

THE SIMPLE C PROGRAMMING LANGUAGE COMPILER

מור פישמן

תאריך: מאי 2024 , תשפ"ד

תוכן עניינים

5.....	 מבוא
5.....	 תקציר
6.....	 מושגים
6.....	1. ניתוח לקסיקלי (Lexical Analysis)
6.....	2. ניתוח תחבירי (Syntax Analysis)
7.....	3. ניתוח סמנטי (Semantic Analysis)
7.....	4. ייצרת קוד בינים (Intermediate Code Generation)
7.....	5. אופטימיזציה (Optimization)
7.....	6. הפקת קוד (Code Generation)
7.....	7. קישור (Linking)
8.....	8. ניפוי שגיאות (Debugging)
8.....	9. כלים פיתוח (Development Tools)
8.....	10. תיאור הנושא
8.....	11. אבני השפה
11.....	12. תכולת השפה
13.....	13. דקדוק השפה
18.....	14. רקע תיאורתי
18.....	15. מהדר (compiler)
19.....	16. System Processing Language
20.....	17. התהליכים שמבצעים מהדר
20.....	1. ניתוח לקסיקלי (Lexical Analysis)
21.....	2. ניתוח תחבירי (Syntax Analysis)
22.....	3. ניתוח סמנטי (Semantic Analysis)
22.....	4. ייצרת קוד בינים (Intermediate Code Generation)
23.....	5. אופטימיזציה של הקוד (Code Optimization)
24.....	6. ייצרת קוד (Code Generation)
25.....	7. טבלת הסימנים (Symbol Table)
27.....	18. מכונת מצבים
27.....	19. שפה פורמלית
27.....	20. סוגים אוטומטיים
29.....	21. תיאור הבעיה האלגוריתמית
29.....	22. ניתוח מילוני (Lexical Analysis)
30.....	23. ניתוח תחבירי (Syntax Analysis)
30.....	24. ניתוח סמנטי (Semantic Analysis)
30.....	25. סריקת אלגוריתמים בתחום הבעיה
30.....	26. Parsing Algorithms

33.....	אסטרטגייה
33.....	ניתוח מילוני (Lexical Analysis)
34.....	אלגוריתם לזהות אסימון ע"י מטריצה
35.....	אלגוריתם לבניית מטריצת מצבים מסויימים נתוניים של שפה
36.....	ניתוח תחבירי (Syntax Analysis)
36.....	מחסנית וטבלאות ה-Parse
38.....	אלגוריתם הניתוח
38.....	מימוש LR Parser
42.....	ניתוח סמנטי (Semantic Analysis)
43.....	Top Down Level Design
43.....	מבט על
43.....	Lexer
44.....	Parser
44.....	Semantic Analyzer
44.....	Code Generator
45.....	Error Handler
46.....	מבנה נתונים
46.....	מנתח מילונים – Lexical Analysis
46.....	מנתח תחבירי – Syntax Analysis
48.....	מנתח סמנטי – Semantic Analysis
49.....	Code Generation
50.....	תיאור סביבת העבודה ושפת התוכנות
50.....	שפת התוכנות.....
50.....	סביבת העבודה.....
51.....	אלגוריתם ראשי
52.....	UML Class Diagram
53.....	מודולים ופונקציות הראשית
53.....	היררכיה המודולים בפרויקטיטים
54.....	Lexical Analysis
54.....	Token
55.....	Lexer Automaton
57.....	Lexical Analyzer
58.....	Syntax Analysis
58.....	Parser
59.....	Parser Stack
60.....	Parser Table
62.....	Parser Tree

64.....	Production Rule
65.....	Symantic Analysis
65.....	Symantic Analayser
70.....	Symbol Table
72.....	Code Generator
72.....	Register
74.....	Code Generator
77.....	Error Handler
77.....	Error Handler
79.....	התוכנית הראשית
79.....	אלגוריתם התוכנית הראשית (main)
80.....	מדריך משתמש
80.....	שימוש
82.....	ביבליוגרפיה

מבוא

תקציר

הפרויקט שלי הינו Compiler, המתרגם קובץ טקסט בעל קוד בשפת C לקוד אסמבילר 32 ביט. שפת תכנות היא למעשה יתיר מחוקים תחביריים ולוגים, שנועד להגדיר פעולות חישוב שבוצע על המחשב. הגדרת השפה היא אינטגרלי במבנה המהדר.

הגדרת שפת תכנות מורכבת משלושה מישורים: **מילוני, תחבירי ולשוני**.

المילוני - הגדרת אבני השפה (Key words) שבהן יהיה ניתן להשתמש. לצורך העניין המילה While היא מילה שמורה שתתקבל אר לעומת זאת Aspks היא מילה ללא משמעות.

התחבירי - מגדרת את הרצפים הלוגים של אבני השפה. לדוגמה, הרץף `x int = 1;` חוקי בשפטינו אך לעומת זאת `z = z - 1` לא מייצג רצף הגיוני.

اللשוני - מיחוס משמעות לרצפים ובודק אותם במישור הפיזי, למשל הצהרה נcona תחבירית יכולה להיות לא נcona לשונית. לדוגמה, שימוש במשפטנה שלא הוצאה לפני.

הקומפיילר הוא לא יותר מתוכנה הבודקת את תקינות הקובץ טקסט לפי חוקי השפה שנגידר, המיצרת ממנו קוד אסמבילר.

אני אדם שאוהב לחשב מחוץ ל קופסה ולהתמודד עם קשיים.

בחירת פרויקט מתאגר כמו קומפיילר הייתה נראית טبيعית עברוי, יש רציתי לקפוץ על ההזדמנויות.

רציתי להבין איך הקסם הזה שנקרא תכנות עובד מאחורי הקלעים, כיצד הקופסה השוחרה הזאת הנקראת Compiler מייצרת קוד אסמבילר מדויק במדויקות מרבית ללא טעויות. אף פעם לא הייתה בנאדים כזה שמקבל את הממציאות כמו שהוא אכן ניגשתי לעבודה שבסתופה אכן כיצד הכל קורה.

המטרה העיקרית שלי הייתה רציתי להבין מהפרויקט הזה היא קודם כל הבניה מעמיקה בנושא הקומפיילרים, תחום שבו לא התעסקתי מימי. כמו שאריסטו אמר "הדרך הטובה ביותר ללמידה עמוקה היא לנסות ליצור אותו". התחלתי מהמן מאמריהם וספריהם ארוכים שמסבירים את התאוריות השונות מאחורי המהדרים רבים שיש, ולאחר חיפוש עמוקה היה לי את הידע הדרוש כדי להתחיל לכתוב את המהדר שלי.

בפרויקט זה רציתי לבחון את יכולותי כמתכנת. להצלח לפתח אלגוריתמים עליים אשר יפתרו את הבעיה העולות בשלבי הקומפיילציה השונים, תוך צבירה ניסיון וידע בנושאים שלא ידעת לפני, כגון מכונות מצבים (Automata), עיצוב שפה ותכנות בשפה C++ אותה לא יצא לי לעבוד בעבר.

כמובן שעלו המונ אटגרים בדרך, חוסר ניסיון בכתיבה C++, אלגוריתמים שבועות ניסיתי לפצח ושעות על גבי שעות של פיצוח מכונות מצבים מדויקת עבור התהליכים השונים.

לסיכום, הפרויקט גבה מני המון. זמן ואנרגיה כאחד אבל כשבסוף עמד מהדר שאינו עיצבתי מ-0. לගמרי לבדי אין תחושה העולה על זאת.

אש mach בסוף זה להציג אתכם יד ביד את התהילך שעברית ביצירת Compiler הראשון שלי, וכמובן שלא האחרון ☺

מושגים

בפרק זה נסקור את המושגים המרכזיים הנדרשים להבנת תהליך בניית מהדר לשפת C. מהדר הוא כל קרייטי להמרת קוד מקור שנכתב בשפת תכנות כלשהי לשפת מכונה. שפת C, אשר פותחה בשנות ה-70 על ידי דניס ריצ'י, היא אחת משפות התכונות הנפוצות והחשובות ביותר, ועדין נמצאת בשימוש רחב בתחוםים רבים.

תהליך הקומpileציה מורכב מכמה שלבים עיקריים:

1. **ניתוח לקסיקלי** (Lexical Analysis)
2. **ניתוח תחבירי** (Syntax Analysis)
3. **ניתוח סמנטי** (Semantic Analysis)
4. **יצירת קוד בינים** (Intermediate Code Generation)
5. **אופטימיזציה** (Optimization)
6. **הפקת קוד** (Code Generation)
7. **קישור** (Linking)

1. ניתוח לקסיקלי (Lexical Analysis)

לקסרים (Lexers)

לקסר הוא רכיב בתהליך ההידור שanford את קוד המקור לרצף של מרכיבים בסיסיים שנקראים Tokens. תהליך זה מכונה גם ניתוח לקסיקלי. כל Token מייצג יחידה בסיסית כמו מילה מפתח, מזהה (identifier), אופרטור או סימן פיסוק.

סימבולים (Tokens)

סימבולים הם היחידות הבסיסיות של הקוד שהתגלו על ידי הלקסר. הם כוללים:

- **מילות מפתח**: כמו `if`, `return`, `int` ..
- **מזהים**: שמות של משתנים ופונקציות.
- **אופרטורים**: כמו `+`, `-`, `*`, `/` ..
- **סימני פיסוק**: כמו `;`, `,`, `{`, `}` ..

2. ניתוח תחבירי (Syntax Analysis)

עצים תחביר (Parse Trees)

עץ תחביר הוא מבנה נתונים המיציג את המבנה היררכי של הקוד. הלקסר מפרק את הקוד למרכיבים בסיסיים, וה-parser מארגן אותם בעץ תחבירי המציג את היחסים בין המרכיבים.

תחביר חופשי-הקשר (Context-Free Grammar)

תחביר חופשי-הקשר הוא קבוצה של כללי המתארים כיצד סימבולים יכולים להיות משלבים ליצור מבנים חוקיים בשפה. שימוש ב-**BNF** (Backus-Naur Form) מאפשר לתאר את הדקדוק של השפה בצורה פורמלית.

3. ניתוח סמנטי (Semantic Analysis)**(Type Checking)**

בדיקות סוגים מבטיחות שכל הביטויים והמשתנים בקוד תואמים מבחינה סוג הנתונים שלהם. זהו שלב קרייטי למניעת שגיאות הרצה הנובעת מאי-התאמה בין סוג נתונים.

סמנטיקה סטטית (Static Semantics)

סמנטיקה סטטית מתיחסת לכלים ולביקורות שניית לבצע ללא צורך בהרצת הקוד, כמו בדיקות נכונות סוגים והקצאות המשתנים.

4. ייצור קוד בינים (Intermediate Code Generation)**(Intermediate Representation)**

קוד בינים הוא ייצוג של הקוד שאינו תלוי במכונה מסוימת. הוא מאפשר ל מהדר לבצע אופטימיזציות ושיפורים לפני הפקת קוד מכונה סופי.

שלבים של תיור

שלבים אלו כוללים ייצור קוד בינים, ביצוע אופטימיזציות עליון, ולבסוף המרתו לקוד מכונה.

5. אופטימיזציה (Optimization)**אופטימיזציות קוד ביןים**

טהיליך זה כולל שיפור של הקוד הביניים שהופק על ידי המהדר. המטרה היא לייצר קודיעיל יותר מבחינת זמן ריצה וניצול זיכרון.

אופטימיזציות ברמת קוד מכונה

אופטימיזציות המתבצעות בשלב הפקת קוד מכונה, כמו הסרת קוד מת (dead code elimination), אינליין של פונקציות ועוד.

6. הפקת קוד (Code Generation)**המרת קוד ביןים לקוד מכונה**

שלב זה כולל המרת הקוד הביניים לקוד מכונה ספציפי עבור הפלטפורמה היעד. מדובר בטהיליך מורכב הכולל תכנון הוראות מכונה וניהול זיכרון.

סביבות יעד

מהדרים צריכים לתמוך בעבדים וסביבות הפעלה שונות, וכך נדרש גמישות בטהיליך הפקת הקוד.

7. קישור (Linking)**קישורים סטטיים ודינמיים**

קישור סטטי משלב את כל הקוד הדרוש בזמן ההידור, בעודו קישור דינמי מאפשר טעינת ספריות חייזניות בזמן הריצה. כל סוג של קישור מצריך טכניקות וכליים שונים.

טיפול בספריות חיצונית
בפרויקטים רבים, יש צורך להשתמש בספריות קוד חיצונית. המהדר והקשר צריכים לנהל תלותים בספריות אלו בצורה נכונה.

ניפוי שגיאות (Debugging)

ניפוי שגיאות קוד מקור

תהליך זה כולל שימוש בכלים וטכניקות לזיהוי ותיקון שגיאות בקוד המקור לפני ואחרי ההידור.

טיפול בשגיאות ריצה

זיהוי ותיקון שגיאות המתרחשות בזמן הריצה, כולל טיפול ב-exceptions, זיכרון וכדומה.

מבחנים יחידתיים (Unit Tests)

בדיקות אלו מתמקדות בבדיקה יחידות קטנות של הקוד (כמו פונקציות או מודולים) באופן עצמאי.

בדיקות אינטגרציה (Integration Tests)

בדיקות אלו מתמקדות בבדיקה השילוב והאינטראקטיבית בין רכיבים שונים בפרויקט.

כלי פיתוח (Development Tools)

IDE ואדייטורים

כלים לעריכת קוד המספקים פונקציות מתקדמות כמו ניפוי שגיאות, השלמה אוטומטית, ובדיקה תחביר בזמן אמת.

תיאור הנושא

בני השפה

קבועים – Constants

קבוע הוא ערך המופיע ישירות בקוד.

קבוע יכול להיות טיפוסים שונים ,תו אם מספר שלם

וטיפוסו יקבע על ידי המהדר למשל:

`8 = x int`

הmahder יודע שנקנץ לשנתנה בגודל של 4 בתים 8 וכן גודל הקבוע יוגדר כ 4 בתים.

משתנים - Variables

משתנה מייצג מקום בזיכרון עם גודל קבוע הניתן להכניס אליו ערכים.
מקום זה מיוצג ע"י שם המשתנה הנקרא גם מזהה (Identifier)

שמות משתנים

שם המשתנה מורכב מאותיות ספורות והטו '_' וחיבר להתחיל באות או ב '.'
אסור לו להיות מילה מוגדרת בשפה ויש הבחנה בין אותיות גדולות וקטנות (Case sensitive)

טיפוסי משתנים

כל המשתנה יש סוג ובמימוש פשוט של C שהכנתי קיימים רק שני סוגי מתוך העשרות של C
מציעה:

Int - משתנה מטיפוס שלם בגודל של 4 בתים. מכיל מספרים שלמים

Char - משתנה מטיפוסתו מכיל ערכים כגון אותיות או מספרים קטנים. גודל בית אחד.

הגדרת משתנים

אOPEN הגדרת המשתנים בשפה לא מאפשר הגדרת משתנה ללא השמה לCN גדרת משתנה בצורה
הבא:

<data-type> <identifier> = <value>;

למשל:

int x = 1;

char c = 'd';

פונקציות – Functions

מייצג קטע קוד אליו ניתן לкопץ כאשר נקרא לו. למעשה פונקציה יכולה להכיל משתנים שביהם
נשתמש רק בתחום הקוד בפונקציה ונפטר מהם לאחר מכן.

שמות פונקציות

שם פונקציה מורכב מאותיות ספורות והטו '_' וחיבר להתחיל באות או ב '.'
אסור לו להיות מילה מוגדרת בשפה ויש הבחנה בין אותיות גדולות וקטנות (Case sensitive)

טיפוסי פונקציות

כל פונקציה יש סוג ובמימוש פשוט של C שהכנתי קיימים רק שני סוגי מתוך העשרות של C
מציעה:

Int - משתנה מטיפוס שלם בגודל של 4 בתים. מחזיר מספרים שלמים

Char - משתנה מטיפוסתו מחזיר ערכים כגון אותיות או מספרים קטנים. גודל בית אחד.

Void – פונקציה לא מחזירה ערך.

משתנים מועברים לפונקציות

נדיר משתנים בראשית הפונקציה אותם נהיה חייבים להעביר בקריאה אליה.

הגדרת פונקציות

אOPEN הגדרת הפונקציות:

<data-type> <identifier>(<passed param>)<scope>

למשל:

```
int func(int number){return 1;}

void func(int number){number = 1;}
```

ביטויים – Expressions**Expression**

ביטוי בשפה מורכב מאופרטורים , משתנים , קבועים וביטויים נוספים. תפקיך המהדר לפרש את היצירופים הללו לערך אותו נחוץ למקומם מסוים בזיכרון או בקובד. תהליך זה נקרא הערכה (Evaluation)

דוגמאות לביטויים:

1+1

$387*9/(function(1,2))$

$x/3 == 3$

Statement

צירוף תחבירי בשפה שנועד לבטא פעולה מסוימת. תוכנת מחשב היא למשר רצף של צירופים כאלה. לדוגמה Statement יכול להיות רכיבים פנימיים כגון Expressions

דוגמאות ל – Statements

תנאים – if, else

לולאות – for , while

קריאה לפונקציות – function()

השמה והצירה – int x = 1, x = 9

אופרטורים - Operators**חשבונים**

- חיבור +

- חיסור -

- כפל *

- חילוק /

- שארית %

לוגיים

- שווה \equiv

- לא שווה \neq

- גדול שווה \geq

- קטן שווה \leq

- גדול >

- קטן <

- לא !

קשרים לוגיים

- או ||

- גם &&

תכלות השפה

בחלק זה נתאר את חלקו של שפה וירכ כל חלק בה נכתב בצורה נcona. כל פקודה בשפה תסתיים עם נקודת פסיק ; למעט לולאות ותנאים

השמה - Assignment

נרצה למשתנה להכניס ערך, צייני לעיל שלא ניתן להגיד בשפה שיצרתי משתנה ללא השמה ולכן הפעולות הללו מתרחשות יחד מלבד השמה למשתנה שהוגדר אז לא צריך הצהרה.

הערך למשתנה יכול להיות קבוע , משתנה או ביטוי .

הסימן העיקרי של השמה זה התו '=' בצורה הזאת

`<Identifier> = <Expression>;`

צריך כזכור שהערך יהיה תואם לטיפוס אליו עושים את ההשמה אבל במקרה של השפה שלנו יש לנו רק שני טיפוסי נתונים שערכם לא סותר אחד את השני ולכן אין בעיה של סוג בשפה חוץ מפונקציית `void` .

תנאים – Conditions

חלק קוד המבוצע בתלות אמת או שקר.

لتנאי יכול להיות שני חלקים

- `If` חובה
- `Else` אופציוני!

אם הביטוי נותן אמת החלק של `IF` הוא זה שיתבצע ואם לא נדלג על חלק זה וחלק `ELSE` יתבצע.

לאחר ביצוע חלקים אלו התוכנית תמשיך לקרוא את שארית הקוד

דוגמא לשימוש ב `if`

`If (<Expression>)`

`<scope>`

...

דוגמא לשימוש ב if else

If (<Expression>)

<scope>

Else

<scope>

...

לולאות – loops

ללאות דומות למבנה ה if אך ההבדל הוא שהן מtbodyות באופן חוזר ונשנה עד שהתנאי לא תקין עוד (כל ריצה של הקוד בתוך הלולאה נקראת איטרציה). בכל איטרציה התנאי יבדק בשנית וכל עוד הוא תקין הקוד ירוץ.

יש לנו שתי סוגים של לולאות:

- While לולאה שבתוכה ניתן להגדיר תנאי, כמו if :

While (<Expression>)

<scope>

...

- For לולאה שניית להגדיר בה עד 3 פעולות שיבוצעו לפני האיטרציה

For (<happens before the first iteration> ; <happens before every iteration> ;

<happens at the end of every iteration>)

<scope>

...

פונקציות – functions

פונקציות הן למעשה קוד שניתן לקבל ערכים ו גם לא יכול להחזיר ערכים וגם לא. המתכוнаראשי לכתבו איזה פונקציה שירצה כדי ליעל את כתיבת הקוד שלו ולקראא לה בעט הצורך

נקרא לפונקציה כך:

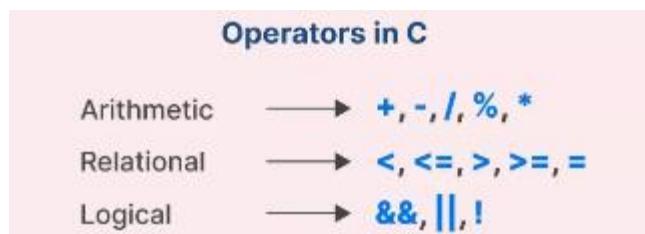
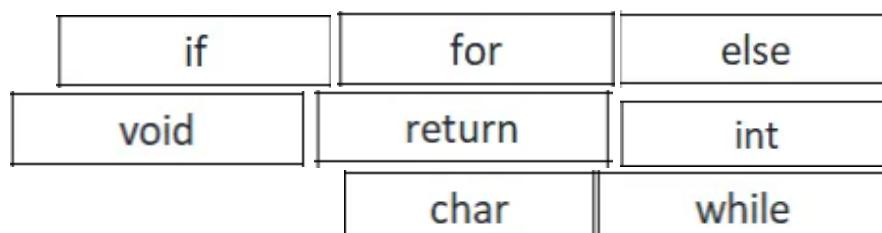
<function name>()

ניתן גם לשים פונקציות שמחזירות ערכים בתוך ביטויים למשל בתנאים או השמות וכו...

דקדוק השפה
לאחר שהגדכנו את תכונות השפה ובני השפה נגדיר את הדקדוק שלה. במלים אחרות Grammar.

תחביר השפה
תחביר השפה שבניתי הוא כמו השפה C רק פשוט יותר שכן הוא תחביר חופשי הקשור (Context Free Grammar) הוא מורכב מאיסימונים שנקלטים מקטע הקוד ומאופיינים ע"י המהדר. לאחר מכן המהדר ממיר אותם ל NonTerminals שהם הגדרות לרצפי סימנים כגון: השמה, לולאה, פונקציה...
תחביר השפה נקבע באמצעות כללים הנקראים כללי יצירה (production rules) שהם מחליטים איזה רצף של תווים ניתן לצמצם ל NonTerminal מסוים.

Tokens
להלן Tokens הנתמכים ע"י השפה שלי.



Increment	→	++
Decrement	→	--
Assignment	→	=

תיאור דקדוק – BNF Notation

צורת תיאור שפות תכנות באמצעות כללי יצירה. היא כוללת סימבולים לא-סופיים, סופיים, כללי יצירה, סימbol התחלת, ופעולות פרשנות. דרך זו מפשטת את התיאור של השפה והופכת אותו לבלתי דו –משמעותי ונוח לקרוא.

```
S -> PROGRAM
PROGRAM -> PROGRAM FUNCTION
PROGRAM -> FUNCTION

FUNCTION -> RETURN_FUNC
FUNCTION -> NONE_RETURN_FUNC

RETURN_FUNC -> RETURN_FUNC_HEAD SCOPE
NONE_RETURN_FUNC -> NONE_RETURN_FUNC_HEAD SCOPE

RETURN_STMT -> return computing_expretion
SCOPE -> { BLOCK }
SCOPE -> { }
BLOCK -> STMT BLOCK
BLOCK -> STMT
STMT -> LINE_STMT ;
STMT -> CONTROL_STATEMENT
STMT -> RETURN_STMT ;
STMT -> FUNC_CALL ;

LINE_STMT -> ASSIGN
LINE_STMT -> FUNC_CALL
LINE_STMT -> VAL_CHANGE_TYPE
LINE_STMT -> LINE_STMT , ASSIGN
LINE_STMT -> LINE_STMT , VAL_CHANGE_TYPE
LINE_STMT -> LINE_STMT , FUNC_CALL

CONTROL_STATEMENT -> IF_STATEMENT
CONTROL_STATEMENT -> WHILE_STATEMENT
CONTROL_STATEMENT -> FOR_STATEMENT
CONTROL_STATEMENT -> IF_ELSE_STATEMENT

IF_STATEMENT -> if ( CONTROL_EXP ) SCOPE
IF_ELSE_STATEMENT -> if ( CONTROL_EXP ) SCOPE else SCOPE

WHILE_STATEMENT -> while ( CONTROL_EXP ) SCOPE

FOR_STATEMENT -> for ( FOR_EXP FOR_EXP CONTROL_EXP ) SCOPE
FOR_STATEMENT -> for ( FOR_EXP FOR_EXP ) SCOPE

TYPE_CONTROL_EXP -> ASSIGN
TYPE_CONTROL_EXP -> VAL_CHANGE_TYPE
TYPE_CONTROL_EXP -> computing_expretion
TYPE_CONTROL_EXP -> FUNC_CALL
CONTROL_EXP -> TYPE_CONTROL_EXP
CONTROL_EXP -> TYPE_CONTROL_EXP , CONTROL_EXP
FOR_EXP -> CONTROL_EXP ;
FOR_EXP -> ;
```

```
VAL_CHANGE_TYPE -> id VAL_CHANGE_OP computing_expretion
VAL_CHANGE_TYPE -> PRE_ONARY_ARITMATIC
VAL_CHANGE_TYPE -> POST_ONARY_ARITMATIC
VAL_CHANGE -> VAL_CHANGE_TYPE
VAL_CHANGE -> VAL_CHANGE_TYPE , VAL_CHANGE
VAL_CHANGE_OP -> +=
VAL_CHANGE_OP -> ==
VAL_CHANGE_OP -> *=
VAL_CHANGE_OP -> !=
VAL_CHANGE_OP -> %=
ONARY_CHANGE_OP -> ==
ONARY_CHANGE_OP -> !=

DECLERATION -> RETURN_TYPE id
DECLERATION -> NONE_RETURN_TYPE id

ASSIGN -> id = computing_expretion
ASSIGN -> DECLERATION = computing_expretion
ASSIGN_OR_DEC -> DECLERATION
ASSIGN_OR_DEC -> ASSIGN
ASSIGN_OR_DEC -> ASSIGN_OR_DEC , ASSIGN
ASSIGN_OR_DEC -> ASSIGN_OR_DEC , DECLERATION

computing_expretion -> bool_expretion
computing_expretion -> aritmatic

bool_expretion -> bool_cmp boolean_combo bool_cmp
bool_expretion -> bool_expretion boolean_combo bool_cmp
bool_expretion -> bool_cmp

bool_cmp -> bool_cmp boolean_compersion expr
bool_cmp -> expr boolean_compersion expr
bool_cmp -> expr

expr -> bool_expr
expr -> aritmatic
bool_expr -> ! aritmatic

aritmatic -> aritmatic + T
aritmatic -> aritmatic - T
aritmatic -> T
T -> T * F
T -> T / F
T -> T % F
T -> F
F -> ( aritmatic )
F -> PRE_ONARY_ARITMATIC
F -> POST_ONARY_ARITMATIC
F -> id
F -> numeric
F -> FUNC_CALL
F -> literal
```

```
PRE_ONARY_ARITMATIC -> id ONARY_CHANGE_OP
POST_ONARY_ARITMATIC -> ONARY_CHANGE_OP id

boolean_compersion -> !=
boolean_compersion -> ==
boolean_compersion -> >=
boolean_compersion -> <=

boolean_combo -> &&
boolean_combo -> ||

FUNC_CALL -> id ( )
FUNC_CALL -> id ( PASS_PARAM )

PASS_PARAM -> computing_expretion
PASS_PARAM -> computing_expretion , PASS_PARAM

RETURN_FUNC_HEAD -> RETURN_TYPE id ( PARAMS )
RETURN_FUNC_HEAD -> RETURN_TYPE id ( )
NONE_RETURN_FUNC_HEAD -> NONE_RETURN_TYPE id ( PARAMS )
NONE_RETURN_FUNC_HEAD -> NONE_RETURN_TYPE id ( )

PARAMS -> DECLERATION
PARAMS -> PARAMS , DECLERATION

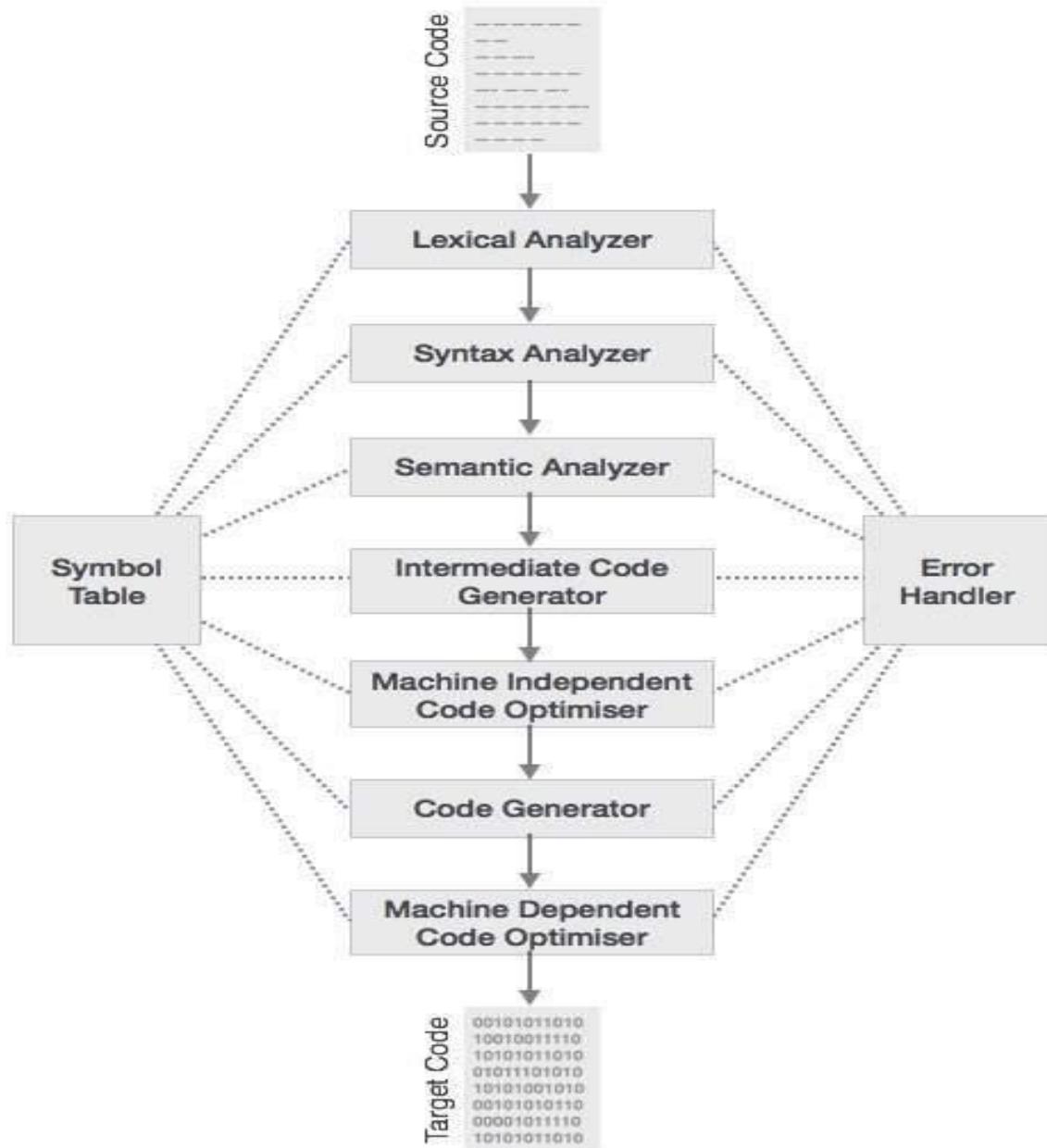
RETURN_TYPE -> int
RETURN_TYPE -> char
NONE_RETURN_TYPE -> void
```

רָקַע תִּיאוֹרֶתִי

מַהְדֵּר (compiler)

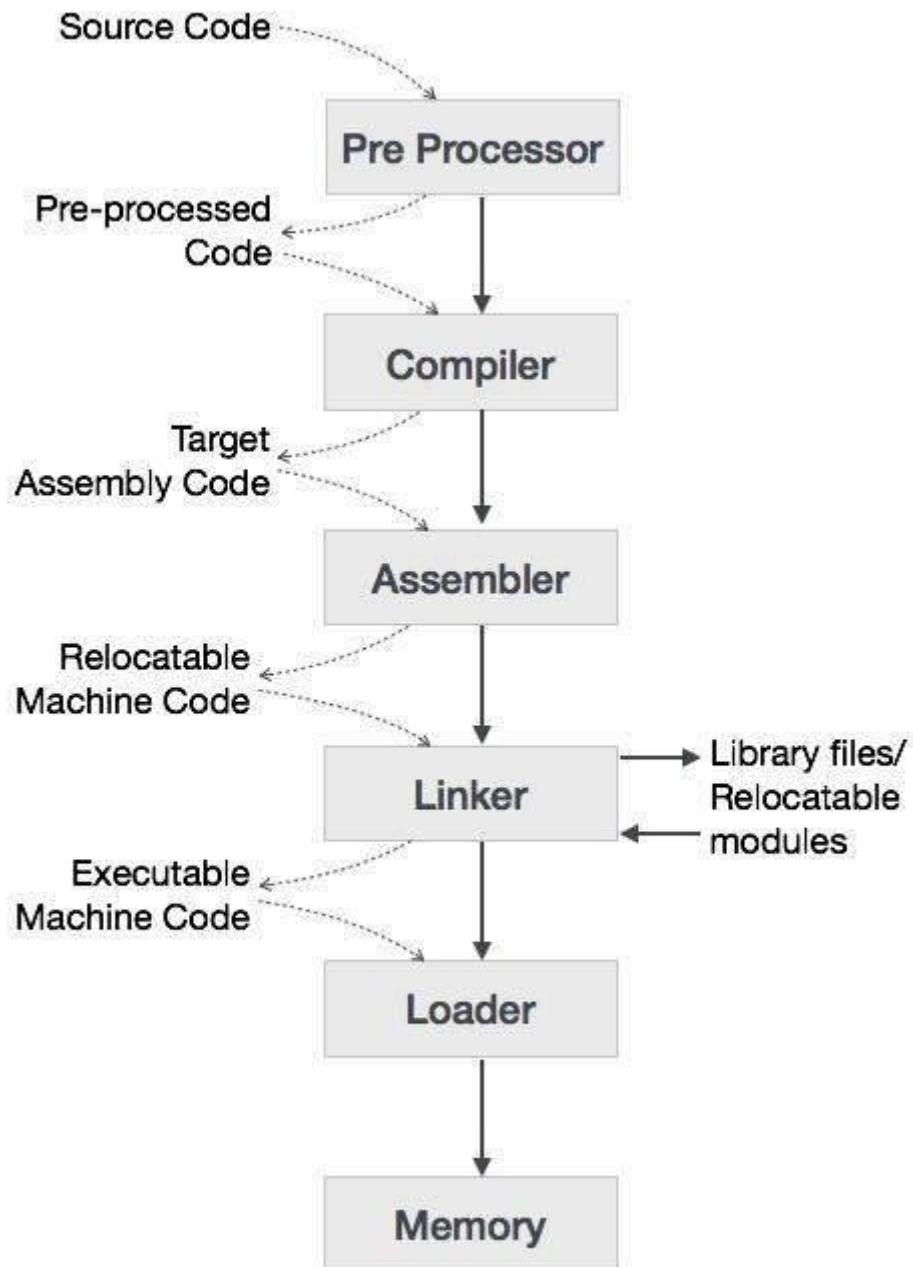
זו תוכנת מחשב שמטרתה היא לתרגם קוד של שפת תכנות מסוימת שהוא לא level low לקוד level low (то есть склонности к записи в постфиксной форме). Находится в пути [here](#).

יש מגוון פעולות שהמהדר מבצע מבחן syntax עד לאופטימיזציה של הקוד אבל מדובר בשלבים בסדר כרונולוגי שכל שלב מקבל את התוצאת של קודמו עד לתוצר הסופי.



Language Processing System

מהדר הוא ייחודה אחת מתוכה מערכת שלמה לקחת קוד שפה עילית וליצור קובץ הרצה ולטען אותו לזכרון, (הشرطוט לעיל מתאר שלבי הקומpileציה אבל גם assembler שזו ייחודה נפרדת מ- מהדר המקבלת קוד assembly ומוחזירה קוד בינארי ב object file)



התהליכיים שבוצע המהדר

קיבלנו קוד הכתוב ב C להלן השלבים שנבצע כדי ליצור קובץ assembly מקביל ל source code :

1. ניתוח לексיקלי (Lexical Analysis)

קלט: source code
 פלט: אוסף כל ה components מהם בניו הקוד, לכל איבר באוסף נקרא **Token**
Token - יחידה לקסיקלית (מילות מפתח, קבועים, מזהים, מחוזות, מספרים, אופרטורים וכו' וכי פיסוק)

נסrok את הקוד כרצף של אותיות כאשר רווח ופסיקים מסמנים על token חדש כל token מייצג פקודת אחרת עם value מסוים
 $\langle \text{attribute-value}, \text{token-name} \rangle$

לדוגמה בשפת C הצהרה על משתנה
`100 = value int`

תכליל את ה Tokens הבאים
`constant (100) (operator) = (identifier) value (keyword) int`

אם נמצא token שלא עומד בתקן השפה נדרש שגיאת קומפיילציה.
 אנו נכתבו כללים למה נחשב token תקין ומה לא באמצעות דפוסים שהיו מוגדרים באמצעות ביטויים רגולריים.

בعال היכולה לתאר את התchapיר של הפקודות לדוגמה:

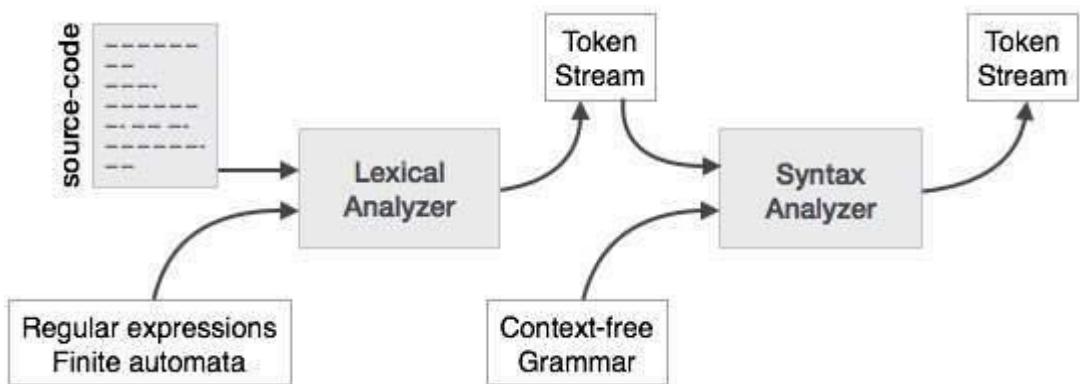
`Decimal = (sign)?(digit)*`
`Identifier = (letter)(letter | digit)*`

אנו נודא כי token שעומד בביטוי regular באמצעות FA - Finite Automation
Finite Automation - דרך בשלבים הבודקת אם הקלט שעומד בתקן מסוים כך שאם הגענו לשלב האחרון סימן שהקלט שעומד בתקן.
 יסביר לעומק בהמשך ...

2. ניתוח תחבירי (Syntax Analysis)

קלט: אוסף כל היחידות ה-לקסיקליות שהתקבלו משלב ה- Lexical Analysis
 פלט: עץ המתאר את סדר הפעולות הכרונולוגיות בקוד ללא בדיקת התאמות טיפוסים ושימושים בפונקציות ...

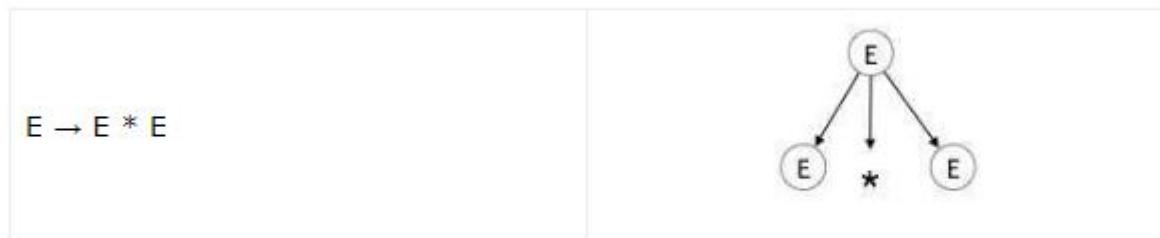
אנו מקבלים אוסף היחידות ה-לקסיקליות שביחד מתארות מבנה של פקודות היוצרות קטע קוד עם סדר הכרונולוגי.
 בשלב זה נבדוק את הנכונות של רצף הפקודות ובנוסף נוצר עץ המתאר את אותו הרצף.
 אנו נתאר את מבנה השפה שלנו באמצעות CFG = Context-free grammar
CFG - דרך בשלבים שבודקת את תקינות המבנה של קלט מסוים לפי חוקיות שנייתה מראש



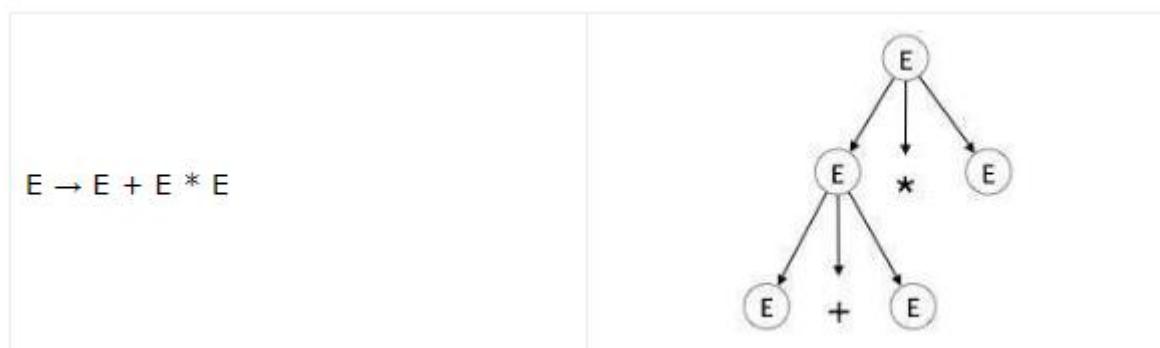
איילוסטרציה לדרך שבה נבנה את העץ הרצוי:

$$\begin{aligned} E &\rightarrow E * E \\ E &\rightarrow E + E * E \\ E &\rightarrow id + E * E \\ E &\rightarrow id + id * E \\ E &\rightarrow id + id * id \end{aligned}$$

Step 1:



Step 2:



3. ניתוח סמנטי (Semantic Analysis)

קלט: עץ המתאר את סדר הפעולות הכרונולוגיות בקוד לא בדיקות התאמה וסוג.

פלט: עץ המתאר את סדר הפעולות הכרונולוגיות בקוד שעבר בדיקת התאמה, סוג ושימושים

מעבר על העץ ובאמצעות **Symbol Table** (שלב בפני עצמו אבל شاملו) אוטומטית לאורך כל השלבים

האחרים, נרჩיב עליו בהמשך).

ונכל לדעת האם המשתנים מקבלים ערכים תקינים, בדיקת תקיןויות של שימושים בפונקציות,

האם אנחנו משתמשים בערך שהוגדר וכו' ..

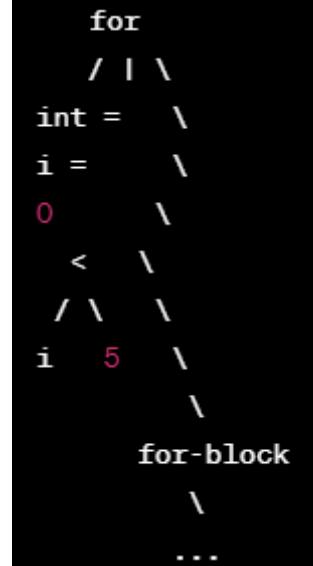
4. ייצור קוד בינים (Intermediate Code Generation)

קלט: עץ המתאר את סדר הפעולות הכרונולוגיות בקוד שעבר בדיקת התאמה ושימושים

פלט: הצגה פשוטה ויחידה של קוד לצורכי בקרה ואנליזה

בשלב זה נמיר את העץ לקוד בינים שקרוב מאוד ל-assembly

למשל קיבלנו עץ מסוים המתאר לולאת for:



אנו נעבור על העץ ונסדר את הפעולות בצורה פשוטה ביותר שניתן בסדרה:

```

t1 = 0      ; Initialization
i = t1      ; Copy the result of initialization to i

L1: t2 = i < 5 ; Condition
    ifFalse t2 goto L2 ; If condition is false, exit the loop
    ; Loop body
    ; (Statements inside the loop body would be here)
    t3 = i + 1 ; Update
    i = t3      ; Copy the result of update to i
    goto L1      ; Go back to condition

L2: ; After the loop
  
```

5. אופטימיזציה של הקוד (Code Optimization)

קלוֹט: TAC - Three Address Code
 פלוֹט: TAC - Three Address Code
 אופטימיזציה

בשלב זה נעבור על ה TAC ונחפש הזדמנויות בהן ניתן לשפר את הקוד ולהחסוך זמן ריצה, לדוגמה:

$a = b + c$

$d = a * 2$

$e = b + c$

$x = d + e$

ניתן לראות שהוא שוחזר על עצמו?
 $c + b$ لكن ניצור משתנה ונקרא לו $temp$ ומבצע את האופטימיזציה הבאה:

$temp = b + c$

$a = temp$

$d = a * 2$

$x = d + temp$

וכך חסכנו זמן מעבד בזכות זהה של pattern שוחזר על עצמו...

6. ייצור קוד (Code Generation)

קלט: TAC - Three Address Code הציג פשרה ואחדיה של קוד לצורכי בקרה ואנליזה לאחר אופטימיזציה
פלט: קוד assembly

בහילך זה נעבור על הקוד הבינים אחרי האופטימיזציה וכל פקודה נעביר לרצף הוראות assembly

למשל:

1. $a = b + c$
2. $d = a * 2$
3. $e = b + c$
4. $x = d + e$

הפרק ל:

```

; Line 1: a = b + c
MOV eax, [b]      ; Load value at memory location b into eax
MOV ebx, [c]      ; Load value at memory location c into ebx
ADD eax, ebx      ; Add values in eax and ebx, result in eax
MOV [a], eax      ; Store result in memory location a

; Line 2: d = a * 2
MOV eax, [a]      ; Load value at memory location a into eax
IMUL eax, 2       ; Multiply eax by 2, result in eax
MOV [d], eax      ; Store result in memory location d

; Line 3: e = b + c
MOV eax, [b]      ; Load value at memory location b into eax
MOV ebx, [c]      ; Load value at memory location c into ebx
ADD eax, ebx      ; Add values in eax and ebx, result in eax
MOV [e], eax      ; Store result in memory location e

; Line 4: x = d + e
MOV eax, [d]      ; Load value at memory location d into eax
MOV ebx, [e]      ; Load value at memory location e into ebx
ADD eax, ebx      ; Add values in eax and ebx, result in eax
MOV [x], eax      ; Store result in memory location x

```

זה הליק מאד ישיר כי הקוד בינים מאד קרוב לשפת assembly

7. טבלת הסימונים (Symbol Table)

טיפולו הנתונים החשוב ביותר שנשתמש בו לאורך כל השלבים של הקומpileציה כל השמות של המשתנים לצד סוג מאוחסנים כאן, טבלה זו מקופה על המהדר לחפש במהירות את המשתנה לפי שם ולakhir אותו כ-נתון וגם משתמש בה כדי לנחל את מה קיים באיזה scope וכך לישם בדיקת סוגים והאם קיימים ביטויים לא נכון מർחבינה הזו...
יכול להיות מיושם על ידי hash table כך שלכל משתנה יש כניסה למשל:

`Int interest`

יהיה מוגדר כך:

`<int , interest>`

arat the Scope Management – מה שחשוב זה שנבדיל בין scopes השונים בתוכנית שלנו כדי שנעשה זאת ה- symbol tables יהיו מסודרים בסדר היררכי:

```

...
int value=10;

void pro_one()
{
    int one_1;
    int one_2;

    {
        int one_3;           \
        int one_4;           |_ inner scope 1
        }
    }

    int one_5;

    {
        int one_6;           \
        int one_7;           |_ inner scope 2
        }
}

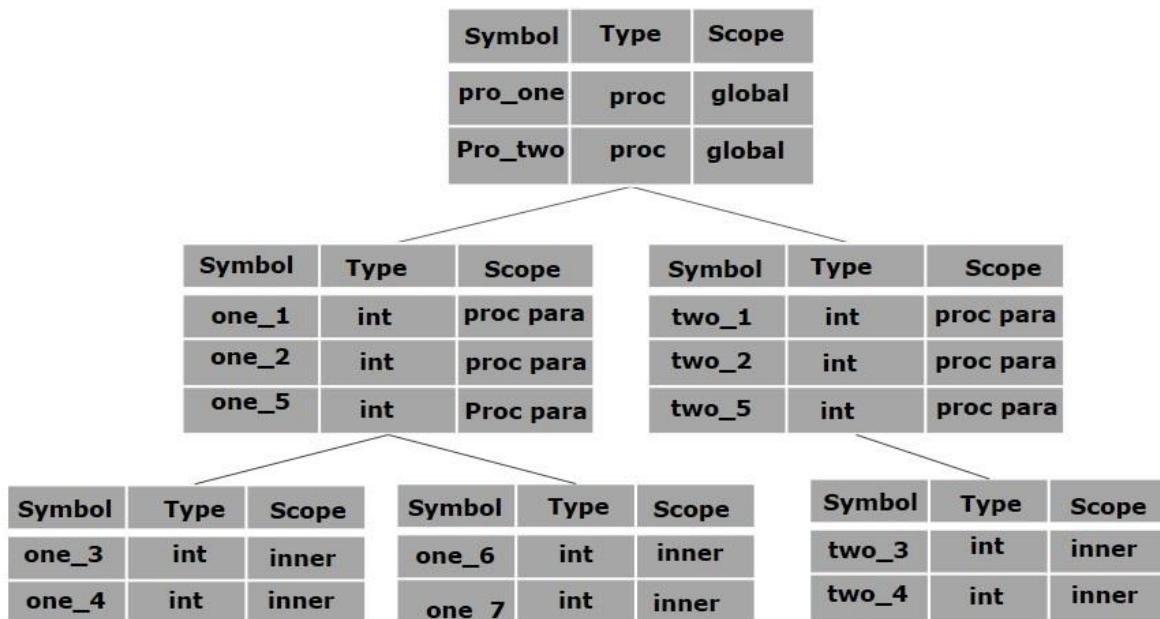
void pro_two()
{
    int two_1;
    int two_2;

    {
        int two_3;           \
        int two_4;           |_ inner scope 3
        }
    }

    int two_5;
}
...

```

נממש את הקוד הבא בצורה זו:



וניתן לראות שכל scope בעלת ענף משלה - ניתן להבדיל ולראות את ההיררכיה בצורה ברורה יותר.

מכונות מצבים

מכונות מצבים לעתים הנקראת אוטומט, היא מודל מתמטי חישובי. אשר נמצאת במצב אחד מוך אוסף סופי של, מצבים בכל רגע נתון. המצב מוחלט לפי הקלט המוגבל לתווך קלט מוגבלים מראהש האוטומט מקבל שפה פורמלית.

שפה פורמלית

שפה פורמלית היא סוג של "שפה" אבסטרקטית שמשמשת לתיאור של מבנים וערכיהם בצורה פורמלית, כלומר, בדרך שבה ניתן להגדיר קטגוריות, כללים ומבנה של מיללים או משפטים תוך שימוש בסימנים ובכלי יצירה.

נניח, לדוגמה, שאני רוצה לתאר באופן פורמלי את קבוצת כל המספרים הטבעיים. אז אני יכול להגיד שפה פורמלית זו על ידי כל יצרה שמתארים את המבנים האפשריים של מספרים טבעיים:

1. כל מספר טבעי מורכב מספרות שאין שליליות.

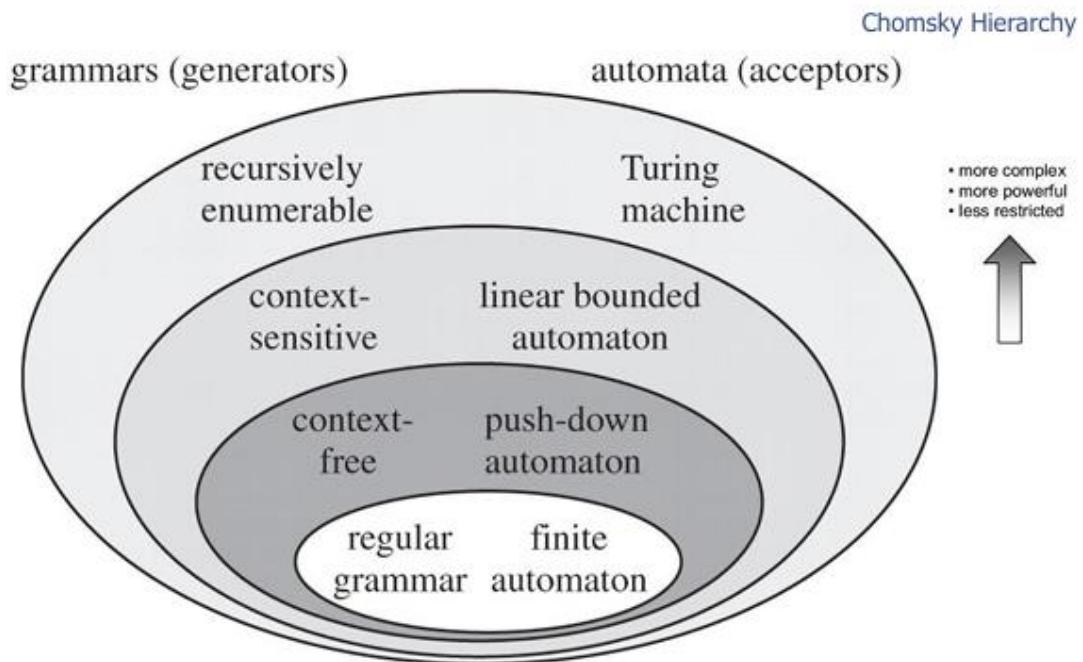
2. כל ספרה במספר יכולה להיות 0, 1, 2, 3, 4, 5, 6, 7, 8 או 9.

באמצעות הגדרת כלים כאלה, אני מטאר בצורה פורמלית את המבנה של המספרים הטבעיים. הקשר בין המושגים, הסימנים והכללים מגדיר את השפה הפורמלית שהוא מגדירים.

בעזרת השפות הפורמליות, ניתן לתאר ולהבין מבנים רבים בתחום המדעים המדוייקים והטכנולוגיים, כמו גם בתחום התכונות והתיאור של שפות תכנות. השפות הפורמליות מסייעות בהבנת המבנה של תכניות ונתונים, וספקות כלים לתיאור ולניתוח דיווני ומדוקן של מבנים וערכיהם באופן מדעי ופורמלי.

סוגי אוטומטים

יש יותר מסוג אחד של אוטומט, כל אוטומט והחזק שלו ואת סיבוכיות הקלט שהוא יכול להתחמודד. את ההיררכיה של האוטומטים ניתן לראות בהיררכיה האוטומטיים שהגדיר Noam Chomsky , החל מהחלש ביותר (אוטומט סופי) ועד למוכנת טירינג (המודל התאורטי המתאר מחשב מודרני). אנו נתרכז בשניים החלשים ביותר.

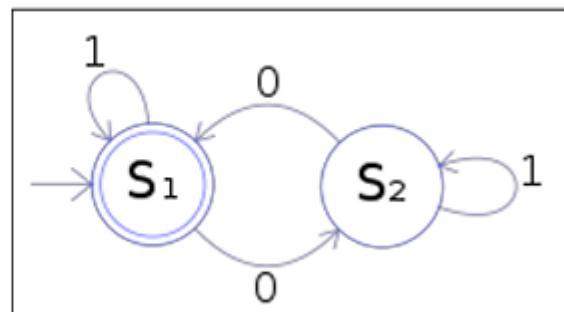


אוטומט סופי (Finite State Machine)

מדובר במבנה מצבים אשר נמצא במצב אחד מתור אוסף של מצבים בכל רגע נתון ועוברת ממצבים רק על ידי תוו קלט מסוים.

המטרה של אוטומט סופי היא לאפיין בדרך כלל שפות רגולריות.

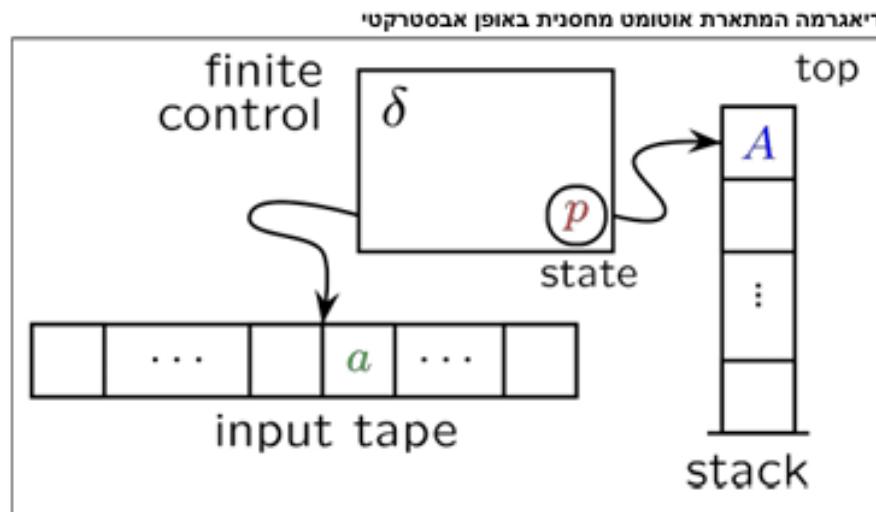
דיאגרמה המתארת אוטומט סופי מעל הא"ב $\{0, 1\}^*$ המקבל רק מחרוזות עם מספר זוגי של אפסים



כל עיגול מייצג מצב ולפי התו הבא אנו נחליט לאיזה מצב לעבור כאשר נשים את הקלט ב 1 או אז' הקלט נכון ונכון ועומד בתנאי השפה, אחרת הקלט שגוי.

אוטומט מחסנית (Push Down Automaton – PDA)

אוטומט הנעזר במחסנית ליביצוע עבודותיו, המעביר בין המצבים מוחלט לא רק על ידי הקלט אלא גם על הערך שבראש המחסנית.



אנו מחליטים לפי הערך שמקבלים מהקלט מהפעולה שנעשה למחסנית (נוציא או נכניס). המחסנית היא האינדיקציה למצב שגוי או נכון. אם בסיום הקלט נישאר עם ערך אחד במחסנית סימן שהצלחנו ליצמצם את הקלט לפי ה *rules production* וסימן שהוא נכון. אם לא אז שהוא נפל באחד מ*the rules production* והוא לא עומד בתקן.

תיאור הבעה האלגוריתמית

כפי שהבנו תהליכי הקומpileציה מורכב משלבים וכל אחד ואחד מהם הוא אתגר אלגוריתמי בפני עצמו.

ניתוח מילוני (Lexical Analysis)
איך ניתן להפוך קובץ טקסט לרצף Tokens בזמן לינארי, למשל כיצד ניתן לקטול מילים לאסימונים
ואיך ניתן להבדיל ביניהם למשל:

כאשר נתקלים ב +, האם להפוך אותו לאסימונ או לחייב כי יכול להגיע אחריו גם עוד + ואז זה האסימונ الآخر ++. צריך פתרון גנרי וחצק.

ניתוח תחבירי (Syntax Analysis)

ידוע כי הלקר מזרים אסימונים לפרסר אבל כיצד ניתן להשיג בזמן ריצה לנארו אלגוריתם המוצא הגיון ברכף האסימונים ומחליט מה הגיוני ומה לא הגיוני בצורה מוחלטת. בנוסף לבנות עץ מדויק המתאר את אופן פעולות הקוד.

איך נדע לאיזה המשך לצפות באופן מדויק למשל:
כאשר אנו מקבלים את האסימונ X שהוא משתנה אנו יכול להמשיך אותו בכמה דרכים כדוגמת:
X = 1;
X->type;
איך נדע למה לצפות, האם להמשיך לקלט הבא או לנסות לצמצם.

ניתוח סמנטי (Semantic Analysis)

אומנם יש לנו עץ המתאר רצף הגיוני של אסימונים אבל אנו עכשו נצטרך לבדוק את הרצף הלוגי של התוכנית. אם נעשה שימוש במשתנים לא מוגדרים, בדיקות סוגים. למעשה אנו נverb על העץ שקיבלו מהניתוח התחבירי ונבדוק את הנכונות הלוגית של הקוד.

איך נעשה זאת בזמן לנארו עם הפתרון הכני גנרי שקיים.

אסור לבצע פה טיעות זה השלב האחרון ובסתורו אם המהדר מאשר אזי הקוד תקין لكن אין מקום לאי דיווקים.

סיקת אלגוריתמים בתחום הבעה

תהליך הניתוח התחבירי הוא המורכב ביותר מבין שלבי הקומpileציה لكن אציג את הדרכים העומדות לרשותך.

Parsing Algorithms

כדי ליצור את ה Parser Tree עליו מבוססת הקומpileציה קיימים מספר אלגוריתמים המתחלקים לשני

סוגים עיקריים:

Top Down Parsing . 1

Bottom Up Parsing . 2

בגלל שפת התוכנות שלנו היא Context free נפרש אותה באמצעות מכונת מצבים המשתמשת במחסנית (Pushdown).

Top Down Parsing

אנו נתחיל מראש עץ התחביר ונרדך לחלק התחתון שבו כאשר נשתמש בכלל שיכתוב השפה bnf.

Recursive Decent Parsing

דרך נפוצה הממשת TDP עץ הניתוח נוצר מראשו עד למטה והוא קוראים את הקלט מהטסוף להתחלה. השיטה שומרת מערכת פונקציות לכל terminal או nonterminal שנמצא ב grammar .

בגלל שאנו נעבור באופן רקורסיבי על הקלט נסבול backtracking אזי יעילות האלגוריתם עלולה להיות אספוננציאלית כלומר ($O(2^n)$)

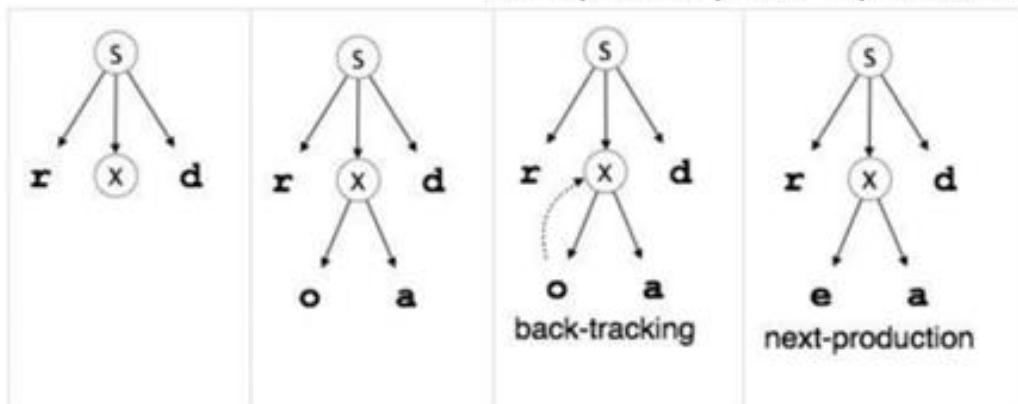
כאשר ה Parser משתמש בשיטה זו ייצרו מצבים בהם המנתח הגיע למבי סתום, בגל לחירה לא נכונה של כל מסויים בדרכ. לכן הוא יחזיר למקום האחרון שבוצע בו ההחלטה וינסה כל אחר... אופן פעולה זאת נקראת Backtracking, רק כאשר הוא ינסה את כל האפשרויות ולא יצליח להתאים את הקלט לתנאי השפה יcriع כי הקלט שגוי.

לדוגמא:

נתון ה – Grammar הבא:

$$\begin{aligned} S &\rightarrow rXd \mid rZd \\ X &\rightarrow oa \mid ea \\ Z &\rightarrow ai \end{aligned}$$

עבור מחרוזת הקלט "read" קך יראה תהליך הניתוח:



Bottom Up Parsing

אנו נתחיל מהתחתי של עץ התחבר וונעלה אל חלקיו העליונים.

כאשר אנו בונים עץ מלמטה עד למעלה אנו רוצים להרכיב את ה Non terminals מרצף ה Terminals שלרשוטנו והם היו הבנים שלו. בעז

אבל כדי להחליט איזה תחת איזה nonterminal הם יהיו נדרש לבצע 2 סוגים של פעולות:

Shift Step

שלב זה מעביר token לטור המחסנית ויוגדר צומת אחד על עץ הניתוח

Reduce Step

רק כאשר המנתח מוצא כלל יוצר שלם משמע שהסמלים הנוכחים מתאימים לאחד מן ה Production rules אנו ניצור צומת עץ חדש בו בניו יהיה הסמלים שהוציאנו מהמחסנית (כמוות הסמלים שנוציא כתובה ב RHS של כלל הייצור) והשורש יהיה RHS של אותו כלל ייצור. אנו למשעה עושים פופ למחסנית ואז דוחפים את הצומת החדש של העץ

עלילות האלגוריתם זהה היא לנארית ☺

למשל נתון הגרמאר הבא:

$S \rightarrow S+S$
$S \rightarrow S-S$
$S \rightarrow (S)$
$S \rightarrow a$

עבור המחרוזת $a1-(a2+a3)$ נבדוק איך יראה התהיליך של הניתות:

Stack contents	Input string	Actions
\$	$a1-(a2+a3)$$	shift a1
\$a1	$-(a2+a3)$$	reduce by $S \rightarrow a$
\$S	$-(a2+a3)$$	shift -
\$S-	$(a2+a3)$$	shift (
\$S-($a2+a3)$$	shift a2
\$S-(a2	$+a3)$$	reduce by $S \rightarrow a$
\$S-(S	$+a3) \$$	shift +
\$S-(S+	$a3) \$$	shift a3
\$S-(S+a3) \$	reduce by $S \rightarrow a$
\$S-(S+S) \$	shift)
\$S-(S+S)	\$	reduce by $S \rightarrow S+S$
\$S-(S)	\$	reduce by $S \rightarrow (S)$
\$S-S	\$	reduce by $S \rightarrow S-S$
\$S	\$	Accept

LR Parser

מנתח אשר משתמש בשיטת BUP ועובד זאת בזמן לינארי, הרעיון האלגוריתמי מאחוריו זאת שיטת Shift Reduce וכאשר נוצר אחד כזה נדרש BNF מדויק כדי שבמצב שיש לנו חוקים מקבילים הוא ידע לקחת את ההחלטה הנכונה בכל מצב.

אסטרטגיה

אציג את האסטרטגיה וגישה הפתרון שלי לבועה האלגוריתמית הנדרשת Compiler. איזג את Complier באמצעות מנגנונים שונים שבעזרתן אשיג יעילות אלגוריתמית של $O(n)$. אפרת בפרק זה איך אסטרטגיה זו בא לידי ביטוי בשלבים השונים בתהליך הקומpileציה.

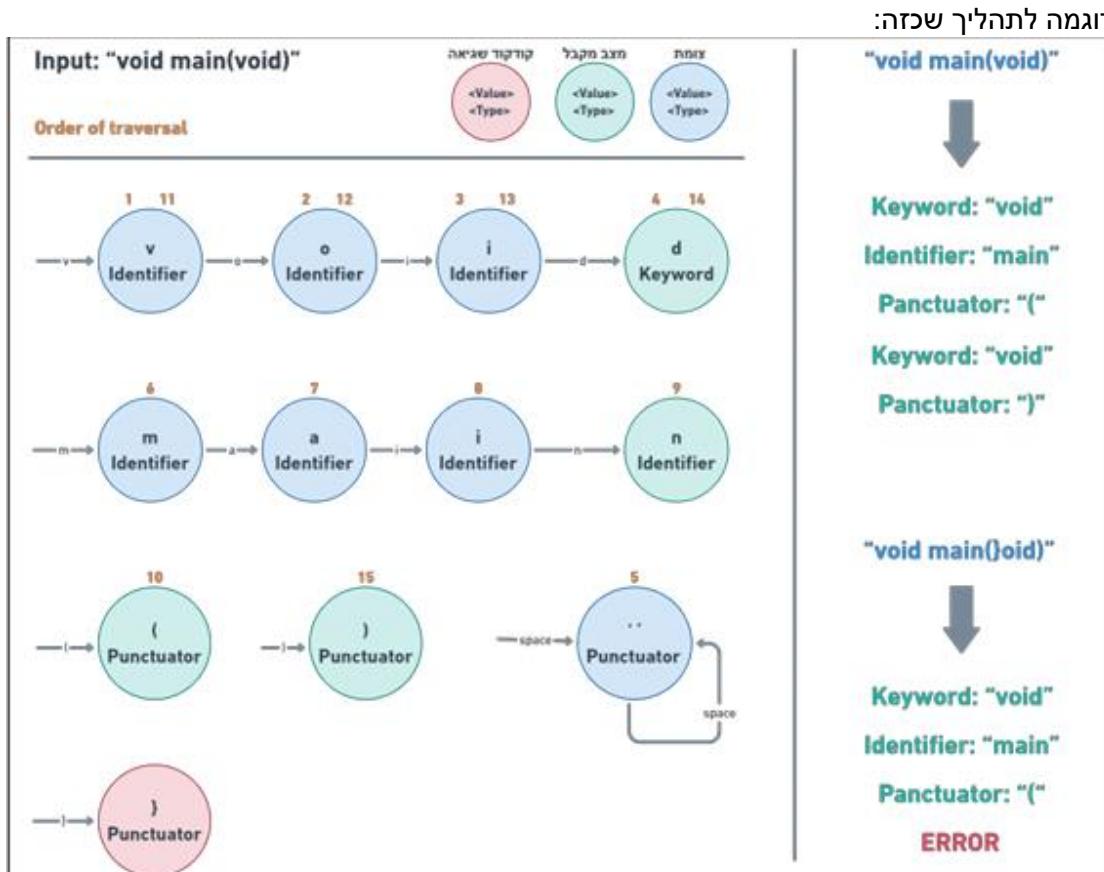
ניתוח מילוני (Lexical Analysis)

אנו שואפים ליעילות לנארית שכן לא נרצה להתייחס לקוד המקורי כתיקסט אלא נתרגם אותו לאסימונים בעלי משמעות. כפי שיקרנו בחלק התיאורטי.

למעשה אנו רוצים ליצור אוטומט סופי המציג ע"י מטריצה בעלת רוחב של כמות התווים בטבלת ASCII כדי שנוכל לסwoג כל تو שנקבל לעמודה, ואורך המטריצה יהיה ככמות המznים הקיימים.

כאשר נכניס למטריצה את התו נוכל ביעילות של 1 להציגו למצב הבא אז התו והמצב הקודם ישמשו לאינדקס של המצב הבא.

וכאשר נגיע למצב בו יופיע Token נחזיר למצב ההתחלה ונכניס את ה Token למערך האסימונים. ככה נוכל בזמן לנארה נסwoג את כל התווים.

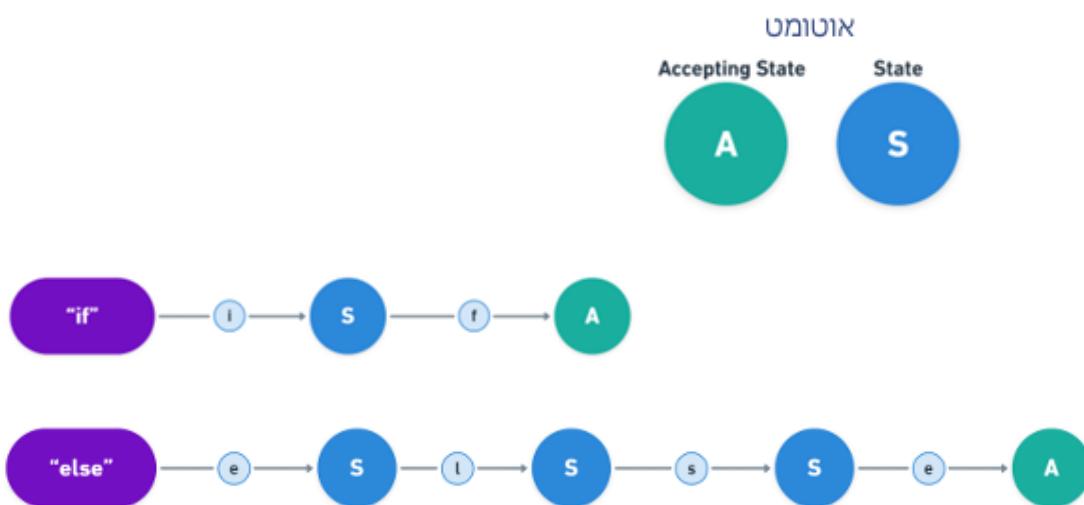


זהו אלגוריתם אשר מוצא אחד בכל פעם כי החלטתי שאני בונה את העץ תוך כדי שלב המילון (בו זמנית):

1. **התחל** בטוקן ריק והתייחס כרגע לתווים בקלט.
2. כל עוד יש קלט וטוקן לא סימנו:
3. העברת התו הבא בקלט למוכנת הסופית וקבלת המצב הנוכחי.
4. אם המצב הוא חיובי:
5. **הוסף** את התו לטוקן הנוכחי.
6. אחרת אם המצב הוא אפס:
7. **הוסף** את התו לטוקן הנוכחי וקבע את המצב הנוכחי לאחרון בטבלת המעברים, והגדיר דגל מצחה לאמת.
8. אחרת:
9. אם הטוקן ריק:
10. **בנה** טוקן מהתו הנוכחי וקבע את הטוקן כਮובן בטבלת המעברים.
11. הסטיים.
12. אחרת:
13. **חפש** את הטוקן בטבלת המעברים.
14. אם המצב הוא מסוף:
15. השתמש בטבלה לאיתור סוג הטוקן (כמו מסוף).
16. אחרת אם המצב הוא מחוזצת:
17. **השתמש** בטבלה לאיתור סוג הטוקן (כמו מחוזצת).
18. אם התו הראשון הוא גרש יחיד:
19. **הסר** את הגרש ובודק האם זהו תו ייחיד, אם כן - טוקן סדרתי, אם לא - שגיאה.
20. אחרת:
21. שגיאה.
22. אחרת:
23. **השתמש** בטבלה לאיתור סוג הטוקן.
24. **בנה** טוקן מהתווים והוסיף אותו לרשימת הטוקנים.
25. הסטיים.
26. כאשר הסטיינו הלוואות, אם יש טוקן:
27. אם זהו תו גרש והוא לא חלק מחוזצת:
28. שגיאה.
29. אחרת:

30. חפש את הトーון בטבלת המעברים.
31. אם המצב הוא מספר:
32. השתמש בטבלה לאיתור סוג הトーון (כמו מספר).
33. אחרת אם המצב הוא מחרוזת:
34. השתמש בטבלה לאיתור סוג הトーון (כמו מחרוזת).
35. אם התו הראשון הוא גרש יחיד:
36. הסר את הגרש ובודק האם זהו תו ייחיד, אם כן - טוון סדרתי, אם לא - שגיאה.
37. אחרת:
38. שגיאה.
39. אחרת:
40. השתמש בטבלה לאיתור סוג הトーון.
41. בנה טוון מהתוונים והוסף אותו לרשימת הトーונים.
42. אם אנחנו הגיעו לסוף הקלט ולא נכנסנו לתוך טוון:
43. חזר מתחילה.
44. החזר את הトーון האחרון.

אלגוריתם לבניית מטריצת מצבים מסויימים נתוניים של שפה
 כאשר אני כותב אני אוחב שהפתרון שלי יהיה פשוט גנרי ולכן במקומות לבנות ידנית את המטריצה הנ"ל כתבתי אלגוריתם שיכל לקבל מערך Tokens של שפה מסוימת ולבנות לה מטריצה שכזו.
 האלגוריתם מtabסס על זה שעיל כל כיוון חדש של רצף אותיות הוא יוצר שורה חדשה במטריצה ומאתחל את העמודות הרלוונטיות בצורה אוטומטית במספרים של המצביעים המתאימים.
 דוגמא של כמה מן האוטומטים שנוצרים בזכות אלגוריתם זה:

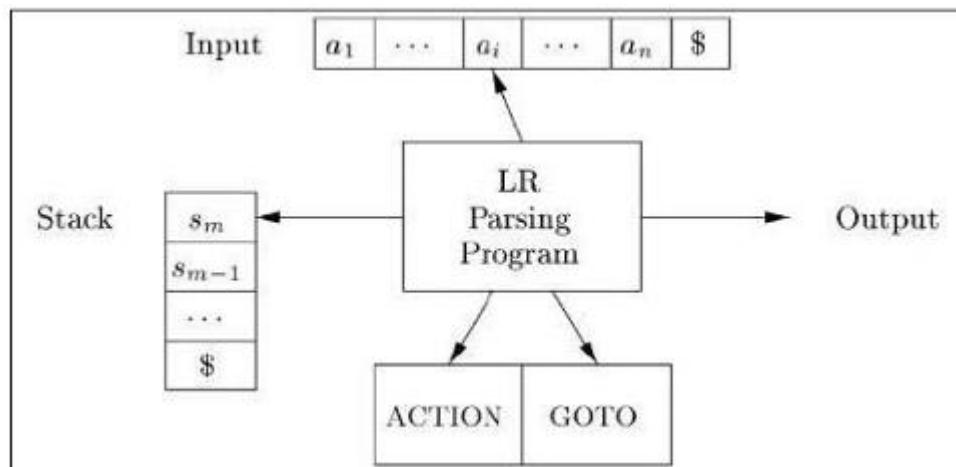


ניתוח תחבירי (Syntax Analysis)

בשלב זה נרצה לבדוק אם רצף האסימונים המתקבלים מה lexer תקין מבחינת Grammar השפה.

זה שלב מורכב יותר, יש לומר שהשלב המורכב ביותר בכל תהליך הקומpileציה. Pushdown Machine_CFG יכול להשתמש באופן אוטומט סופי כי מדובר בתחריר CFG لكن בשימוש בגישה Pushdown Machine. אוטומט מחסנית.

בגלל שאנחנו שואפים לזמן ריצה לינארית, מבין כל האופציות שיש לנו אלגוריתם Shift Reduce. NRAEA האופציה המשכנית ביותר לאור החופש Backtracking והצורה שנראית טבעיות כך הכל לבנות עץ מלמטה למעלה. لكن בשימוש Parser LR.



מחסנית וטבלאות LR Parser

נשתמש במחסנית ובשתי טבלאות, Goto Table – | Action Table.

מחסנית

המחסנית תחיל בתוכה צמתים של עץ שעמידים להתחבר לכל צומת מספר מצב ובראש המחסנית תמיד נמצא מספר המצב הבא אליו צריכים לעבור, כשמתחל התחılır מספר זה הוא 0.

כל פעולה SHIFT מטרת לדוחוף למחסנית את TERMINAL הנוכחי ופעולה ACTION אומרת להוציא מהמחסנית X צמתים לצמצם אותם לצומת חדשה ולהכניס למחסנית.

Action Table

מצינית למנתח בהתאם לקלט שהוא מקבל ולמספר שבראש המחסנית איזה פעולה לנוקוט.

Accept, Reduce, Shift

כל שורה מייצגת מצב ועל עמודה מייצגת אסימון terminal בלבד אזי אנו משתמשים בתווים מהקלט כדי לדעת איזו עמודה.

אם הגיענו לתא ריק סימן שהגענו למצב שלא יכול להתקיים ממשע שגיאה.

Goto Table

מצינית למנתח לאיזה מצב הוא צריך לעבור לאחר ביצוע פעולה Reduce, שאנו מוצאים את כמות האסימונים לפי RHS-Ano נציב ב GOTO את המצב בתווך השורה את ה LHS כעמודה וכן נדע איזה מצב להכניס למחסנית.

Action & Goto Tables
אנו נזכיר ל BNF הבא טבלאות Action ו Goto מתאימות

```

E' -> E
E -> E + T
E -> T
T -> T * F
T -> F
F -> ( E )
F -> id
  
```

STATE	ACTION						GOTO			
	+	*	()	id	\$	E'	E	T	F
0			s4		s5			1	2	3
1	s6					acc				
2	r2	s7		r2		r2				
3	r4	r4		r4		r4				
4		s4		s5			8	2	3	
5	r6	r6		r6		r6				
6		s4		s5				9	3	
7		s4		s5					10	
8	s6			s11						
9	r1	s7		r1		r1				
10	r3	r3		r3		r3				
11	r5	r5		r5		r5				

`id + id * id`

אחרי שיצרנו את הטבלאות המתאימות בוא ננסה ליצור עץ תחבירי עבור הביטוי:

Trace				Tree			
Step	Stack	Input	Action	E	+ T	* F	id
1	0	id + id * id \$	s5				
2	0 id 5	+ id * id \$	r6				
3	0 F	+ id * id \$	3				
4	0 F 3	+ id * id \$	r4				
5	0 T	+ id * id \$	2				
6	0 T 2	+ id * id \$	r2				
7	0 E	+ id * id \$	1				
8	0 E 1	+ id * id \$	s6				
9	0 E 1 + 6	id * id \$	s5				
10	0 E 1 + 6 id 5	* id \$	r6				
11	0 E 1 + 6 F	* id \$	3				
12	0 E 1 + 6 F 3	* id \$	r4				
13	0 E 1 + 6 T	* id \$	9				
14	0 E 1 + 6 T 9	* id \$	s7				
15	0 E 1 + 6 T 9 * 7	id \$	s5				
16	0 E 1 + 6 T 9 * 7 id 5	\$	r6				
17	0 E 1 + 6 T 9 * 7 F	\$	10				
18	0 E 1 + 6 T 9 * 7 F 10	\$	r3				
19	0 E 1 + 6 T	\$	9				
20	0 E 1 + 6 T 9	\$	r1				
21	0 E	\$	1				
22	0 E 1	\$	acc				

עץ ניתוח

בנוסף לבדיקה הרצפים אנו בונים עץ ניתוח שיישמש אותנו לאורך המשך הקומpileציה.

בניתו تعدה כאשר אנומבצע REDUCE האסימונים אותם נוציא הם הינם בצורת שבת השורש הוא ה-SHA של החוק "יצור הנוכחי" וכאשר אנו געשה זאת לאורך כל הקלט במידה והוא נכון במחסנית-Amor להישאר רק שורש של עץ אחד ומורכב מכל הקלט וזהו העץ אותו אנו מחפשים.

אלגוריתם הניתוח

הינה הפסאודו קוד עבור האלגוריתם של המנתח עבור אסימון בכל פעם אזי הוא נבנה כדי שברגע שהוא מקבל אסימון ועוד מהלך סר' יהיה ניתן לשולח אותו לפרסר:

1. **בדיקה אם הטוקן הוא NULL והחזרת 0** במקרה חיובי.
2. **בדיקה האם הטיפוס של הטוקן קיים** במאפיין המפרידים של המפענה, והחזרת 1 אם כן.
3. **בדיקה האם הטוקן הוא מזהה (IDENTIFIER)**, ואם כן - אין פעולה נוספת.
4. **הצב ערך העליון** של המחסנית במשתנה המצביע הנוכחי.
5. **קבל** הטיפוס של הטוקן ומספר המצביע שבו אנו נמצאים.
6. **בדיקה של הפעולה** הבאה בטבלת הפעולות של המפענה.(Action Table)
7. אם אין פעולה מתאימה, החזרת 0.
8. אם יש פעולה, בדיקה אם היא פעולה בסיס "הזזה" (SHIFT).
9. אם כן, **יצירת** צומת חדש והכנסתו **למחסנית**, והחזרת 1.
10. אם לא, המשך לשלב הבא.
11. אם הפעולה היא פעולה קיבוץ (REDUCE):
12. מציאת החוק המתאים בטבלת החוקים.
13. **הורדת** מספר המצביעים הנדרשים לפי החוק מהמחסנית.
14. **יצירת** עץ חדש והכנסתו **למחסנית**, והחזרת -1.
15. אם הפעולה היא פעולה אחרת, **החזרת 1**.

מימוש LR Parser

לאחר ביסוס התאוריה נצטרך למצוא דרך לבנות את הטבלאות הללו.

כדי למש את ה-LR PARSER נצטרך לחלק זאת לשני חלקים:

1. **ליקוי דיאגרמת מעברים** המייצג את תחביר השפה.
2. **יצירה של ACTION ושל TABLES GOTO** בעזרת דיאגרמה זו.

The Dot Notation

באמצעות נקודה הנמצאת בין חלקים שונים של ה-SHA נוכל לדעת כמה מחר היצור הספציפי כבר כיסינו.

לדוגמא:

$A \rightarrow \cdot XYZ$
 $A \rightarrow X \cdot YZ$
 $A \rightarrow XY \cdot Z$
 $A \rightarrow XYZ \cdot$

נתן לנו 4 אופציונות: $A \rightarrow XYZ$

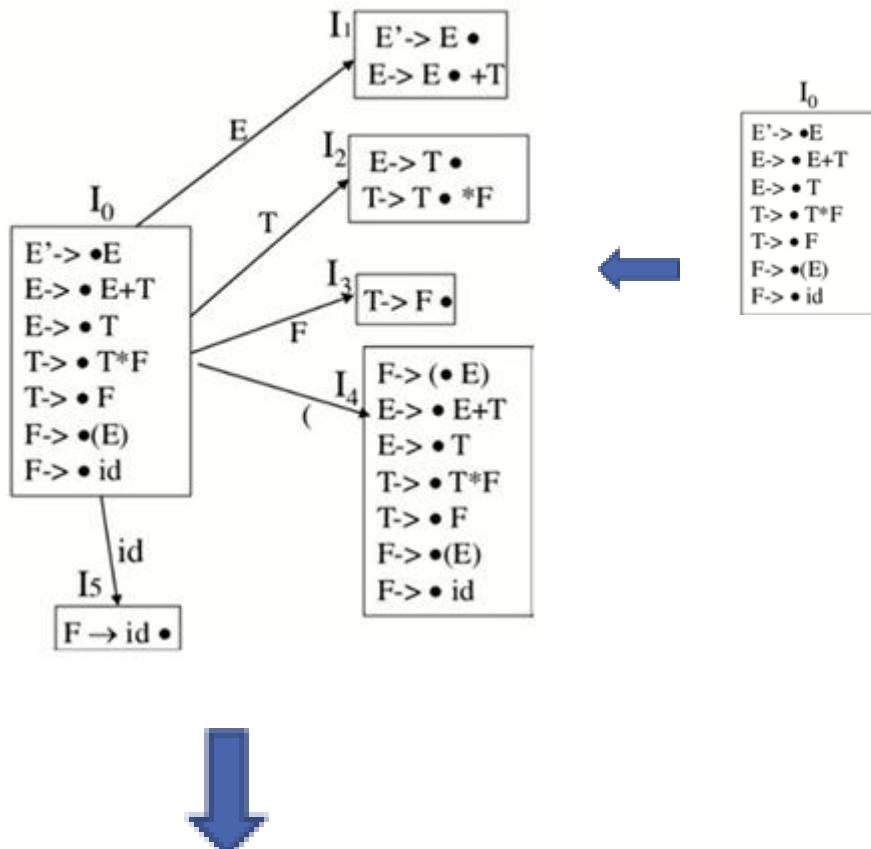
בנייה דיאגרמת המעברים
באמצעות הnotation Dot Notation אנו יוצרים גרף של אפשרויות:

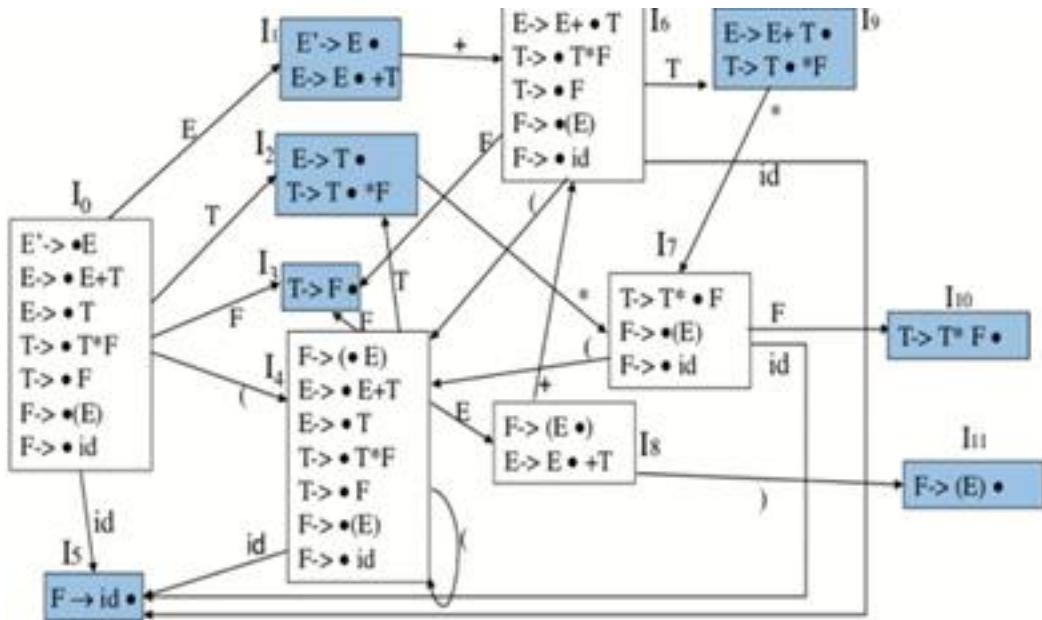
ניקח את ה **Grammar** הבא

$E' \rightarrow \bullet E$
$E \rightarrow \bullet E + T$
$E \rightarrow \bullet T$
$T \rightarrow \bullet T * F$
$T \rightarrow \bullet F$
$F \rightarrow \bullet (E)$
$F \rightarrow \bullet id$

ונמשיך עבור כל מצב את ה **rules** האפשריים
האפשריים עבורו

נדיר אותו במצב ראשון:





נושיר לעשות זאת עד שהגענו לכל מצב הkaza האפשרים וכך תיראה דיאגרמת ה GRAMMAR שלנו

וכעת קיבלנו דיאגרמה שמייצגת את האוטומט שנדרת לנו כל מה שאנו צריכים לדעת, איזה חוקים אפשריים בכל מצב, ובעור איזה סמל עוביים לאיזה מצב.

בנייה הטבלאות

באמצעות השלב הקודם קיבלנו את דיאגרמת המעברים ממנה נרצה לייצר את שתי הטבלאות ההכרחיות לפרסר.

מעבר על קשיות - SHIFT

מעבר על קשיות הדיאגרמה ובעור קשת עם משקל Nonterminal נעדכן מעבר זה בביטול ה GOTO, בעור קשת עם משקל terminal נעדכן בittelot the Shift Action פעולה Shift וצמוד אליה מספר המצביע אליו צריך לעבור לאחר ביצוע הפעולה.

מצב הטבלאות לאחר שעברנו על כל הקשיות שבDİagram מהשלב הקודם:

State	Action Table						Goto Table		
	id	+	*	()	\$	E	T	F
0	S5			S4			1	2	3
1		S6				ACC			
2			S7						
3									
4	S5			S4			8	2	3
5									
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9			S7						
10									
11									

מעבר על המצבים - REDUCE

מעבר על המצבים בעלי הנקודה בסוף ה - Production Rule בדיאגרמה מהשלב הקודם המציגים reduce שהגנוו לסוף ה Production Rule וניתן לבצע reduce וכן מסמנים את כל פעולות ה Follow set First set | action. אבל נctrיך להכיר את המושגים First set |

First & Follow sets

המ מספקים את המיקום המדוייק של כל Terminal ב – החלפה כדי להקל על קבלת ההחלטה reduce בפועלות ה

از First set הם כל ה Terminals שיכולים להופיע ישירות לאחר שימוש החלפה לאז Nonterminal זה או אחר, First set הם כל ה Terminals שיכולים להופיע מיד לאחר Nonterminal זה או אחר.

ה First set של nonterminal נקרא לו A יהי כל ה terminals שיכולים להופיע מיד לאחר A בכל Production Rule בהם A מופיע. גם אם יש NonTerminals בין נתעלם מהם עד שנגיע לשalls terminals שיכולים להיות אחר A:

לדוגמה, כך יראה ה – Follow set Grammar עבור המחלבים הקודמים

- (0) $E' \rightarrow E$
- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

$$\begin{aligned} \text{Follow}(E) &= \{ \$, +,) \} \\ \text{Follow}(T) &= \{ *, +,), \$ \} \\ \text{Follow}(F) &= \{ *, +,), \$ \} \end{aligned}$$

אלגוריתם לחישוב Follow set

- אם A הוא המצב ההתחלתי, אז $\$ \in \text{Follow}(A)$
- אם A הוא Non-terminal ויש לו Production כדוגמת $S \rightarrow A$, אז $\text{Follow}(A) \subseteq \text{Follow}(S)$
- אם A הוא Non-terminal ויש לו Production כדוגמת $S \rightarrow AB$, אז $\text{Follow}(A) \subseteq \text{First}(B)$
- אם A הוא Non-terminal ויש לו Production כדוגמת $S \rightarrow ABC$, אז $\text{Follow}(A) \subseteq \text{First}(C)$

כאשר אנו ידעים לחשב את Follow Set נוכל להמשיך למלא את טבלת ה Action בפועלות ה reduce.

עבור כל מצב בו הנקודה נמצאת בסוף ה rule Production נסמן בכל העמודות שבשורה של המצב הנוכחי שנמצאות ב Follow Set של המצב הנוכחי, Reduce ועל פי המספרים שמסומנים בסוגרים בחום לעיל. למעשה אנו מבצעים את פעולה ה reduce המתאימה רק בעמודות הנמצאות ב follow set של המצב הנוכחי ובכך רואים קדימה אסימון אחד בכל שלב.

כך יראו הtáבלאות לאחר סיום שלב זה. אלו הТАבלאות הסופיות איתן לבצע את תהליכי הניתוח התחבירי.

State	Action Table						Goto Table		
	id	+	*	()	\$	E	T	F
0	S5			S4			1	2	3
1		S6				ACC			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4			9	3	
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

ניתוח סמנטי (Semantic Analysis)

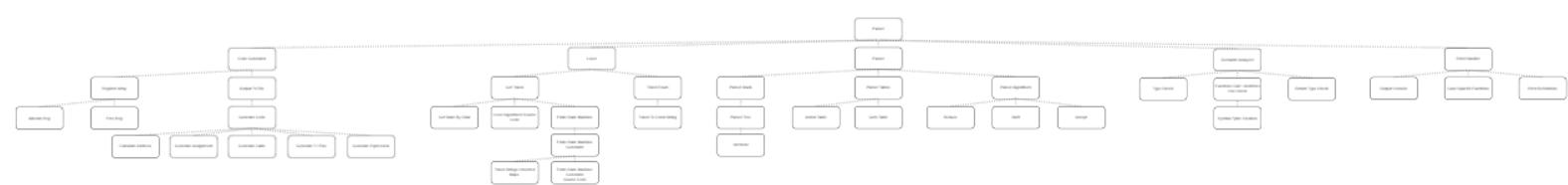
במהלך הניתוח הסמנטי עבור על העץ ומטרתי תהיה לבדוק את תקינותו מבחינה סמנטית.

בדיקת נכונות סוגים משתנים – נבדוק על ידי ריצה על העץ בצורה רקורסיבית האם המשתנים שווים בסוגם (השמה, ביטויים וכו').

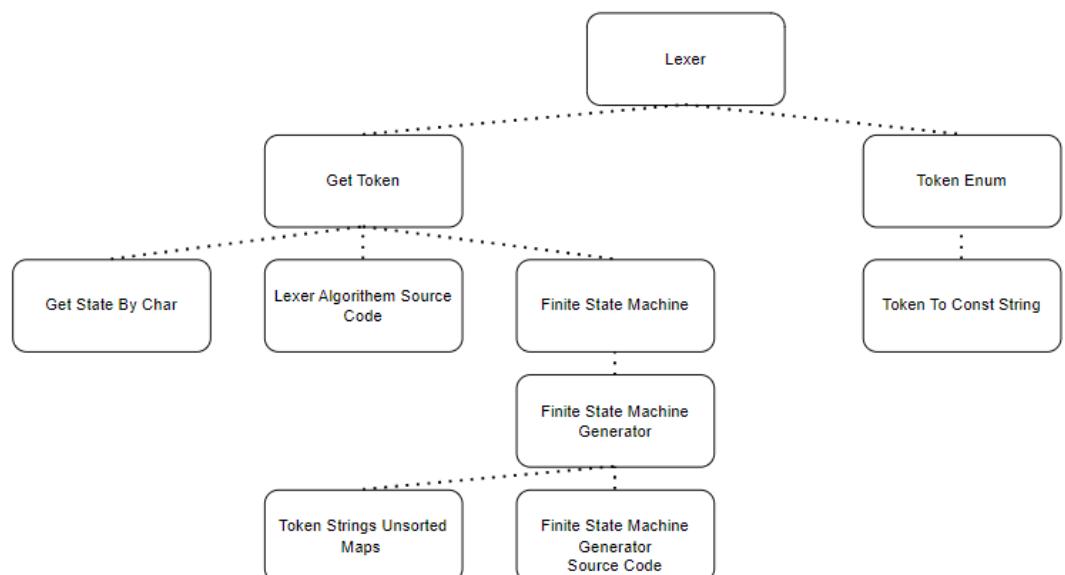
בדיקת נכונות שימוש במשתנים – האם המשתנה \ פונקציה קיימים וייחודם זאת אנו משיגים כאשר אנחנו מנהלים מבנה נתונים בשם **Symbol table** בתוכו נחזק את המשתנים שנתקלנו בהם ונבדוק מה קיימים.

Top Down Level Design

מבט על



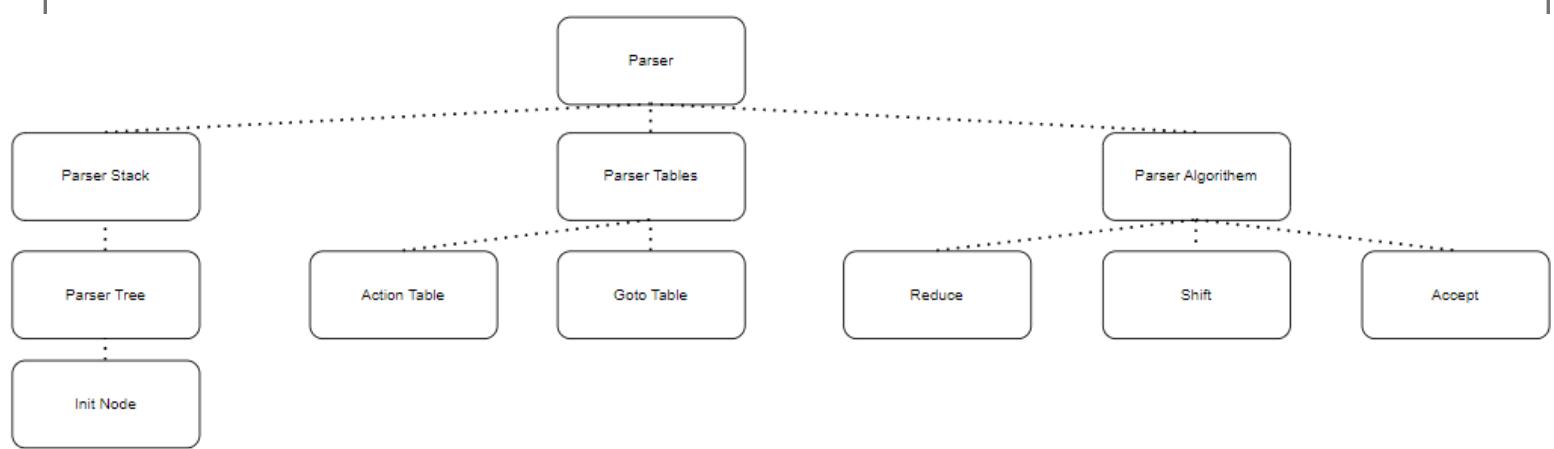
Lexer



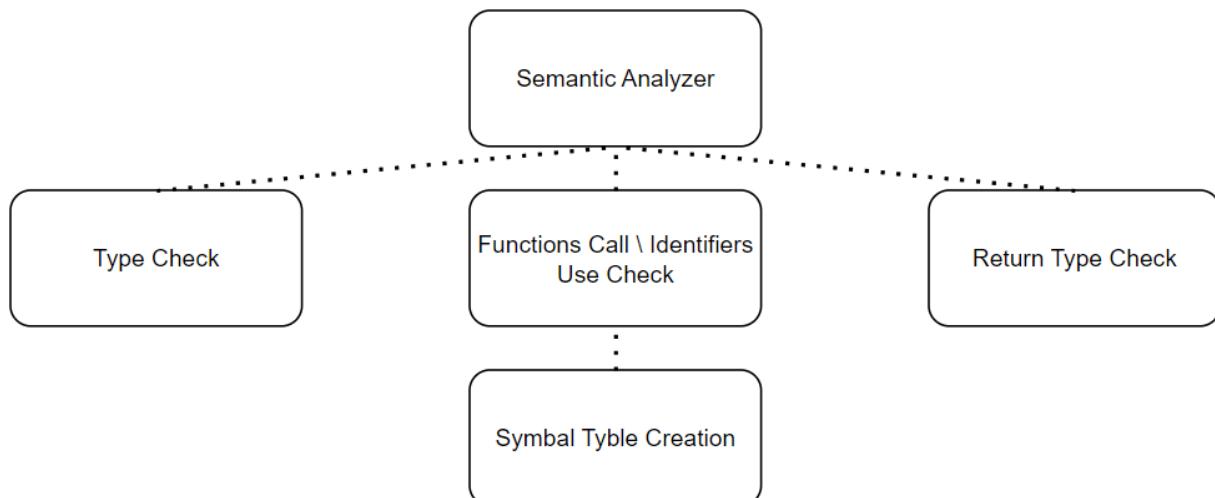
The Simple C Compiler

מוך פישמן

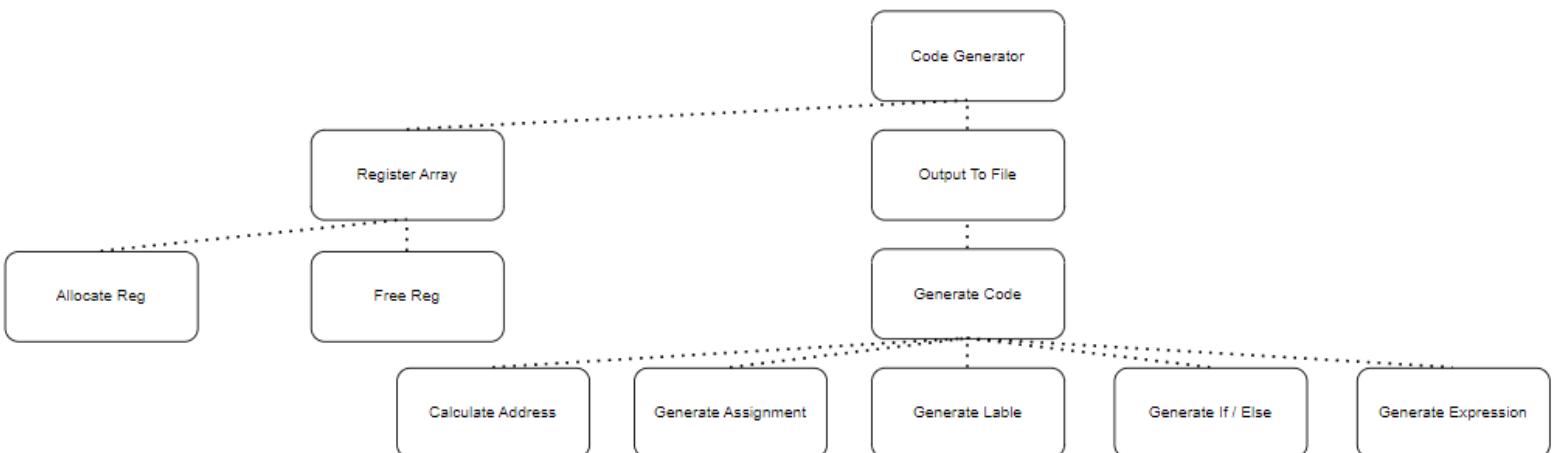
Parser



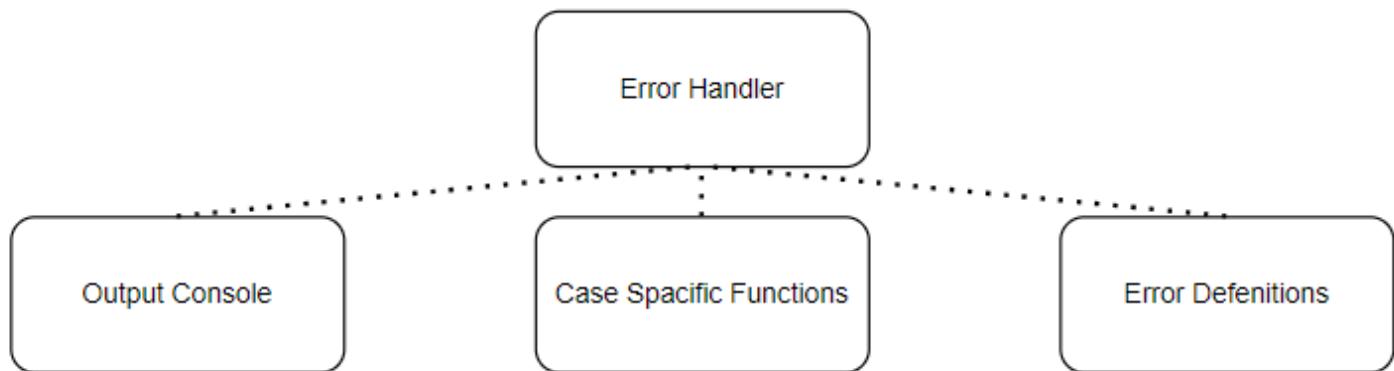
Semantic Analyzer



Code Generator



Error Handler



מבנה נתונים

בפרק זה אציג את מבני הנתונים בהם נזקרתי לצורך בניית הפרויקט

מנתח מילונים – Lexical Analysis

כפי שהוצג בפרקים הקודמים המנתח המילוני מורכב מאלגוריתם שבוסס על מכונת מצבים סופית, כדי לייצג אותה בצורה העיליה ביותר נעזר במטריצה זו ממדית כאשר כל טור מייצגת אות ascii וכל שורה מייצגת מצב במטריצה.

מבנה זה מאפשר לנו ביעילות של O(1) להשיג את המצב הבא במטריצה כאשר ניגש אליו באמצעות האינדקס המורכב מהאות וה מצב הקודם.

תרשים

```
line: 0. e: 5, f: 2, i: 4, t: 1, w: 3,
line: 1. n: 6,
line: 2. a: 7, o: 8,
line: 3. h: 9,
line: 4. f: 10,
line: 5. l: 11,
line: 6. u: 12,
line: 7. l: 13,
line: 8. r: 14,
line: 9. i: 15,
line: 10.
line: 11. s: 16,
line: 12. e: 17,
line: 13. s: 18,
line: 14.
line: 15. l: 19,
line: 16. e: 20,
```

מנתח תחבירי – Syntax Analysis

כפי שהוצע לעיל מימוש המנתח התחבירי נעשה על ידי אוטומט מחסנית, ראשית אוטומט מחסנית מייצג בצורה אבסטרקטית על ידי גרפ ולאחר מכן ניתן להפוך אותו לשני הטבלאות (Action , Goto ,) .

לאחר שאנו מבצעים חישובים שבעזרתם בונים את הטבלאות הללו נגיע לשתי מטריצות שמייצגות את הטבלאות האלו.

מבנה נתונים זה מאפשר לנו לקבל החלטות בכל מצב אפשרי במהלך תהליך הניתוח. בנוסף אנו משתמשים במילון כדי לייצג את ה Production Rules כך נשים יעילות של O(1) כאשר נרצה לגלות מהו המצב הבא. כאשר נרצה לדוחף למחסנית אנו נשתמש במבנה נתונים זה שייחסן מצבים לעציהם. אותם עצים בסופו יתחברו לעץ המנתח התחבירי הסופי.

תרשים של הגרף שהוא מייצגים באמצעות מטריצות

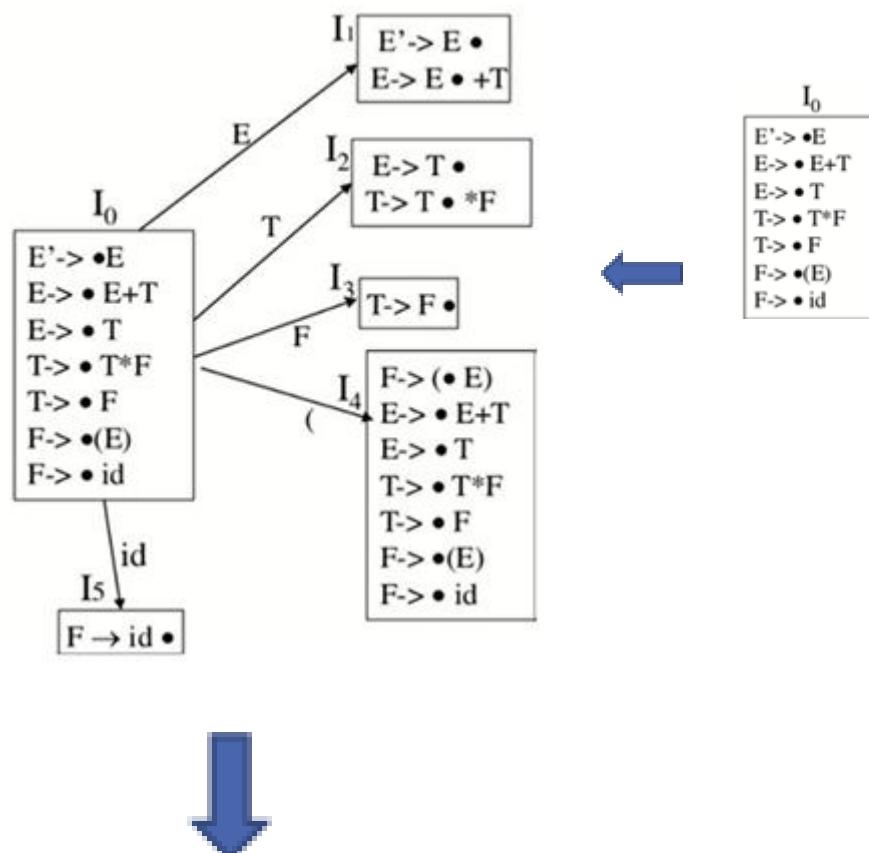
ניקח את ה Grammar הבא

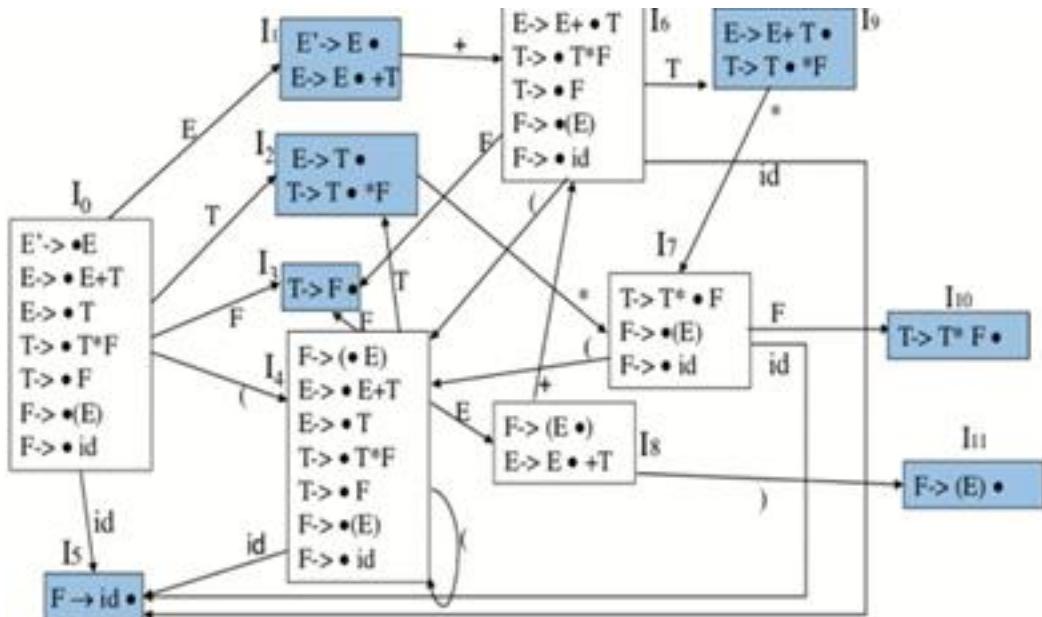
$$\begin{aligned} E' &\rightarrow \bullet E \\ E &\rightarrow \bullet E + T \\ E &\rightarrow \bullet T \\ T &\rightarrow \bullet T * F \\ T &\rightarrow \bullet F \\ F &\rightarrow \bullet (E) \\ F &\rightarrow \bullet id \end{aligned}$$

ונמשיך עבור כל מצב את המילים האפשריות

האפשריות
עבורו

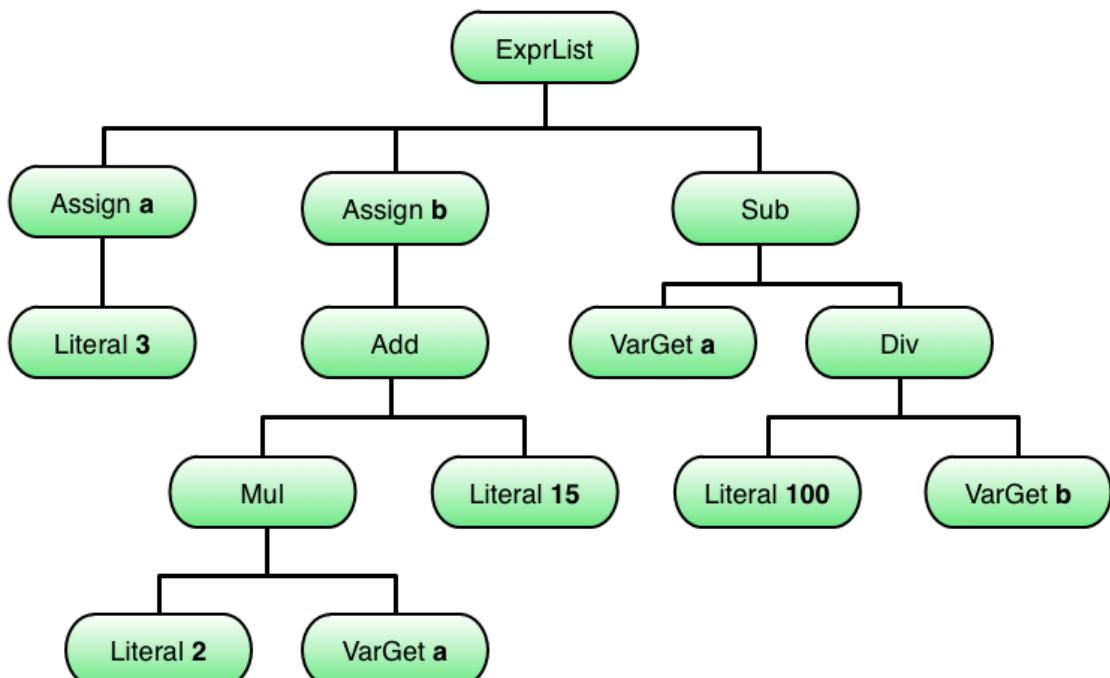
נדיר אותו במצב ראשוני:





תרשים של עץ הניתוח

$a=3; b=2*a + 15; a=100/b$



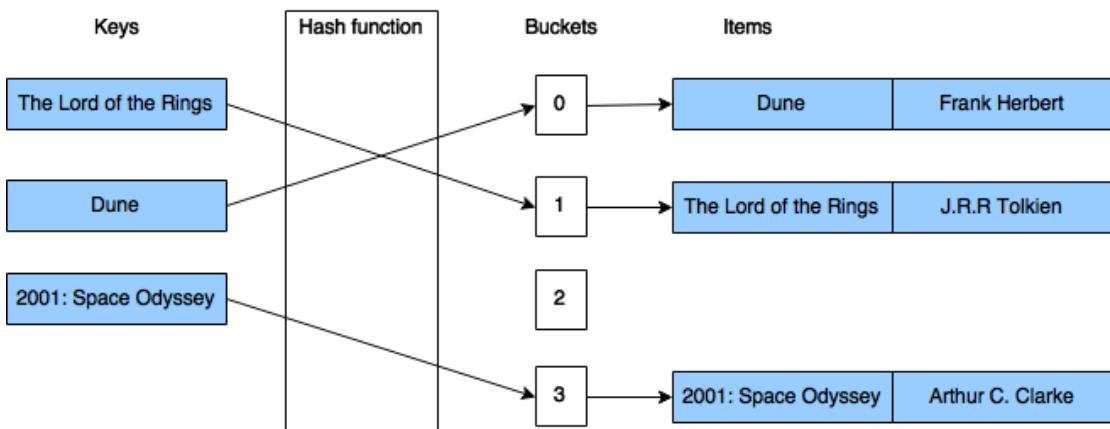
מנתח סמנטי – Semantic Analysis

עבור המנתח הסמנטי נכתב פונקציות בהן נשתמש לטובת סריקת עץ התחביר, כאשר נסרק את העץ עבור כל NonTerminal האם קיימת פונקציה היכולה לסרוק אותו במילון הפונקציות.

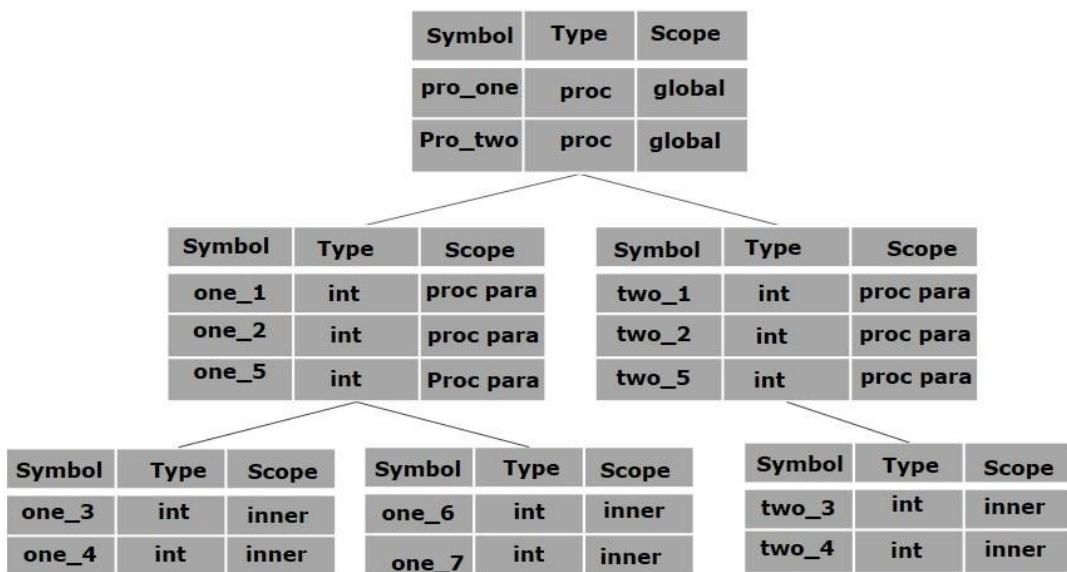
כך שב(1) נמצא את הבדיקה המתאימה לכל צומת בעץ.

במהלך המנתח הסמנטי נבנה את ה Symbol Table שיעזר לנו ביצירת הקוד הסופי ולכן נשתמש במבנה נתונים יעיל יאפשר לנו לדעת מה ה Identifiers בכל scope .

תרשים מלון הפונקציות



תרשים ה Symbol Table



Code Generation

כאשר ניצור את קוד האסמבילר הסופי נדרש להקצות Registers לכט נחזיק מערך בו כל תא יהיה מצביע למחלקה המייצגת Register שנותנת מידע אם הוא תפוס או שניתן להשתמש בו וכו'...

תרשים

r	0	1	2	3	4	5	6
name	%rbx	%r10	%r11	%r12	%r13	%r14	%r15
inuse	X		X				

טיאור סביבת העבודה ושפה התכנות

שפה התכנות



שפה התכנות שבה השתמשתי לפיתוח הקומpileר היא שפת C++.



וכמוון נעשה שימוש בסקריפטים של בדיקה ויצירת המוצר הסופי. Batch

סביבה העבודה



- Windows 10

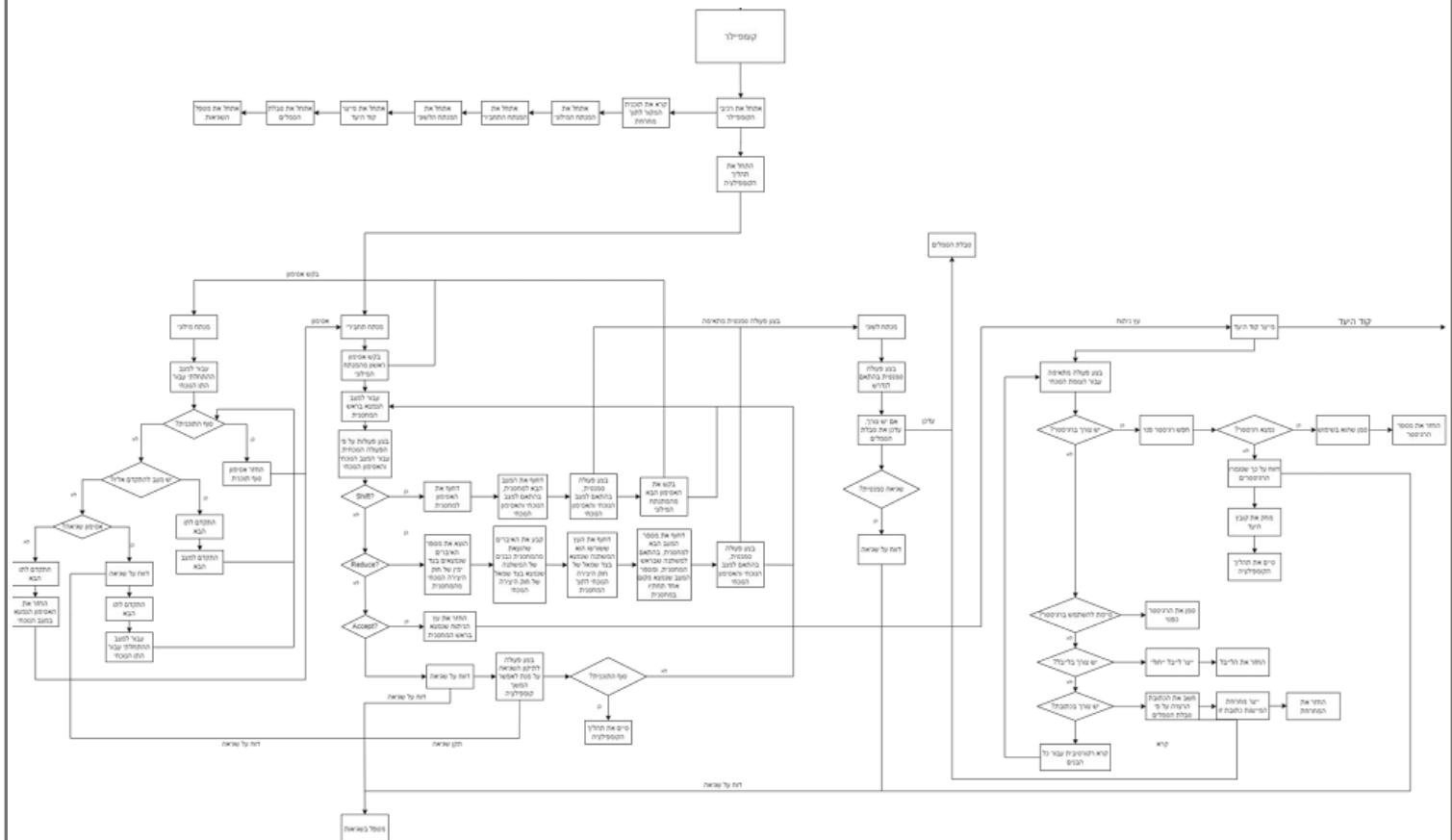


- Visual Studio 2019



– Masm32

אלגוריתם ראש

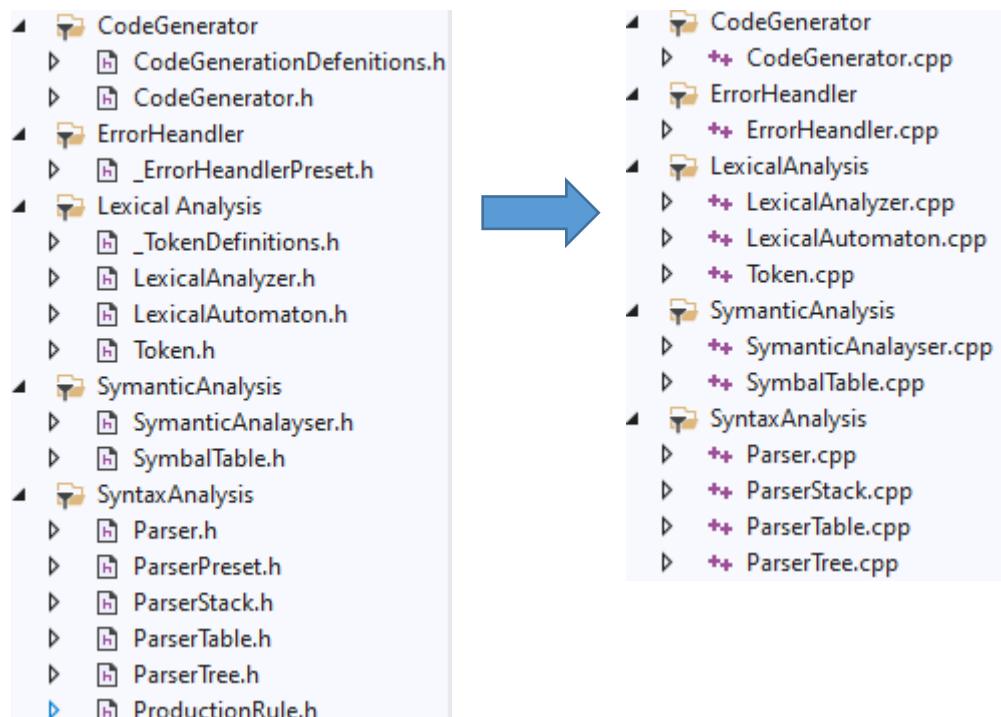


UML Class Diagram



מודולים ופונקציות ראשיות

היררכיה המודלים בפרויקט
תרשים של המחלקות והקוד בפרויקט.



Lexical Analysis

מטרת המודלים, להגדיר את מבנה המנתח הלקסיקלי כולל תת המבנים למשל Tokens.

Token

מודל על מבנה האסימון והשימוש בו.

```

class Token : public ParseNodeData {
public:
    std::string tokenValue;
    TokenType tokenType;
    size_t serialNumber;
    size_t line_number;

public:
    Token(const std::string& value, TokenType type, size_t serialNumber);

    const std::string& getValue() const;
    TokenType getType() const;
    size_t getSerialNumber() const;

    // Implementation of the print function from the base class
    void print() const override;
};

```

קבצים

Token.h

Token.cpp

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	פונקציה בונה	מצבייע לאובייקט שנוצר	מחוץות, סוג הnnen , מספר הגילוי	Token
O(1)	Getter בשבייל לדעת את סוג הnnen Token	סוג הnnen Token	לא	getType
O(1)	Getter בשבייל לדעת את המספר הסידורי	מספר הגילוי	לא	getSerialNumber
O(1)	מדפס את הnnen Token למסך	הדפסה למסך	לא	print

Lexer Automaton

מודל הקובע את מבנה האוטומט הסופי.

```

class LexicalAutomaton {
private:
    int currentState;
    std::vector<std::vector<int>> transitionMatrix;
    int maxTokenSize;
    std::unordered_map<TokenType, std::vector<int>> tokenStateMap;

public:
    int getCurrentState() const;
    void transition(char inputChar);
    void resetCurrentState();
    void setCurrentState(int state);
    void generateTransitionMatrix();
    const std::vector<std::vector<int>>& getTransitionMatrix() const;

private:
    void generateTokenStateMap();
};

```

קבצים
 LexicalAutomaton.h
 LexicalAutomaton.cpp

פונקציות

פונקציה	קלט	פלט	טיואר	יעילות
getCurrentState	לא	מספר שלם המיצג את המצב הנוכחי של האוטומט.	Getter בשבייל לדעת המצב הנוכחי.	O(1)
transition	תו מ�ור הקלט כדי להחליף את המצב	לא	פונקציה שנועדה לעדכן את המצב באוטומט	O(1)
resetCurrentState	לא	לא	פונקציה שנועדה לאפס את המצב של המטריצה ל 0	O(1)
setCurrentState	מספר שלם המיצג את המצב	לא	Setter בשבייל לעדכן את המצב הנוכחי	O(1)

			המצב המיועד.	
$O(n^2)$	פונקציה שיזכרת את המטריצה המייצגת אוטומט סופי מיועד.	לא	לא	generateTransitionMatrix
$O(1)$	פונקציה המחזירה את המטריצה המייצגת אוטומט סופי מיועד.	מטריצה דו ממדית	לא	getTransitionMatrix

Lexical Analyzer

מודל המיצג את האלגוריתם בעזרתו נאחד את כל המודלים האחרים לטובת המנתה הלקסיקלי.

```
class LexicalAnalyzer {
private:
    std::vector<Token*> recognizedTokens;
    std::string inputString;
    int inputIndex;
    LexicalAutomaton automaton;

public:
    LexicalAnalyzer
    (const std::string& input, const LexicalAutomaton& lexerAutomaton);

    void _BOOT_();
    Token* getNextToken();
    const std::vector<Token*>& getRecognizedTokens() const;
};

};
```

קבצים

LexicalAnalyzer.h

LexicalAnalyzer.cpp

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	פונקציה בונה שנוועדה ליציאת המנתה הלקסיקלי	כתובות המחלקה בצירוף	מחוזצת, אוטומט סופי	LexicalAnalyzer
O(1)	פונקציה שנוועדה לאותל את המנתה הלקסיקלי	לא	לא	_BOOT_
O(1)	פונקציה שנוועדה להחזיר את Token מהרצף האחרון מילים לאחריו שהיינו בו	כתובות של Token שנוצר מהמחוזצת שנקלטה בפונקציית הבנייה	לא	getNextToken
O(1)	Getter שנוועד להחזיר את התוצר של המנתה הלקסיקלי	ערך המקלט שנוצר	לא	getRecognizedTokens

Syntax Analysis Parser

מודל המיצג את האלגוריתם בעזרתו נאחד את כל המודלים האחרים לטובת המנתה התחברי.

```
class Parser {
private:
    ParserTable* parserTable;
    ParseTreeNodeStack* parserStack;
    std::vector<ProductionRule*> productionRules;
    void initProductionRule();

public:
    // Constructor to initialize Parser class members
    Parser(const std::string& gotoTable__, const std::string& actionTable__, ParseTreeNodeStack* stack);

    int syntaxAnalays(Token* tokenToParse);
    ParseTreeNode* extractTree();

    void printTree();
};
```

קבצים
Parser.h
Parser.cpp

פונקציות

פונקציה	קלט	כתובת	טיור	יעילות
Parser	מחוזת, מחוזת, כתובת של מחסנית המנתה התחברי	כתובת המחלקה בזיכרון	פונקציה בונה שנועדה ליציא את המנתה התחברי	O(1)
syntaxAnalays	Token מוצג האיסימונים מהמנתה הלוקסיקלי	מספר המצב	הfonקציה הראשית של המנתה התחברי שבעצם מקבלת Token ומחזירה את מספר המצב אותו קיבלנו.	O(n)
extractTree	לא	כתובת של עץ המנתה התחברי	פונקציה שונועדה להחזיר התוציר הסופי של המנתה התחברי	O(1)

Parser Stack

מודל המיצג את המחסנית של המנתה התחבירי.

```

class ParseTreeNodeStack {
private:
    std::stack<ParseTreeNode*> nodeStack;

public:
    ~ParseTreeNodeStack();
    // Push a new node onto the stack
    void push(ParseTreeNode* node);

    // Pop a node from the stack
    ParseTreeNode* pop();

    // Get the top node from the stack without removing it
    ParseTreeNode* top();

    // Check if the stack is empty
    bool isEmpty() const;
};

```

קבצים

ParserStack.h
ParserStack.cpp

פונקציות

פונקציה	קלט	פלט	תיאור	יעילות
push	כתובת של צומת תחביר	לא	פונקציה שנועדה לדוחף למחסנית ערך	O(1)
pop	לא	כתובת של צומת של עץ תחביר	פונקציה המחזירה את הערך בראש המחסנית ומוציאיה אותו	O(1)
top	לא	כתובת של עץ המנתה התחבירי	פונקציה שנועדה להחזיר התוצאת הסופי של המנתה התחבירי	O(1)

O(1)	פונקציה הבודקת האם המחסנית ריקה	תוצאה בוליאנית	לא	isEmpty
------	---------------------------------	----------------	----	---------

Parser Table

מודל המיצג את האוטומטים של המנתה התחבירי.

```

struct Action
{
    ActionType actionType;
    int toState;
};

class ParserTable {
private:
    std::unordered_map<int, std::unordered_map<TokenType, Action >> actionTable;
    std::unordered_map<int, std::unordered_map< NonTerminal, int >> gotoTable;

public:
    ParserTable(const std::string& gotoTable__, const std::string& actionTable__);
    Action* actionTableCheck(int currentState, const TokenType* tokenFromInput);
    int gotoTableCheck(int currentState, const NonTerminal* tokenFromStack);

private:
    void initActionTable(const std::string& fileName);
    void initGotoTable(const std::string& fileName);
};

```

קבצים

ParserTable.h

ParserTable.cpp

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(n)	פונקציה בונה של המחלקה, שסורקת את הקבצים ובונה את המטריצות הרצויות	כתובות של המחלקה בזיכרון	מחוזת, מחוזת,	ParserTable
O(1)	פונקציה שנועדה כדי לדעת מה הפעולה שנעשתה במצב הנוכחי והאטיון.	הפעולה אותה נבצע	מצב נוכחי, האטיון מהקלט	actionTableCheck
O(1)	פונקציה שנועדה כדי לדעת המצב שנדחף למחרנית לאחר reduce	ה מצב שנדחוף למחרנית	מצב נוכחי, האטיון מהקלט	gotoTableCheck
O(n)	אתחול המטריצה המתארת את האוטומט של ה Action Table	לא	מחוזת	initActionTable
O(n)	אתחול המטריצה המתארת את האוטומט של ה Goto Table	לא	מחוזת	initGotoTable

Parser Tree

מודל המיצג את העץ של המנתה התחבירי.

```

class ParseTreeNode : public TreeNode {
private:
    ParseNodeData* data;
    int stateNumber;
    std::vector<ParseTreeNode*> childrens;
    bool side;
    TokenType* type;
    int maxDepgh;
    int line_number;

public:
    int register_number;
    ParseTreeNode(ParseNodeData* nodeData, int stateNumber);
    TokenType* getType();
    void setType(TokenType t);
    int getMaxDepgh();
    int getMaxDepgh(ParseTreeNode* p);
    int get_line_number();
    int get_line_number(ParseTreeNode* p);
    ~ParseTreeNode();
    std::vector<ParseTreeNode*> getChildrens();
    int getStateNumber() const;
    void setStateNumber(int newState);
    ParseNodeData* getData() const;
    void addChild(ParseTreeNode* child);
    void PrintNode(int indentation = 0) const;
    void print(int depth = 0) const override;
};


```

קבצים

ParserTree.h

ParserTree.cpp

פונקציות

פונקציה	קלט	פלט	טיואר	יעילות	O(1)
ParseTreeNode	נתוני צומת (ParseNodeData*, int) מספר מצב	לא	קונסטרוקטו ר: יוצר אובייקט חדש עם נתוני צומת ומספר מצב		
getType	סוג טוקן (TokenType)	לא	מחזיר את סוג הטוקן של הצומת		O(1)
setType	סוג טוקן (TokenType)	לא	קובע את סוג הטוקן של הצומת		O(1)
getMaxDepth	צומת פרט (ParseTreeNode*)	(int) עומק מקסימלי	מחזיר את העומק המקסימלי של הצומת הננטונה		O(n^2)
getChildrens	ולא	וקטור של צמתים פרט (std::vector<ParseTreeNode*>)	מחזיר וקטור של הילדים של הצומת		O(1)
addChild	צומת ילך (ParseTreeNode*)	לא	מוסיף ילך חדש לצומת		O(1)

Production Rule
מודל המיצג החוקים של השפה

```
struct ProductionRule {  
    NonTerminal nonTerminalType;  
    int ruleLength;  
};
```

קבצים
ProductionRule.h

Symantic Analysis

Symantic Analayser

מודל המיצג את המנתח הלקוטיקלי.

```

class SymanticAnalaser {
private:
    // Functions for the Functions Map
    GenericSymbol* IdentifierInSymbolTable(SymbolTable& sTable, Token* token);
    GenericSymbol* FunctionInSymbolTable(SymbolTable& sTable, Token* token, std::vector<TokenType*>* passed_param, int state);
    bool FunctionName(SymbolTable& sTable, Token* token, std::vector<TokenType*>* passed_param, int state);
    TokenType* RETURN_TYPE(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool DECLARATION(ParseTreeNode& treeNode, SymbolTable& sTable);
    TokenType* F(ParseTreeNode& treeNode, SymbolTable& sTable, TokenType* emptyType);
    TokenType* POST_ONARY_ARITMATIC(ParseTreeNode& treeNode, SymbolTable& sTable);
    TokenType* PRE_ONARY_ARITMATIC(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool POST_ONARY_ARITMATICA(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool PRE_ONARY_ARITMATICA(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool VAL_CHANGE_TYPE(ParseTreeNode& treeNode, SymbolTable& sTable);
    //with return type for the expretion calc....
    TokenType* FUNC_CALL_CHECK(ParseTreeNode& treeNode, SymbolTable& sTable);
    //not with return type for the line stmt...
    bool FUNC_CALL(ParseTreeNode& treeNode, SymbolTable& sTable);
    void PASSED_PARAMS(ParseTreeNode& treeNode, SymbolTable& sTable, std::vector<TokenType*>* returnal);
    //חוליסט מה זה היה בסוף
    //הקיים יבצע בדיקה האם יש שרך אחד או לא וככה יחליט מהי הפעונקציה המתאימה ביזור לביטוי
    //במקורה והוא צריך להתאים את עצמו אז הוא יאפשר
    TokenType* COMPUTING_EXPRETION(ParseTreeNode& treeNode, SymbolTable& sTable, TokenType* emptyType);
    bool ASSIGN(ParseTreeNode& treeNode, SymbolTable& sTable);
    //return type;
    bool RETURN_FUNC(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool NONE_RETURN_FUNC(ParseTreeNode& treeNode, SymbolTable& sTable);
    SymbolTable* NONE_RETURN_FUNC_HEAD(ParseTreeNode& treeNode, SymbolTable& sTable);
    int CONSIDERING_SCOPE(ParseTreeNode& treeNode, SymbolTable& sTable, TokenType* save_type);
    bool BLOCK(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool STMT(ParseTreeNode& treeNode, SymbolTable& sTable);
    //NEED FIX
    bool CONTROL_STATEMENT(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool IF_ELSE_STATEMENT(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool WHILE_STATEMENT(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool FOR_STATEMENT(ParseTreeNode& treeNode, SymbolTable& sTable);
    bool FOR_EXP(ParseTreeNode& treeNode, SymbolTable& sTable);
    //NEED FIX
    bool CONTROL_EXP(ParseTreeNode& treeNode, SymbolTable& sTable);
    //NEED FIX
    bool LINE_STMT(ParseTreeNode& treeNode, SymbolTable& sTable);
    //it isnot the best and not bad
    bool RETURN_STMT(ParseTreeNode& treeNode, SymbolTable& sTable);
    SymbolTable* RETURN_FUNC_HEAD(ParseTreeNode& treeNode, SymbolTable& sTable);
    void PARAMS(ParseTreeNode& treeNode, SymbolTable& sTable, std::vector<TokenType*>* returnal);
private:
    ParseTreeNode* treeNodeToScan;
    SymbolTable* symbolTable;
    std::unordered_map<NonTerminal, NonTerminalFunction>* functionMap;
    //when entering to calcution_expretion
    std::stack<TokenType*>* return_type;
    Error_handler* err;
public:
    SymanticAnalaser(ParseTreeNode* treeNodeToScan, Error_handler* err);
    ~SymanticAnalaser();
    void setFunctionMap();
    void addFunctionToMap(NonTerminal nonTerminal, bool (SymanticAnalaser::* function)(ParseTreeNode&, SymbolTable&));
    bool Symantic_Aalisers();
    SymbolTable* extract_symbolTable();
private:
    bool Symantic_Aalysis(ParseTreeNode* ptner);
};


```

קבצים
 SymanticAnalayser.h
 SymanticAnalayser.cpp

פונקציות

פונקציה	קלט	פלט	תיאור	יעילות
IdentifierInSymbolTable	טבלת סמלים, טוקן	סמל גנרי (GenericSymbol*)	בודקת אם מזהה נמצאת בטבלת הסמלים ומחזירה את הסמל הגנרי המתאים	O(n)
FunctionInSymbolTable	וקטור סוג טוקנים, מספר מצב	טבלת סמלים, טוקן, סמל גנרי (GenericSymbol*)	בודקת אם פונקציה נמצאת בטבלת הסמלים עם הפרמטרים שנמסרו ומחזירה את הסמל הגנרי המתאים	O(n)
FunctionName	וקטור סוג טוקנים, מספר מצב	טבלת סמלים, טוקן, בוליאני (bool)	בודקת אם שם הפונקציה קיים בטבלת הסמלים ומחזירה נכון או לא נכון	O(n)
RETURN_TYPE	צומת עץ פרס, טבלת סמלים	סוג טוקן (TokenType*)	מחזירה את סוג ההחזרה של הצומת בעץ הפרס	O(n)

(n)0	בודקת אם יש הצהרה תקינה בצוותם בעז הפרס ומחזירה נכון או לא נכון	(bool) בוליאני	צומת עץ פרס, טבלת סמלים	DECLARATION
(n)0	בודקת וביצעת חישוב אריתמטי פוסט אונרי על הצומת בעז הפרס	TokenType*) סוג טוקן	צומת עץ פרס, טבלת סמלים	POST_ONARY_ARITMATIC
(n)0	בודקת וביצעת חישוב אריתמטי פורה אונרי על הצומת בעז הפרס	TokenType*) סוג טוקן	צומת עץ פרס, טבלת סמלים	PRE_ONARY_ARITMATIC
(n)0	בודקת קריית פונקציה בצוותם בעז הפרס ומחזירה נכון או לא נכון	(bool) בוליאני	צומת עץ פרס, טבלת סמלים	FUNC_CALL
(n)0	מחשבת ביטוי בצוותם בעז הפרס ומחזירה את סוג הטוקן	TokenType*) סוג טוקן	צומת עץ פרס, טבלת סמלים, סוג טוקן	COMPUTING_EXPRESSION
(n)0	בודקת השמה בצוותם בעז הפרס ומחזירה נכון או לא נכון	(bool) בוליאני	צומת עץ פרס, טבלת סמלים	ASSIGN

O(n)	בודקת את טווח המשתנים בצוותם בעץ הפרס ומחזירה נכון או לא נכון	צומת עץ פרס, (bool) בוליאני	טבלת סמלים, סוג טוקן	SCOPE
O(n)	בודקת הזרה-if (תנאי else) בצוותם בעץ הפרס ומחזירה נכון או לא נכון	צומת עץ פרס, (bool) בוליאני	טבלת סמלים	IF_ELSE_STATEMENT
O(n)	בודקת הזרה לולאת while בצוותם בעץ הפרס ומחזירה נכון או לא נכון	צומת עץ פרס, (bool) בוליאני	טבלת סמלים	WHILE_STATEMENT
O(n)	בודקת הזרה לולאת for בצוותם בעץ הפרס ומחזירה נכון או לא נכון	צומת עץ פרס, (bool) בוליאני	טבלת סמלים	FOR_STATEMENT
O(n)	בודקת הזרה החזרה בצוותם בעץ הפרס ומחזירה	צומת עץ פרס, (bool) בוליאני	טבלת סמלים	RETURN_STMT

		נכון או לא נכון			
O(1)	קונסטרוקטו ר: יוצר אובייקט חדש של מנתח סמנטי	לא	צומת עץ פרס, מנהל שגיאות	SymanticAnal aser	
O(n)	מגדיר את מפת הֆונקציות	לא	לא	setFunctionM ap	
O(n)	מבצע ניתוח סמנטי ומחזירה נכון או לא נכון	(bool) בוליאני	לא	Symantic_Aali sers	

Symbol Table

מודל המציג את טבלת הסמלים של המנתח התחביבי.

```

class SymbolTable {
public:
    std::vector<GenericSymbol*>* tokensInScope;
    int currentScope;
    std::vector<SymbolTable*>* lowerScops;
    SymbolTable* parent;
    int used_space;
    int param_known;
    int found;
    int scope_found;
    int number_of_identifiers;
    int function_number;
    SymbolTable();
    int calculateMatchAccuracy(Function* func, std::vector<TokenType*>* passed_params);
    int getIdNum();
    bool IsNameTaken(Token* token);
    bool IsNameTakenInLowerScopes(Token* tokenizer, int scope_level);
    GenericSymbol* searchFunction(Token* token, std::vector<TokenType*>* passed_params);
    GenericSymbol* searchIdentifier(const std::string& str);
    void addToken(GenericSymbol* name);
    SymbolTable* newScope();
    GenericSymbol* searchFunctionInLowerScopes(Token* tokenizer, int scope_level, std::vector<TokenType*>* passed_params);
    GenericSymbol* searchIdentifierInLowerScopes(const std::string& str, int scope_level, int fn);
};


```

קבצים
SymbolTable.h
SymbolTable.cpp

פונקציות

פונקציה	קלט	פלט	תיאור	יעילות
SymbolTable	לא	לא	קונסטרוקטו ר: יוצר אובייקט חדש של טבלת סמלים	O(1)
calculateMatchAccuracy	פונקציה (Function*, std::vector<TokenType*>*) סוגי טוקנים (std::vector<TokenType*>*)	(int) מספר שלם	מחשב את מידת ההתאמה של הפונקציה לפי הפרמטרים שנמסרו	O(n)

O(n)	בודק אם השם כבר תפוס בטבלת הסמלים	bool בוליאני	Token* טוקן (Token*)	IsNameTaken
O(n)	מחזיר את העומק המקסימלי של הצומת הנowntה	GenericSymbol* (GenericSymbol*) סמל גנרי	Token*, וקטור סוג טוקנים std::vector<TokenType*> (std::vector<TokenType*>)	searchFunction
O(n)	מחפש מזהה בטבלת הסמלים לפי שם ומחזירה את הסמל המתאים	GenericSymbol* (GenericSymbol*) סמל גנרי	const std::string& מחרוזת	searchIdentifier
O(1)	يוצר בטבלת סמלים חדשה לטווח חדש ליד חדש לצומת	טבלת סמלים (SymbolTable*)	לא	newScope

Code Generator

Register

מודל המיצג את הרגיסטר שמחזיק במידע עלייו ושימוש.

```
class Register {
public:
    std::string BIT32;
    std::string HBIT8;
    std::string LBIT8;
    bool BIT8;
    bool ARITMATIC;
    bool inuse;

public:
    Register(std::string BIT32, std::string HBIT8, std::string LBIT8, bool BIT8, bool ARITMATIC);
    std::string* GetName();
    bool InUse();
    void EndInUse();
    void SetInUse();
};

};
```

קבצים

CodeGenerator.h

CodeGenerator.cpp

פונקציות

פונקציה	קלט	פלט	תיאור	יעילות
Register	מחוזות: שם הרגיסטר 32 ביט, שם הרגיסטר 8 ביט גובה, שם הרגיסטר 8 ביט נמור, בוליאנים: האם הוא 8 ביט, האם הוא אריתמטי	לא	קונסטרוקטו ר: יוצר אובייקט חדש של רגיסטר עם הנתונים שנמסרו	O(1)
GetName	לא	מצבייע למחוזות (std::string*)	מחזיר את שם הרגיסטר 32 ביט	O(1)

O(1)	מחזיר נכון אם הרגיסטר בשימוש	(bool) בוליאני	לא	InUse
O(1)	מסמן שהרגיסטר אינו בשימוש	לא	לא	EndInUse
O(1)	מסמן שהרגיסטר בשימוש	לא	לא	SetInUse

Code Generator

מודל המיצג את יוצר הקוד.

```
class Code_Generator {  
private:  
    std::vector<Register*>* registers;  
    FILE* destFile;  
    ParseTreeNode* ast;  
    SymbolTable* symbolTable;  
    int functions_wrote;  
    int temp;  
public:  
    Code_Generator(ParseTreeNode* ast, SymbolTable* symbolTable);  
    bool init(std::string* file_name);  
    void push_all(int number);  
    void setup_sp_down_by_num(SymbolTable* sTable, int number);  
    void update_sp_up_by_number(SymbolTable* sTable, int number);  
    void update_sp_down(SymbolTable* sTable);  
    void setup_sp_down(SymbolTable* sTable, bool func);  
    void update_sp_up(SymbolTable* sTable);  
    void pop_all(int number);  
    int register_alloc(bool low, bool aritmatic);  
    void register_free(int r);  
    std::string* register_name(int r, bool size);  
    std::string* label_create();  
    std::string symbol_address(GenericSymbol* entry);  
    void output(const char* format, ...);  
    void output_data_segment();  
    void generate();  
    void generate_code(ParseTreeNode* treeNode);  
    void generate_function(ParseTreeNode* treeNode);  
    int decl_params(Function* f, SymbolTable* sTable);
```

```

void generate_scope(ParseTreeNode* treeNode, SymbolTable* sTable);
int generate_return(ParseTreeNode& treeNode, SymbolTable& sTable);
void generate_scope_stmt(ParseTreeNode* treeNode, SymbolTable* sTable);
void CONTROL_STATEMENT(ParseTreeNode& treeNode, SymbolTable& sTable);
void gen_IF_ELSE_STATEMENT(ParseTreeNode& treeNode, SymbolTable& sTable);
void gen_IF_STATEMENT(ParseTreeNode& treeNode, SymbolTable& sTable);
void gen_WHILE_STATEMENT(ParseTreeNode& treeNode, SymbolTable& sTable);
void gen_FOR_STATEMENT(ParseTreeNode& treeNode, SymbolTable& sTable);
int FOR_EXP(ParseTreeNode& treeNode, SymbolTable& sTable);
int CONTROL_EXP(ParseTreeNode& treeNode, SymbolTable& sTable);
void generate_line_stmt(ParseTreeNode* treeNode, SymbolTable* sTable);
int VAL_CHANGE_TYPE(ParseTreeNode& treeNode, SymbolTable& sTable);
void ASSIGN(ParseTreeNode& treeNode, SymbolTable& sTable);
GenericSymbol* DECLARATION(ParseTreeNode& treeNode, SymbolTable& sTable);
void PASSED_PARAMS(ParseTreeNode& treeNode, SymbolTable& sTable, std::vector<TokenType*>* returnal);
int FUNC_CALL(ParseTreeNode& treeNode, SymbolTable& sTable, int reg_number);
GenericSymbol* GetFunction(SymbolTable& sTable, Token* token, std::vector<TokenType*>* passed_param);
GenericSymbol* GetIdentifier(SymbolTable& sTable, Token* token);
int COMPUTING_EXPERSION(ParseTreeNode& treeNode, SymbolTable& sTable);
int generate_bool_expretn(ParseTreeNode* treeNode, SymbolTable* sTable);
int generate_bool_cmp(ParseTreeNode* treeNode, SymbolTable* sTable);
int generate_expr(ParseTreeNode* treeNode, SymbolTable* sTable);
void generate_not(int reg);
int generate_f(ParseTreeNode* treeNode, SymbolTable* sTable);
int POST_ONARY_ARITMATICA(ParseTreeNode& treeNode, SymbolTable& sTable);
int PRE_ONARY_ARITMATICA(ParseTreeNode& treeNode, SymbolTable& sTable);
int generate_aritmatic(ParseTreeNode* treeNode, SymbolTable* sTable);

void generate_mul(int reg_one, int reg_two);
void generate_div(int reg_one, int reg_two);
void generate_mod(int reg_one, int reg_two);
void generate_add(int reg_one, int reg_two);
void generate_sub(int reg_one, int reg_two);
void generate_push(int reg);
void generate_pop(int reg);
void generate_call(std::string function_name);
void code_generator_decl(SymbolTable* sTable);
void code_generator_mov_literal(int register_number, Token* token);
void code_generator_mov_identifier(int register_number, GenericSymbol* identifier, TokenType type);
void code_generator_mov_identifier_back(int register_number, GenericSymbol* identifier, TokenType type);
};

ביבטום
CodeGenerator.h
CodeGenerator.cpp

```

פונקציות

פונקציה	קלט	פלט	תיאור	יעילות
Code_Generator	עツ תחבירי (ParseTreeNode*), טבלת סמלים (SymbolTable*)	לא	קונסטרוקטו ר: יוצר אובייקט חדש של מחולק קוד עם עץ	O(1)

	תחכירים וטבלת סמלים			
O(1)	מאתחל את מחולל הקוד עם שם קבוע היעד	bool (bool) בוליאני	מחרוזת (std::string*)	init
O(n)	מייצר את הקוד המבוסס על עצם התחכيري	לא	לא	generate
O(n)	מייצר קוד עבור צומת מסוים בעץ התחכيري	לא	צומת עצם התחכiri (ParseTreeNode*)	generate_code
O(n)	מקצתה רגיסטר בהתאם לפרמטרים שנמסרו ומחזיר את מספר הרגיסטר שהוקצתה	(int) מספר רגיסטר	בוליינים: האם הרגיסטר נמור, האם הרגיסטר אריתמטי	register_alloc
O(1)	משחרר את הרגיסטר שהוקצתה	לא	(int) מספר רגיסטר	register_free
O(1)	מחזיר את הכתובת של הסמל	std::string) מחרוזת	סמל גנרי (GenericSymbol*)	symbol_addresses
O(n)	כותב את הפלט לקובץ היעד בהתאם למבנה ולפרמטרים שנמסרו	לא	מחרוזת תבנית (const char*), פרמטרים נוספים	output

O(n)	מייצר קוד עבור פונקציה מסוימת בעץ תחבירי	לא	צומת עץ תחבירי (ParseTreeNode*)	generate_function
O(n)	מייצר קוד עבור תחום מסוים בעץ תחבירי	לא	צומת עץ תחבירי (ParseTreeNode*), טבלת סמלים (SymbolTable*)	generate_scope

Error Handler**Error Handler**

מודל המציג את הריגיסטר שמחזיק במידע עליון ושימוש.

```

class Error_heandler {
    Error_Type* type_error;
    std::string* file_name;
public:
    Error_heandler(std::string* file_name);
    void error_handler_report(int line, Error_Type error_type);
    const char* error_handler_error_to_str(Error_Type error_type);
    void custume_error_handler_report(int line, Error_Type error_type, const char* format);
private:
    std::string getLineFromFile(const std::string& filename, int lineNumber);
    void error_handler_report(int line, Error_Type error_type, const char* format, ...);
};

```

קבצים
ErrorHeandlerPreset.h
ErrorHeandler.cpp

פונקציות

יעילות	תיאור	פלט	קלט	פונקציה
O(1)	קונסטרוקטו ר: יוצר אובייקט חדש של מנהל שגיאות עם שם קבוע	לא	מחרוזת (std::string*)	Error_heandler
O(n)	מדוח על שגיאה בקובץ בשורה מסוימת	לא	(int) מספר שורה, סוג שגיאה (Error_Type)	error_handler_report
O(1)	מכיר סוג שגיאה למחוזות	(const char*) מצביע לטו	סוג שגיאה (Error_Type)	error_handler_error_to_str
O(n)	מדוח על שגיאה מותאמת אישית בקובץ בשורה מסוימת בהתאם לתבנית	לא	(int) מספר שורה, סוג שגיאה (Error_Type), מחרוזת התבנית (const char*)	custume_error_handler_report
O(n)	מחזיר את השורה המבוקשת מקובץ	מחרוזת (std::string)	(const std::string&), (int) מספר שורה	getLineFromFile
O(n)	מדוח על שגיאה בקובץ בשורה מסוימת בהתאם לתבנית ולפרמטרים נוספים	לא	(int) מספר שורה, סוג שגיאה (Error_Type), מחרוזת התבנית (const char*), פרמטרים נוספים	error_handler_report

התוכנית הראשית

אלגוריתם התוכנית הראשית (main)

זהו אלגוריתם אשר מאגד את כל המודלים בקומpileר ויוצר תוכנה שפועלת במטרה אחת, לדוגמה:

1. האם לא קיבלנו מספיק פרמטרים 2 שמות של קבצים.
2. הדפס למסך "for the compiler pass end-to-end files"
3. אחרת
4. הכנס לתוך המשתנה `input_file_name` את שם הקובץ הראשון
5. צור משתנה `Er` שיטפל במקרי שגיאה
6. צור מצביע לקובץ `inputFile`
7. אם הקובץ לא נפתח בהצלחה, יודפס הודעה שגיאה והתוכנית תחזיר עם סטטוס שגיאה.
8. הכנס למשתנה `input` את התוכן בקובץ הקלט
9. צור אוטומט והכנס אותו למנתח לסקיל
10. אתחל מנתח תחבירי
11. קבל את הטוקן הבא מטורק קובץ הקלט ועבورو בצע את הנימוחות התחבירי
12. אם הטוקן הוא רווח או שורה חדשה התעלם
13. אם המנתח התחבירי החזר מצב 0
14. הדפס שגיאה למסך.
15. החזר 0.
16. צור מנתח סמנטי
17. בדוק האם המנתח הסמנטי עבר בהצלחה אם כן
18. צור כותב קוד
19. כתוב את הקוד
20. נקה את כל הזיכרון החופשי שהוקצה במהלך התוכנית.
21. החזר 1

מדריך משתמש

שימוש

כיצד ניתן להשתמש בקומpileר כדי לкомפל קוד בשפת c לשפת assembly 32 bit בהנחה והכל על המחשב

```
int power(int num) {
    if(num == 0)
    {
        return 1;
    }
    return num*power(num-1);
}

void print_num(int number){
    if(number != 0){
        print_num(number/10);
        print_c(number%10 + '0');
    }
}

void triangle(int size){
    size--;
    for(int i = 0; i <= size; i++){
        for(int z = 0; z <= size-i; z++){
            print_c(' ');
        }
        for(int z = 0; z <=i; z++){
            print_c('*');
        }
        for(int z = 1; z <=i; z++){
            print_c('*');
        }
        print_c(13);
        print_c(10);
    }
}
```

נכונות תוכנת C :
התוכנית מבקשת מספר ובודקת אותו בעצרת
ומדפיסה משולש בגובה המספר בעצרת...

```
void main(){
    print_c('h');
    print_c('i');
    print_c(' ');
    print_c('e');
    print_c('n');
    print_c('t');
    print_c('e');
    print_c('r');
    print_c(' ');
    print_c('n');
    print_c('u');
    print_c('m');
    print_c('b');
    print_c('e');
    print_c('r');
    print_c(':');

    int g = power(get_c());
    print_c(13);
    print_c(10);
    triangle(g);
}
```

>CompilerScript.bat test.txt final.asm : command line בelow

התוצאות:

```
C:\Users\User\OneDrive\שולחן העבודה\asm 32>CompilerScript.bat test.txt final.asm
compiling stage:

[Lexical Analysis      -> succsed]
[Syntax Analysis       -> succsed]
[Symantic Analysis     -> succsed]
[Generating             -> succsed]

Cleaning up memory....
->      succsed
assembling stage:
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

Assembling: final.asm

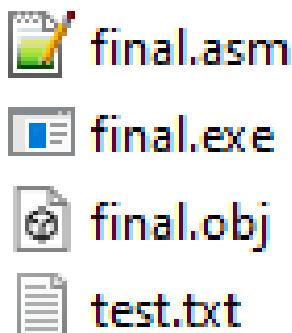
*****
ASCII build
*****

linking stage:
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

running executable - final.exe...
hi enter number:3

*
 ***
 ****
 *****
```

למעשה התוכנית יצרת שלושה קבצים , קובץ האסמבלר, קובץ קוד המכונה (.obj) ותוכנת ה .exe



ביבליוגרפיה

Atwood, J. & Spolsky, J. (2008, Sep). **Stack Overflow**. Retrieved from <https://stackoverflow.com>

Bergman, S. D. (2017, May). **Compiler Design**. Retrieved from <https://bit.ly/3vkW5so>

Dancis, K., & Schoeffel, S. (2017, July). **Whimsical**. Retrieved from <https://whimsical.com>

Darveshi, R. S. (2011, November). **E.C.E**. Retrieved from Youtube <https://bit.ly/3vhk7R0>

Grandinetti, P. (2019, October). **Compilers**. Retrieved from grandinetti <https://bit.ly/3MpMEtm>

Hoyt, B. (2021, March). **Implement Hash Table in C**. Retrieved from <https://bit.ly/3uA00xs>

Hsu, T. S. (1996, Fall). **Symbol Table**. Retrieved from Academia Sinica <https://bit.ly/3klSkxA>

Sanger, L. (2001, January). **Wikipedia**. Retrieved from <https://www.wikipedia.org>

Simonson, S. (2010, May). **Automata Theory**. Retrieved from Youtube <https://bit.ly/3HfEiJR>

Simonson, S. (2010, May). **Theory of Computation**. Retrieved from Youtube <https://bit.ly/3pGrM7g>

Simonson, S. (2017, January). **Algorithms**. Retrieved from Youtube <https://bit.ly/3q0SDzV>

Soshnikov, D. (2021, April). **Building a Parser**. Retrieved from Youtube <https://bit.ly/3IRKJM8>

Thain, D. (2017, Fall). **Code Generation**. Retrieved from Youtube <https://bit.ly/3HZLb9T>

Tutorialspoint. (2015). **Learn Compiler Design**. Retrieved from Tutorialspoint: <https://bit.ly/35pWuLi>

Uncode. (2013, July). **Uncode**. Retrieved from Youtube <https://bit.ly/3vGe9c4>