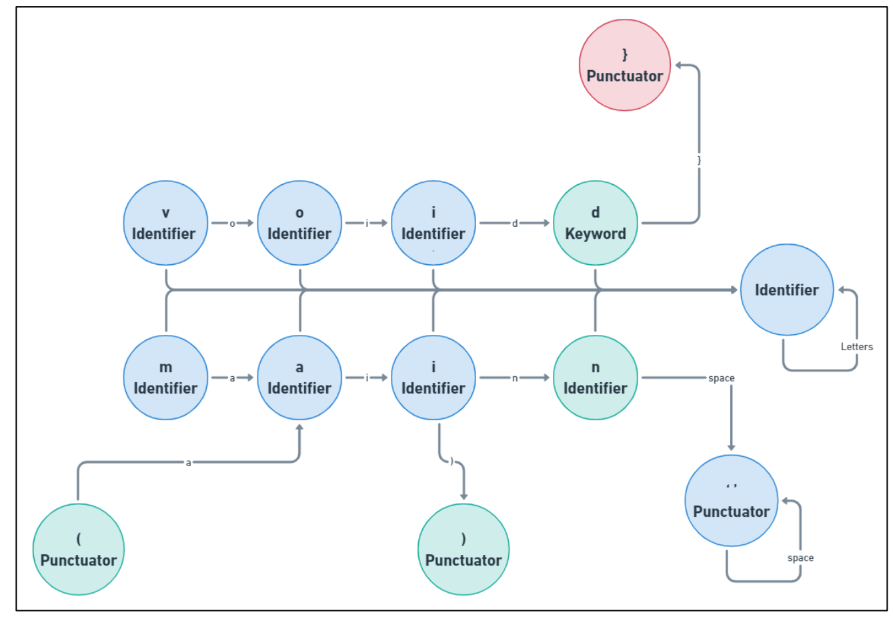
# מבני נתונים

## מנתח מילוני – Analysis Lexical

עבור המנתח המילוני השתמשתי במכונת מצבים סופית המיוצגת על ידי גרף.

משקלי הקשתות בגרף הן אותיות, כך שעבור כל אות ומצב נוכחי נעבור למצב ספציפי אחר )אוטומט דטרמיניסטי(. כאשר נגיע למצב בו אין לאן להתקדם עבור התו הבא מהקלט, סימן שהגענו למצב סופי, ונחזיר את ה – Token השמור במצב זה. עבור תו לא צפוי נגיע למצב שגיאה.

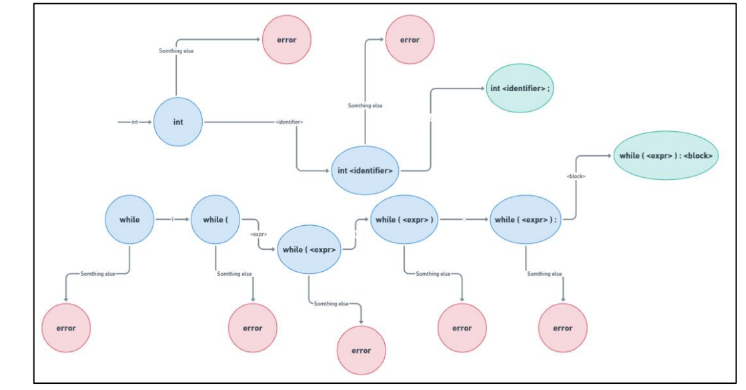
מבנה נתונים זה בעצם מאפשר לנו לשמור על יעילות זמן ריצה לינארית, (n(O, משום שאנו לא צריכים לשאול שום שאלות, או לבדוק מספר אופציות עבור כל מצב. כל המצבים האפשריים נמצאים בתוך מבנה הנתונים וכך בכל מצב אנחנו יכולים להכריע איזו פעולה עלינו לעשות.



## מנתח תחבירי – Analysis Syntax

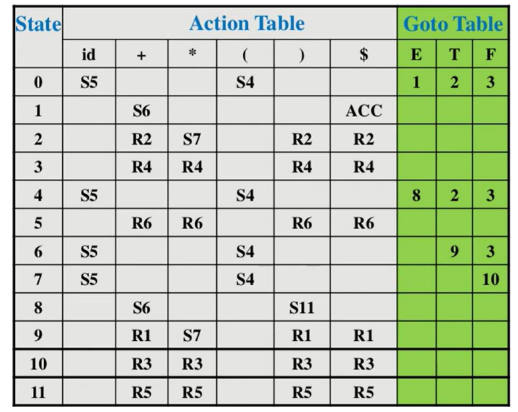
עבור המנתח התחבירי השתמשתי באוטומט מחסנית המיוצג על ידי גרף. כל צומת בגרף מייצג מצב נוכחי, מיקום, בחוק בתחביר השפה, והקשתות בגרף היו ה – Tokens שקיבלנו מהמנתח המילוני. עבור מצב נוכחי, וה – Token הבא מהקלט נדע איזו פעולה עלינו לעשות בניתוח מטה מעלה )Error ,Accept ,Reduce ,Shift). עבור Token לא צפוי מהקלט נגיע למצב שגיאה.

מבנה נתונים זה מאפשר לנו לקבל החלטה עבור כל מצב אפשרי במהלך תהליך הקומפילציה. אנו לא צריכים לשאול שום שאלות, או לבדוק מספר אופציות, מה שעוזר לנו לשמור על יעילות זמן ריצה לינארית, (n(O.



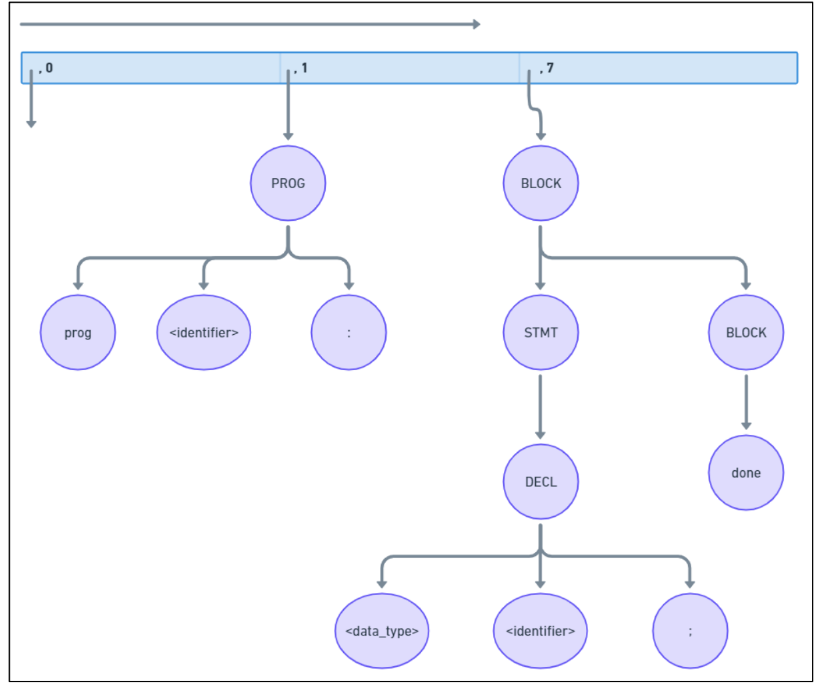
## Parse Table

על מנת לממש את הגרף שמייצג את אוטומט המחסנית השתמשתי בשתי טבלאות, tables Goto & Action, אשר יחד נקראות table Parse. אלו בעצם שתי מטריצות כך שבטבלת ה – Action כל תא הוא הפעולה שצריך לעשות עבור המצב והאסימון הנוכחי, ובטבלת ה – Goto כל תא הוא מספר המצב אליו צריך לעבור עבור המצב הנוכחי וה – terminal-Non שבראש מחסנית הניתוח. הסבר מפורט על טבלאות ה – Action וה – Goto ניתן למצוא ב - Stack & table Parsing בפרק האסטרטגיה. גישה לתאים השונים במטריצות תתבצע ביעילות אלגוריתמית קבועה, (1)O, ובכך תעזור לנו לשמור על יעילות זמן ריצה לינארית, (n(O, של הקומפיילר.



## Parse Stack

עבור אוטומט המחסנית אנו כמובן צריכים מחסנית. עבור המחסנית של האוטומט השתמשתי במחסנית בה כל איבר מכיל את שורש העץ שנבנה עד כה בתא זה במחסנית, ובנוסף כל איבר במחסנית שומר לאיזה מצב צריך ללכת לאחר מכן. המחסנית מאפשרת לנו "לזכור" את הדברים שראינו כך שבהמשך נוכל להוציא אותם מהמחסנית ולהשתמש בהם במקום לחפשם שוב. בכך היא עוזרת לשמור על יעילות זמן ריצה לינארית, (n(O, של הקומפיילר.



## דקדוק השפה – Rules Production

על מנת לשמור את כללי היצירה של השפה נשתמש במערך המכיל עבור כל כלל יצירה את ה – terminal-Non שב – LHS שלו, ואת מספר הסמלים שיש ב – RHS של כלל היצירה.

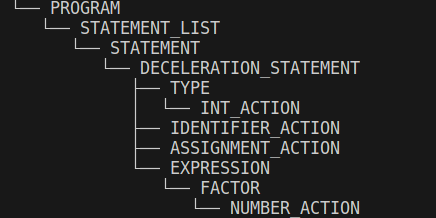
כך נוכל לדעת בסיבוכיות זמן ריצה קבועה, (1)O, כמה איברים צריך להוציא מהמחסנית בכל Reduce, ובנוסף לכך נוכל לדעת איזה terminal-Non צריך להיות בשורש העץ לאחר פעולת ה – Reduce.

## עץ הניתוח – Tree Parse

נרצה שהמנתח התחבירי יחזיר לנו עץ המייצג את התוכנית.

כל צומת בעץ יכיל Terminal או terminal-Non. לכל צומת המכיל terminal-Non, יהיה מערך של הבנים שלו, וכל עלה, צומת המכיל Terminal, יחזיק Token מקוד המקור. העץ נבנה אט אט כ תתי עצים בתוך ה – Stack Parse, והוא נבנה כך שאם נעבור עליו בסריקה תוכית, Inorder

traversal, נקבל את תוכנית המקור. העץ מייצג את מבנה התוכנית, ובעצם מתוך המבנה שלו אנו יכולים לקבל החלטות עבור כל מיקום בעץ. דבר זה עוזר לנו לשמור על יעילות אלגוריתמית לינארית, (n(O, של הקומפיילר.

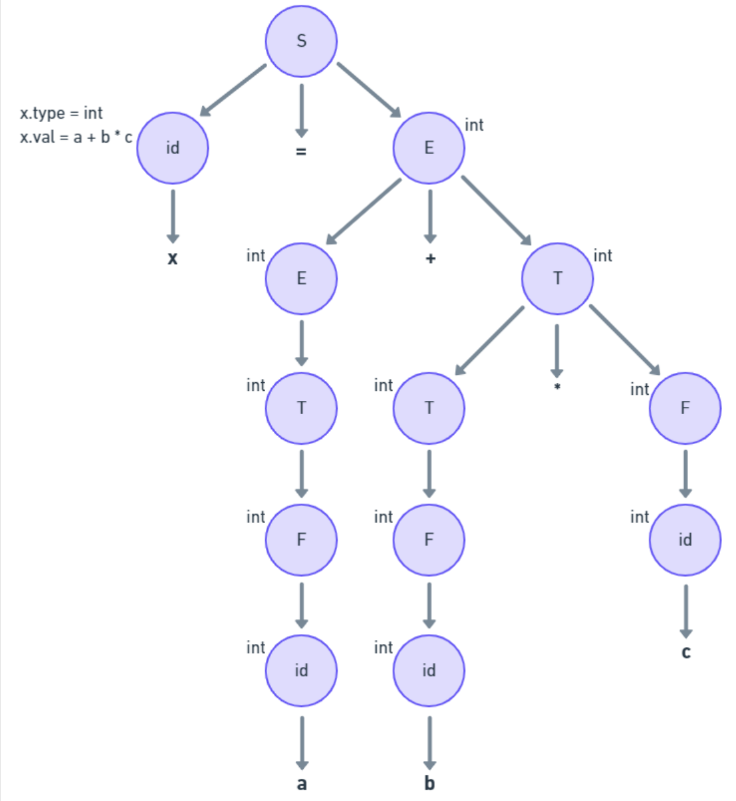


## 

## מנתח סמנטי – Analysis Semantic

עבור המנתח הסמנטי נוסיף לכל צומת בעץ הניתוח Attribute שיוסיף לנו סמנטיקה בנוסף למבנה של התוכנית. כך נוכל למצוא ולנתח שגיאות סמנטיות אשר לא נתפסות בשלבים הקודמים של הקומפילציה.

התכונות שלעיל נוספות לעץ תוך כדי תהליך הניתוח ביעילות זמן ריצה קבועה, (1)O, ובכך לא פוגעות ביעלות זמן הריצה הלינארית, (n(O, של הקומפיילר.



## Hash table

מטרתה של טבלת הסמלים היא שמירה ואחזור של מידע אודות המשתנים השונים בתוכנית. על מנת שתבצע את עבודתה בצורה היעילה ביותר, נשתמש ב – table Hash.

על כל שם משתנה נפעיל פונקציית Hash אשר תי תן לנו את מיקום שם המשתנה בטבלת ה – Hash. כך נוכל לגשת למיקום המשתנה הספציפי הזה במערך ביעילות זמן ריצה קבועה, (1)O.

# 

# סביבת העבודה

## עורך קוד

עבדתי בסביבת visual studio code

## מערכת הפעלה

מערכת הפעלה linux mint



## שפת תכנות

שפת c++



## קומפיילר

g++ compiler

# 

# מדריך למשתמש

## דרישות

וודאו תחילה כי מותקן על מכשיר הלינוקס שלכם קומפיילר G++ עדכני וגם תוכנת NASM

1. write a file with .ace in type and follow the language rules

2. Compile the files with g++ \*.cpp -o ace command

3. Run ./ace

4. That's it!

# סיכום אישי

## מסע מרתק בעולם קומפיילרים: סיפורו של סטודנט צעיר

עמדתי בפני אתגר עצום. פרויקט גמר בקורס מדעי המחשב, לא פחות. המשימה: בניית קומפיילר משלי. עולם קומפיילרים, תחום מורכב ומרתק, נראה בתחילה בלתי אפשרי. אבל אני, עידו הירש, סטודנט צעיר עם רצון עז ללמוד ולצמוח, לא היססתי. קפצתי אל תוך האתגר וצעדתי במסע מרתק רצוף גילויים, תובנות והתפתחויות אישיות.

המסע לא היה קל. התמודדתי עם אתגרים תיאורטיים ואלגוריתמיים רבים. הייתי צריך ללמוד כמויות עצומות של חומר מורכב, לפתח פתרונות יצירתיים לבעיות מורכבות, ולהתמודד עם אתגרים תכנותיים ומעשיים, כמו חלוקת קוד יעילה, מימוש תאוריות מורכבות, פתרון באגים, וניהול פרויקט בסדר גודל משמעותי.

אבל לאורך כל הדרך, פיתחתי כלים רבים שעזרו לי להתמודד עם האתגרים

יכולת למידה עצמית: גיליתי יכולת מרשימה ללמוד באופן עצמאי נושאים מורכבים, לחקור מקורות מידע שונים, ולהרכיב ידע תיאורטי נרחב.

פתרון בעיות יצירתי: פיתחתי יכולת לנתח בעיות מורכבות, לפתח אלגוריתמים יעילים, ולמצוא פתרונות יצירתיים תוך שימוש במבני נתונים מתאימים.

ניהול פרויקטים: צברתי ניסיון ניהולי משמעותי, תוך תכנון, ניהול זמן, ועבודה יעילה בסביבה מורכבת.

עולם מדעי המחשב, עולם מלא אפשרויות ויצירתיות, שימש מקור השראה עבורי. מצאתי השראה בדמויות מובילות בתחום, כמו אלן טיורינג, אבי מדעי המחשב, דניס ריצ'י, מפתח שפת C, ולינוס טורוואלדס, מפתח מערכת ההפעלה לינוקס.

הצלחתי להשיג את כל המטרות שהצבתי לעצמי

פיתוח אישי: פיתחתי את עצמי הן מבחינה מקצועית והן מבחינה אישית. רכשתי ידע נרחב, פיתחתי מיומנויות חשובות, וחיזקתי את ביטחונו העצמי.

בניית קומפיילר: הצלחתי לבנות קומפיילר פעיל, תוך שימוש באלגוריתמים ומבני נתונים יעילים.

שיתוף ידע: כתבתי ספר פרויקט מפורט וברור, המאפשר לקוראים להבין לעומק את התהליך של בניית קומפיילר. הפרויקט שלי מהווה דוגמה למסע אישי מרתק בעולם מדעי המחשב. עבודתי מדגישה את חשיבות הלמידה העצמית, ההתמדה, והיצירתיות. הוכחתי שעם רצון עז, עבודה קשה, והתמדה, ניתן להגשים חלומות ולפתח יכולות יוצאות דופן.

אבל זה לא נגמר פה.

המסע שלי רק התחיל. אני נרגש להמשיך וללמוד, לחקור ולפתח את עצמי בתחום מדעי המחשב. אני יודע שיש לי עוד הרבה מה ללמוד, ויש לי הרבה מטרות שאני רוצה להגשים. אני בטוח שאמשיך להתמודד עם אתגרים, לפתח פתרונות יצירתיים, ולצמוח כאדם וכמקצוען.

העתיד נראה מרתק.

## במידה והיה לי זמן נוסף בפרויקט הייתי רוצה לבצע מספר דברים

אופטימיזציה של קוד:

אקדיש זמן רב לאופטימיזציה של קוד המקור של הקומפיילר.

אלמד טכניקות שונות לאופטימיזציה, כגון ניתוח זרימה של נתונים, הסרת קוד מיותר, וחיזוק קוד.

אני מאמין שתוצאות אופטימיזציה אלו ישפרו משמעותית את הביצועים של הקומפיילר, הן מבחינת מהירות ההידור והן מבחינת יעילות השימוש בזיכרון.

תמיכה בשפות תכנות נוספות:

ארחיב את הקומפיילר כך שיתמוך בשפות תכנות נוספות מלבד השפה המקורית שתמכתי בה.

זה יהיה תהליך מאתגר ומלמד, שנדרש ממני להבין לעומק את ההבדלים הסינטקטיים והסמנטיים בין שפות התכנות השונות.

אני מאמין שתהליך זה יעניק לי ידע רב יותר בתחום שפות התכנות ויאפשר לי לפתח קומפיילר גמיש ורב-תכליתי יותר.

פיתוח ממשק משתמש גרפי:

בנוסף, אפתח ממשק משתמש גרפי (GUI) עבור הקומפיילר.

ממשק המשתמש יאפשר למשתמשים להזין קוד מקור בצורה נוחה וקלה יותר, לראות את תוצאות ההידור, ולקבל מידע נוסף על תהליך ההידור.

אני מאמין שממשק משתמש גרפי יהפוך את הפרויקט שלי לנגיש וקל לשימוש עבור משתמשים אחרים, ויאפשר להם להבין טוב יותר את תהליך ההידור.

השתתפות בפרויקטים Open-Source:

אני מרגיש צורך לחלוק את הידע והניסיון שרכשתי עם הקהילה.

לכן, אצטרף לפרויקטים Open-Source בתחום הקומפיילרים.

אשתף קוד לפרויקטים קיימים, וסייע למפתחים אחרים לפתור בעיות טכניות.

אני מאמין שהשתתפות בפרויקטים אלו תאפשר לי ללמוד מפתחים מנוסים אחרים, לשפר את כישורי עבודת הצוות שלי, ולתרום לקהילת התוכנה.

## סיכום:

הרחבת הפרויקט שלי תרמה רבות ללמידה שלי ולצמיחה שלי כאיש מקצוע. למדתי טכניקות חדשות, פיתחתי מיומנויות חשובות, והרחבתי את הידע שלי בתחום מדעי המחשב. יתרה מזאת, נהניתי מאוד מהתהליך ומהאתגרים שהציב בפניי. אני בטוח שניסיון זה ישמש אותי רבות בעתיד, הן בלימודים האקדמיים שלי והן בקריירה המקצועית שלי.

# ביבליוגרפיה

בפרק זה אפרט את מקורות המידע שלי

מקור 1: [ספר בנושא קומפיילרים](https://lib.fbtuit.uz/assets/files/Alfred-V.AhoMonicaS.LamRaviSethiJeffreyD.Ullman-Compilers-PrinciplesTechniquesandTools-Pearson_AddisonWesley2006.pdf)

מקור 2: [ספר של האוניברסיטה הפתוחה בנושא אוטומטים](https://books.google.co.il/books?id=G3-kKfR-tNQC&printsec=frontcover&hl=iw&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false)

מקור 3: [ספר אינטרנטי על BNF](https://condor.depaul.edu/ichu/csc447/notes/wk3/BNF.pdf)

מקור 4: [סרטוני יוטיוב](https://www.youtube.com/playlist?list=PLGNbPb3dQJ_5FTPfFIg28UxuMpu7k0eT4)

מקור 5: [גיטהאב](https://github.com/DoctorWkt/acwj)

מקור 6: [סדרת סרטונים של גברת הודית מיוטיוב](https://www.youtube.com/playlist?list=PL6xbXi2C3seOXoXK5nL92Frnk_gOT-_2b)

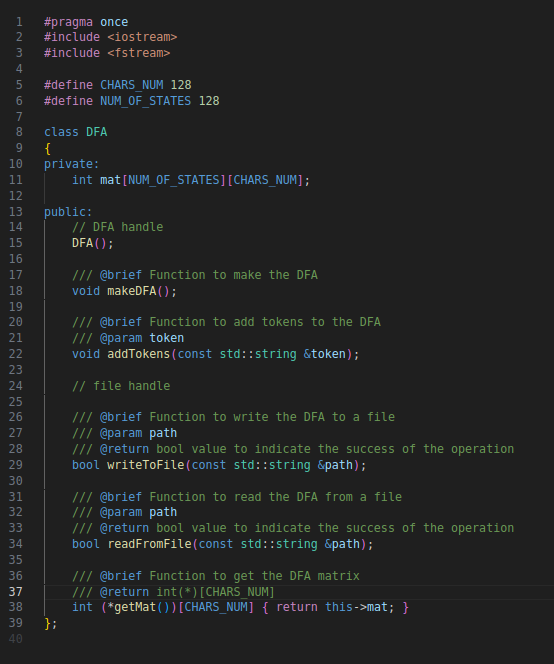
# 

# קוד הפרויקט

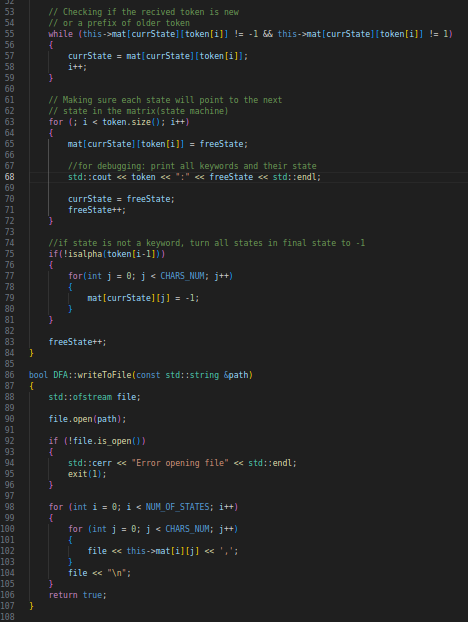
הרחבה על כל חלקי הפרוייקט

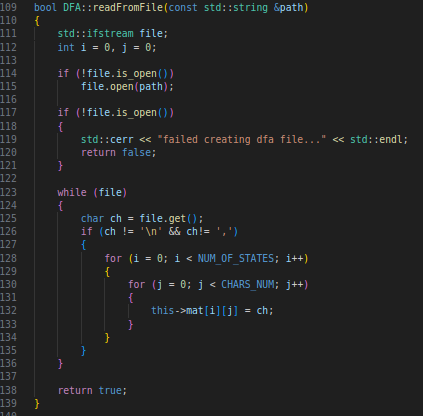
## Dfa

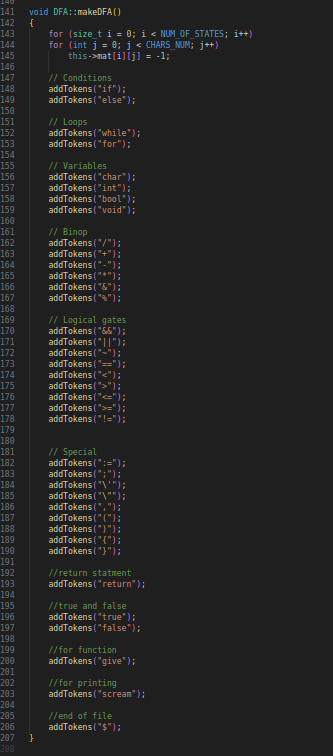
מחלקה האחראית על תפעול וכתיבת מכונת המצבים



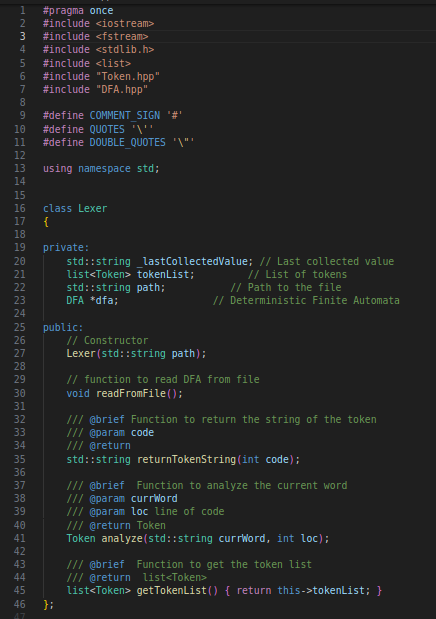


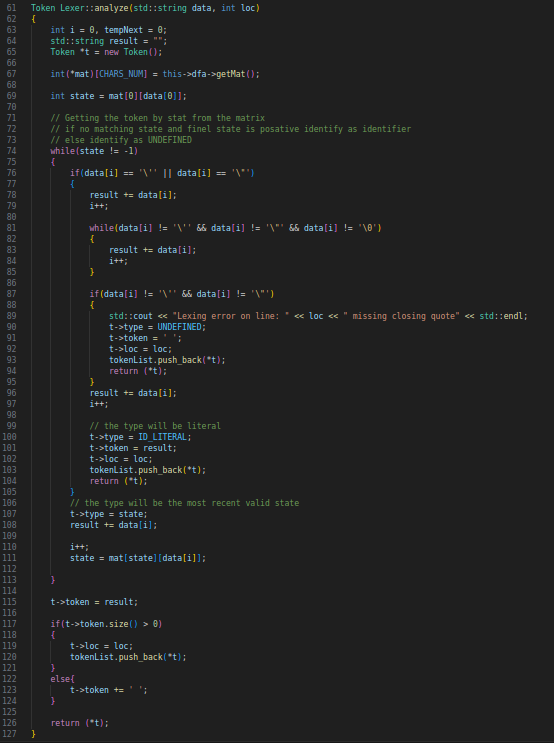
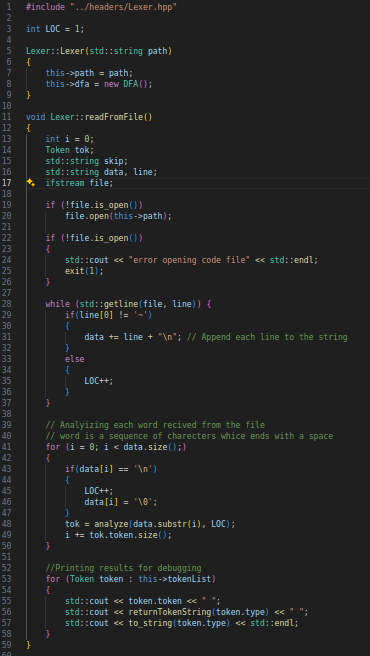


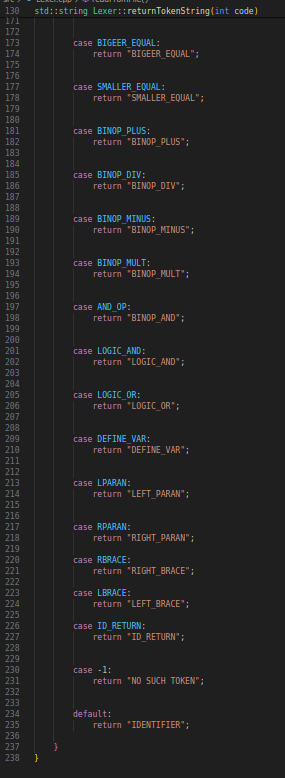




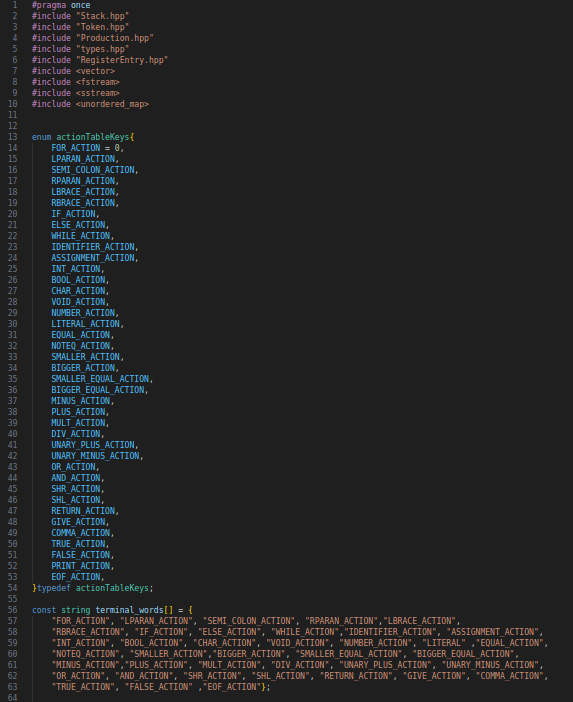
## Lexer

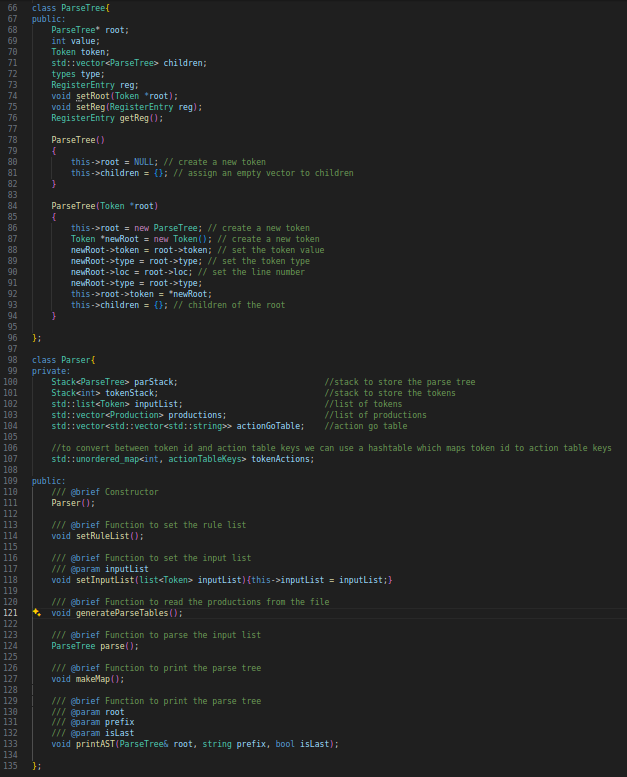


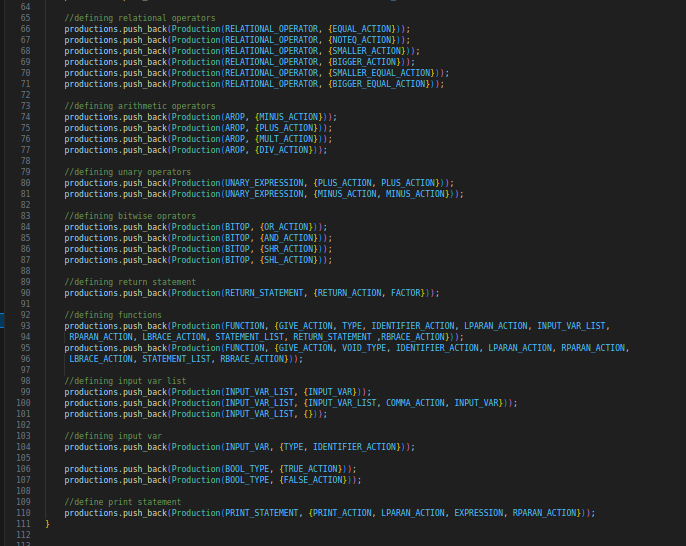
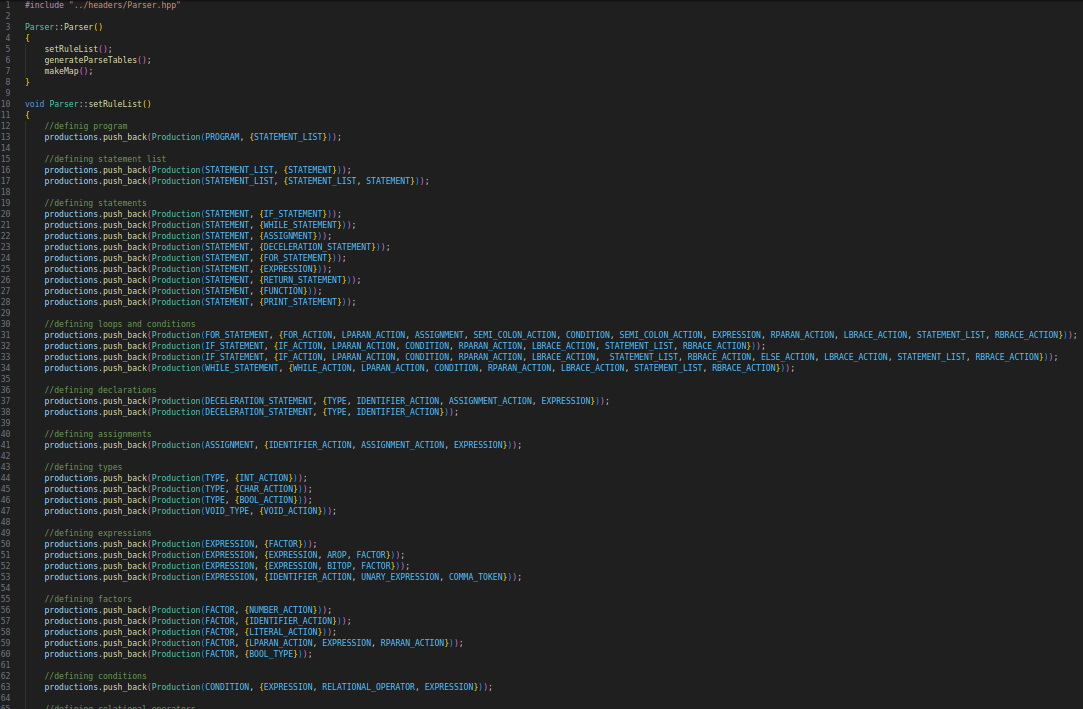




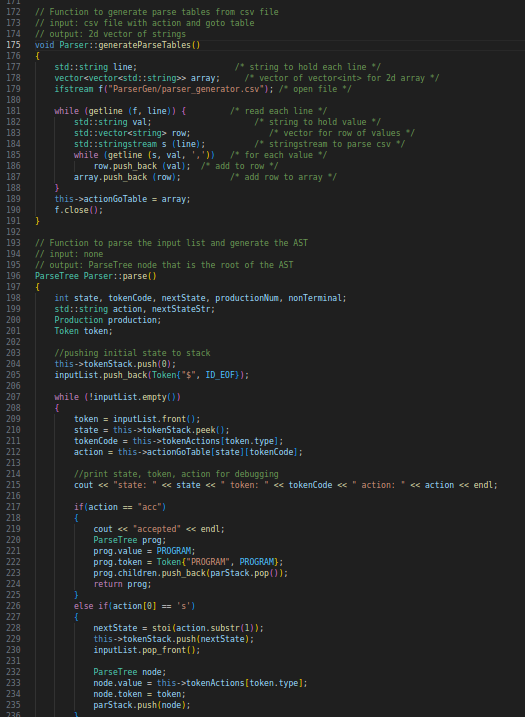
## Parser

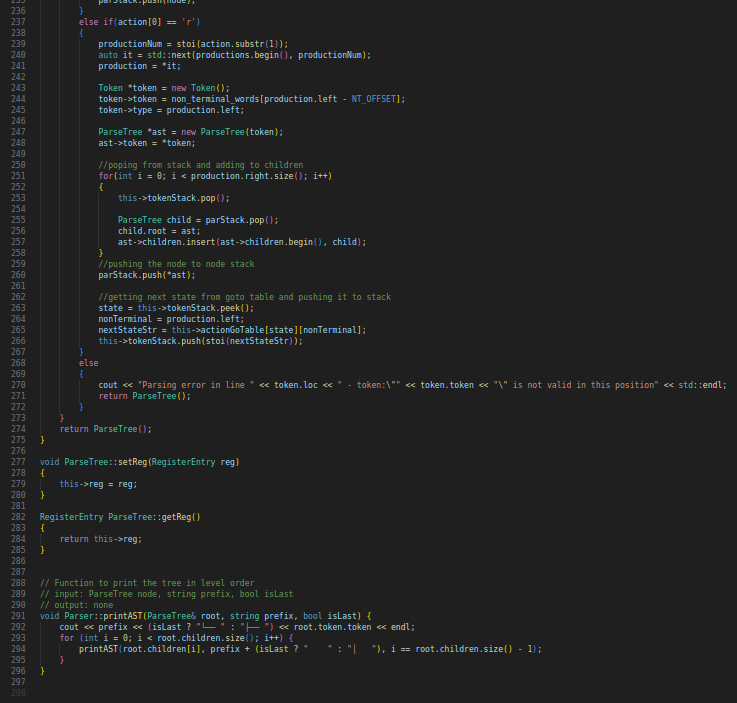




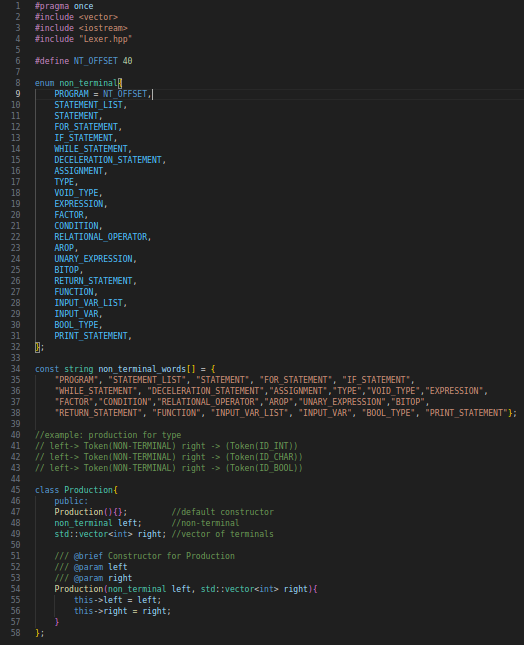




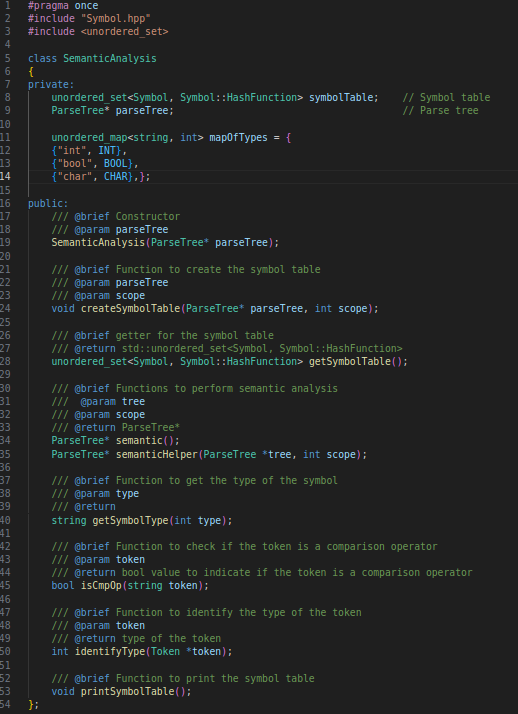


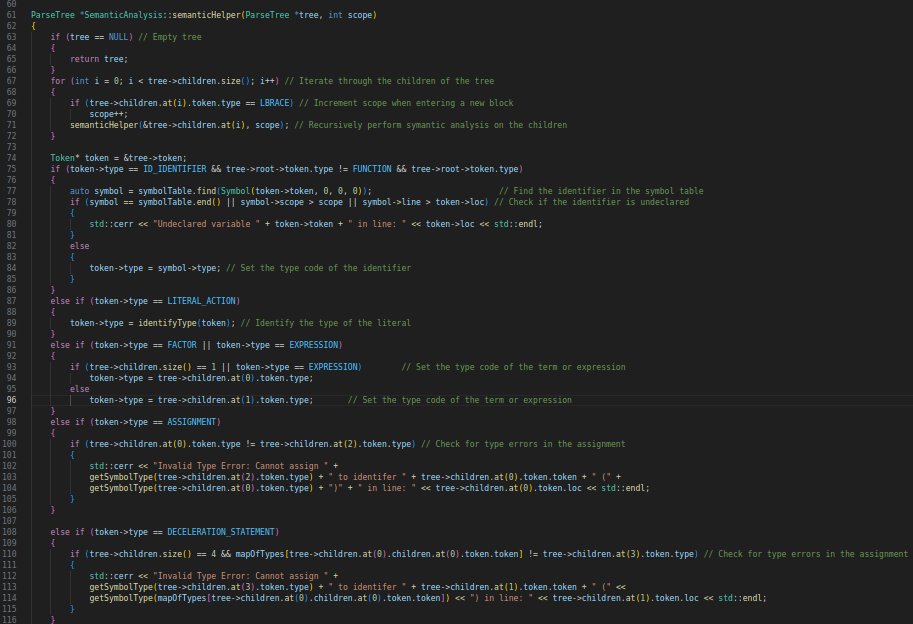
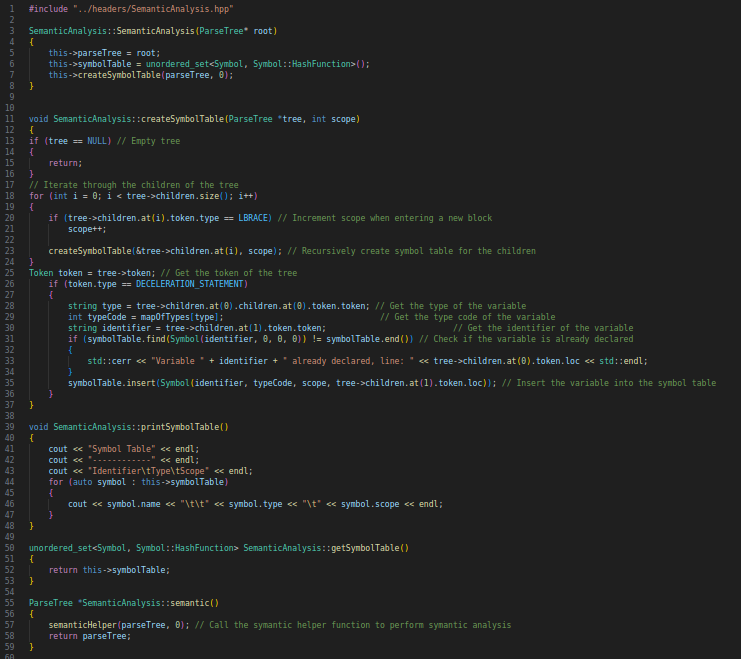


## Production

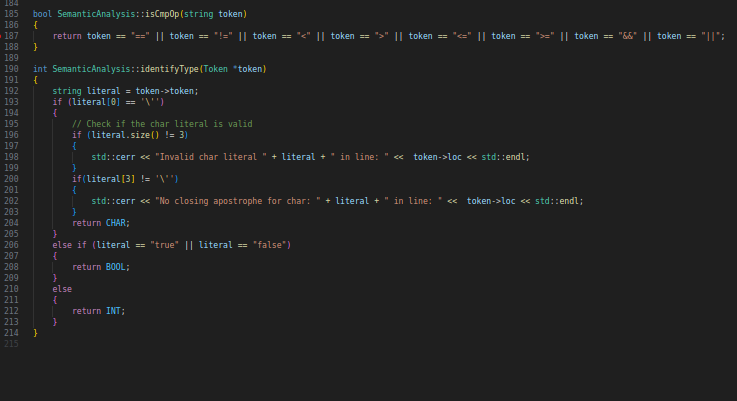


## semanticAnalysis



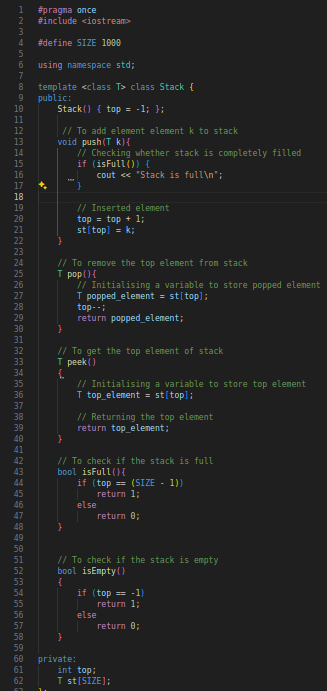




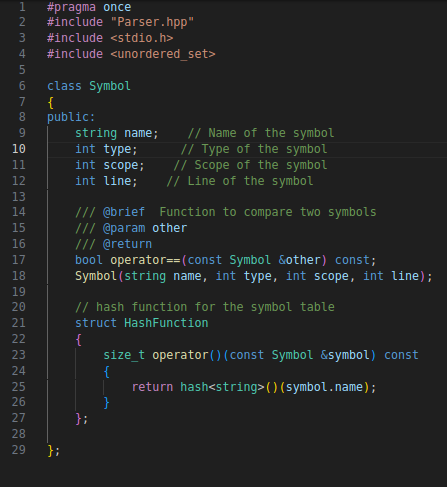


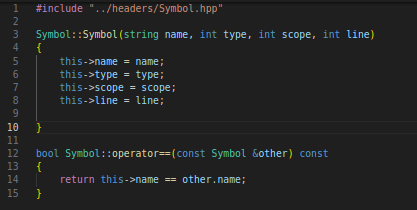
## 

## Stack.hpp

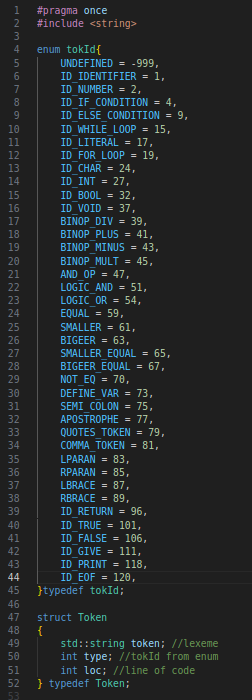


## Symbol

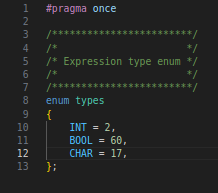




## Token.hpp

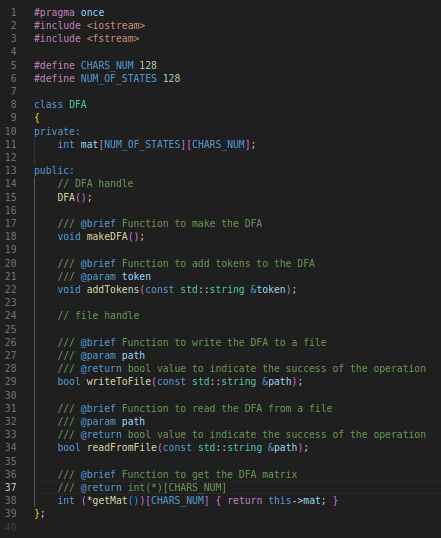


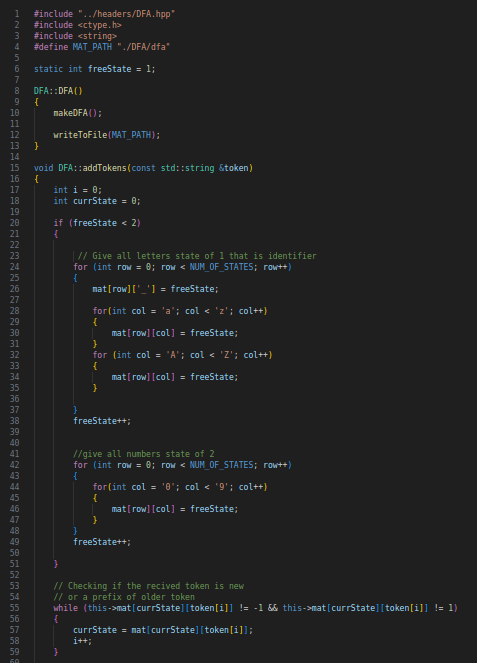
## Types.hpp

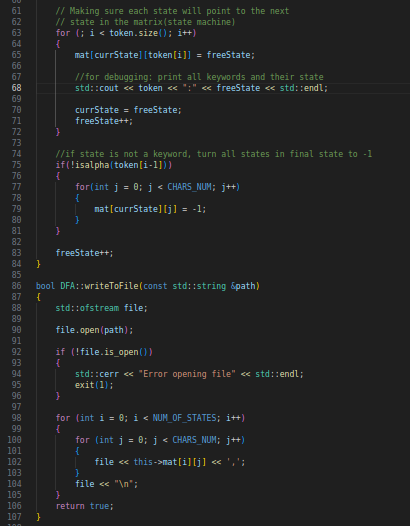


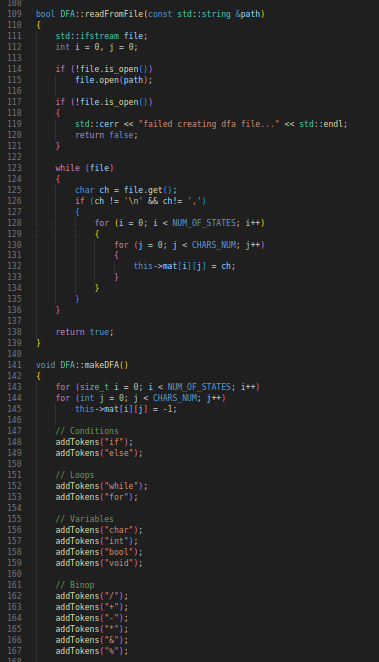
## 

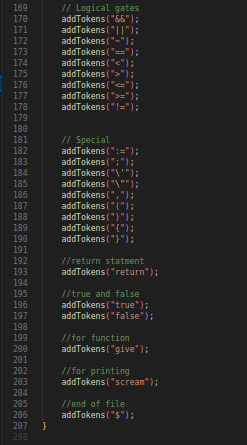
## Dfa











## Code generation

