**COMPILER for**

**MyC Programming Language**

**עבודת גמר**

**לקבלת תואר טכנאי תוכנה**

**מגיש** : **אהד קידרון**

**ת.ז.**: **216268516**

**מנחים** : **מיכאל צ'רנובילסקי**

תמונה שמכילה גרפיקה, לוגו, עיצוב גרפי, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.**אפריל  2025 תשפ"ה**

תוכן עניינים

[תקציר 9](#_Toc196597879)

[מושגים 10](#_Toc196597880)

[שפות תכנות 10](#_Toc196597881)

[קומפיילר - Compiler 10](#_Toc196597882)

[ביטוי רגולרי – Regular expressions 10](#_Toc196597883)

[אוטומט – Automata 10](#_Toc196597884)

[דקדוק חסר הקשר – Context Free Grammar 10](#_Toc196597885)

[Production Rule 10](#_Toc196597886)

[Derivation 11](#_Toc196597887)

[תיאור הפרויקט 12](#_Toc196597888)

[הקדמה 12](#_Toc196597889)

[אבני השפה - MyC 13](#_Toc196597890)

[קבועים -Constants 13](#_Toc196597891)

[סוגי נתונים -Data Types 13](#_Toc196597892)

[מילות מפתח - Keywords 13](#_Toc196597893)

[אופרטורים - Operators 14](#_Toc196597894)

[משתנים - Variables 14](#_Toc196597895)

[תכולת השפה 15](#_Toc196597896)

[הגדרת משתנה - Declaration 15](#_Toc196597897)

[השמת ערך למשתנה - Assignment 15](#_Toc196597898)

[אמת/שקר – True/False 15](#_Toc196597899)

[תנאים - Conditions 16](#_Toc196597900)

[לולאות - loops 16](#_Toc196597901)

[הגדרת פונקציה 17](#_Toc196597902)

[קריאה לפונקציה 17](#_Toc196597903)

[דקדוק השפה 18](#_Toc196597904)

[Tokens 18](#_Toc196597905)

[תחביר השפה 18](#_Toc196597906)

[רקע תיאורטי 20](#_Toc196597907)

[מהו קומפיילר? 20](#_Toc196597908)

[סוגי מעבדי שפות 20](#_Toc196597909)

[מתורגמן - Interpreter 20](#_Toc196597910)

[מהדר - Compiler 21](#_Toc196597911)

[- Compiler && Interpreterיתרונות וחסרונות 21](#_Toc196597912)

[תהליך הקומפילציה 22](#_Toc196597913)

[עיבוד מקדים(preprocessing) 22](#_Toc196597914)

[ניתוח מילוני(Lexical analysis) 22](#_Toc196597915)

[ניתוח תחביר(syntax analysis) 22](#_Toc196597916)

[ניתוח סמנטי( ( Semantic Analysis 22](#_Toc196597917)

[יצירת קוד ביניים( (Intermediate Code Generation 22](#_Toc196597918)

[אופטימיזציה(code optimization) 23](#_Toc196597919)

[יצירת הקוד(Code Generation) 23](#_Toc196597920)

[Front End/Back End 23](#_Toc196597921)

[תורת האוטומטים 24](#_Toc196597922)

[תורת האוטומטים-מבוא 24](#_Toc196597923)

[אוטומט סופי דטרמיניסטי-DFA 24](#_Toc196597924)

[אוטומט מחסנית – PDA 25](#_Toc196597925)

[ייצוג אוטומט על ידי מטריצה 25](#_Toc196597926)

[שפת היעד - אסמבלי nasm x64 27](#_Toc196597927)

[תנאים 28](#_Toc196597928)

[לולאות 28](#_Toc196597929)

[פונקציות 28](#_Toc196597930)

[קריאה לפונקציות 28](#_Toc196597931)

[תיאור הבעיה האלגוריתמית 29](#_Toc196597932)

[ניתוח מילוני Lexical Analysis – 29](#_Toc196597933)

[ניתוח תחבירי – Syntax Analysis 29](#_Toc196597934)

[ניתוח סמנטי – Semantic Analysis 29](#_Toc196597935)

[סקירת אלגוריתמים בתחום הבעיה 30](#_Toc196597936)

[Parser Categories 30](#_Toc196597937)

[ניתוח מלמעלה למטה – Top Down 30](#_Toc196597938)

[ניתוח מלמטה למעלה – Bottom Up 30](#_Toc196597939)

[Left-most derivation 30](#_Toc196597940)

[Right Most Derivation 31](#_Toc196597941)

[Left - > Right 31](#_Toc196597942)

[LL(1) 31](#_Toc196597943)

[LR(1) 31](#_Toc196597944)

[SLR(1) 31](#_Toc196597945)

[Parsing Problems 32](#_Toc196597946)

[Left recursion 32](#_Toc196597947)

[Left Factoring 33](#_Toc196597948)

[Shift/Reduce Conflict 33](#_Toc196597949)

[Parsing algorithms 33](#_Toc196597950)

[עקיבה לאחור – Back Tracking 33](#_Toc196597951)

[Recursive decent 34](#_Toc196597952)

[Predictive Parser 34](#_Toc196597953)

[Shift Reduce Parser 35](#_Toc196597954)

[אסטרטגיה 36](#_Toc196597955)

[ניתוח מילוני – Lexical Analysis 36](#_Toc196597956)

[מכונת המצבים 37](#_Toc196597957)

[אלגוריתם לזיהוי אסימון 37](#_Toc196597958)

[פסאודו קוד 38](#_Toc196597959)

[ניתוח תחבירי – Syntax Analysis 38](#_Toc196597960)

[Shift, Reduce, Accept 39](#_Toc196597961)

[Parsing Table 40](#_Toc196597962)

[עץ הניתוח 41](#_Toc196597963)

[מחסנית - Stack 42](#_Toc196597964)

[אלגוריתם ניתוח התחבירי 43](#_Toc196597965)

[פסאודו קוד 43](#_Toc196597966)

[MyC Grammar 44](#_Toc196597967)

[Syntax diagram 45](#_Toc196597968)

[ניתוח סמנטי – Semantic Analysis 51](#_Toc196597969)

[שגיאות סמנטיות 51](#_Toc196597970)

[Symbol Table 53](#_Toc196597971)

[Scope 53](#_Toc196597972)

[אסטרטגיה בשלב הסמנטי 54](#_Toc196597973)

[יצירת הקוד – Code Generation 56](#_Toc196597974)

[שימוש באוגרים 56](#_Toc196597975)

[אסטרטגיה ב-Code Generation 56](#_Toc196597976)

[Error handling 58](#_Toc196597977)

[Error handling – Lexical 58](#_Toc196597978)

[Error handling – Parsing 59](#_Toc196597979)

[Error handling - Semantic 61](#_Toc196597980)

[ארכיטקטורה - Top-down Level Design 62](#_Toc196597981)

[Lexer 62](#_Toc196597982)

[Parser 63](#_Toc196597983)

[Semantic 64](#_Toc196597984)

[Code Generation 64](#_Toc196597985)

[UML Use Cases 66](#_Toc196597986)

[מבנה נתונים 67](#_Toc196597987)

[מבנה הנתונים בLexer- 67](#_Toc196597988)

[מבנה הנתונים ב-Parser 68](#_Toc196597989)

[Parsing Table 68](#_Toc196597990)

[Stack 69](#_Toc196597991)

[Parsing tree 70](#_Toc196597992)

[מבנה הנתונים ב-Semantic 71](#_Toc196597993)

[Symbol Table 71](#_Toc196597994)

[Scope Tree 71](#_Toc196597995)

[תיאור סביבת העבודה ושפות תכנות 72](#_Toc196597996)

[סביבת העבודה: 72](#_Toc196597997)

[Operating System 72](#_Toc196597998)

[Code Editor 72](#_Toc196597999)

[C Compiler 72](#_Toc196598000)

[שפות תכנות 72](#_Toc196598001)

[אלגוריתם ראשי 73](#_Toc196598002)

[האלגוריתם הראשי - Lexer 73](#_Toc196598003)

[האלגוריתם הראשי – Parser 73](#_Toc196598004)

[UML Class Diagram 74](#_Toc196598005)

[76](#_Toc196598006)

[78](#_Toc196598007)

[תיאור המחלקות הראשיות בפרויקט 79](#_Toc196598008)

[Token 79](#_Toc196598009)

[Lexer\_dfa 80](#_Toc196598010)

[struct 80](#_Toc196598011)

[פונקציות 80](#_Toc196598012)

[Lexer 81](#_Toc196598013)

[פונקציות 81](#_Toc196598014)

[ParserTable 82](#_Toc196598015)

[struct 82](#_Toc196598016)

[פונקציות 83](#_Toc196598017)

[ParserTree 83](#_Toc196598018)

[struct 83](#_Toc196598019)

[פונקציות 84](#_Toc196598020)

[ParserStack 85](#_Toc196598021)

[Struct 85](#_Toc196598022)

[פונקציות 85](#_Toc196598023)

[Parser 86](#_Toc196598024)

[פונקציות 86](#_Toc196598025)

[Symbol Table 87](#_Toc196598026)

[struct 87](#_Toc196598027)

[פונקציות 88](#_Toc196598028)

[Scope 89](#_Toc196598029)

[struct 89](#_Toc196598030)

[פונקציות 89](#_Toc196598031)

[ScopeTree 90](#_Toc196598032)

[פונקציות 90](#_Toc196598033)

[Code Generator 91](#_Toc196598034)

[Struct 91](#_Toc196598035)

[פונקציות 91](#_Toc196598036)

[ErrorReporting 93](#_Toc196598037)

[פונקציות 93](#_Toc196598038)

[התוכנית הראשית 94](#_Toc196598039)

[מדריך למשתמש 95](#_Toc196598040)

[סיכום אישי 97](#_Toc196598041)

[רפלקציה אישית 97](#_Toc196598042)

[ביבליוגרפיה 99](#_Toc196598043)

[נספח - קוד הפרויקט 100](#_Toc196598044)

[Token 100](#_Toc196598045)

[Token.h 100](#_Toc196598046)

[Token.c 102](#_Toc196598047)

[Lexer 104](#_Toc196598048)

[Lexer\_dfa.h 104](#_Toc196598049)

[Lexer\_dfa.c 106](#_Toc196598050)

[Lexer.h 116](#_Toc196598051)

[Lexer.c 117](#_Toc196598052)

[Parser 132](#_Toc196598053)

[ParserStack.h 132](#_Toc196598054)

[ParserStack.c 132](#_Toc196598055)

[ParserTable.h 134](#_Toc196598056)

[ParserTable.c 134](#_Toc196598057)

[ParserTree.h 174](#_Toc196598058)

[ParserTree.c 174](#_Toc196598059)

[Parser.h 175](#_Toc196598060)

[Parser.c 180](#_Toc196598061)

[Semantic 196](#_Toc196598062)

[Scope.h 196](#_Toc196598063)

[Scope.c 197](#_Toc196598064)

[SymbolTable.h 202](#_Toc196598065)

[SymbolTable.c 205](#_Toc196598066)

[ScopeTree.h 213](#_Toc196598067)

[ScopeTree.c 215](#_Toc196598068)

[Code Generator 255](#_Toc196598069)

[CodeGenerator\_Base.h 255](#_Toc196598070)

[CodeGenerator.h 257](#_Toc196598071)

[CodeGenerator.c 259](#_Toc196598072)

[ErrorReporting 294](#_Toc196598073)

[ErrorReporting.h 294](#_Toc196598074)

[ErrorReporting.c 296](#_Toc196598075)

[Compiler.h 298](#_Toc196598076)

[Compiler.c 299](#_Toc196598077)

# תקציר

מאז ומתמיד אני זוכר את עצמי כילד סקרן, מאז שנכנסתי אל תחום המחשבים, מאוד עניין אותי הפער בין התוכנה לחומרה. מחשבה זאת נשארה בתור מעיין BlackBox מסתורי שבמשך הרבה מאוד זמן פשוט קבלתי את זה כמו שזה.

אני אדם שאוהב לאתגר את עצמו. כאשר המורה שלי שיבח את נושא פרויקט זה ואמר "זה פרויקט לתותחים" ישר נדלקתי על הרעיון, ונדהמתי לגלות כמה פרויקט זה יתרום לי מקצועית וירחיב לי את הידע.

בפרויקט זה ארצה להפיק את המירב. ארצה להעשיר את הידע המקצועי שלי ולרכוש כלים חדשים.  
ארצה להבין לעומק את תהליך הקומפילציה. להתמודד עם פיתוח אלגוריתמים השונים ויעילותם, וכמו כן תכנון והנדוס מבני הנתונים.

ספר פרויקט זה מסקר ומסביר את כל שלבי תהליך הקומפילציה. הוא מתאים לאנשים אשר רוצים גם הם לבנות שפה משלהם, ולתרגם אותה לשפת מכונה, ולעזר בספר זה בתור מדריך. אך כמו כן ספר זה מתאים גם לאנשים סקרנים אשר רוצים להעשיר את הידע שלהם בתחום הבעיה.

קריאה מהנה 😊

# מושגים

אתחיל בליישר קו איתכם - הקוראים, ואסביר על המושגים אשר ילוו אותנו במשך כל הספר, אשר כדאי שתכירו.

שפות תכנות  
שפת תכנות היא אוסף של סימנים ומשפטים אשר משמשת להגדרת פעולות שהמחשב יבצע. כל פעולה העומדת בפני עצמה, מבצעת משהו קבוע, אך כאשר מחברים מספר פעולות בצורה מסוימת, האינטגרציה שלהם יכולה לממש אלגוריתמים שונים.  
כיום יש מספר רב של שפות תכנות, כאשר לכל שפה יש את הדקדוק והסמנטיקה שלה.

## קומפיילר - Compiler

קומפיילר הוא תוכנה שבמחשב מתרגמת קוד כתוב בשפה גבוהה (source language) לשפה נמוכה (target language) כדי ליצור תוכנית ניתנת להרצה.  
בזירה המעשית, הקומפיילר עובר סדרת שלבים הכוללת ניתוח לקסיקלי, ניתוח תחבירי, ניתוח סמנטי, יצירת קוד ביניים, אופטימיזציה ויצירת קוד סופי.

## ביטוי רגולרי – Regular expressions

ביטוי רגולרי (regex) הוא רצף תווים המגדיר תבנית לחיפוש או התאמה בתוך מחרוזת טקסט. ביטויים רגולריים נמצאים בשימוש במגוון יישומים — מנועי חיפוש, כלי עיבוד טקסט ועד חלק מניתוח לקסיקלי בסורקי טוקנים בקומפיילרים.

## אוטומט – Automata

אוטומט הוא מודל חישובי מופשט המשמש לתיאור ולניתוח של שפות פורמליות באמצעות “מכונה” בעלת מצבים ומעבר בין מצבים לפי סמל קלט.  
אוטומטים סופיים (Finite Automata) הם מכונות עם קבוצה סופית של מצבים המזהות שפות רגולריות על ידי מעבר בין מצבים בהתאם לסמל הנקלט.

## דקדוק חסר הקשר – Context Free Grammar

דקדוק חסר-הקשר (CFG) הוא מערכת חוקים פורמלית שבה כל כלל מחליף סימון לא-טרמינלי אחד במחרוזת של טרמינלים ולא-טרמינלים, ללא תלות בהקשרו.  
כל כלל בדקדוק חסר-הקשר נכתב בצורת A → α, כאשר A הוא סימון לא-טרמינלי ו־α רצף של טרמינלים ולא-טרמינלים.  
דקדוקים אלה מהווים את הבסיס לאלגוריתמים לניתוח תחבירי בקומפיילר כדי לפענח ולהבין את מבנה התוכנית.

### Production Rule

כאשר נגדיר את השפה שלנו, לפי חוקי שפה חופשית הקשר, נשתמש באוסף של ביטויים, כל ביטוי הוא הגדרה כללית ברורה, הנקרא Production rule.

בכל Production rule יש לנו שני צדדים:

* צד שמאלי, שבוא נמצא סמל לא-סופי(NonTerminal) אשר אותו ניתן להחליף ברצף הסמלים אשר נמצאים בצד ימין לProduction rule. נקרא גםLHS(Left Hand Side).
* צד ימני, בוא יכול להיות סמלים לא-סופיים או סמלים סופיים הם מייצגים את רצף הסמלים אשר יכולים להחליף את צד שמאל בProduction rule. נקרא גם RHS(Right Hand Side).

### Derivation

Derivation, או גזירה בעברית, הוא אוסף כל ה Production rulesהמוגדרים אצלנו. איתם הקומפיילר יעבוד ויבדוק אם הקוד תואם את חוקי השפה.

# תיאור הפרויקט

## הקדמה

הפרויקט שלי הוא פיתוח מהדר, אשר מקבל כקלט קוד בשפה-MyC, אל קוד אסמבלי 64 ביט בסביבת windows 64.

השפה שאני המצאתי נקראת MyC, וכמו שמתחרז, היא בנויה בהשראת שפת C, בא ממשתי חלק מהיכולות אשר שפת C מציעה.

שפת תכנות היא אוסף של הגדרות לוגיות אשר מצייתות לחוקים תחביריים וסמנטיים של השפה. כאשר אנו מדברים על הגדרות ומבנה השפה, אנו מתייחסים למספר היבטים:

* מילוני(Lexical) – מגדיר אלו מילים שייכות לשפה, לדוגמה: המילה putchar שייכת לשפה שלי, אולם המילה blabla לא.
* תחבירי(Symantec) – מגדיר את האופן שבוא מילות השפה מחוברות זו לזו, לדוגמה: השרשור: if(1 > 2){…} חוקי, אולם השרשור if 1 > 2 לא.
* לשוני(Semantic) – מגדיר כללי התנהגות שהשפה נדרשת לציין להם, כמו: לא ניתן לבצע השמה עם ערך תווי למשתנה מסוג שלם, לא ניתן להכריז שם של משתנה פעמיים תחת אותו Scope.

## אבני השפה - MyC

### קבועים -Constants

קבוע הוא ערך המופיע בצורתו הסופי בקוד התוכנית.

בMyC קבוע יכול להיות בשני תצורות, ערך תווי, או ערך מספרי שלם.

ערך תווי יהיה בנוי מהמבנה הבא: '<Any-Printable-Ascii>', כאשר כל תווי שניתן להדפיסו יהיה כבול בין שני גרשיים.

ערך מספרי יהיה רצץ של ספרות בתחום: 0 – 9.

### סוגי נתונים -Data Types

בשפת MyC ישנם שלושה סוגים של טיפוסי נתונים: int, char, void.

Int – הוא טיפוס של מספר שלם, גודלו הוא שני בתים.

Char – הוא טיפוס מסוג תווי, גודלו הוא בית.

Void – הוא טיפוס מסוג תהום, משמע הוא מגדיר אי החזרה של ערך.

Int ו char יכולים להופיע המספר תפקידים בקוד השפה, כמו בטיפוס קבוע, טיפוס משתנה, ערך חזרה מפונקציה ועוד.

Void – הוא טיפוס שונה מעט, שהוא למעשה לא מגדיר טיפוס לערך, לא ניתן להגדיר void number; לדוג', אלה הוא משתמש במקום אחד בלבד, להגדיר פונקציה שלא מחזירה ערך.

### מילות מפתח - Keywords

מילות מפתח הם שמות של מילים המאפשרת לנו כל מני פונקציונליות לתוכנית. להלן רשימת מילות המפתח המצויות בשפת MyC:

* If – משתמש להגדרת תנאי.
* else if – משתמש להגדיר תנאי בתנאי שהתנאים הקודמים לו, לא התקיימו.
* else – משתמש להגדיר תנאי שיקרה בתנאי ושום תנאי קודם לו לא התקיים.
* while – משמש להגדיר לולאה.
* putchar – מוציא לפלט את התווי הרצוי.
* getchar – מקבל כקלט תווי מהמשתמש.
* return - מחזיר מקריאה לפונקציה את הערך המצורף.

### אופרטורים - Operators

#### אופרטורים חשבוניים:

* חיבור - +
* חיסור - –
* כפל - \*
* חילוק / -
* שארית - %

#### אופרטורים לוגיים – Logical Operators

* שווה ל - ==
* לא שווה ל - !=
* קטן מ - >
* גדול מ - <
* קטן או שווה ל - <=
* גדול או שווה ל - >=
* לא - !

#### אופרטורים של קשרים לוגיים – Logical Connectors Operators

* וגם - &&
* או - ||

#### סוגריים - Brackets

* סוגריים עגולים - ( )
* סוגריים מסולסלים - { }

### משתנים - Variables

משתנה הוא דבר השומר ערך תחת מקום מסוים בזיכרון.

כאשר נרצה לגשת או לשנות אך ערך המשתנה נשתמש בשם המשתנה. שם המשתנה נקרא גם המזהה – identifier.

שם משתנה נדרש לקיים את התנאים הבאים:

* השם לא זהה לשם של keyword.
* השם יכול להכיל אך ורק: תווים משפת האנגלית, ספרות, או התווי '\_'.
* השם לא יכול להתחיל בתווי שהוא ספרה.

כמו כן כדאי לציין כי שמות המשתנים הם Case sensitive, לצורך העניין: המשתנה main שונה לחלוטין מהמשתנה MAIN.

## תכולת השפה

בחלק זה אסביר על מבנה השפה וכיצד מוגדר כל חלק בשפה מבחינה דקדוקית.

כל פעולה בMyC אשר לא משתמשת עם יצירת Scope חדש, תסתיים עם נקודה פסיק ;.

### הגדרת משתנה - Declaration

בכדי להגדיר משתנה, ולמעשה "לתפוס" מקום בזיכרון, ישנה התבנית הכללית הבאה:



לדוגמה:



### השמת ערך למשתנה - Assignment

בכדי להשים ערך למשנה, נתבונן בהגדה הכללית הבאה:



לדוגמה:



כאשר נבצע השמה, אנו לא נדרשים שטיפוס ההשמה יהיה זהה לטיפוס המשתנה, במידה והשמנו מספר תווי למספר שלם, או מספר שלם לתוך תווי, הקומפיילר ידע להתמודד עם מצב זה. לדוגמה, הדבר הבה גם הוא חוקי לחלוטין:



### אמת/שקר – True/False

אמת או שקר מיוצר על ידי ערך מסוג int.

False זה ערך השווה לאפס.

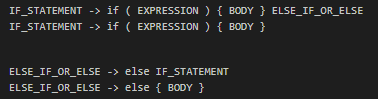
True זה כל ערך שלא שווה לאפס.

### תנאים - Conditions

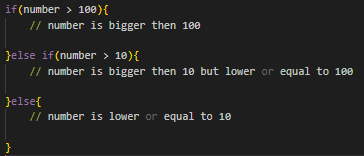
תנאי הוא הגדרת קטע קוד, שיהיה בתוך Scope ייעודי, אשר ירוץ אם ורק אם התנאי שהגדרנו יכיל ערך אמת(מספר שלם השונה מאפס).

בנוסף, ניתן להגדיר תנאי אשר יבוצע אם ורק אם התנאים הקודמים לו לא התבצעו, וכמו כן ערך התנאי הנ"ל החזיר אמת. ולבסוף, ניתן להגדיר תנאי אשר יבוצע אם ורק אם שום תנאי קודם לא התבצע.

ההגדרה הכללית של סוגי התנאים הללו נראה כך:



לדוגמה:



### 

### לולאות - loops

לולאות הוא הגדרת קטע קוד, אשר נמצא תחת Scope ייעודי, אשר ירוץ מספר מפעמים.

קטע הקוד ירוץ כל עוד תנאי הלולאה מניב ערך אמת, משמע שונה מאפס.

להלן הגדרה כללית של לולאה:



תמונה שמכילה טקסט, גופן, צילום מסך

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

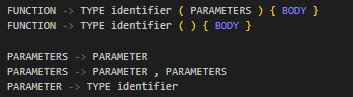
### הגדרת פונקציה

כל הרעיון של השפה, הוא שאנו רושמים את קטעי הקוד תחת פונקציות, לא ניתן לכתוב קוד בGlobal scope מלבד כתיבת פונקציות.

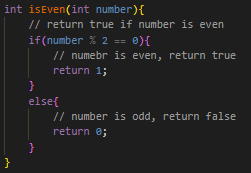
לכל פונקציה יכולה להחזיר שלושה סוגי ערכים: int, char, void. וכמו כן כל פונקציה יכולה לקבל מספר בלתי מוגבל של פרמטרים. לכל פרמטר נדרש להגדיר את שמו וטיפוסו.

טיפוס של פרמטר אינו יכול להיות void.

להלן הגדרה פורמלית של יצירת פונקציה:

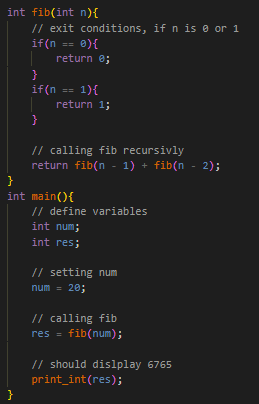


להלן דוגמה של יצירת פונקציה:



### קריאה לפונקציה

מאחר וכל השפה שלנו מושתת על גפי פונקציות, נדרש להיות יכולים לקרוא להם. במידה והפונקציה מחזירה ערך חזרה, נוכל לקבל אותו על ידי ביצוע השמה.

תמונה שמכילה טקסט, צילום מסך, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.הלן הגדרה פורמלית לקריאה לפונקציה.

להלן דוגמה של קטע קוד המבצע קריאה לפונקציה   
ומחשב את האיבר ה-20 בסדרת פיבונאצ'י.

## דקדוק השפה

לאחר שהסברתי את מבנה השפה, אצלול לעומק ואסביר על מבנה הדקדוקי המלא של השפה. ה Grammarשל השפה.

### Tokens

Tokens או אסימונים, הם יחידות המידע ברמה הכי נמוכה אשר מייצגים את קוד השפה. כאשר נרצה להדר את השפה, נעבוד עם האסימונים.

להלן כל הToken-ים הנמצאים בשפה MyC:



### תחביר השפה

תחביר השפה MyC , יחד עם עוד הרבה שפות תכנות פופולריות כיום, הוא תחביר חופשי הקשר.

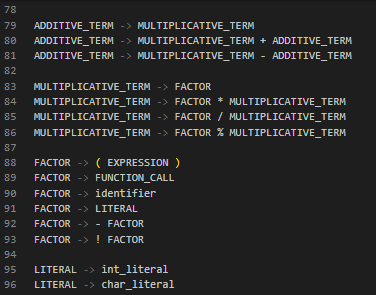
שפה חופשית הקשר (Context-Free Grammar - CFG) היא מודל מתמטי המשמש לייצוג שפות פורמליות. היא מגדירה שפה באמצעות אוסף כללים פשוטים וברורים המכונים production rules.

בכל Production rule יש לנו שני צדדים:

* צד שמאלי(LHS), שבוא נמצא סמל לא-סופי(NonTerminal) אשר אותו ניתן להחליף ברצף הסמלים אשר נמצאים בצד ימין לProduction rule.
* צד ימני(RHS), בוא יכול להיות סמלים לא-סופיים או סמלים סופיים הם מייצגים את רצף הסמלים אשר יכולים להחליף את צד שמאל בProduction rule.

סמליים סופיים נקראים Terminals, והם גם למעשה אותם Token-ים עליהם דברנו.

להלן תחביר השפה של שפת התכנות MyC:

תמונה שמכילה טקסט, צילום מסך, גופן, תוכנה

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.תמונה שמכילה טקסט, צילום מסך, תוכנה

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

# רקע תיאורטי

## מהו קומפיילר?

מעבד הוא הרכיב במחשב האחראי על הרצת שפות התכנות. אולם מעבד ולמעשה כל שאר רכיבי המחשב, אינם יודעים לציית לפקודות בשפת תכנות, אלה הוא יודע לקרוא רצפים של אפסים ואחדות.

לבני האדם יש הרבה מאוד יכולות וכישורים, אך לבנות ולתכנת קטעי קוד על ידי חיבור רצפים של אפסים ואחדות, זהו לא אופציה.

בני האדם, שהבינו את הכוח הפוטנציאלי של המחשב, חפשו דרך להעפיל על הפער זה, כך שיהיה ניתן לתכנן קטעי קוד ולבצע פעולות, אך בצורה יותר נוחה מלעבוד מול אפסים ואחדות. הפתרון שהם מצאו הוא להמציא ולהשתמש במהדר.

קומפיילר, או בעברית-מהדר, זהו שם של תוכנית מחשב אשר תפקידה הוא לתרגם קוד בשפת מחשב א' לשפת מחשב ב'. המהדר הקלאסי, מקבל כקלט שפה הרשומה בשפה עילית ומתרגם אותה לשפה ברמה נמוכה יותר.  
תפקידי מעבדי השפות הוא להמיר שפות שמובנות לבני האדם, לשפות שמובנות על ידי המעבד(או לשפות ברמה יותר נמוכה).



## סוגי מעבדי שפות

מתורגמן - Interpreter  
שפת תכנות המשתמשת בעיבוד מסוג מתורגמן תפעיל את המתורגמן בזמן הריצה של התוכנית. המתורגמן כל פעם ידע מהי השורה הבאה לעיבוד, יתרגם אותה לשפת מכונה ויבקש מהמעבד שיבצע אותה.

לרוב יחשבו מתורגמנים כיותר איטיים מאשר מהדרים(compilerים), שכן המתורגמן יבצע את תהליך העיבוד לכל שורה בנפרד בעוד שהמהדר יעבד ישר את הקוד במלואו.

מהדר - Compiler  
שפת תכנות המשתמשת בעיבוד מסוג מהדר תעבור על הקוד במלואו לפני הרצת התוכנית. המהדר ימיר את הקוד לקובץ בינארי(לדוגמה קובץ (EXEהמכיל קוד בשפת מכונה, ולאחר מכן יהיה ניתן להריץ את אותו הקובץ ישירות.  
בתהליך הקומפילציה, המהדר מסוגל לאתר שגיאות תחביריות בקוד, ולהתריע למשתמש כי לא נין להשלים את תהליך ההידור.

בנוסף, המהדר עשוי לבצע כל מני אופטימיזציות במהלך תהליך בקומפילציה, שיכולות לשפר את ביצועי התוכנית (כגון שיפור ניצול הזיכרון או מהירות הריצה).

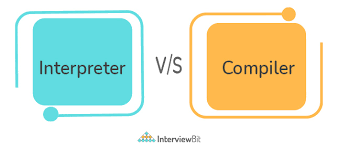
### - Compiler && Interpreterיתרונות וחסרונות

יתרונות של המהדר על פני המתורגמן:

1. המהדר מסוגל מבצע אופטימיזציה של הקוד, בעוד שמתורגמן לרוב לא.
2. מהדר מבצע בדיקה של נכונות הקוד גם לפני הריצה, בעוד שמתורגמן עולה על שגיאות רק בזמן ריצה

יתרונות של המתורגמן על פני המהדר:

1. שפות שעובדות עם מתורגמן נחשבות כיותר "גמישות", הוא מבצע את ההמרה לקוד מכונה בזמן הריצה וזה מאפשר לתוכנית לפעול בצורה יותר דינמית, הקוד יכול להתאים את עצמו במהלך ההרצה, לשנות ערכים או אפילו לעבוד עם נתונים מבלי לדעת את סוגם מראש.



## תהליך הקומפילציה

עיבוד מקדים(preprocessing)  
השלב שקודם לתהליך הקומפילציה. בשלב זה הקוד עובר עיבוד ראשוני שבו מתבצעת פעולות כמו: השתמטת הערות, השמה של קבועים לערכם, ועוד. מטרת העיבוד המקדים הוא להנגיש את הקוד בכדי שיהיה מוכן ומותאם.

### ניתוח מילוני(Lexical analysis)

בתהליך זה הקומפיילר מבצע סריקה של כל קוד המקור, ומקטלג אותו לרצף של Tokens(אסימונים). כל Token הוא יחידת המידע הבסיסית, היכול לתאר רצף של אותיות כאלמנט דקדוק. לדוגמה עם קוד מקור: int num1 = 5;  
רצף האסימונים יהיה: [semicolon][Literal][operator][identifier][keyword]

כמו כן בשלב זה הקומפיילר יכול לזהות שגיאות מילוניות ולהתריע על כך למשתמש.

### ניתוח תחביר(syntax analysis)

בשלב זה לקומפיילר יש את אוסף האסימונים מהשלב הקודם. כעת הקומפיילר ממיר את רצף האסימונים הזה למבנה המייצג את הכללים התחביריים של השפה – עץ תחבירי(Parse Tree).

לא תמיד הקומפיילר יצליח לבנות את העץ התחבירי, ובמידה והקומפיילר יכשל זאת, משמע שיש לנו טעות תחבירית בקוד השפה שלנו. במקרה זה הקומפיילר יתריע על כך למשתמש.

### ניתוח סמנטי( ( Semantic Analysis

בחלק זה אנו עוברים על אותו מבנה שבנינו, ומוודאים שצורת המבנה עומדת בדרישות השפה מבחינת ההיגיון הסמנטי, ולא רק מבחינת התחביר.

דבר זה בא לידי ביטוי באופנים הבאים: שימוש במשתנה שלא הוגדר, הגדרת משתנה פעמיים באותו Scope, ועוד.

### יצירת קוד ביניים( (Intermediate Code Generation

בחלק זה, לאחר שראינו את נכונות הקוד מבחינה תחבירית וסמנטית, נבצע סריקה סופית על העץ התחבירי שלנו, לאחר שהוכחנו את נכונות הקוד, ונבנה קוד בשפת ביניים. שפה זאת היא שפה שמייצגת את המבנה הלוגי של התוכנית בצורה פשוטה יותר, כך שיהיה הרבה יותר פשוט לאחר מכן להמיר לשפת מכונה או אל השפה הרצוייה.

### אופטימיזציה(code optimization)

בחלק זה התוכנית תעבור על הקוד ביניים שנוצר, ותנסה לשפר אותה. הן מבינת יעילות בזמני ריצה, והן מבחינת יעילות בשימוש בזיכרון. אופן השיפור יכול לבוא לידי ביטוי במחיקת שורות מיותרות, הפשטת ביטויים קבועים ועוד. מטרת האופטימיזציה היא להפיק קוד יעיל ומהיר יותר לביצוע, תוך שמירה על הפונקציונליות.

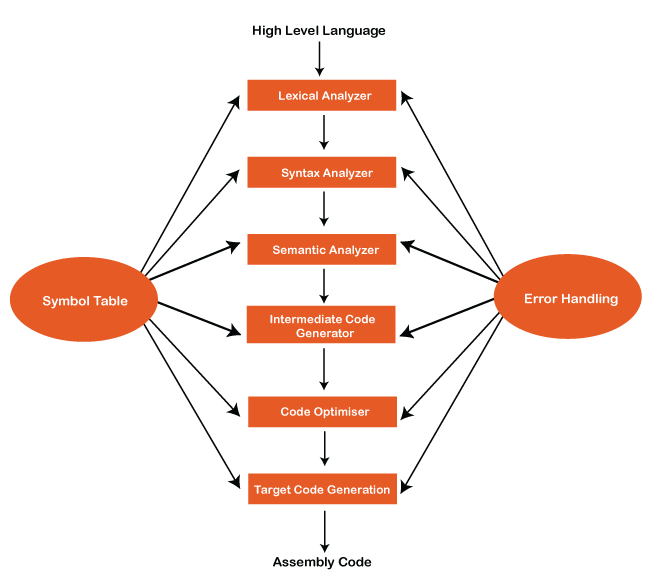
### יצירת הקוד(Code Generation)

בחלק זה אנו נשתמש בקוד הביניים שייצרנו מקודם, ונתרגם אותו לשפה הרצויה.

## Front End/Back End

כאשר אנו מדברים על שלבי הקומפילציה, נהוג לחלק אותם לשני חלקים: Front End וBack End. Lexical, Syntax, Semantic, Intermediate Code Generation, הם נחשבים כשלבי הFront End. מנגד, הCode Generation נחשב כשלב הBack End.

הסיבה לחלוקה זאת היא מאחר ושלבי הFront End אינם מתייחסים כלל לסביבה בא נמיר את הקוד, כמו שפת היעד, מערכת הפעלה וכדומה. לעומת זאת, שלבי הBack End אכן מתייחסים לפרטים הטכניים ולסביבה בה יהיה קוד היעד.



## תורת האוטומטים

### תורת האוטומטים-מבוא

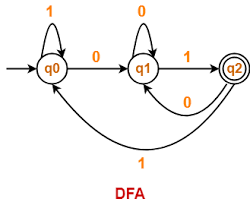
תורת האוטומטים היא תחום במתמטיקה ומדעי המחשב אשר עוסק בבנייה ועיצוב מודלים חישוביים. מכונות אשר מקבלות קלט מסוים ויודעות לטפלו על ידי מעבר בין מצבים מוגדרים. למודלים אלו יש שימוש רב בהרבה תחומים כמו בניתוחי שפות, קומפילציה, בינה מלאכותית, ועוד.

### אוטומט סופי דטרמיניסטי-DFA

אוטומט סופי דטרמיניסטי הוא סוג ספציפי של מודל בתורת האוטומטים, אשר יש לו את התכונות הבאות:

* לכל קלט במצב מסוים יש מעבר למצב אחד בלבד, אין קונפליקטים.
* אין מעברים ריקים(Epsilon).
* יש מצב התחלתי אחד, בעוד שניתן להגדיר מספר מצבים סופיים.
* האוטומט מטפל בקלט תו-אחר-תו.

לדוגמה נתבונן באוטומט הבא, המקבל שפה המורכבת מ: {0, 1}



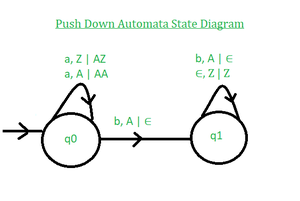
* q0 הוא המצב ההתחלתי (מסומן ע"י חץ שלא מגיע משום מצב).
* q2 הוא המצב הסופי(מסומן ע"י עיגול כפול).
* בכל מצב באוטומט, אין קישוריות לשני מצבים על ידי אותו תו(תכונת הDFA).
* בכל מצב, חייב לעבור על ידי תוו לא ריק.

### אוטומט מחסנית – PDA

בדומה לאוטומט הסופי הדטרמיניסטי, אוטומט המחסנית מורכב מאוסף של [מצבים](https://he.wikipedia.org/wiki/%D7%9E%D7%A6%D7%91_(%D7%9E%D7%93%D7%A2%D7%99_%D7%94%D7%9E%D7%97%D7%A9%D7%91)), המקבילים למצבי הזיכרון בהם יכולה להימצא מכונת החישוב במהלך ביצוע חישוב. דרך הפעולה של האוטומט מבוססת על קריאת מילת [קלט](https://he.wikipedia.org/wiki/%D7%A7%D7%9C%D7%98) באופן סדרתי, אות אחת בכל פעם, עד אשר מסתיימת קריאת המילה.

ההבדל העקרי בין אוטומט מחסנית לבין אוטומט סופי, הוא התקן זיכרון נוסף שעומד לרשות אוטומט המחסנית-מחסנית. כאשר עובדים עם אוטומט מחסנית, נבצע החלטות לא רק על ידי המצב והתווי הנוכחי, אלה גם לפיי מצב המחסנית.

להלן דוגמה של אוטומט מחסנית פשוט.

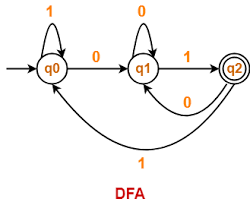


### ייצוג אוטומט על ידי מטריצה

ניתן לייצג אוטומט סופי דטרמיניסטי(ללא מחסנית) על ידי מטריצה, מערך דו ממדי של איברים.

* כל שונה במטריצה תייצג לי State, מצב בוא אני יכול להימצאות.
* כל עמודה תייצג לי תווי אלפביתי המוגדר בשפה.
* כל איבר במטריצה אייצג לי את המהלך הבא, ערך הState אליו אליך במידה ואני נמצא תחת הState והתווי הנוכחי.

לדוגמה, מוצג אותו אוטומט אשר הצגתי מקודם:



יהיה לי את המטריצה הבאה:

|  |  |  |
| --- | --- | --- |
| Input 1, index: 1 | Input 0, index: 0 |  |
| Goto: q0 | Goto: q1 | q0, index: 0 |
| Goto: q2 | Goto: q1 | q1, index: 1 |
| Goto: q0 | Goto: q1 | q2, index: 2 |

המקומות אשר צבועים ב: ■ מייצגים את המטריצה עצמה בעוד שאלו הצבועים ב: ■מייצגים את המיקום במטריצה.

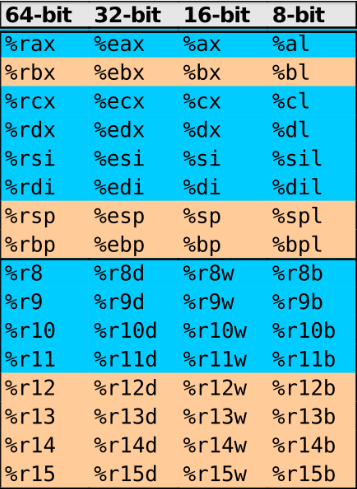
לדוגמה, אם אני במצב q1 וקבלתי את התו '0', אגש למטריצה במיקום:

Mat[1][0], שבוא השלב הבא יהיה להגיע למצב q1.

כאשר אנו רוצים לעבוד עם אוטומט מחסנית, נממש זאת בדרך דומה, נייצג באותו אופן אוטומט סופי דטרמיניסטי(ללא מחסנית), ופשוט נוסיף מחסנית ככלי עזר.

## שפת היעד - אסמבלי nasm x64

לאחר שכבר הסברתי בצורה די מעמיקה על אופי השפה המומצאת – MyC, ארצה להסביר על השפה השנייה – שפת היעד. בחלק זה אדבר על האסמבלי nasm x64 ואתייחס באופן ספציפי על האופן שבוא נהוג לעבוד עימו בסביבת windows.

בnasm x64, יש לנו אוגרים עם: 64, 32, 16, 8 בתים. להלן תמונה של כל האוגרים הקיימים:

אותם אוגרים מתחלקים לקבוצות שונות:

**אוגרי כלל‑מטרה (General‑Purpose Registers )**  
כוללים RAX, RBX, RCX ו‑RDX. משמשים לאחסון משתנים זמניים, תוצאות חישוב ושימושים גנריים בתוך הפונקציה.

**אוגרי פרמטרים ומצביעי זיכרון (Index & Argument Registers)**  
RSI ו‑RDI משמשים על נתונים בזיכרון (מקור ויעד), בהתאם ל‑

**אוגרי הרחבה (Extended Registers R8–R15)**

אוגרים כלליים נוספים המשמשים להעברת פרמטרים נוספים, אחסון ערכים ועוד.

**אוגרי מחסנית ומסגרת (Stack & Frame Pointers)**  
RBP ו- RSP משמשים בכדי לשמור ולגשת לנתונים השמורים במחסנית.

### תנאים

ב NASM x64 תנאים מבוססים על שילוב של פקודות השוואה(CMP או TEST) ופקודות קפיצה כמו(JE, JG, JL ועוד). להלן שימוש בתנאים באסמבלי.

### לולאות

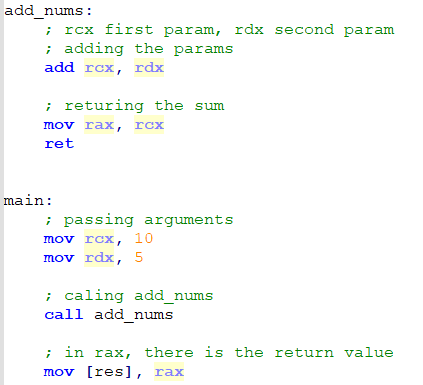
ב NASM x64 לולאות while מבוססות על חזרה על קטע קוד כל עוד תנאי הלולאה מניב אמת, להלן דוגמה של לולאת while באסמבלי.

### פונקציות

ב־NASM x86‑64, פונקציות מוגדרות באמצעות תווית (label), לאחר מכן גוף הפונקציה ובסופו של דבר הkeyword – ret , המורה על חזרה למקום שממנו הפונקציה נקראה. להלן מבנה פונקציה באסמבלי.

### קריאה לפונקציות

ב־NASM x86‑64, קריאה לפונקציה נעשית על ידי שימוש בkeyword – call ולאחר מכן label הפונקציה. בשביל להעביר פרמטים, המוסכמה בwindows היא להעביר את ארבעת הארגומנטים הראשונים על ידי האוגרים:

במידה והפונקציה מחזירה ערך, היא תחזיר ערך זה על גבי האוגר: RAX.

להלן מופיע דוגמה לשימוש בקריאה לפונקציות עם ארגומנטים וערך חזרה:

# תיאור הבעיה האלגוריתמית

תהליך הקומפילציה הוא תהליך די ארוך, אשר מצריך מספר שלבים, לכל שלב והבעיה האלגוריתמית שלה.

בפרק זה אציג עבורכם את הבעיה האלגוריתמית עבור כל שלב מרכזי בפרויקט, אנתח אותו ואתן דוגמאות.

## ניתוח מילוני Lexical Analysis –

כפי שכבר הסברתי, בשלב זה המטרה העיקרית היא להמיר את כל קוד המקור שלנו, הרשום כתווי-תווי, לרצף של אסימונים(Tokens).

הבעיה האלגוריתמית הינה כיצד לנתח את רצף התווים שלנו, ולבנות את רצף האסימונים שלנו. וזאת תוך כדי שמירה על יעילות זמן ריצה. כמו כן, נדרש לבנות אלגוריתם אשר ידע לפרש את רצף התווים בצורה חד-חד ערכית, כך שלא יהיה בלבול כלל. לדוגמה: מה קורה כאשר האלגוריתם נמצא במצב בוא הוא עיבד את רצף התווים: "retu", האם הוא יגדיר משתנה חדש בשם: "retu"? או יבדוק אם התווים הבאים הם: "rn", כך שבסופו של דבר הוא יזהה כי זוהי בכלל מילת המפתח "return".

## ניתוח תחבירי – Syntax Analysis

כפי שכבר הסברתי, בשלב זה הקומפיילר המיר את קוד המקור אל רשימה של Token-ים. כעת בשלב זה הקומפיילר ינסה לבנות עץ תחביר(Abstract Syntax Tree).

על מנת לבנות את עץ התחביר, נדרש להגדיר הגדרות מדויקות אשר ייצגו את המבנה התחבירי של השפה, וההיגיון שבה. האתגר העיקרי בשלב ניתוח התחביר של השפה הוא בניית אלגוריתם אשר יודע לעבוד עם ההגדרות ולבנות את עץ התחביר, תוך כדי שמירה על יעילות מרבית.

יתר על כך, נצטרך שהאלגוריתם יהיה דטרמיניסטי וכמו כן שידע לבצע החלטות מושכלות תוך כדי התחשבות במלוא הקלט.

## ניתוח סמנטי – Semantic Analysis

כפי שכבר הסברתי, בשלב זה, הקומפיילר הוכיח את נכונות הקוד מבחינה מילונית ותחבירית. אך דברים אלו לא מספיקים, ויש שלב אחד נוסף שאנו נדרשים לוודא, וזהו לראות אם הקוד עובר את הדרישות הסמנטיות. בשלב זה אנו בודקים כי אנחנו לא משתמשים במשנה או קוראים לפונקציה לא מוכרזת. בודקים כי אין ערבוב בין שני טיפוסים בעת פעולה חישובית, בודקים שאין הגדרה של שני משתנים עם שמות זהים תחת אותו Scope, ועוד.

הבעיה האלגוריתמית העיקרית היא מציאת אלגוריתם, אשר ידע לטפל בכל אשר נאמר תוך כדי שמירה על יעילות זמן ריצה טובה. האלגוריתם נדרש לעבור על העץ התחבירי, ולראות כי תקין מבחינה סמנטית.

# סקירת אלגוריתמים בתחום הבעיה

בפרק זה אסקר את הפתרונות הקיימים בתחום הCompiler design. הLexer, Semantic, Code generation, כל שלבים אלו אומנם מורכבים, אך רוב הפתרונות הקיימים כיום ודי דומים זה לזה. בפרק זה אתמקד באופן ספציפי על שלב הParsing שבקומפיילר.

שלב זה הוא השלב הכי מורכב מבחינה אלגוריתמית, ומציע הרבה אלגוריתמים ושיטות עבודה שונות. כל אחת ויתרונותיה, בפרק זה אסקור את חלקם.

## Parser Categories

### ניתוח מלמעלה למטה – Top Down

גישה זו מתחילה בסמל השורש של הדקדוק(הProduction rule הראשון בDerivation) ומנסה ליצור את מחרוזת הקלט על ידי יישום כללי הדקדוק בצורה הדרגתית. כל פעם התוכנית מחליפה את אחד הNon Terminal-ים בRHS, אל הRHS של הProduction Rule בוא הLHS זהה לאותו Non Terminal.

גישה זו יעילה עבור דקדוקים פשוטים יחסית, הניתנים לחיזוי מראש. אך מתקשה מאוד ואף נכשלת עם סוגי דקדוקים יותר מורכבים.

### ניתוח מלמטה למעלה – Bottom Up

גישה זו פועלת בכיוון ההפוך, מתחילה בקלט ומנסה לבנות את מבנה העץ התחבירי עד להגעה לסמל השורש. כל פעם התוכנית ממירה רצף של Non Terminal-ים וTerminal-ים הנמצאים בRHS, לLHS של ה Production ruleהמתאים.

התוכנית תסתיים כאשר נקבל את הProduction rule הנמצא בראש הDerivation.

שיטה זו נחשבת כיותר חזקה ומסוגלת למתמודד עם דקדוקים מורכבים הרבה יותר, ומבצעת זאת תוך כדי שמירה על יעילות זמן מירבת.

גישה זו מתאימה לשפות מורכבות ומיושמת בדרך כלל במהדרים מתקדמים.

### Left-most derivation

בכל שלב בוחרים את הסימול (non-terminal) השמאלי ביותר בשרשרת ולא את הטרמינל.

מחליפים אותו בהתאם לכלל ב - production rule, ואז ממשיכים שוב תמיד עם הסימול השמאלי ביותר הבאה.

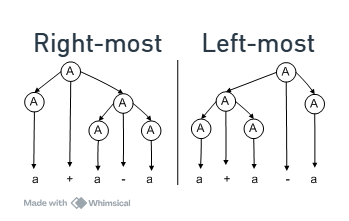
תוצאת הגזירה מוצגת כעץ שבו ההרחבות מתבצעות מהשמאל לימין.

### Right Most Derivation

בכל שלב בוחרים את הסימול (non-terminal) הימני ביותר בשרשרת.

מחליפים אותו בהתאם לכלל ב - production rule, ואז ממשיכים שוב תמיד עם הסימול הימני ביותר הבאה.

תוצאת הגזירה מוצגת כעץ שבו ההרחבות מתבצעות מהימין לשמאל.



### Left - > Right

"Left->Right" אומר כי סריקת האסימונים מתבצעת משמאל לימין. מה זה אומר? זה אומר כי הParser- קורא את רצף הטוקנים בדיוק כפי שהם מופיעים בקוד, מהטוקן הראשון (השמאלי ביותר) ועד לטוקן האחרון (הימני ביותר), מבלי לחזור אחורה.

### LL(1)

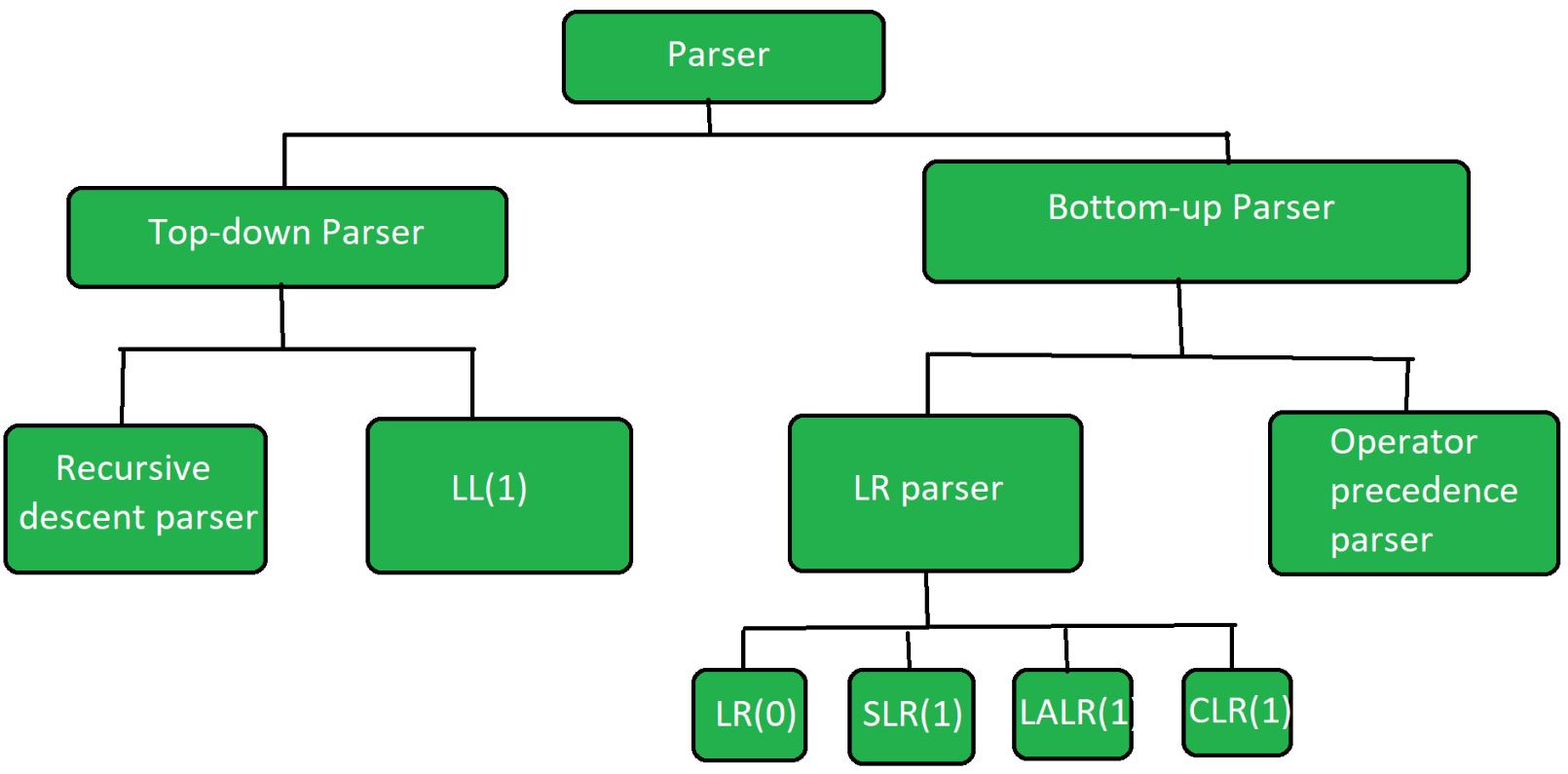
top-down עם left-to-right + Left-most-derivation. מתאים לדקדוקים ללא left recursion.

### LR(1)

Bottom-up עם left-to-right + right-most-derivation. עם הסתכלות אחת קדימה. משתמש בשיטת shift reduce.

### SLR(1)

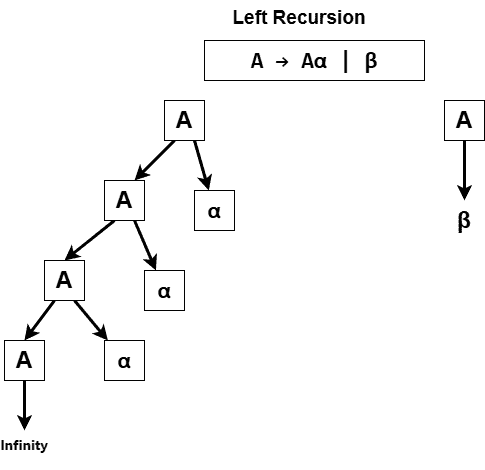
Simple LR(1). תת-קטגוריה של LR(1) הנחשבת כפשוטה יותר. אך פחות עוצמתית במקרים של shift-reduce conflicts.



## Parsing Problems

### Left recursion

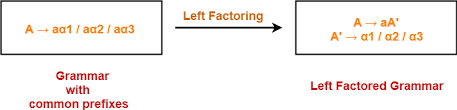
בעיה מוכרת בשלב ניתוח התחבירי. בעיה זו נוצרת כאשר אנו מגדירים Production Rule אשר הNon Terminal הראשון לפי הסדר בRHS זהה לסימן ה LHSשל אותו Production Rule. מצב זה יוצר שרשרת אין סופית. ישנם סוגי מנתחים אשר נכשלים כאשר יש Left recursion ויש כאלו שמתמודדים. כאשר נבחר סוג מנתח, נתחשב בפרמטר זה.



### Left Factoring

בעיה מוכרת בשלב ניתוח התחבירי בתכנון קומפיילר, אך בעייתי רק לניתוח מלמעלה ומטה.

הוא קורה כאשר מוגדר לנו כמה Production Rules אשר בהם הNon Terminal הראשון זהה. במקרה זה התוכנית מגיעה למצב בוא היא לא יודעת עם איזה Production rule להשתמש.



### Shift/Reduce Conflict

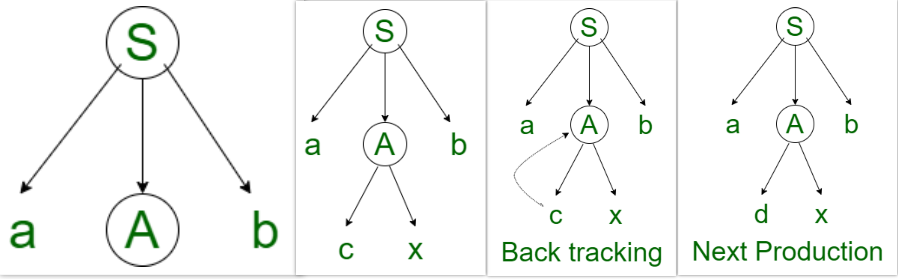
בעיית Shift/Reduce Conflict קיימת בסוגי מנתחים מסוג LR. אשר משתמשים בטבלת parser table. בעיה זו נוצרת כאשר לפי הgrammar שלנו, במצב הנוכחי אנו יכולים גם לבצע shift, וגם לבצע reduce. זוהי בעיה מאחר והאוטומט לא דטרמיניסטי ולא יודע איזה פעולה לבצע. כאשר בוחרים סוג מנתח, אנו נדרשים להתייחס לבעיה זו.

## Parsing algorithms

כעת אסביר על מספר אלגוריתמים מוכרים ונפוצים בשלב הParsing.

### עקיבה לאחור – Back Tracking

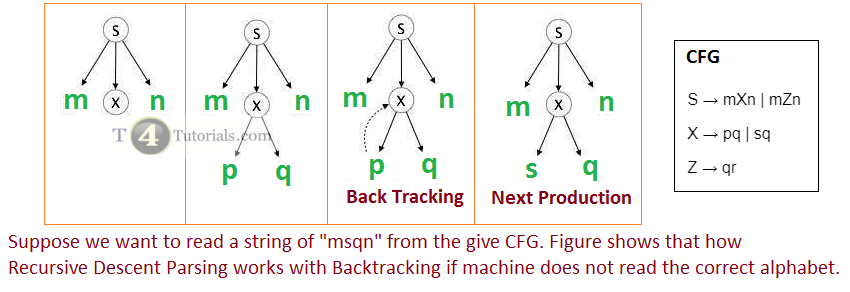
עקיבה לאחור היא שיטה לפתרון בעיות חיפוש הפועלת לפי העיקרון: בודקים אפשרות מסוימת, אם היא נכונה, הידד! נמשיך בחיפוש, אך אם היא לא נכונה, נסוג לאחור וננסה אפשרות אחרת. אם נתקלים בדרך סתומה, נחזור אחורה וננסה דרך אחרת. נבצע תהליך זה עד למציאת פתרון או להבנה שאין פתרון אפשרי.



### Recursive decent

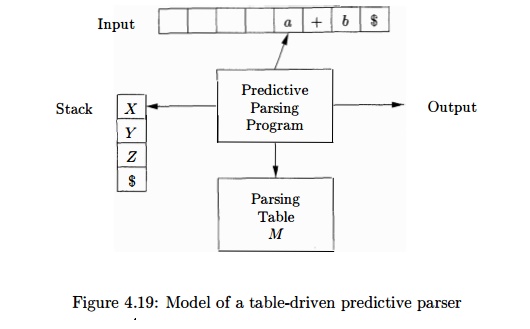
Recursive decent היא שיטה מסוג TDP, בא אנו קוראים את הקלט משאל לימין. באלגוריתם זה, עץ התחביר נוצר מלמעלה ומטה. שיטה זאת משתמשת בפונקציות עבור כל Terminal וNon Terminal אשר שייכים לGrammar של השפה, ואופן הפעולה הוא קריאות רקורסיבית בין הפונקציות. אלגוריתם זה סובל מLeft factored ובכדי להתגבר על מכשול זה, אלגוריתם זה יבצע Back-Tracking וינסה אפשרות אחרת. אלגוריתם זה משומש בד"כ בשפות שהתחביר שלה פשוט יחסית.

מאחר ואלגוריתם זה משתמש בBack-Tacking, סיבוכיות זמן ריצתו היא אקספוננציאלית, O(2^n).



### Predictive Parser

Predictive Parser הוא סוג של Parser אשר מסתכל קדימה אל עבר הקלט וכך מסוגל להחליט עבור כל כלל איזה Production Rule יחול עליו, ומכך הוא גם לא מבצע Back-Tracking בשום מצב. הוא דטרמיניסטי.

סוג אלגוריתם זה הוא גם Top-Down-Parser. אלגוריתם זה משתמש במחסנית ויעילותו הוא O(n).

### Shift Reduce Parser

Shift Reduce Parser הוא אלגוריתם מסוג Bottom-Up-Parsing אשר מקבל את קלט הקוד ועושה דרכו למעלה, מחליף כל פעם Terminal-ים וNon Terminal-ים בNon Terminal-ים לפי הProduction rules, עד אשר קבלנו את הProduction Rule הראשוני בDerivation. הוא יודע להתמודד עם Left factoring.

אלגוריתם זה משתמש במחסנית, ופועל על פי שני עקרונות:

* Shift – האלגוריתם מכניס לראש הרשימה את הTerminal/Non Terminal הנוכחי.
* Reduce – ברגע שהאלגוריתם מזהה כי בראש המחסנית ישנם ערכים הזהים לRHS של איזשהו Production Rule, האלגוריתם ישלוף את ערכים אלו ויחליפם בLHS של אותו production Rule.

אלגוריתם זה נחשב כאופציה מצוינת בהרבה מקרים, סיבוכיות זמן הריצה שלו הוא O(n).



# אסטרטגיה

בפרק זה אציג את האסטרטגיה לפתרון הבעיה האלגוריתמית בפרויקט.

כפי שכבר ציינתי, פרויקט זה מורכב ממספר בעיות אלגוריתמיות מרכזיות, מטרתנו הוא לבנות את הקומפיילר אשר יקמפל את הקוד ביעילות זמן ריצה המינימלית. אחלק חלק זה עבור שלבי הפרויקט.

## ניתוח מילוני – Lexical Analysis

זהו החלק הראשון בהחלט בפרויקט. כפי שכבר הסברתי, ניתוח המילוני מקבל את קוד המקור שהמשתמש הזין, ורוצה לקמפל. ניתוח המילוני יעבור על קוד המקור, ויצור סריה של אסימונים המייצגים את הקוד.

שוב אזכיר כי המטרה שלנו זה לזהות וליצור את רצף האסימונים ביעילות זמן הריצה המרבית. לכן לא נסכים על להשתמש בקינון תנאי If, או פתרון דומה. נרצה למצוא פתרון יעיל.

הפתרון הוא שימוש באוטומט.

סוג האוטומט בוא נרצה להשתמש הוא אוטומט סופי דטרמיניסטי, או אס"ד בקיצור. באוטומט נשתמש בשפה המורכבת מהתווים החוקיים בשפת המקור. נבנה מסלול עבור כל אחד מהאסימונים שיש לנו בשפה. המסלול יסתיים כאשר נגיע לקודקוד שהוא עלה. במידה ונריץ רצף תווים לא חוקיים באוטומט, נגיע לאחד מין הצמתים המייצגים שגיאה.

על ידי שימוש זה אנו יכולים לזהות בצורה חד-חד ערכית כל אחד מין האסימונים, אין שום מצב של חוסר ודאות. בשימוש זה אנו בכל פעם עוברים תווי-תווי, כך שסך בכל ניתן למצוא את כל האסימונים בזמן ריצה של O(n).

### מכונת המצבים

להלן מופיע מכונת המצבים עבור שפת התכנות MyC. מאחר ודיאגרמה זו די גדולה, קשה לקרוא אותה פה. בנספחים שייכתי לכם קישור לדיאגרמה כך שתוכלו לראותה בצורה טובה.

תמונה שמכילה תרשים, צילום מסך, טקסט, צבעוני

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

* מצב בצבע כחול מסמל keyword
  + מצב בצבע כחול מלא מסמל מצב סופי
* מצב בצבע כתום מסמל ביטוי השוואה
  + כתום מלא מסמל אוטומט סופי
* מצב בצבע אפור מלא מסמל סוגריים
* מצב בצבע חום מלא מסמל פעולה מתמטית
* מצב בצבע אדום מסמל מצב שגיאה

### אלגוריתם לזיהוי אסימון

* נתחיל תמיד בצומת ההתחלה - q0.
* נרוץ על גבי האוטומט עד להגעת צומת שהוא עלה.
* אם העלה הוא עלה סוף, ניצור אסימון חדש לפי סוג העלה.
* אם העלה הוא עלה שגיאה, נוציא לפלט את ההודעה המתאימה ונכנס לPanic mode.

### פסאודו קוד

להלן מופיע פסאודו קוד של האלגוריתם למציאת אסימון:

1. **אתחל** את מטריצת הסמיכויות לפי אוטומט השפה.
2. **עבור** לצומת ההתחלה – q0.
3. **כל עוד** לא הגעת לצומת שהוא עלה, בצע:
   1. **עבור** לצומת הבאה לפי הצומת והתווי הנוכחי.
   2. **קדם** את הקלט בתווי אחד.
4. **צור** אסימון לפי מספר העלה.
5. **החזר** את האסימון.

## ניתוח תחבירי – Syntax Analysis

זהו החלק השני בקומפיילר. כפי שכבר הסברתי, בחלק זה קבלנו את רצף האסימונים משלב המילוני, ומכך גם אנו יודעים כי קוד הקלט תקין מבחינה מילונית. כעת נרצה להוכיח את נכונות הקוד מבחינה תחבירית, וננסה לבנות את עף התחביר – Abstract Syntax Tree. בפרק הקודם הצגתי מספר אלגוריתמים מוכרים עבור שלב זה, האלגוריתם שבחרתי להטמיע בפרויקט הינו Shift Reduce Parser.

בחרתי באלגוריתם זה מאחר והוא יודע להתמודד עם Left factoring, ובעל סיבוכיות זמן ריצה מרבית.

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 הוא מורכב יותר. שלב זה פועל לפי התחביר(חופשי הקשר) אשר מתאר לנו את תחביר השפה. וכתוצאה מכך לא ניתן לקבוע את מעבר המצבים על ידי אוטומט סופי דטרמיניסטי. אלה נזדקק להשתמש באוטומט מחסנית.

על מנת לממש את אלגוריתם ניתוח התחבירי shift reduce. נשתמש במחסנית, ובשתי טבלאות הנקראות יחדיו Parsing Table.

להלן תמונה של Shift Reduce Parser.



### Shift, Reduce, Accept

Accept, Shift, Reduce הם שלושת המהלכים אשר לפי הם אנו פועלים בעת ביצוע האלגוריתם. האופן שבוא אנו מתייחסים לאותם מהלכים מהווה את יסוד האלגוריתם.

#### Shift

מצב בוא אנחנו מחליטים לבצע Shift, משמע לדחוף את התווי הנוכחי(Terminal/Non Terminal) למחסנית.

ניתן לדמיין מצב זה כאשר אנו מתקדמים קדימה עם האוטומט.

#### Reduce

מצב בוא האלגוריתם זיהה כי n איברים האחרונים הנמצאים בראש המחסנית שווים לRHS של איזשהו Production Rule. נשלוף את אותם n איברים מהמחסנית ונדחוף במקומם את הLHS של אותו Production Rule.

ניתן לדמיין מצב זה כאשר אנו נסוגים לאחור חזרה באוטומט.

#### Accept

מצב באלגוריתם בוא הוא מזהה כי סיימנו לטפל בכל הקלט, וקבלנו את עץ-התחביר במלואו. מצב זה מתריע כי האלגוריתם סיים בהצלחה.

### Parsing Table

לאחר שהסברתי על שלושת המצבים בהם האלגוריתם פועל(Accept, Shift, Reduce), עולה השאלה, כיצד האלגוריתם יודע מתי הוא צריך לבצע איזשהו מצב? כיצד הוא מתייחס לדקדוק ולתחביר של השפה שלנו? פה נכנס לתמונה הParsing Table .

הParsing Table מייצג לי אוטומט סופי דטרמיניסטי(DFA), אשר בכל נקודה יודע איזה פעולה לבצע ולאן להמשיך.

הParsing Table מחולק לשני תתי טבלאות: Goto Table וAction Table.

ניתן יהיה להבחין כי הטבלה לא מלאה בכל נקודה קיימת במטריצה, אלה היא לוקה בחלקים ריקים. הסיבה לכך היא שלא כל מהלך אפשרי הוא חוקי בשפה. במידה הגענו לנקודה בparsing table שבה אין לנו לבצע שום פעולה(shift או reduce) נבין כי יש לנו שגיאה תחבירית.

#### Action Table

מציין למנתח מהי הפעולה הבאה הרצויה, בהתאם למצב הנוכחי והתווי הבא בקלט.

כל שורה תייצג לי מצב, וכל עמודה תייצג לי אסימון(Terminal/Token) אשר נמצא בשפה.

בנוסף לסוג פעולה הבאה, המנתח מחיר עבורי ערך מסוים עבור כלShift או Reduce:

עבור shift, הערך ייצג לי את מספר המצב אליו אני נדרש לעבור.

עבור Reduce, הערך ייצג לי את מספר הProduction Rule שאתו אשתמש כאשר אבצע את הReduce.

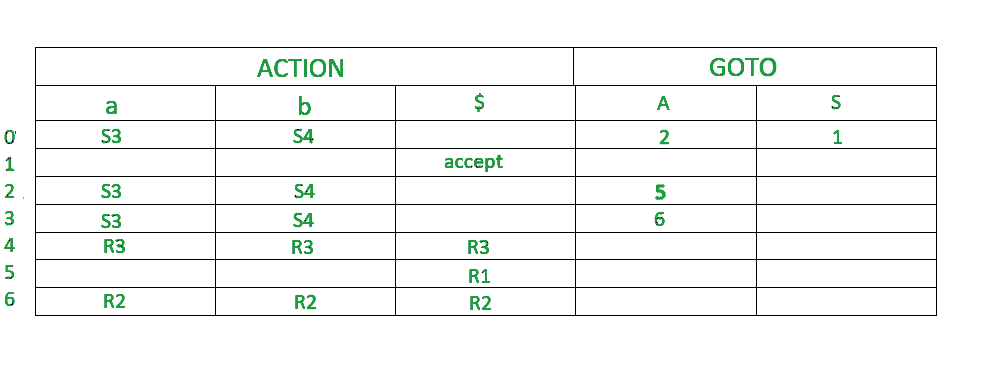
מאחר ויש לי רק מקרה אחד ויחיד בוא אני מקבל את הקלט, ואומר כי הוא שייך לשפה, יהיה לי בדיוק תא אחד בודד אשר ייצג לי את פעולת ה-Accept.

#### Goto Table

טבלת הGoto מייצגת עבור המנתח, מהו המצב אליו היא צריכה לחזור, לאחר ביצוע מהלך ה-Reduce, לפי המצב הנוכחי וה- LHS(Non Terminal) שבוא השתמשנו במהלך ה-Reduce.

כאשר הסברתי על הReduce, נתתי דוגמה לכן שניתן לדמיין כי אנחנו נסוגים אחורה באוטומט, מחליפים את רצף התווים שלנו בתוו אחר. ניתן לחשוב על טבלת הGoto כטבלה אשר אומרת לנו לאיזה מצב אנחנו צריכים לחזור לאחר ה-Reduce.

טבלת הGoto אומרת לנו את מצב אליו נחזור, ומכך לא תכיל שום ערך נוסף.

להלן דוגמה של Parsing Table. 

* טבלה זאת מכילה את הTerminal-ים(Token-ים): a, b, $
* טבלה זאת מכילה את הNon-Terminal-ים: A, S
* אסימון הדולר מסמן את סיום הקלט

### עץ הניתוח

עץ התחבירי (AST) מייצג את קלט הקוד במבנה היררכי המיצג את המבנה התחבירי של הקוד. הוא כמו מפת הדרכים של הקוד לאחר שעבר את תהליח ניתוח התחבירי.

כל צומת בעץ מייצגת חלק מהקוד, כמו חישוב, קריאה לפונקציה או הוראת בקרה. יכול להכיל Terminal או Non-Terminal. לכל צומת יכול להיות כמות שונה של בנים, על פי תחביר ומבנה השפה.

עץ הניתוח ילווה אותנו בשלבים הבאים של הקומפיילר ובעזרנו ניצור את קוד בשפת היעד.

להלן תמונה להמחשה את מבנה עץ הניתוח.

### מחסנית - Stack

כפי שכבר הסברתי, אלגוריתם הParsing אינו יכול להתקיים רק על ידי אוטומט סופי דטרמיניסטי(אס"ד), אלה הוא צריך גם לשמור נתונים על גבי מחסנית.

כל אלמנט הנשמר במחסנית מכיל את שני הנתונים הבאים:

1. המצב הנוכחי בוא אנו נמצאים
2. תמונה שמכילה תרשים, שרטוט, ציור

   תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי. מצביע לראש צומת של עץ ניתוח.

### אלגוריתם ניתוח התחבירי

כעת אציג עבורכם את האלגוריתם המלא בפסאודו קוד, המשתמש במחסנית והParsing Table.

### פסאודו קוד

1. **אתחל** את הParsing Table.

2. **אתחל** את המחסנית.

3. **צור** משתנים: v, s, t, e.

4. **צור** מצביע לעץ ניתוח p.

5. **דחוף**-למחסנית(0, Null).

6. **כל עוד** לא עברנו על כל אסימוני הקלט, **בצע**:

6.1. **הכנס** למשתנה t את האסימון הבא.

6.2. **הכנס** למשתנה e את הצץ-למחסנית().

6.3. **הכנס** למשתנה s את חלץ-ערך-מצב(e).

6.4. **אם** Action[s, t] = Shift v, **אז**:

6.4.1. **צור** צומת עץ ניתוח מסוג Terminal עם הערך t והכנס את כתובתו לp.

6.4.2. **דחוף** למחסנית(v, p).

6.5. **אחרת**, אם Action[s, t] = Reduce v, אז:

6.5.1. **מצא** את הProduction Rule במקום ה-v.

6.5.2. **הכנס** למשתנה t את ערך הLHS של אותו Production Rule.

6.5.3. **צור** צומת בעץ ניתוח מסוג Non-Terminal עם הערך t והכנס את כתובתו לp.

6.5.4. **עבור** מספר האלמנטים בRHS של אותו Production Rule, בצע:

6.5.4.1. **שלוף**-מהמחסנית() לתוך e.

6.5.4.2. **הכנס** חלץ-מצביע-לעץ(e) לתוך בן חדש של צומת p.

6.5.5. **הכנס** למשתנה e את הצץ-למחסנית().

6.5.6. **הכנס** למשתנה s את חלץ-ערך-מצב(e).

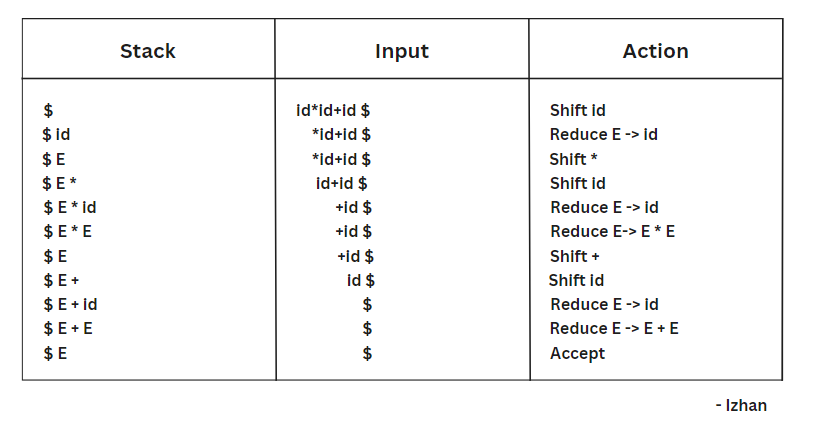
6.5.7. **הכנס** למשתנה v את הערך Goto[s, t].

6.5.8. **דחוף**-למחסנית(p, v).

6.6. **אחרת**, **אם** Action[s, t] = Accept, **אז**:

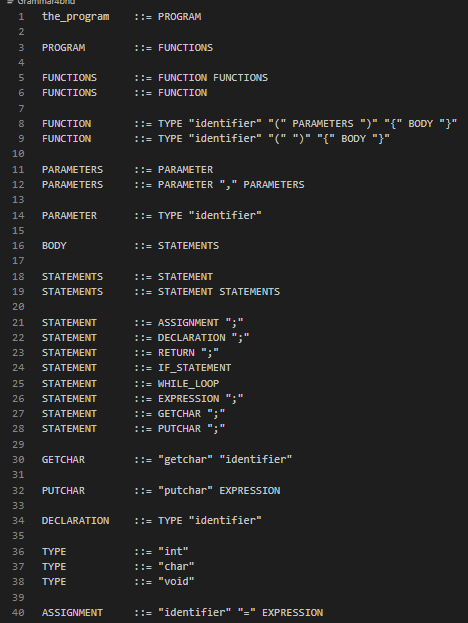
6.6.1. **שלוף**-מהמחסנית לתוך e.

6.7. **אחרת**, **דווח** על שגיאה



### MyC Grammar

תמונה שמכילה טקסט, צילום מסך, תפריט

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.להלן הgrammar של השפה המומצאת Mטבפורמט bnf.

תמונה שמכילה טקסט, צילום מסך, תוכנה, תכונות מולטימדיה

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

### Syntax diagram

להלן מופיע דיאגרמת התחביר עבור אותו grammar שהצגתי.

**THE PROGRAM:**



**PROGRAM:**

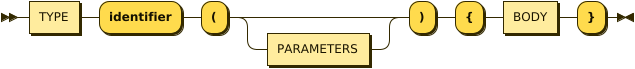


**FUNCTIONS:**

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

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

**FUNCTION:**

****

**PARAMETERS:**

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

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

**PARAMETER:**

****

**BODY:**

****

**STATEMENTS:**

****

**STATEMENT:**

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

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

**GETCHAR:**

****

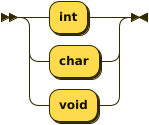
**PUTCHAR:**

****

**DECLARATION:**

****

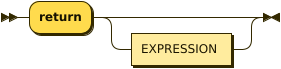
**TYPE:**

****

**ASSIGNMENT:**

****

**RETURN:**

****

**FUNCTION\_CALL:**

**תמונה שמכילה צילום מסך, גופן, לוגו, גרפיקה

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

**ARGUMENTS:**

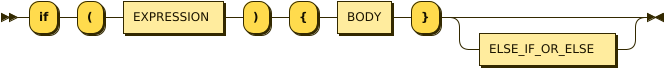
**תמונה שמכילה גופן, צילום מסך, גרפיקה, לוגו

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

**ARGUMENT:**

****

**IF\_STATEMENT:**

****

**ELSE\_IF\_OR\_ELSE:**

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

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

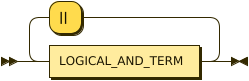
**WHILE\_LOOP:**

****

**EXPRESSION:**

****

**LOGICAL\_OR\_TERM:**

****

**LOGICAL\_AND\_TERM:**

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

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

**RELATIONAL\_TERM:**

**תמונה שמכילה צילום מסך, טקסט

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

**ADDITIVE\_TERM:**

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

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

**MULTIPLICATIVE\_TERM:**

**תמונה שמכילה צילום מסך, גופן, מספר

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

**FACTOR:**

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

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

**LITERAL:**

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

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

## ניתוח סמנטי – Semantic Analysis

כפי שכבר הסברתי, זהו השלב השלישי בתהליך הקומפילציה. בשלב זה הקומפיילר כבר הוכיח את נכונותו של קוד המקור מבחינה לשונית ותחבירית. אך דברים אלו לא מספיקים, ועדיין יש מקום לשגיאות בקוד. לדוגמה, מה אם הגדרתי משתנה מסוג void, מה אם הגדרתי פונקציה שמחזירה ערך מסוג integer, אך בתוכה אני לא מחזיר כלום, וכך העלה. כל השגיאות הללו נקראות שגיאות סמנטיות ובשלב זה המטרה של הקומפיילר הוא לוודאות כי אין שגיאות מסוג זה.

בשלב זה נשתמש עם עץ הAST שנוצר עליו, נעבור עליו ונבדוק את תקינותו.

אך לפני הכל, נצטרך לדעת מהם סוגי השגיאות הסמנטיות. בכדי שנוכל לזהותם בהמשך.

### שגיאות סמנטיות

כעת אעבור על כל השגיאות הסמנטיות האפשריות הנמצאות בשפת התכנות שהמצאתי - MyC.

#### הגדרת משתנה מסוג void

אי אפשר להגדיר משתנה מטיפוס void, כי void מציין העדר ערך ולא יכול לאחסן נתונים. כל ניסיון כזה יגרום לשגיאה בזמן הקומפילציה.

#### הגדרה כפולה של משתנה תחת אותו Scope

שני משתנים עם אותו השם באותו תחום יוצרים קונפליקט בזיהוי הסמל. יש להבטיח שמות ייחודיים לכל משתנה בתוך אותו Scope.

#### הגדרת פרמטר של פונקציה מסוג void

פרמטרים אינם יכולים להיות מטיפוס void, שכן אין משמעות לארגומנט שאינו מחזיק ערך. יש לבחור טיפוס תקף שמאפשר העברת ערכים.

#### קריאה לפונקציה לא מוגדרת

אם קוראים לפונקציה שלא הוכרזה או הוגדרה מוקדם יותר, הקומפיילר לא ימצא סימול מתאים לקישור. חובה להכריז את הפונקציה לפני הקריאה אליה.

#### יותר מדי ארגומנטים בקריאה לפונקציה

הבאת יותר ערכים מאלה שהוגדרו בפרמטרים יוצרת אי-התאמה בין קריאה להגדרה. יש להתאים במדויק את מספר הארגומנטים למספר הפרמטרים.

#### פחות מדי ארגומנטים בקריאה לפונקציה

אי סיפוק מספיק ערכים מאלה שהוגדרו בפרמטרים של הפונקציה יוצרת אי-התאמה בין קריאה להגדרה. יש להתאים במדויק את מספר הארגומנטים למספר הפרמטרים.

#### ארגומנט עם טיפוס שונה מהפרמטר המקביל אליו בקריאה לפונקציה

טיפוסי הארגומנטים חייבים להתאים לטיפוסי הפרמטרים כדי להבטיח שהערכים יפורשו נכון. העברת טיפוס שגוי עלולה לגרום להתנהגות בלתי צפויה או לקריסה.

#### שימוש עם משתנה שלא הוגדר

שימוש בשם משתנה שלא עבר הכרזה מונע הקצאה תקינה של זיכרון וגורם לשגיאה. יש להכריז ולהגדיר את המשתנה לפני כל שימוש בו.

#### החזרת טיפוס שלא תואם את הגדרת הפונקציה

כל פונקציה נדרשת להחזיר טיפוס מסוים התואם את סוג הטיפוס המוגדר בה.

#### הגדרה כפולה של שם פונקציה

שני הגדרות או הכרזות של פונקציה עם אותו השם באותו תחום יגרמו קונפליקט. יש להקפיד על ייחודיות שמות הפונקציות.

#### הגדרה כפולה של שם פרמטר בכותרת פונקציה

פרמטרים בכותרת הפונקציה צריכים להיות עם שמות שונים כדי למנוע בלבול. לא ניתן להגדיר שני פרמטרים בעלי אותו השם באותה פונקציה. דומה לאיסור הגדרת שני משתנים בעלי אותו שם תחת אותו Scope.

#### שם הפונקציה: 'main' לא קיימת

הפונקציה main" " היא נקודת הכניסה הנחוצה לתוכנית. העדר פונקציה זאת תמנע את הרצת התוכנית.

#### שימוש בפונקציה שמחזירה void בתוך ביטוי

פונקציה מטיפוס void אינה מחזירה ערך ולכן לא יכולה להשתלב בתוך ביטוי המחזיק תוצאה. יש לקרוא לה כהצהרה נפרדת ולא כחלק מחישוב ערך.

#### חריכה מכמות המקסימלית של פרמטרים בהגדרת פונקציה

בשפת מומצאת זאת – MyC, ישנה הגבלה על מספר הפרמטרים הניתן להגדיר לפונקציה, לא ניתן לעבור הגבלה זאת.

### Symbol Table

הקומפיילר נעזר במבנה אשר נקרא symbol table, מבנה זה אחראי על לעקוב ולשמור מידע עבור כל המשתנים, הפונקציות, והמזהים של התוכנית. הוא שומר את כל המידע עליהם, כמו : שם, סוג טיפוס, מיקום בזיכרון ועוד. מבנה זה משמש בכדי לעקוב ולוודות שימוש נכון במשתנים ופונקציות. טבלה זאת משתמש כלי מרכזי ונבנית בשלב תהליך הסמנטי.

### Scope

Scope הוא מבנה אשר מייצג לנו Scope מסוים בתוכנית. לכל Scope שמור בתוכו את הSymbol Table של כל המזהים.

כל Scope שומר אחד מין שני סוגי המזהים הבאים בטבלת הSymbol table שלו, על פי סוג הScope.

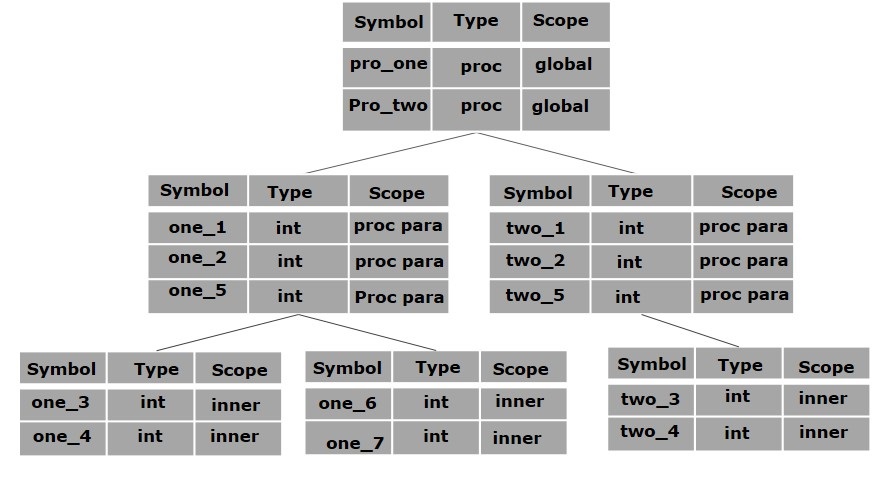
#### Global Scope

סקופ גלובלי אשר בוא נשמר כל הPrototype של הפונקציות, כמו שמם, ערך חזרה, הפרמטרים וטיפוסם. ישנו רק Scope גלובלי אחד ויחיד.

#### Local Scope

סקופ לוקלי הוא סקופ אשר שייך לפונקציה מסוימת, או לScope פנימי שלה. Scope זה שומר משתנים שהוגדרו תחת Scope זה בלבד.

כל Scope שומר את הScope החיצוני שלו, ואת כל הScope-ים הפנימיים שלו. כאשר אוסף כל הScope-ים נקראת Scope Tree.להלן דוגמה של Scope Tree.



### אסטרטגיה בשלב הסמנטי

כעת אסביר על האסטרטגיה לפייה אני מבצע את הבדיקות הסמנטיות.

ראשית אזכיר כי כל צומת בעץ התחביר שלנו מציין Terminal, או Non-Terminal. אם זהו Terminal, צומת זאת היא מסוג עלה(אין לה בנים).

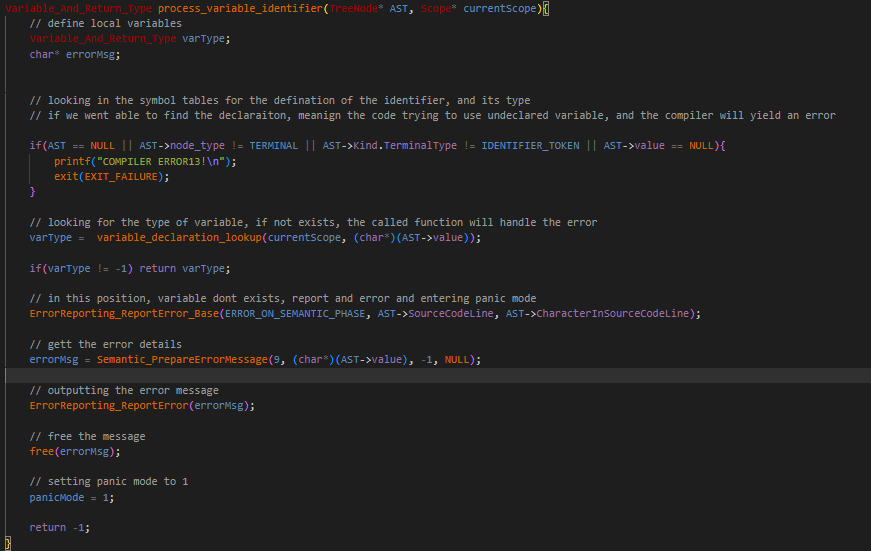
האסטרטגיה בנויה על רעיון די פשוט אך עוצמתי: לכל Production Rule שיש לנו בGrammar בין אם זה expression, if stmt וכך העלה, יהיה פונקציה סמנטית מתאימה. כל פונקציה כזאת תתנהג כפונקציה משגיחה, אשר תבדוק את תקינות הסמנטית בProduction rule המתאים. הפונקציה תקרא באופן רקורסיבי לפונקציות השונות לפי חוקי grammar השפה. על פי רעיון זה אנו מטיילים בעץ הניתוח שלנו, ובכל פעם בודקים את התקינות הסמנטית שלו.

דבר זה מעט מסובך להבנה, לכן אסביר זאת על ידי דוגמה:

להלן מופיע הפונקציה בקוד שלי אשר אחראית על בדיקת התקינות הסמנטית עבור לולאת while.

תמונה שמכילה טקסט, צילום מסך

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

להלן פונקציה מתוך התוכנית שלי אשר בודקת את התקינות הסמנטית כאשר אנו משתמשים עם מזהה(משתנה).

## יצירת הקוד – Code Generation

סוף סוף! הגענו אל(כמעט) השלב האחרון בפרויקט. זהו שלב הBack End. כפי שכבר הסברתי, בשלב זה, הוכחנו את נכונותו של קוד הקלט מבחינה מילונית, תחבירית וסמנטית. כעת אנו בטוחים בנכונות הקוד ומוכנים להמירו לקוד המקור.

מטרתנו בשלב זה הוא להמיר את עץ התחביר – AST Tree, להוראות בשפת האסמבלי.

האסמבלי אליו אמיר את הקוד הוא אסמבלי nasm 64 ביט.

### שימוש באוגרים

באסמבלי, בשונה משפות גבוהות, אנחנו עובדים מול אוגרים. אך יש לנו כמות מוגבלת של אוגרים, נרצה לעקוב אחר האוגרים אשר נמצאים בשימוש(איתם לא נרצה לעבוד) והאוגרים החופשיים, בהם נשתמש.

נצטרך לנהל את שימוש האוגרים בדרך בטוחה ויציבה.

תמונה שמכילה טקסט, צילום מסך, גופן, מספר

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.נבנה מבנה אשר יאפשר לנו להקצות אוגר פנוי, להלן תמונה לדוגמה.

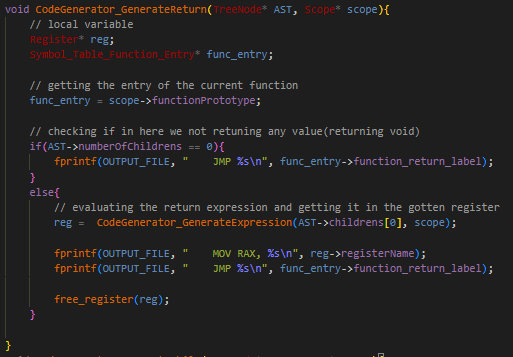
### אסטרטגיה ב-Code Generation

שלב זה די דומה לאופן שבוא פעלנו בשלב הקודם. בשלב זה אנו בונים פונקציות עזר עבור כל צומת אפשרית ב-AST. כל פונקציה תדע לטפל במקרה בצומת המתאימה ותקראה לפונקציות הבאות בהתאם. בשלב זה כבר הוכחנו את נכונותו של קוד המקור, ולכן בחלק זה אנו לא צריכים לבצע בדיקות נוספות.

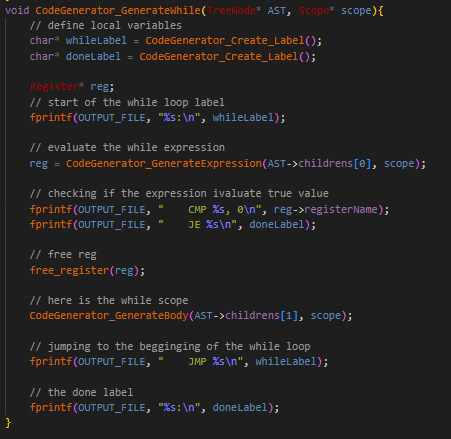
להלן דוגמה של מספר פונקציות:

#### CodeGenerator\_GenerateReturn

פונקציה זאת נקראת כאשר הגענו אל צומת ב-AST מסוג Non-Terminal RETURN. פונקציה זאת אחראית להשים את אוגר RAX בערך שהפונקציה מחזירה ולאחר מכן חוזרת מין הקוד(על ידי קפיצה לסוף הפונקציה).



#### CodeGenerator\_GenerateWhile

פונקציה זאת נקראת כאשר הגענו אל צומת ב-AST אשר מייצג Non-Terminal מסוג WHILE. פונקציה זאת אחראית על יצירת לולאת while. ראשית היא נכנסת ללולאה, בודקת אם התנאי מתקיים, אם מתקיים נשארת, אם לא היא קופצת החוצה.

## Error handling

לאחר שסקרתי והסברתי על כל תהליך הקומפילציה, בפרק זה אתן דגש ואסקור את האופן שבוא הקומפיילר יודע להגיב במצבים בהם התגלתה שגיאה.

כפי שכבר הסברתי, ישנם שלושה שלבים בהם הקומפיילר מזהה שגיאות: Lexical, Syntax, Semantic.

בכד אחד מהשלבי הקומפיילר, במידה והקומפיילר יזהה שגיאה הוא יכנס למצב panic mode ויסיים את התוכנית מיד לאחר סיום אלגוריתם השלב הנוכחי. זאת אומרת כי כאשר אנחנו נכנסים למצב panic mode לעולם לא נצליח לקמפל את קוד המקור.

### Error handling – Lexical

ארענן את זיכרוננו ואזכיר כי בשלב זה המטרה שלנו הוא לקטלג את כל קלט הקוד, מתצורה של תווי-תווי, לתצורה של אסימונים(Tokens). אל אסימון מיצג את מבנה הקוד ברמה הבסיסית ביותר. נשתמש במכונת מצבים בכדי לזהות את האסימונים.

כמו כן, בנינו מצבים אשר מייצגים מקרה של שגיאה, בהם האוטומט זיהה תוו או רצף של תווים לא חוקיים תחבירית. זיהוי שגיאה יהיה כאשר נגיע לאותם מצבים.

#### Lexer panic mode

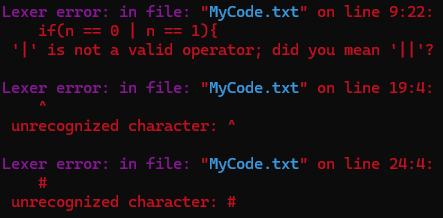
כאשר בשלב בLexer זיהינו שגיאה, התרענו זאת למשתמש, אך ברצוננו להמשיך בשלב זה ולנסות לאתר שגיאות מילוניות נוספות, ולאחר כך לסיים את התוכנית ללא הידור הקוד. דבר זה נקרא Panic mode.

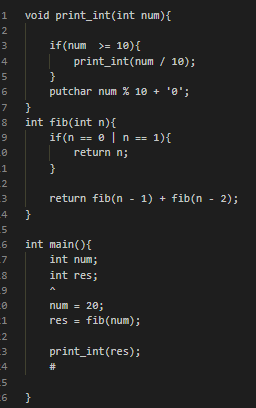
מייד לאחר שזיהינו שגיאה והתרענו למשתמש, לא נרצה ישר לחזור לריצה במכונת המצבים. לדוגמה, אם היה לי את רצף התווים הבא: retu^rn" ". מכונת המצבים הייתה מזהה בהצלחה שיש שימוש בתווי '^' אשר אינו מקובל בשפה, אך לא היינו רוצים שהיא תמשיך לעבד את הקלט, היינו רוצים שהיא תדלג על התווים הקרובים, עד אשר היינו מגיעים להתחלה של אסימון חדש.

בכדי לעשות זאת, נעשה כך שמקרה זה, לאחר שה Lexer - זיהה שגיאה, הוא "יתעלם" מהתווים הבאים, על הגעה לאחד מין התווי delimiter.

נתבונן בקטע הקוד הבא בשפת התכנות – MyC.

ונראה את פלט הקומפיילר, אשר זיהה שגיאות מילוניות.





### Error handling – Parsing

ארענן את זיכרוננו ואזכיר בשלב זה אנו ממירים את אוסף האסימונים שלנו אל עץ התחביר, שלב זה מכיל שימוש במכונת מצבים ומחסנית.

הדרך בא אנו מזהים שגיאה תחבירית היא כאשר אנו רוצים להתקדם באוטומט מחסנית, אך הגענו למצב לא אפשרי(מצב שלא הוגדר בטבלה, גם לא shift וגם לא reduce בparsing table). במקרה כזה נתריע למשתמש על בעיה תחבירית, וכעת נכנס למצב panic mode.

#### Parser panic mode

לאחר שזיהינו שגיאה והתרענו למשמש, לא נרצה באופן מידי לסיים את התוכנית, אלה נשאף להמשיך באלגוריתם וננסה לזהות בעיות תחביריות פוטנציאליות נוספות.

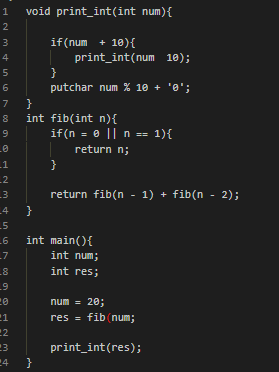
במצב הזה, הparser מתחיל "לדחות" ולדלג על אסימונים מהקלט עד אשר הוא נתקל בטוקן מסנכרן, כלומר טוקן שנחשב "עוגן" במבנה התחביר של השפה, כמו נקודה-פסיק, סוגריים סוגרים.

לאחר שמזהים טוקן מסנכרן, הפרסר מנקה את המחסנית על ידי שליפה של מצבים, עד אשר הוא מגיע למצב המאפשר פעולה חוקית (shift או reduce) עם האסימון המסנכרן. ברגע שזה קורה, הפרסר דוחף את המצב החדש אשר שנמצא על ידי האסימון המסנכרן וממשיך את האלגוריתם באופן רגיל במטרה ולמצוא עוד שגיאות.

אחדד ואסביר כי בשלב זה, לאחר panic mode לא מובטח לנו שנצליח לאתר את כל השגיאות התחביריות הפוטנציאליות שייתכן וקיימות בקוד המקור.

נתבונן בקטע הקוד הבא בשפת התכנות – MyC.

ונראה את פלט הקומפיילר, אשר זיהה שגיאות תחביריות.



תמונה שמכילה טקסט, צילום מסך, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

### Error handling - Semantic

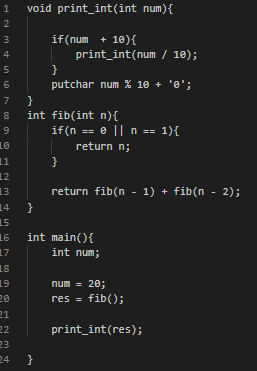
אזכיר כי בשלב זה המטרה היא לעבור על עץ הניתוח – AST ולבדוק את תקינות הקוד מבחינה סמנטית.

כאשר נזהה שגיאה, נדפיס התרעה למשתמש, נעבור למצב panic mode ונמשיך לעבד את העץ כרגיל. בחלק זה אין צורך בהתייחסות מיוחדת למצב בוא זיהינו שגיאה.

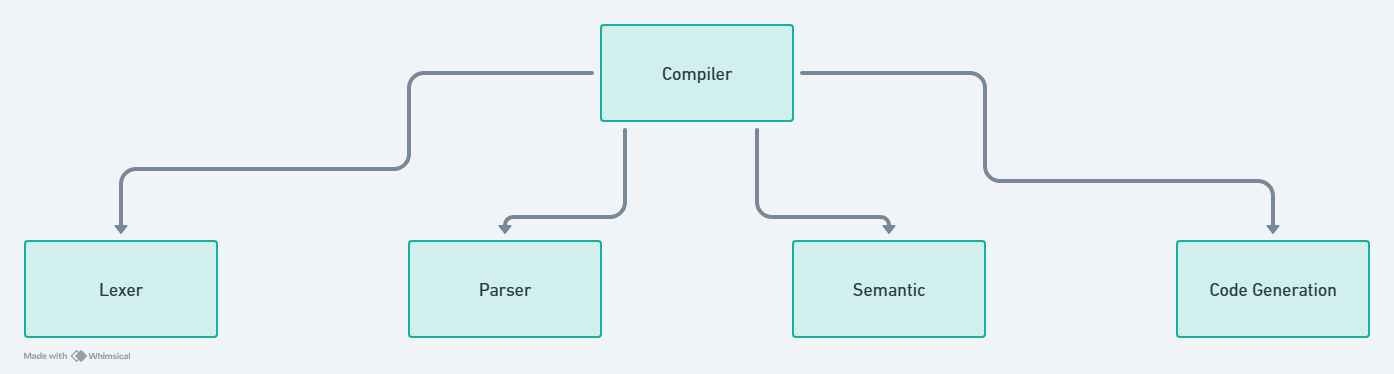
נתבונן בקטע הקוד הבא בשפת התכנות – MyC.

ונראה את פלט הקומפיילר, אשר זיהה שגיאות סמנטיות.

תמונה שמכילה טקסט, צילום מסך, גופן

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

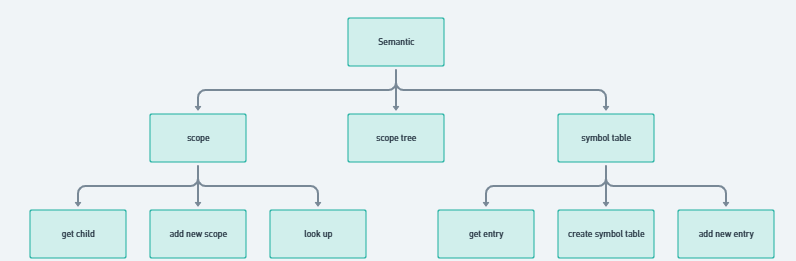
# ארכיטקטורה - Top-down Level Design



## תמונה שמכילה טקסט, צילום מסך, גופן, תרשים תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.Lexer

## תמונה שמכילה צילום מסך, תרשים, ריבוע, מלבן תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.Parser

## Semantic



## Code Generation

# UML Use Cases

# מבנה נתונים

בפרק זה אסביר על מבני הנתונים בהם השתמשתי לאורך הפרויקט בכל אחד מהשלבים.

## מבנה הנתונים בLexer-

אזכיר כי בשלב ניתוח המילוני המטרה שלנו זה להמיר את קוד הקלט, אשר רשום בצורת תווי-תווי לצורה של אסימון-אסימון. בשלב זה נדרשנו לממש אלגוריתם אשר יצליח ביעילות לזהות את האסימונים השונים.

ממשתי זאת על ידי שימוש באוטומט סופי דטרמיניסטי, וייצגתי אותו בעזרת מטריצה. מבנה נתונים זה אפשר לנו להתקדם באוטומט בפרק זמן O(1), כך שיעילות זמן הריצה של מציאת כל האסימונים היא ליניארית, O(n).

תמונה שמכילה תרשים, צילום מסך, טקסט, צבעוני

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.כאשר אני מייצר כל אסימון, אני מצרך אותו לתור מערך של אסימונים. כל אסימון ממוקם לפי הזמן שבוא הוא נמצא. כאשר השלב השני מעבד את המערך, הוא מבצע סריקה, עובר משמאל לימין, כך שיוצא כי מעבר על כל האסימונים יוצא ביעילות זמן ריצה ליניארית, כלומר O(n).

## מבנה הנתונים ב-Parser

אזכיר כי בשלב ניתוח התחבירי המטרה שלנו הוא להמיר את רצץ האסימונים לעץ תחביר אשר מייצג לנו את קלט הקוד בצורה ההיררכית המותאמת לשפה שלנו, על פי הכללים התחביריים של השפה.

כדי לבצע ניתוח תחבירי, השתמשתי במודל של אוטומט מחסנית המיוצג באמצעות גרף. כל צומת בגרף מייצג נקודת זמן מסוימת בתהליך הניתוח, הכוללת את המצב הנוכחי, המיקום בקוד, והכללים התחביריים הרלוונטיים. המעברים בין הצמתים, כלומר הקשתות בגרף, מיוצגים על ידי ה-Tokens שהתקבלו מהמנתח המילוני.

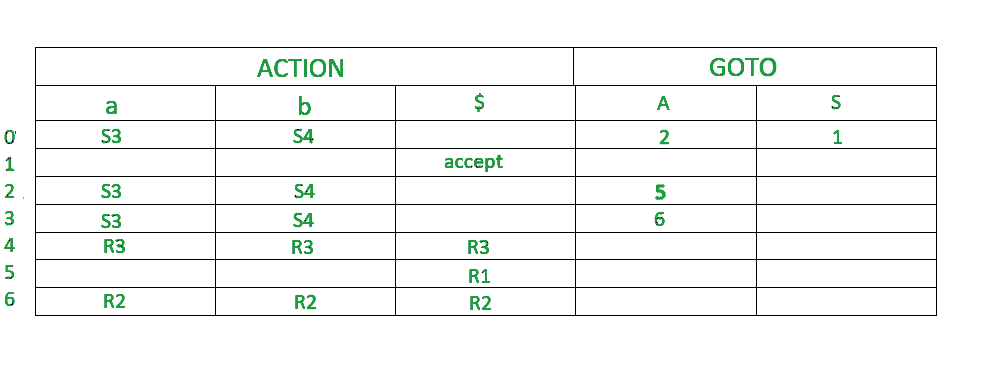
בכל רגע נתון, בהתבסס על המצב הנוכחי ועל ה-Token הבא מהקלט, המערכת קובעת את הפעולה שיש לבצע: Shift, Reduce, Accept או Error. במידה וה-Token הבא אינו צפוי בהתאם לחוקי השפה, המערכת תעבור למצב שגיאה.

מבנה נתונים זה מאפשר לקבל החלטות באופן עקבי בכל שלב בתהליך הקומפילציה. אין צורך בתשאול או בחינת מספר אפשרויות, דבר התורם לשמירה על יעילות זמן ריצה לינארית (O(n)).

### Parsing Table

על מנת לממש את הגרף השתמשתי בשני מטריצות: Goto Table, Action Table. שני הטבלאות יחד מרכיבות לי את הParsing Table. אודות לטבלה זאת, יכולתי לבצע החלטות דטרמיניסטיות עבור תהליך ניתוח התחביר.

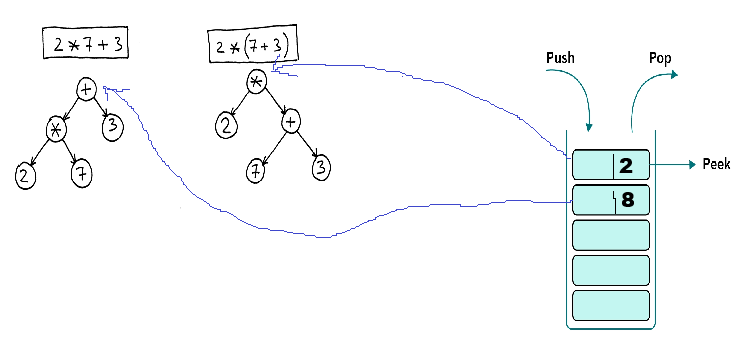
גישה לכל מקום במטריצה היא ביעילות זמן קבועה, משמע O(1). כך שהשימוש בטבלאות אלו במהלך תהליך ניתוח התחביר היה ביעילות זמן ליניארית, O(n).



### Stack

כמו כן ציינתי שהשתמשתי באוטומט מחסנית, לכן לא יכולתי להסתפק במטריצה, אשר מייצגת לי אוטומט סופי דטרמיניסטי, נדרשתי למחסנית.

המחסנית מחזיקה לי שני סוגי נתונים תחת כל אלמנט. הראשון הוא מספר מסוג שלם, השני הוא מצביע לעץ התחביר.

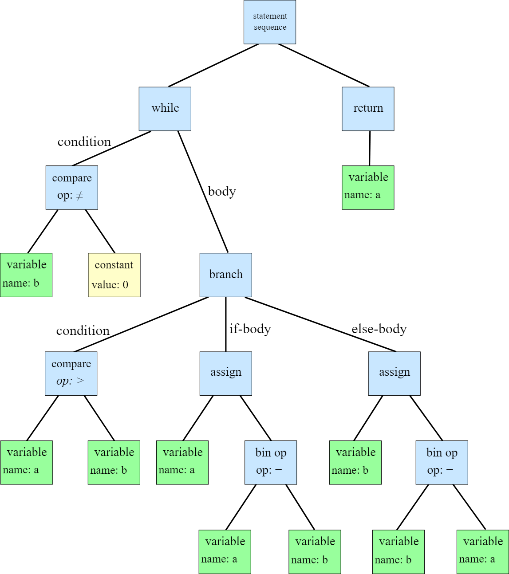


שימוש זה מאפשר לי להכניס ולהוציא אלמנטים מין המחסנית ביעילות זמן ריצה קבועה, כלומר O(1).

### Parsing tree

הParsing Tree הוא מבנה הנתונים אשר מאפשר לנו לייצג את קלט הקוד בצורה המתאימה לתחביר של השפה שהמצאנו.

לכל צומת בעץ ייתכן ויהיה מספר שונה של בנים, דבר זה תלוי בסוג הצומת ומבנה הקלט. כאשר אנו עוברים על עץ התחביר, אנו מבקרים בכל צומת בדיוק פעם אחת, ומך סיבוכיות זמן הריצה היא ליניארית, כלומר O(n).



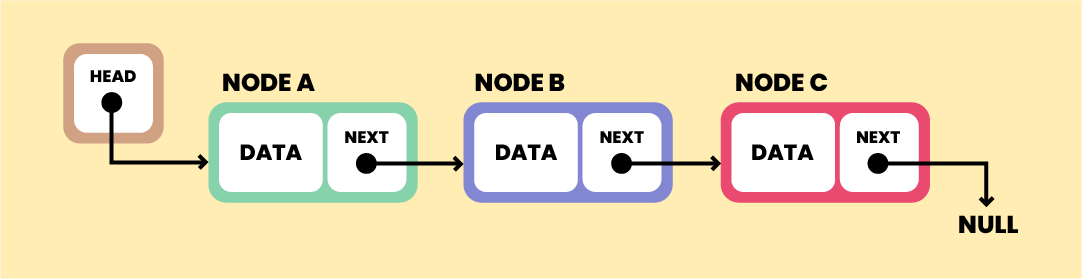
## מבנה הנתונים ב-Semantic

אזכיר כי בשלב ניתוח הסמנטי המטרה שלנו היא לוודא כי הקוד שלנו עומד בדרישות הסמנטיות של השפה. אופן הפעולה בו אנחנו בודקים דבר זה הוא לטייל על גבי ה-AST ותוך כדי בנייה ושימוש במבנה הנתונים – ScopeTree. בשלב הקודם סיקרתי את מבנה הנתונים של עץ הניתוח(AST), ובחלק זה אסקור את מבנה ה Scope Tree.

### Symbol Table

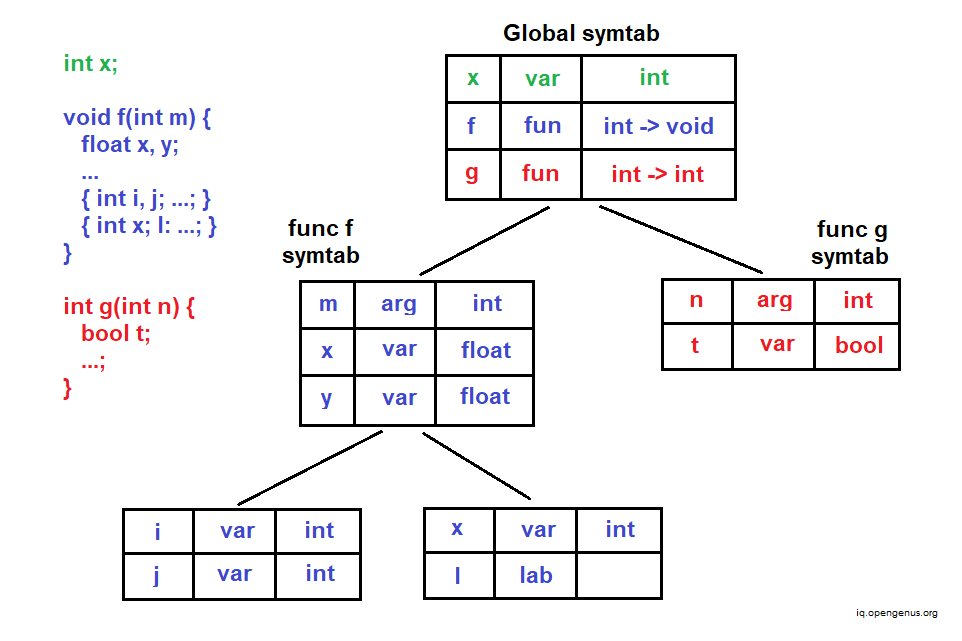
כפי שכבר הסברתי, הsymbol table הוא מבנה אשר נמצא בכל אחד מהscope-ים. במבנה זה נשמר הנתונים עבור כל אחד מהמזהים שבתוכנית, הנמצאים תחת אותו scope. הsymbol table מורכב ממספר שדות ומרשימה מקושרת.

כל חולייה ברשימה המקושרת מכילה מזהה ומידע עליו.

מכך, כאשר אנו רוצים לחפש מזהה תחת איזושהי טבלה סימבולית, נוכל לבצע זאת ביעילות של O(n).

### Scope Tree

כפי שכבר הסברתי, הScope Tree הוא למעשה מבנה עץ אשר כל חולייה היא מסוג Scope. כאשר אנו רוצים לחפש איזשהו מזהה, נחפש אותו בsymbol table של הScope הנוכחי, או בScope שהוא האבא של הScope הנוכחי, וכך העלאה. נתייחס למקרה הממוצע ונניח כי יש כמות (יחסית) קבועה של scope-ים, כך שיעילות זמן הריצה עבור ריצה בין scope לאביו הוא O(1).

כך שסך הכך, כאשר אנו מבצעים look up ומחפשים משתנה, יעילות זמן הריצה היא O(n).

# תיאור סביבת העבודה ושפות תכנות

## Visual-studio-codeסביבת העבודה:

### Operating System

Windows Operating System 11

### Code Editor

Microsoft Visual Studio Code

### C Compiler

Gcc

## Introduction to C Programming Lab Manual for VTU (BESCK104E/204E) - MyBlogosphereשפות תכנות

C Programing Language

# אלגוריתם ראשי

## האלגוריתם הראשי - Lexer

1. **אתחל** את מטריצת הסמיכויות לפי אוטומט השפה.
2. **עבור** לצומת ההתחלה – q0.
3. **כל עוד** לא הגעת לצומת שהוא עלה, בצע:
   1. **עבור** לצומת הבאה לפי הצומת והתווי הנוכחי.
   2. **קדם** את הקלט בתווי אחד.
4. **צור** אסימון לפי מספר העלה.
5. **החזר** את האסימון.

## האלגוריתם הראשי – Parser

1. **אתחל** את הParsing Table.

2. **אתחל** את המחסנית.

3. **צור** משתנים: v, s, t, e.

4. **צור** מצביע לעץ ניתוח p.

5. **דחוף**-למחסנית(0, Null).

6. **כל עוד** לא עברנו על כל אסימוני הקלט, **בצע**:

6.1. **הכנס** למשתנה t את האסימון הבא.

6.2. **הכנס** למשתנה e את הצץ-למחסנית().

6.3. **הכנס** למשתנה s את חלץ-ערך-מצב(e).

6.4. **אם** Action[s, t] = Shift v, **אז**:

6.4.1. **צור** צומת עץ ניתוח מסוג Terminal עם הערך t והכנס את כתובתו לp.

6.4.2. **דחוף** למחסנית(v, p).

6.5. **אחרת**, אם Action[s, t] = Reduce v, אז:

6.5.1. **מצא** את הProduction Rule במקום ה-v.

6.5.2. **הכנס** למשתנה t את ערך הLHS של אותו Production Rule.

6.5.3. **צור** צומת בעץ ניתוח מסוג Non-Terminal עם הערך t והכנס את כתובתו לp.

6.5.4. **עבור** מספר האלמנטים בRHS של אותו Production Rule, בצע:

6.5.4.1. **שלוף**-מהמחסנית() לתוך e.

6.5.4.2. **הכנס** חלץ-מצביע-לעץ(e) לתוך בן חדש של צומת p.

6.5.5. **הכנס** למשתנה e את הצץ-למחסנית().

6.5.6. **הכנס** למשתנה s את חלץ-ערך-מצב(e).

6.5.7. **הכנס** למשתנה v את הערך Goto[s, t].

6.5.8. **דחוף**-למחסנית(p, v).

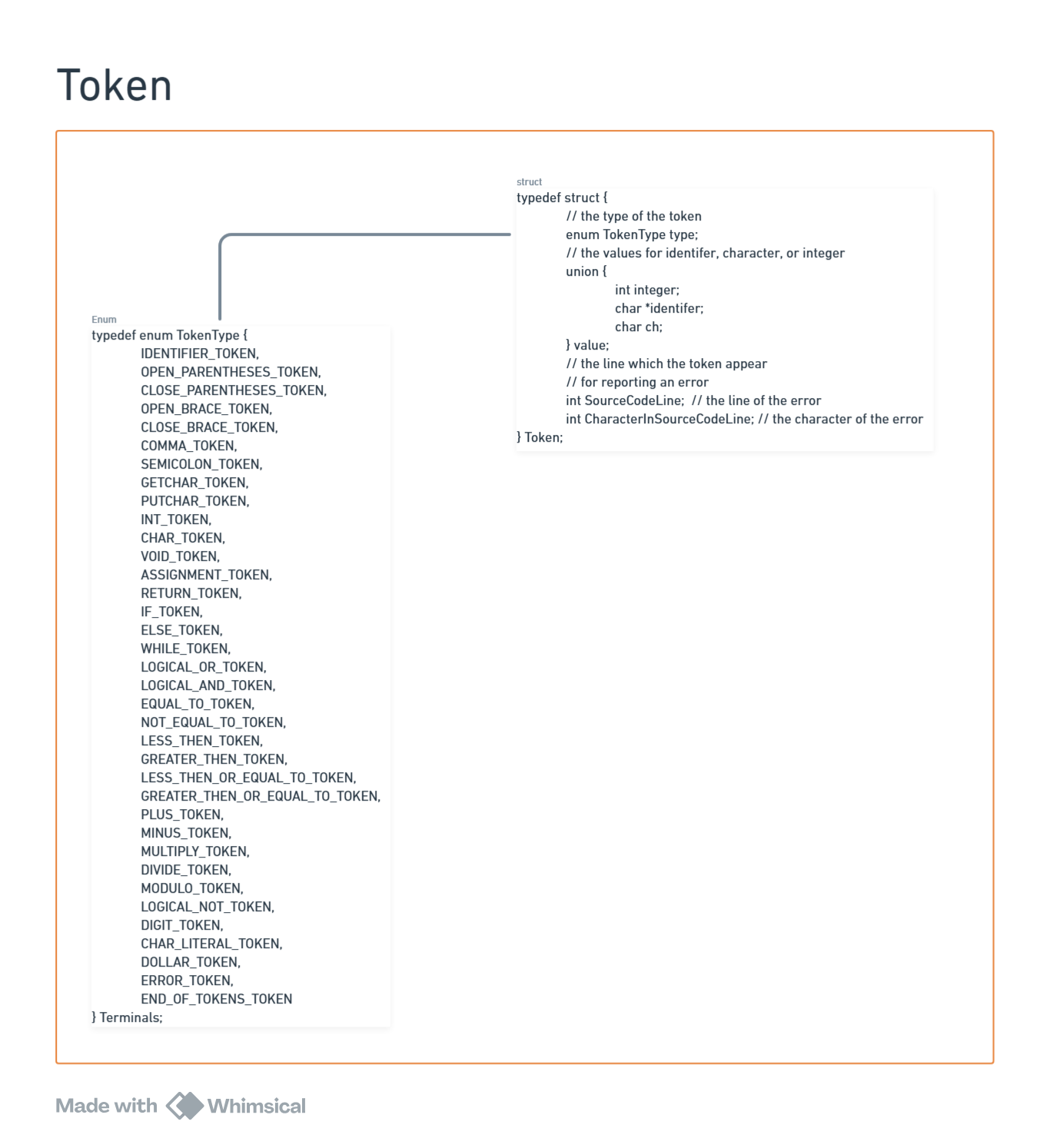
6.6. **אחרת**, **אם** Action[s, t] = Accept, **אז**:

6.6.1. **שלוף**-מהמחסנית לתוך e.

6.7. **אחרת**, **דווח** על שגיאה

# UML Class Diagram

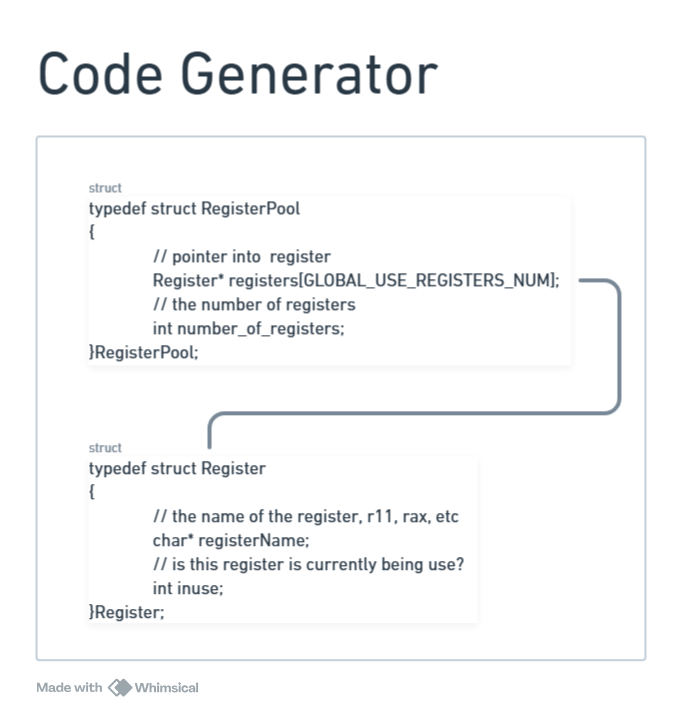
להלן תרשים הClass UML. תרשים זה אינו נכנס בצורה נוחה, לכן אמליץ לראותו דרך whimsical. קישור נמצא בנספחים.



תמונה שמכילה טקסט, צילום מסך, גופן, מספר

תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.

# תמונה שמכילה טקסט, צילום מסך, גופן, עיצוב תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.תמונה שמכילה טקסט, צילום מסך, גופן, עיצוב תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.





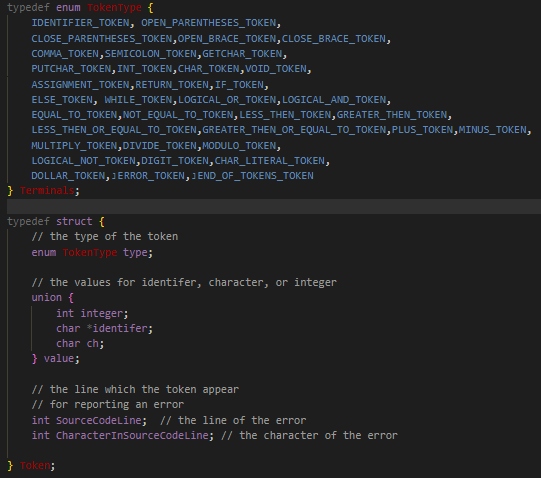
# 

# תיאור המחלקות הראשיות בפרויקט

בפרק זה אסקור את המודלים השנים ואפרט על מה תפקידם, אסביר על מבני ה - struct-ים, ועל הפונקציות הראשיות של כל מודל.

## Token

מודל המגדיר את המבנה של אסימון.

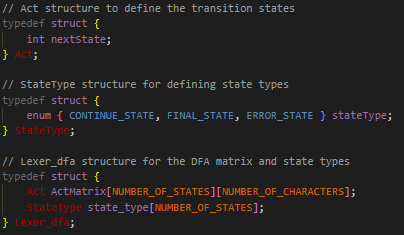


למבנה זה אין פונקציות עיקריות.

## Lexer\_dfa

מודל זה מגדיר את במבנה של האוטומט סופי דטרמיניסטי המשתמש בכדי לבצע אלגוריתם חלק זה.

### struct



### פונקציות

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| init\_dfa | - | Lexer\_dfa\* | הפונקציה מייצרת Lexer\_dfa חדש | O(1) |
| init\_lexer\_dfa | Lexer\_dfa\* | - | הפונקציה מאתחלת את הLexer לפי האוטומט | O(1) |
| getTokenType | int | Terminals | הפונקציה מחזירה את ערך הEnum התואם למצב הסופי באוטומט לפי מספר המצב | O(1) |
| getAct | Lexer\_dfa\*, int, char | Act | הפונקציה מחזירה את הAct הבא שיש לבצע לפי המצב הנוכחי באוטומט והתוו הבא | O(1) |
| getStateType | Lexer\_dfa\*, int | StateType | הפונקציה מחזירה את סוג המצב לפי מספר מצב הקלט | O(1) |

## Lexer

מודל זה אחראי על מציאת רצף האסימונים המרכיבים את קוד הקלט.

לחלק זה אין מבנה struct, אלה הוא משתמש עם מבני ה: Lexer\_dfa, Token.

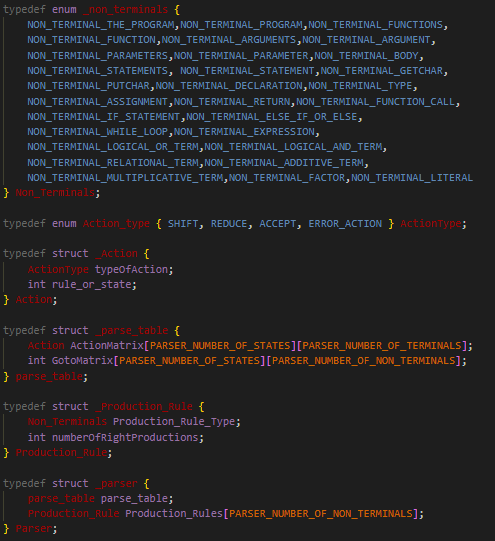
### פונקציות

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| getTokens | FILE\* | Token\* | הפונקציה מקבלת מצביע לקובץ המקור, מעבדת אותו ומחזירה את רצץ האסימונים כמערך | O(n) |
| addNewToken | Token\*\*, int\*, Terminals, void\*, int, int | - | הפונקציה מקבלת מערך אסימונים ונתוני אסימון חדש, הפונקציה מייצרת אסימון חדש ומוסיפה אותו לסוף המערך | O(1) |
| Lexer\_PrepareErrorMessage | Int, char | Char\* | הפונקציה מקבלת מספר המייצג מצב סופי מסוג שגיאה ותוו, הפונקציה מייצרת ומחזירה את מחרוזת הודעת השגיאה | O(1) |

## ParserTable

מבנה זה מחזיר את הParsing Table של הParser ומהווה את הליבה של אלגוריתם הParsing. מבנה זה מהווה את האוטומט סופי דטרמיניסטי בשלב הParsing.

### struct



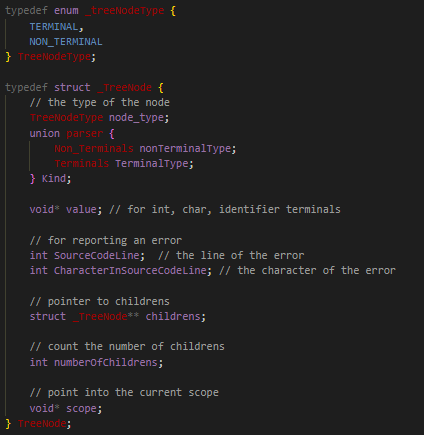
### פונקציות

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| init\_action\_table | Parser\* | - | הפונקציה מקבלת את הParser ומאתחלת את Action Table שלו | O(1) |
| init\_goto\_table | Parser\* | - | הפונקציה מקבלת את הParser ומאתחלת את Goto Table שלו | O(1) |
| init\_Production\_Rules | Parser\* | - | הפונקציה מקבלת את הParser ומאתחלת את רשומות הProduction rules שלו |  |

## ParserTree

מבנה זה אחראי על ייצוג קוד המקור, מייצג את עץ התחביר.

### struct



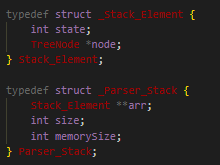
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| parser\_treeNode\_create\_TreeNode | - | TreeNode\* | הפונקציה מקצה זיכרון ומייצרת ומחזירה מבנה חדש מסוג TreeNode | O(1) |
| parser\_treeNode\_addChild | TreeNode\*,TreeNode\*, int | - | הפונקציה מקבלת שני צמתי עץ, משימה את הצומת השנייה בכן של הצומת הראשונה | O(1) |
| parser\_treeNode\_set\_type | TreeNode\*, TreeNodeType | - | הפונקציה מגדירה את סוג הצומת | O(1) |
| parser\_treeNode\_set\_terminal | TreeNode\*, TreeNodeType | - | הפונקציה מגדירה את סוג הTerminal של הצומת | O(1) |
| parser\_treeNode\_set\_non\_terminal | TreeNode\*, TreeNodeType | - | הפונקציה מגדירה את סוג הNon Terminal של הצומת | O(1) |

### פונקציות

## ParserStack

מבנה זה הוא מחסנית אשר שומרת צומת עץ ניתוח ומספר שלם כאלמנט מחסנית. מהווה חלק חשוב בשלב הניתוח והופכת את האוטומט סופי דטרמיניסטי לאוטומט מחסנית.

### Struct



### פונקציות

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| parser\_stack\_createStack | - | Parser\_Stack\* | הפונקציה מקצה זיכרון מייצרת ומחזירה מבנה parsing stack חדש | O(1) |
| parser\_stack\_push | Parser\_Stack \*, Stack\_Element \* | - | הפונקציה דוחפת איבר חדש למחסנית | O(1) |
| parser\_stack\_pop | Parser\_Stack \* | Stack\_Element\* | הפונקציה שולפת את האיבר הכי עליון מהמחסנית | O(1) |
| parser\_stack\_top | Parser\_Stack \* | Stack\_Element\* | הפונקציה מחזירה את האיבר הכי עליון מהמחסנית ללא שליפתו | O(1) |

## Parser

מודל זה אחראי על מימוש האלגוריתם של השלב השני בקומפיילר. מודל זה משתמש עם שאר המודלים בנמצאים תחת אותו שלב((Parsing table, TreeNode, parser\_stack, ולכן למודל זה אין מבנה משלו.

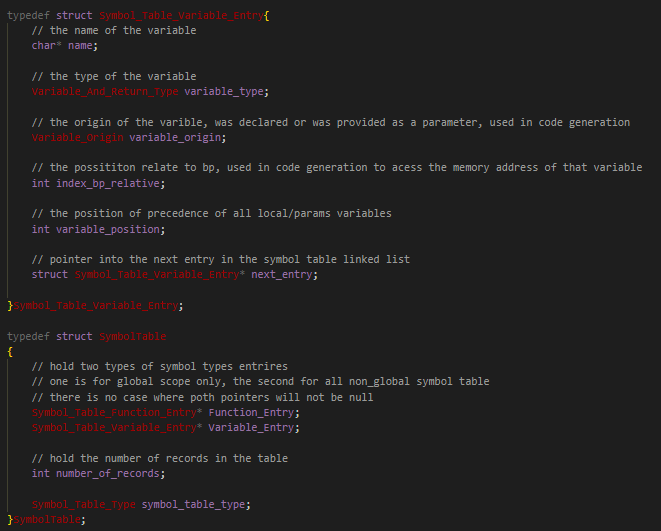
### פונקציות

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| parser\_init | - | Parser\* | הפונקציה מקצה זיכרון ומייצרת מבנה Parser חדש | O(1) |
| parser\_get\_action | Parser\*, int, Terminals | Action | הפונקציה מחזירה את המהלך הבא שיש לעשות בparsing לפי המצב והTerminal הנוכחי | O(1) |
| parser\_get\_goto | Parser\*, int, Non\_Terminals | int | הפונקציה מחזירה את המצב הבא שיש ללכת אליו לאחר ביצוע reduce | O(1) |
| parser\_shift | Parser\_Stack\*, Token\*, int | - | הפונקציה מבצעת את מהלך הshift באלגוריתם | O(1) |
| parser\_reduce | Parser\*, Parser\_Stack\*, int | - | הפונקציה מבצעת את מהלך הreduce באלגוריתם | O(1) |
| parser\_accept | Parser\_Stack\* | TreeNode\* | הפונקציה מבצעת את מהלך הaccept באלגוריתם, מחזירה את עץ הניתוח | O(1) |
| parser\_recovery\_error | Parser\*, Parser\_Stack\*, Token\*\* | - | הפונקציה אחראית על טיפול התמודדות עם זיהוי שגיאה בכדי להמשיך את הparsing בpanic mode | O(n) |
| Parser\_PrepareErrorMessage | Parser\*, int,Token\* | Char\* | הפונקציה אחראית על יצירת הודעת שגיאה מותאמת עבור מקרה בוא זיהינו שגיאה תחבירית | O(n) |
| create\_abstact\_syntax\_tree | Parser\*, Token\* | TreeNode\* | הפונקציה הראשית של אלגוריתם הניתוח, מחזירה את עץ ה – AST | O(n) |

## Symbol Table

מבנה זה אחראי על שמירת נתונים על מזהים המופיעים תחת אותו Scope. כל מזהה נשמר בטבלה כחוליה בתוך רשימה מקושרת. כל symbol table יכול לשמור מזהים מסוג פונקציה או מזהים מסוג משתנה.

### תמונה שמכילה טקסט, צילום מסך, גופן, תוכנה תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.תמונה שמכילה טקסט, צילום מסך, גופן תוכן שנוצר על-ידי בינה מלאכותית עשוי להיות שגוי.struct



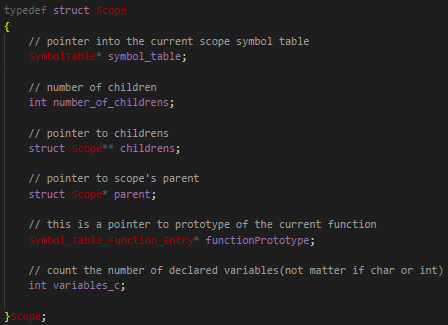
### פונקציות

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| create\_symbol\_table | Symbol\_Table\_Type | SymbolTable\* | הפונקציה מקצה זיכרון ומייצרת מבנה SymbolTable חדש | O(1) |
| create\_parameter | Variable\_And\_Return\_Type | Function\_Parameter\* | הפונקציה מקצה זיכרון ומייצרת מבנה Function\_Parameter חדש | O(1) |
| add\_function\_record  \_into\_global\_symbol  \_table | SymbolTable\*, Variable\_And\_Return\_Type, char\*, Function\_Parameter\*, int | Symbol\_Table\_Function\_Entry | הפונקציה מייצרת חולייה חדשה המייצגת פונקציה ברשימת הפונקציות של הsymbol table הגלובלית | O(1) |
| add\_variable\_record  \_into\_local\_symbol\_table | SymbolTable\*, Variable\_And\_Return\_Type, char\*, Variable\_Origin | - | הפונקציה מייצרת חולייה חדשה המייצגת משתנה ברשימת המשתנים של הsymbol table הלוקלי | O(1) |
| isVariableExistsInTable | SymbolTable\*, char\* | int | הפונקציה בודקת אם שם המשתנה שקיבלה קיים בטבלה | O(n) |
| isFunctionExistsInTable | SymbolTable\*, char\* | int | הפונקציה בודקת אם שם הפונקציה שקבלה קיימת בsymbol table הגלובלית | O(n) |
| get\_Varaible\_Type\_From\_Table | SymbolTable\*, char\* | Variable\_And\_Return\_Type | הפונקציה מחזירה את סוג הטיפוס של שם המשתנה שקיבלה | O(n) |

## Scope

מודל זה הוא מבנה אשר תפקידו הוא לנהל את הScope-ים שיש בתוכנית וההיררכיה בניהם. מבנה זה מאפשר לנו וידוי כי בקוד המקור אנו לא מבצעים מהלך לא חוקי מבחינת שימוש במשתנים.

### struct



### פונקציות

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| create\_Functionn\_Params | TreeNode\*, int, int\* | Function\_Parameter\* | הפונקציה משתמשת עם עץ הניתוח ויוצרת את הרשימה המקושרת של הפרמטרים של הפונקציה | O(n) |
| create\_global\_scope | - | Scope\* | הפונקציה מקצה זיכרון ומייצרת scope גלובלי חדש | O(1) |
| create\_local\_scope | - | Scope\* | הפונקציה מקצה זיכרון ומייצרת scope לוקלי חדש | O(1) |

## ScopeTree

מודל זה משתמש עם מבנה ה-Scope וה-Symbol Table בכדי לבצע את אלגוריתם הסמנטי בקומפיילר. אי לכך, למודל זה אין מבנה struct.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| process\_program | TreeNode\*, Scope\*\* | - | זוהי הפונקציה הראשית בשלב ניתוח הסמנטי, היא קוראת לשאר הפונקציות בהתאם | O(n) |
| Semantic\_PrepareErrorMessage | int, char\*, int, char\* | Char\* | פונקציה זו מייצרת את הודעות השגיאה המותאמות | O(1) |
| process\_statement | TreeNode\*, Scope\* | - | פונקציה זו מטפלת בבדיקה הסמנטית עבור statement | O(1) |
| process\_declaration | TreeNode\*, Scope\* | - | פונקציה זו מטפלת בבדיקה הסמנטית עבור הגדרת משתנה | O(1) |
| process\_while | TreeNode\*, Scope\* | - | פונקציה זו מטפלת בבדיקה הסמנטית עבור לולאת while | O(1) |
| process\_expression | TreeNode\*, Scope\* | Variable\_And\_Return\_Type | פונקציה זו מטפלת בבדיקה הסמנטית עבור expression | O(1) |
| process\_if | TreeNode\*, Scope\* | - | פונקציה זו מטפלת בבדיקה הסמנטית עבור תנאי אם | O(1) |
| process\_putchar | TreeNode\*, Scope\* | - | פונקציה זו מטפלת בבדיקה הסמנטית עבור הדפסת תווי למסך | O(1) |

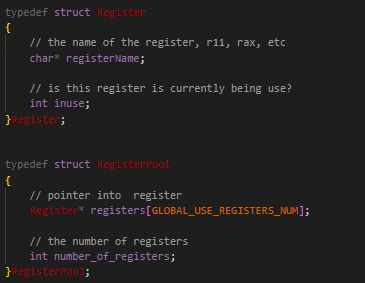
### פונקציות

## Code Generator

מודל זה אחראי על יצירת קוד היעד – nasm x64. משתמש במבנה הTreeNode- ו-Scope.

למודל זה יש מבנה נתונים קטן, אשר באמצעותו אנו מקצים ומשתמשים עם האוגרים הכלליים(r8 – r15).

### Struct



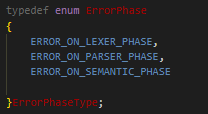
### פונקציות

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| CodeGenerator\_CreateRegister | Char\*, int | Register\* | הפונקציה מקצה זיכרון ומייצרת מבנה Register  חדש | O(1) |
| CodeGenerator\_InititeRegistersPool | - | - | הפונקציה מאתחלת את בריכת האוגרים שעל ידה נקצה ונשחרר שימוש באוגרים | O(1) |
| allocate\_register | - | Register\* | הפונקציה מחזירה מבנה Register פנוי הניתן לשימוש | O(1) |
| free\_register | Register\* | - | הפונקציה מגדירה את אוגר הקלט כאוגר חופשי | O(1) |
| CodeGenerator\_Create\_Label | - | Char\* | הפונקציה מייצרת ומחזירה שם label ייחודי חדש | O(1) |
| CodeGenerator\_FetchFunctionEntry | SymbolTable\*, char\* | Symbol\_Table\_Function\_Entry\* | הפונקציה מחזירה את הentry של הפונקציה אשר שמה זהה למחרוזת הקלט | O(n) |
| CodeGenerator\_FetchVariableEntry | SymbolTable\*, char\* | Symbol\_Table\_Variable\_Entry\* | הפונקציה מחזירה את הentry של המשתנה אשר שמו זהה למחרוזת הקלט | O(n) |
| CodeGenerator\_Generate | TreeNode\*, Scope\*, FILE\* | - | הפונקציה הראשית בתוכנית, אחראית ליצירת קוד היעד וקריאה לשאר הפונקציות באופן דמוי רקורסיה | O(1) |
| CodeGenerator\_GenerateFunction | TreeNode\*, Scope\* | - | הפונקציה אחראית על יצירת קוד היעד עבור פונקציה | O(1) |
| CodeGenerator\_GenerateIf | TreeNode\*, Scope\* | - | הפונקציה אחראית על יצירת קוד המקור המייצר תנאי אם | O(1) |
| CodeGenerator\_GenerateDeclaration | TreeNode\*, Scope\* | - | הפונקציה אחראית על יצירת קוד המקור המייצר הגדרת משתנה | O(1) |
| CodeGenerator\_GenerateExpression | TreeNode\*, Scope\* | Register\* | הפונקציה אחראית על יצירת קוד המקור המייצר ביטוי מתמטי. הפונקציה מחזירה את האוגר המחזיר בערך הביטוי | O(1) |
| CodeGenerator\_GenerateFunctionCall | TreeNode\*, Scope\* | Register\* | הפונקציה אחראית על יצירת קוד המקור המייצר קריאה לפונקציה\פרוצדורה. הפונקציה מחזירה את האוגר המחזיר בערך הביטוי שהוחזר מין הפונקציה(במידה ולא מחזירה void) | O(1) |
| CodeGenerator\_GeneratePutchar | TreeNode\*, Scope\* | - | הפונקציה אחראית על יצירת קוד היעד עבור הדפסת תווי לconsole. | O(1) |
| CodeGenerator\_GenerateWhile | TreeNode\*, Scope\* | - | הפונקציה אחראית על יצירת קוד היעד עבור לולאת while | O(1) |
| CodeGenerator\_GenerateReturn | TreeNode\*, Scope\* | - | הפונקציה אחראית על יצירת קוד היעד עבור החזרת ערך מתוך פונקציה(במידה ולא מחזירה void) | O(1) |

## ErrorReporting

קובץ זה אחראי על הדפסת הודעות השגיאה במקרה ויש שגיאה בקוד המקור בשלבי :Lexer, parser, semantic.

לחלק זה אין מבנה נתונים. אלה רק מגדיר Enum קטן.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| שם הפונקציה | קלט הפונקציה | פלט הפונקציה | תיאור הפונקציה | יעילות הפונקציה |
| ErrorReporting\_ReportError\_Base | ErrorPhaseType, int, int | - | זוהי הפונקציה אשר מדפיסה את חלק הכללי בהודעת השגיאה(מס' שורה, מס' תווי, תוכן שורה וכו'). | O(1) |
| ErrorReporting\_ReportError | Char\* | - | זוהי הפונקציה אשר מדפיסה את ההודעה המותאמת עבור כל שגיאה | O(1) |
| ErrorReporting\_PrintSourceCodeLine | int | - | פונקציה זו מדפיסה את תוכן השורה הרצויה בקוד המקור. | O(1) |
| ErrorReporting\_Switch\_To\_Red | - | - | פונקציה זו משנה את צבע פלט הConsole לאדום | O(1) |
| ErrroReporting\_Switch\_To\_Purpule | - | - | פונקציה זו משנה את צבע פלט הConsole לסגול | O(1) |
| ErrroReporting\_Switch\_To\_Cyan | - | - | פונקציה זו משנה את צבע פלט הConsole לכחול בהיר | O(1) |

### פונקציות

# התוכנית הראשית

כעת אציג את התוכנית הראשית שלי בפרויקט. אשר למעשה היא מאוד פשוטה וכל מה שהיא מבצעת הוא קריאה לפונקציות הראשיות בשלבים השונים, ולבסוף, מבצעת את ההמרה לobj ו-linking.

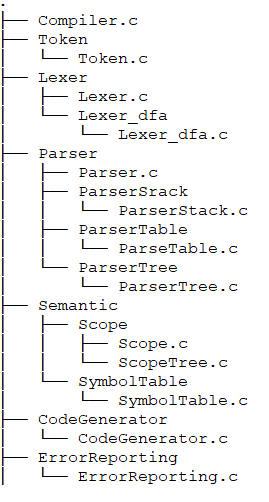
פונקציית main(int argc, char\* argv):

1. **אם** argc != 2, **בצע**:
   1. **הוצא לפלט** הודעת שגיאה.
   2. **סיים** תוכנית
2. **אם** לא בדוק-שם- קובץ-תקין(argv[1]), **בצע**:
   1. **הוצא לפלט** הודעת שגיאה.
   2. **סיים** תוכנית.
3. **אם** פתח קובץ -> file == null, **בצע:**
   1. **הוצא לפלט** הודעת שגיאה.
   2. **סיים** תוכנית
4. **קרא ל** קבל-אסימונים(file) -> tokens
5. **אם** panic\_mode == 1, **בצע**:
   1. **סיים** תוכנית.
6. **קרא ל** קבל-עץ-ניתוח(tokens) -> AST
7. **אם** panic\_mode == 1, **בצע**:
   1. **סיים** תוכנית.
8. **קרא** **ל** בדוק-סמנטי(AST) -> ScopeTree.
9. **אם** panic\_mode == 1, **בצע**:
   1. **סיים** תוכנית.
10. **צור** קובץ nasm חדש -> nfile.
11. **קרא ל** צור-קוד-יעד(AST, ScopeTree, nfile).
12. **אם** panic\_mode == 1, **בצע**:
    1. **סיים** תוכנית.
13. **צור** קובץ obj עבור nfile.
14. **צור** קובץ exe -> efile
15. **בצע** linkingעבור obj וספריית kernel32.libלתוך efile.

# מדריך למשתמש

כעת אדריך כיצד להתקין את הקומפיילר של השפה המומצאת – MyC. אציין כי קומפיילר MyC עובד במיוחד עבור סביבת Windows.

ראשית אסביר על היררכית הקבצים שלי:



אמליץ כי במידה ותתקינו את הקומפיילר על הסביבה שלכם, תשמרו על אותו מבנה היררכי בקבצים. כמובן שיש גם לצרף את כל קבצי ה-.h בהתאם.

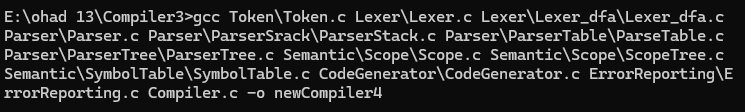
ישנם שלושה דברים המשתנים מסביבה לסביבה. והם שלושת הנתונים הבאים, הנמצאים תחת Compiler.h, אותם כל משתמש נדרש להתאים בשבילו.



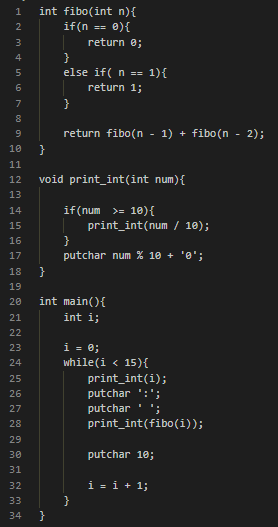
KERNEL32\_FILE\_PATH – הנתיב בוא נמצא הספרייה kernel32.  
NASM\_FILE\_PATH – הנתיב בוא נמצא nasm.exe, אשר ממיר קובץ nasm לobj.  
LINKER\_FILE\_PSTH – הנתיב בוא נמצא linker.exe, אשר מבצע Linking עם kernel32.lib וobj התוכנית שלנו, יחדיו נקבל את קובץ הexe של התוכנית.

במידה ומוגדר לכם משתנה סביבה עבור אחד מין הנתיבים, ניתן לרשום את שם הexe בלבד.

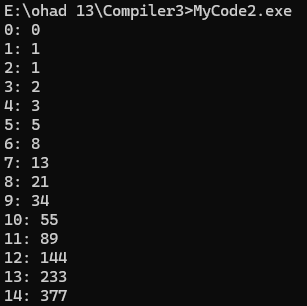
לסיום, תדרשו לקמפל את כל הקבצים בעזרת אחד מין הקומפיילרים המוצעים לשפת c. להלן דוגמה לקימפול הקבצים בעזרת gcc.



להלן דוגמה של קוד בשפת MyCהמדפיסה את 15 האיברים הראשונים בסדרת פיבונאצ'י.

נרשום בcmd את שם הקומפיילר ולאחר מכן את שם קובץ התוכנית.

לאחר ביצוע דבר זה תיווצר עבורכם קובץ חדש הנקרא: [your\_file\_name].exe אותו ניתן להריץ.

 נריץ את התוכנית ונראה את הפלט:

# סיכום אישי

## רפלקציה אישית

ואוו, איך מסכמים תהליך ארוך שכזה? איפה בכלל מתחילים?

הכל התחיל לאחר קבלתי למסלול "סיגיט" ביחידת 8200 אשר מתחיל בלימודי י"ג תוכנה. שנה שבה הרחבתי את אופקי בתחום מדעי המחשב והסייבר.

במסגרת שנת הלימודים התבקשנו לבצע פרויקט, פרויקט אשר וילווה אותנו במהלך כל השנה עליו נעבוד. אני זוכר כי בהתחלה לא היה לי שום מושג איזה פרויקט אני רוצה לעשות, ולקח לי המון זמן להחליט.

המורה שלי הסביר על סוגי הפרויקטים האפשריים, אחד מהם היה לבנות קומפיילר לשפה מומצאת. כאשר המורה דיבר על נושא זה ואמר "זה פרויקט לתותחים", אני בתור אדם שמאוד אוהב לאתגר את עצמו, ישר נדלקתי על הרעיון, חקרתי על נושא זה ונדהמתי לגלות כמה פרויקט זה עשיר ומרתק. ובסופו של דבר החלטתי שזה הפרויקט שאני עומד לעשות.

כאשר בחרתי את הפרויקט, לא ידעתי או הכרתי את השלבי העבודה, אלה צללתי פנימה ו"נכנסתי אל המסתורין".

פרויקט אשר בהתחלה חשבתי שייקח לי חודשיים לעשות, התגלגל למעל חצי שנה. במהלך הפרויקט עמדתי בפני הרבה אתגרים, חלקים מורטי עצבים והרבה מאוד Debugging. הפרויקט בהחלט לא היה קל, אך כאשר אני מתבונן מהסוף, אני מבין כי רכשתי הרבה מאוד ידע חדש והצלחתי לעמוד ביעדיי, אני מבסוט על עצמי.

העבודה על הפרויקט עבורי הייתה מאוד עוצמתית. אחד מין התכונות שבלטו אלצי במהלך העשייה הוא משמעת, לעבוד קשה ולעמוד ביעדים שהצבתי לעצמי.

מפרויקט זה אני לוקח איתי הרבה כלים. פרויקט זה לימד אותי הרבה תחומים מקצועיים, כמו שימוש באוטומטים ובמבני נתונים, שיטות ניתוח, ואלגוריתמים שונים. אך זה לא נגמר פה, למדתי לתכנן ולהתנהל בצורה נכונה מול פרויקט בסדר גודל רב, לעמוד ביעדים וחלוקת זמנים שהצבתי על עצמי.

במהלך הדרך חוויתי מספר לא מצומצם של אתגרים ורגעי משבר. רגעים בהם הבנתי כי עשיתי טעות ועכשיו לפתע אני צריך לחזור שבוע אחורה ולהמשיך משם, מקרים בהם היה לי באגים ושגיאות בלוגיקה שלקחו לי ימים לזהות ולתקן. רגעים בהם קיללתי את הפרויקט ושאלתי את עצמי למה לא לקחתי נושא פרויקט יותר קל.

כאשר התחלתי את הפרויקט הרגשתי כאלו אני נכנס לתוך מרוץ. שבוא אני צריך לשמור על "קצב" כדי שאסיים בזמן. לא הקדשתי מספיק זמן ללמידה תיאורטית, על compilers design, אלגוריתמיקה, וארכיטקטורה. זוהי הייתה טעות. זמן יקר התבזבז לי עקב כתיבת קוד ותכנון לדברים שבדיעבד היו שגויים ולא נכונים. אם הייתי מתחיל היום מחדש, הייתי מקדיש הרבה יותר זמן ללמידה על עולם הקומפיילרים, בכדי שהעשייה שלי תהייה יותר רציפה וחלקה.

למרות כל הבעיות וקשיים שחוויתי בדרך, אני בהחלט שמח עם התוצר שלי. הצבתי לעצמי מטרה ולא וויתרתי, עבדתי קשה ולבסוף הצלחתי. הפרויקט הזה העשיר את הידע שלי רבות והביא לי כלים שאני לוקח איתי לעתיד.

אני רוצה להודות לכם הקוראים, שהקדשם זמן עבור קריאת הספר, מקווה שהפקתם תובנות וידע רב .

# ביבליוגרפיה

בפרק זה אציג את מקורות המידע והכלים בהם נעזרתי במהלך הפרויקט.  
את קוד המקור המלא ניתן למצוא בעמוד ה-GitHub שלי בקישור הבא:

<https://github.com/ohadki12/MyC-Compiler.git>

**כלים בהם השתמשתי**

* **SLR Parser Generator**  
  אתר ליצירת טבלת Action ו-Goto מתוך תחביר שהוזן: <https://jsmachines.sourceforge.net/machines/slr.html>
* **Whimsical**  
  פלטפורמה ליצירת דיאגרמות ותרשימים בצורה גרפית נוחה: <https://whimsical.com/>
* **Railroad Diagram Generator**  
  כלי ליצירת דיאגרמות תחביר שפת תכנות:  
   <https://rr.red-dove.com/ui>

**מקורות מידע**

* **Bergman, S. D. (2017, May). Compiler Design. Retrieved from** [**https://bit.ly/3vKw55o**](https://bit.ly/3vKw55o)
* **Darveshi, R. S. (2011, November). E.C.E. Retrieved from Youtube** [**https://bit.ly/3vHxR7o**](https://bit.ly/3vHxR7o)
* **Tutorialspoint. (2015). Learn Compiler Design. Retrieved from Tutorialspoint:** [**https://bit.ly/35pWuLi**](https://bit.ly/35pWuLi)

**תרשימים**

* **Lexer DFA:** [**https://whimsical.com/Sgr62UL1KWcyCw8i3Qafeb**](https://whimsical.com/Sgr62UL1KWcyCw8i3Qafeb)
* **UML Class Diagram:** <https://whimsical.com/Xeco6XxPzoXggpMLwB12RC>
* **UML Use Cases:** [**https://whimsical.com/JfZNkdehK9vwn4CrgYxxRL**](https://whimsical.com/JfZNkdehK9vwn4CrgYxxRL)
* **Register:** [**https://whimsical.com/GKqWX8GkPxZPZwwm7XULrb**](https://whimsical.com/GKqWX8GkPxZPZwwm7XULrb)
* **Tokens:** [**https://whimsical.com/C57DvrF8iccaysg7Fd6emh**](https://whimsical.com/C57DvrF8iccaysg7Fd6emh)

# נספח - קוד הפרויקט

## Token

### Token.h

|  |
| --- |
| #pragma once  #include<stdio.h>  #include<stdlib.h>  // Token.h contents  typedef enum TokenType {      IDENTIFIER\_TOKEN,      OPEN\_PARENTHESES\_TOKEN,      CLOSE\_PARENTHESES\_TOKEN,      OPEN\_BRACE\_TOKEN,      CLOSE\_BRACE\_TOKEN,      COMMA\_TOKEN,      SEMICOLON\_TOKEN,      GETCHAR\_TOKEN,      PUTCHAR\_TOKEN,      INT\_TOKEN,      CHAR\_TOKEN,      VOID\_TOKEN,      ASSIGNMENT\_TOKEN,      RETURN\_TOKEN,      IF\_TOKEN,      ELSE\_TOKEN,      WHILE\_TOKEN,      LOGICAL\_OR\_TOKEN,      LOGICAL\_AND\_TOKEN,      EQUAL\_TO\_TOKEN,      NOT\_EQUAL\_TO\_TOKEN,      LESS\_THEN\_TOKEN,      GREATER\_THEN\_TOKEN,      LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN,      GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN,      PLUS\_TOKEN,      MINUS\_TOKEN,      MULTIPLY\_TOKEN,      DIVIDE\_TOKEN,      MODULO\_TOKEN,      LOGICAL\_NOT\_TOKEN,      DIGIT\_TOKEN,      CHAR\_LITERAL\_TOKEN,      DOLLAR\_TOKEN,      ERROR\_TOKEN,      END\_OF\_TOKENS\_TOKEN  } Terminals;  typedef struct {      // the type of the token      enum TokenType type;      // the values for identifer, character, or integer      union {          int integer;          char \*identifer;          char ch;      } value;      // the line which the token appear      // for reporting an error      int SourceCodeLine;  // the line of the error      int CharacterInSourceCodeLine; // the character of the error    } Token;  void printTokenTypeName(int token);  void printTokenArray(Token\*); |

### Token.c

|  |
| --- |
| #include"Token.h"  void printTokenTypeName(int token) {      // Static array of token names, matching the updated TokenType enum      static const char\* tokenNames[] = {          "IDENTIFIER\_TOKEN",          "OPEN\_PARENTHESES\_TOKEN",          "CLOSE\_PARENTHESES\_TOKEN",          "OPEN\_BRACE\_TOKEN"          ,"CLOSE\_BRACE\_TOKEN",           "COMMA\_TOKEN",           "SEMICOLON\_TOKEN",          "GETCHAR\_TOKEN",           "PUTCHAR\_TOKEN",           "INT\_TOKEN",           "CHAR\_TOKEN",           "VOID\_TOKEN",           "ASSIGNMENT\_TOKEN",          "RETURN\_TOKEN",          "IF\_TOKEN",          "ELSE\_TOKEN",          "WHILE\_TOKEN",          "LOGICAL\_OR\_TOKEN",          "LOGICAL\_AND\_TOKEN",          "EQUAL\_TO\_TOKEN",          "NOT\_EQUAL\_TO\_TOKEN",            "LESS\_THEN\_TOKEN",            "GREATER\_THEN\_TOKEN",            "LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN",          "GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN",          "PLUS\_TOKEN",          "MINUS\_TOKEN"          ,"MULTIPLY\_TOKEN"          ,"DIVIDE\_TOKEN",          "MODULO\_TOKEN",          "LOGICAL\_NOT\_TOKEN",          "DIGIT\_TOKEN",          "CHAR\_LITERAL\_TOKEN",            "DOLLAR\_TOKEN",            "ERROR\_TOKEN",            "END\_OF\_TOKENS\_TOKEN"      };      // Ensure the token is within the valid range of the enum values      if (token >= 0 && token < sizeof(tokenNames) / sizeof(tokenNames[0])) {          printf("The token name is: %s   ", tokenNames[token]);      } else {          printf("Unknown token\n");      }  }  void printTokenArray(Token \*tokenarray){      // helper function who prints all the tokens in a visible and ease way       for(int i = 0; tokenarray[i].type != END\_OF\_TOKENS\_TOKEN; i++){            printTokenTypeName(tokenarray[i].type);            if(tokenarray[i].type == DIGIT\_TOKEN){              printf("DIGIT: %d\n", tokenarray[i].value.integer);          }          else if(tokenarray[i].type == IDENTIFIER\_TOKEN){              printf("IDENTIFIER TEXT: %s\n", tokenarray[i].value.identifer);          }          else if(tokenarray[i].type == CHAR\_LITERAL\_TOKEN){              printf("TOKEN CHAR: '%c'\n", tokenarray[i].value.ch);          }          else{              printf("\n");          }          printf("\nthe token line: %d\n", tokenarray[i].SourceCodeLine);          printf("the token char in line: %d\n\n\n", tokenarray[i].CharacterInSourceCodeLine);      }  } |

## Lexer

### Lexer\_dfa.h

|  |
| --- |
| #pragma once  #include"../../Token/Token.h"  #include<string.h>  #include<ctype.h>  // Lexer\_dfa.h contents  #define NUMBER\_OF\_STATES 92  #define NUMBER\_OF\_CHARACTERS 95  // Enum for token types - Already defined in Token.h, so no need to repeat.  // Act structure to define the transition states  typedef struct {      int nextState;  } Act;  // StateType structure for defining state types  typedef struct {      enum { CONTINUE\_STATE, FINAL\_STATE, ERROR\_STATE } stateType;  } StateType;  // Lexer\_dfa structure for the DFA matrix and state types  typedef struct {      Act ActMatrix[NUMBER\_OF\_STATES][NUMBER\_OF\_CHARACTERS];      StateType state\_type[NUMBER\_OF\_STATES];  } Lexer\_dfa;  // Function prototypes for Lexer\_dfa  Terminals getTokenType(int FinalStage);  Lexer\_dfa\* init\_dfa();  int getIndexFromSymbol(char symbol);  char getSymbolFromIndex(int index);  void set\_dfa\_matrix\_row(Lexer\_dfa\* dfa, int state, int idenState, int delState, int errState);  void set\_dfa\_keyword\_token(Lexer\_dfa\* dfa, int states[], char symbols[]);  void set\_dfa\_state\_symbol(Lexer\_dfa\* dfa, int state, char symbol, Act act);  void set\_dfa\_int(Lexer\_dfa\* dfa);  void set\_dfa\_char(Lexer\_dfa\* dfa);  void set\_dfa\_if(Lexer\_dfa\* dfa);  void set\_dfa\_return(Lexer\_dfa\* dfa);  void set\_dfa\_getchar(Lexer\_dfa\* dfa);  void set\_dfa\_putchar(Lexer\_dfa\* dfa);  void set\_dfa\_else(Lexer\_dfa\* dfa);  void set\_dfa\_void(Lexer\_dfa\* dfa);  void set\_dfa\_while(Lexer\_dfa\* dfa);  void set\_dfa\_semicolon(Lexer\_dfa\* dfa);  void set\_dfa\_operators(Lexer\_dfa\* dfa);  void set\_dfa\_brackets(Lexer\_dfa\* dfa);  void set\_dfa\_comama(Lexer\_dfa\* dfa);  void set\_dfa\_comparisons(Lexer\_dfa\* dfa);  void set\_dfa\_digit\_literal(Lexer\_dfa\* dfa);  void set\_dfa\_char\_literal(Lexer\_dfa\* dfa);  void set\_dfa\_logical\_and(Lexer\_dfa\* dfa);  void set\_dfa\_logical\_or(Lexer\_dfa\* dfa);  void set\_dfa\_stages\_types(Lexer\_dfa\* dfa);  void init\_lexer\_dfa(Lexer\_dfa\* dfa);  Act getAct(Lexer\_dfa \*dfa, int curState, char curChar);  StateType getStateType(Lexer\_dfa \*dfa, int curState);  int toStepBack(int fState); |

### Lexer\_dfa.c

|  |
| --- |
| #include"Lexer\_dfa.h"  Terminals getTokenType(int FinalStage){      switch (FinalStage)      {          case 16:              return IF\_TOKEN;          case 4:              return INT\_TOKEN;          case 14:              return VOID\_TOKEN;          case 9:              return  CHAR\_TOKEN;          case 22:              return WHILE\_TOKEN;          case 36:              return PLUS\_TOKEN;          case 37:              return MINUS\_TOKEN;          case 38:              return MULTIPLY\_TOKEN;          case 39:              return DIVIDE\_TOKEN;          case 40:              return MODULO\_TOKEN;          case 80:              return OPEN\_PARENTHESES\_TOKEN;          case 81:              return CLOSE\_PARENTHESES\_TOKEN;          case 78:              return OPEN\_BRACE\_TOKEN;          case 79:              return CLOSE\_BRACE\_TOKEN;          case 30:              return DIGIT\_TOKEN;          case 86:              return LOGICAL\_AND\_TOKEN;          case 88:              return LOGICAL\_OR\_TOKEN;          case 84:              return CHAR\_LITERAL\_TOKEN;          case 77:              return SEMICOLON\_TOKEN;          case 45:              return ASSIGNMENT\_TOKEN;          case 52:              return EQUAL\_TO\_TOKEN;          case 46:              return LOGICAL\_NOT\_TOKEN;          case 51:              return NOT\_EQUAL\_TO\_TOKEN;          case 47:              return GREATER\_THEN\_TOKEN;          case 50:              return GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN;          case 48:              return LESS\_THEN\_TOKEN;          case 49:              return LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN;          case 91:          case 25:          case 26:          case 29:          case 90:          case 89:              return ERROR\_TOKEN;          case 59:              return RETURN\_TOKEN;          case 28:              return IDENTIFIER\_TOKEN;          case 35:              return ELSE\_TOKEN;          case 67:              return GETCHAR\_TOKEN;          case 75:              return PUTCHAR\_TOKEN;          case 76:              return COMMA\_TOKEN;      }      return -1;  }  Lexer\_dfa\* init\_dfa(){      // creates a new lexer dfa object      // define varivales      int x, y;      Act act;      // allocating memeory for the dfa      Lexer\_dfa \*dfa = (Lexer\_dfa\*)malloc(sizeof(Lexer\_dfa));      if(dfa == NULL){          printf("malloc error!");          exit(1);      }      //printf("allocated memory for dfa with pointer: %d", dfa);      act.nextState = - 1;      // intiating all indexes in the matrix to -1, indicate a spot unintallize      for(y = 0; y < NUMBER\_OF\_STATES; y++){          for(x = 0; x < NUMBER\_OF\_CHARACTERS; x++){              dfa->ActMatrix[y][x] = act;          }      }      // returning the lexer dfa object      return dfa;  }  int getIndexFromSymbol(char symbol){      // returning the index of a given symbol in the matrix      return symbol - 32;  }  char getSymbolFromIndex(int index){      // retutn the char that is in the given index      return index + 32;  }  void set\_dfa\_matrix\_row(Lexer\_dfa\* dfa, int state, int idenState, int delState, int errState){      // classifying the characters      char identifiers[] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789\_";      char delimiters[] = "(),;{}<>=!\*+-/%|&' ";      char errors[] =  "@#$^`~?.:#\"[]^\\";      // initiating integer for loop      int i;      // setting the identifiers, identifier state is in q27      for(i = 0; i < sizeof(identifiers) / sizeof(identifiers[0]) - 1; i++){          // checking that we are not overriding an existing act          if(dfa->ActMatrix[state][getIndexFromSymbol(identifiers[i])].nextState == - 1){              set\_dfa\_state\_symbol(dfa, state, identifiers[i], (Act){idenState});          }      }      // setting the delimiters, which is identifier final state in q28      for(i = 0; i < sizeof(delimiters) / sizeof(delimiters[0]) - 1; i++){          // checking that we are not overriding an existing act          if(dfa->ActMatrix[state][getIndexFromSymbol(delimiters[i])].nextState == - 1){              set\_dfa\_state\_symbol(dfa, state, delimiters[i], (Act){delState});          }      }      // setting the errors state, going into invaild char error final state in q29      for(i = 0; i < sizeof(errors) / sizeof(errors[0]) - 1; i++){          // checking that we are not overriding an existing act          if(dfa->ActMatrix[state][getIndexFromSymbol(errors[i])].nextState == - 1){              set\_dfa\_state\_symbol(dfa, state, errors[i], (Act){errState});          }        }  }  void set\_dfa\_keyword\_token(Lexer\_dfa\* dfa, int states[], char symbols[]){      // setting a token path in the matrix      // define variables      int i;        // initiating the states acoording to the symbols      set\_dfa\_state\_symbol(dfa, states[0], symbols[0], (Act){states[1]});      for(i = 1; symbols[i] != '\0'; i++){          //q27 -> identifier state          //q28 -> identifier final state          //q29 -> unknown symbol final state error          set\_dfa\_matrix\_row(dfa, states[i], 27, 28, 29);          set\_dfa\_state\_symbol(dfa, states[i], symbols[i], (Act){states[i + 1]});      }      set\_dfa\_matrix\_row(dfa, states[i], 27, states[i + 1], 29);  }  void set\_dfa\_state\_symbol(Lexer\_dfa\* dfa, int state, char symbol, Act act){      // intiating a symbol in a state to do the given act      dfa->ActMatrix[state][getIndexFromSymbol(symbol)] = act;  }  void set\_dfa\_int(Lexer\_dfa\* dfa){      // setting int token...      set\_dfa\_keyword\_token(dfa, (int[]){0,1,2, 3, 4}, "int");  }  void set\_dfa\_identifier(Lexer\_dfa\* dfa){      set\_dfa\_matrix\_row(dfa, 27, 27, 28, 29);  }  void set\_dfa\_char(Lexer\_dfa\* dfa){      // setting char token...      set\_dfa\_keyword\_token(dfa, (int[]){0,5,6,7,8,9}, "char");  }  void set\_dfa\_if(Lexer\_dfa\* dfa){      // setting if token...      set\_dfa\_keyword\_token(dfa, (int[]){0,1, 15, 16}, "if");  }  void set\_dfa\_return(Lexer\_dfa\* dfa){      // setting return token      set\_dfa\_keyword\_token(dfa, (int[]){0,53,54,55,56,57,58,59}, "return");    }  void set\_dfa\_getchar(Lexer\_dfa \*dfa){      // setting the getchar token keyword      set\_dfa\_keyword\_token(dfa, (int[]){0, 60, 61, 62, 63, 64, 65, 66, 67}, "getchar");  }  void set\_dfa\_putchar(Lexer\_dfa\* dfa){      // setting the putchar token keyword      set\_dfa\_keyword\_token(dfa, (int[]){0, 68, 69, 70, 71, 72, 73, 74, 75}, "putchar");  }  void set\_dfa\_else(Lexer\_dfa\* dfa){      // setting the else token      set\_dfa\_keyword\_token(dfa, (int[]){0,31,32,33,34,35}, "else");  }  void set\_dfa\_void(Lexer\_dfa\* dfa){      // setting the void token      set\_dfa\_keyword\_token(dfa, (int[]){0,10, 11, 12, 13, 14}, "void");  }  void set\_dfa\_while(Lexer\_dfa\* dfa){      // setting the while token in the dfa matrix      set\_dfa\_keyword\_token(dfa, (int[]){0,17,18,19,20,21, 22}, "while");  }  void set\_dfa\_semicolon(Lexer\_dfa\* dfa){      // setting the token for ; token        set\_dfa\_state\_symbol(dfa, 0, ';', (Act){77});  }  void set\_dfa\_operators(Lexer\_dfa\* dfa){      // settting the tokens for operators        set\_dfa\_state\_symbol(dfa, 0, '+', (Act){36});      set\_dfa\_state\_symbol(dfa, 0, '-', (Act){37});      set\_dfa\_state\_symbol(dfa, 0, '\*', (Act){38});      set\_dfa\_state\_symbol(dfa, 0, '/', (Act){39});      set\_dfa\_state\_symbol(dfa, 0, '%', (Act){40});  }  void set\_dfa\_brackets(Lexer\_dfa\* dfa){      // set brackets      set\_dfa\_state\_symbol(dfa, 0, '{', (Act){78});      set\_dfa\_state\_symbol(dfa, 0, '}', (Act){79});      set\_dfa\_state\_symbol(dfa, 0, ')', (Act){81});      set\_dfa\_state\_symbol(dfa, 0, '(', (Act){80});  }  void set\_dfa\_comama(Lexer\_dfa\* dfa){      //setting the comma token in the dfa matrix      set\_dfa\_state\_symbol(dfa, 0, ',', (Act){76});  }  void set\_dfa\_comparisons(Lexer\_dfa\* dfa){      // set comperisions      set\_dfa\_state\_symbol(dfa, 0, '=', (Act){41});      set\_dfa\_matrix\_row(dfa,41, 45, 45, 45);      set\_dfa\_state\_symbol(dfa, 41, '=', (Act){52});      set\_dfa\_state\_symbol(dfa, 0, '!', (Act){42});      set\_dfa\_matrix\_row(dfa,42, 46, 46, 46);      set\_dfa\_state\_symbol(dfa, 42, '=', (Act){51});      set\_dfa\_state\_symbol(dfa, 0, '>', (Act){43});      set\_dfa\_matrix\_row(dfa,43, 47, 47, 47);      set\_dfa\_state\_symbol(dfa, 43, '=', (Act){50});      set\_dfa\_state\_symbol(dfa, 0, '<', (Act){44});      set\_dfa\_matrix\_row(dfa,44, 48, 48, 48);      set\_dfa\_state\_symbol(dfa, 44, '=', (Act){49});  }  void set\_dfa\_digit\_literal(Lexer\_dfa\* dfa){      // setting the path for identifying a digit token      int i;      set\_dfa\_matrix\_row(dfa, 23,26, 30,29);      set\_dfa\_matrix\_row(dfa, 24, 26, 91, 29);      for(i=0 ; i < 10; i++){          set\_dfa\_state\_symbol(dfa, 0, '0' + i, (Act){23});          set\_dfa\_state\_symbol(dfa, 23, '0' + i, (Act){23});          set\_dfa\_state\_symbol(dfa, 24, '0' + i, (Act){25});      }      set\_dfa\_state\_symbol(dfa, 23, '.', (Act){24});  }  void set\_dfa\_char\_literal(Lexer\_dfa\* dfa){      // setting the path for a char litetal      int i;      // setting the ' start path      set\_dfa\_state\_symbol(dfa,0, '\'', (Act){82});      set\_dfa\_matrix\_row(dfa, 82, 83, 83, 83);      set\_dfa\_state\_symbol(dfa,83, '\'', (Act){84});  }  void set\_dfa\_logical\_and(Lexer\_dfa\* dfa){      set\_dfa\_state\_symbol(dfa, 0, '&', (Act){85});      set\_dfa\_matrix\_row(dfa, 85, 89,89,89);      set\_dfa\_state\_symbol(dfa, 85, '&', (Act){86});  }  void set\_dfa\_logical\_or(Lexer\_dfa\* dfa){      set\_dfa\_state\_symbol(dfa, 0, '|', (Act){87});      set\_dfa\_matrix\_row(dfa, 87, 90,90,90);      set\_dfa\_state\_symbol(dfa, 87, '|', (Act){88});  }  void set\_dfa\_stages\_types(Lexer\_dfa\* dfa){      int finals[] = {16, 4, 14, 9, 22, 36, 35, 37, 38, 39, 40, 80, 81, 78, 79,          30, 86, 88, 84, 77, 45, 52, 46, 51, 47, 50, 48, 49,          59, 28, 67, 75, 76};      int errors[] = {91, 25, 26, 29, 89, 90};      int continues[] = {0, 1, 2, 3, 5, 6, 7, 8, 10, 11, 12, 13, 15, 17, 18, 19,          20, 21, 23, 24, 27, 31, 32, 33, 34, 41, 42, 43, 44,          53, 54, 55, 56, 57, 58, 60, 61, 62, 63, 64, 65,66,68, 69,          70, 71, 72, 73,74, 82, 85, 87};        int i;      // setting all the finals      for(i = 0; i < sizeof(finals) / sizeof(finals[0]); i++){          dfa->state\_type[finals[i]].stateType = FINAL\_STATE;      }      // setting all the erros      for(i = 0; i < sizeof(errors) / sizeof(errors[0]); i++){          dfa->state\_type[errors[i]].stateType = ERROR\_STATE;      }      // setting all the continues states      for(i = 0; i < sizeof(continues) / sizeof(continues[0]); i++){          dfa->state\_type[continues[i]].stateType = CONTINUE\_STATE;      }  }  void init\_lexer\_dfa(Lexer\_dfa\* dfa){        set\_dfa\_matrix\_row(dfa, 0, 27, 28, 29);      set\_dfa\_state\_symbol(dfa, 0, ' ', (Act){0});        // firstly, setting the types of all states      set\_dfa\_stages\_types(dfa);        // setting the paths to a operators( + - \* / %)      set\_dfa\_operators(dfa);        //  setting the paths to a patanteses( (, ), {, })      set\_dfa\_brackets(dfa);        // setting the paths for comperrisions tokens( ==, !=, <=, >=)      set\_dfa\_comparisons(dfa);        // setting the path for semicolon      set\_dfa\_semicolon(dfa);        // setting the digits leteral path      set\_dfa\_digit\_literal(dfa);      // setting the char literal path      set\_dfa\_char\_literal(dfa);      // setting the comma character      set\_dfa\_comama(dfa);      // setting the || and && tokens      set\_dfa\_logical\_and(dfa);      set\_dfa\_logical\_or(dfa);      set\_dfa\_identifier(dfa);        // setting the paths to a keywords      set\_dfa\_char(dfa); // char      set\_dfa\_if(dfa); // if      set\_dfa\_else(dfa); // else      set\_dfa\_int(dfa); // int      set\_dfa\_return(dfa); // return      set\_dfa\_getchar(dfa); // getchar(i/o)      set\_dfa\_putchar(dfa); // putchar(i/o)      set\_dfa\_while(dfa); // while      set\_dfa\_void(dfa); // void    }  int toStepBack(int fState){      // return 1 if the final state ended after any DELIMITER character, else return 0      if(fState == 4 || fState == 9 || fState == 14 || fState == 16 || fState == 22 || fState == 30 || fState == 28 || fState == 35      || fState == 45 || fState == 46 || fState == 47|| fState == 48|| fState == 59 || fState == 67 || fState == 75){          return 1;      }      return 0;  }  Act getAct(Lexer\_dfa \*dfa, int curState, char curChar){      // returing the act in the current indexes      Act act = dfa->ActMatrix[curState][getIndexFromSymbol(curChar)];      return act;  }  StateType getStateType(Lexer\_dfa \*dfa, int curState){      // returning the state of the currat      StateType st = dfa->state\_type[curState];      return st;  } |

### Lexer.h

|  |
| --- |
| #pragma once  #include"Lexer\_dfa/Lexer\_dfa.h"  #include"../ErrorReporting/ErrorReporting.h"  #include <stdio.h>  #include <stdlib.h>  #include <ctype.h>  #include <string.h>  typedef struct LexerSourceCodeStorage  {      int number\_of\_lines;      char\*\*lines;  }LexerSourceCodeStorage;  #define MAX\_LINE\_LENGTH 256  char\* getNextLine(FILE\* fp);  Token\* getTokens(FILE\* fp);  void lexer\_AddNewLineIntoStorage(char\* line);  LexerSourceCodeStorage\* lexer\_CreateNewCodeStorage();  char\* getSourceCodeLine(int line);  char\* Lexer\_GetTokenName(Token\* token);  void SetsourceCodeFileName(char\* fname);  char\* getSourceCodeFileName(); |

### Lexer.c

|  |
| --- |
| #include"Lexer.h"  // create global storage struct that will be alive untill program exit  LexerSourceCodeStorage\* SourceCodeStorage;  // create a gloval storage that will keep the name of the souce code  char\* SOURCE\_CODE\_FILE;  char\* Lexer\_GetTokenName(Token\* token) {      // Static array of token names, matching the updated TokenType enum      char\* tokenNames[] = {          "identifier",          "(",          ")",          "{",          "}",          ",",          ";",          "getchar",          "putchar",          "int",          "char",          "void",          "=",          "return",          "if",          "else",          "while",          "||",          "&&",          "==",          "!=",          "<",          ">",          "<=",          ">=",          "+",          "-",          "\*",          "/",          "%",          "!",          "int\_literal",          "char\_literal",          "$",          "ERROR\_TOKEN", // No corresponding name provided, keeping original          "END\_OF\_TOKENS\_TOKEN" // No corresponding name provided, keeping original      };        //if(token->type > 36) return NULL;      return tokenNames[token->type];  }  char\* getNextLine(FILE\* fp) {      // creating a temporery buffer to hold the line data      char\* buffer = (char\*)malloc(sizeof(char) \* MAX\_LINE\_LENGTH + 2);      char\* temp = NULL;      if(buffer == NULL){          printf("MALLOC ERROR");          exit(-1);      }      if (fp == NULL || fgets(buffer, MAX\_LINE\_LENGTH, fp) == NULL) {          return NULL; // Return NULL if end of file or error      }      // add the buffer with delimiter at the end      temp = buffer;      while (\*temp != '\0') temp++;      \*(temp) = ' ';      \*(temp + 1) = 0;        return buffer; // Return the line read  }  void addNewToken(Token\*\* tokens,int \*tokensLen, Terminals TokenType, void\* value, int TokenLine, int TokenCharacterInLine){      //printf("\n\n\ni M BEING PRINTED\n\n\n");      if(\*tokens == NULL && \*tokensLen == 0){          \*tokens = (Token\*)malloc(sizeof(Token) \* 1);          if(\*tokens == NULL){              printf("MALLOC ERROR!");              exit(1);          }          \*tokensLen = 1;        }      else if(\*tokens != NULL && \*tokensLen != 0){          // increasing the size of the array          (\*tokensLen)++;          \*tokens = (Token\*)realloc(\*tokens, sizeof(Token) \* \*tokensLen);          if(\*tokens == NULL){              printf("MALLOC ERROR!");              exit(1);          }      }      // setting the token type      (\*tokens)[\*tokensLen - 1].type = TokenType;      //printf("Creating new token element\n");      // if type identifier, number or character, add their value      if(TokenType == IDENTIFIER\_TOKEN){          //printf("this is identifier, with the data: %s", (char\*)(value));          (\*tokens)[\*tokensLen - 1].value.identifer = (char\*)(value);      }      else if(TokenType == DIGIT\_TOKEN){          (\*tokens)[\*tokensLen - 1].value.integer = \*(int\*)(value);          free(value);      }      else if(TokenType == CHAR\_LITERAL\_TOKEN){            (\*tokens)[\*tokensLen - 1].value.ch = \*(char\*)(value);          free(value);      }      // setting the token error values      (\*tokens)[\*tokensLen - 1].SourceCodeLine = TokenLine;      (\*tokens)[\*tokensLen - 1].CharacterInSourceCodeLine = TokenCharacterInLine;    }  void\* extractNumberFromStream(char\* stream, size\_t stream\_size) {      if (stream\_size == 0) {          return NULL;      }      int endIndex = 0;      while (!isdigit(stream[endIndex])) {          endIndex--;      }      int startIndex = endIndex - 1;      while (isdigit(stream[startIndex]))      {          startIndex--;      }      startIndex++;        int token\_length = endIndex - startIndex + 1;      //printf("[DEBUG] Number starts at index %d, length: %d\n", startIndex, token\_length);      char\* numStr = malloc(token\_length + 1);      if (!numStr) {          perror("[ERROR] malloc failed");          exit(EXIT\_FAILURE);      }      strncpy(numStr, stream + startIndex, token\_length);      numStr[token\_length] = '\0';      //printf("[DEBUG] Extracted numeric string: %s\n", numStr);      long number = strtol(numStr, NULL, 10);      free(numStr);      int\* numPtr = malloc(sizeof(int));      if (!numPtr) {          perror("[ERROR] malloc failed");          exit(EXIT\_FAILURE);      }      \*numPtr = (int)number;      //printf("[DEBUG] Converted number: %d\n", \*numPtr);      if(stream\_size < startIndex \* -1){          perror("[ERROR]");          exit(EXIT\_FAILURE);      }      return numPtr;  }  void\* extractIdentifierFromStream(char\* stream, size\_t stream\_size) {      stream--;      if (stream\_size == 0) {          //printf("[ERROR] Stream size is zero. Returning NULL.\n");          return NULL;      }      int endIndex = 0;        while (!isalnum(stream[endIndex]) && stream[endIndex] != '\_')      {          endIndex--;      }      int startIndex = endIndex - 1;      while (isalnum(stream[startIndex]) || stream[startIndex] == '\_')      {          startIndex--;      }      startIndex++;      int token\_length = endIndex - startIndex + 1;      char\* identStr = malloc(token\_length + 1);      if (!identStr) {          perror("[ERROR] malloc failed");          exit(EXIT\_FAILURE);      }      strncpy(identStr, stream + startIndex, token\_length);      identStr[token\_length] = '\0';      if(stream\_size < startIndex \* - 1){          perror("[ERROR]");          exit(EXIT\_FAILURE);      }      return identStr;  }  void\* extractCharacterFromStream(char\* stream, size\_t stream\_size) {      if (stream == NULL || stream\_size < 3) {          //printf("[ERROR] Stream is NULL or too short to contain a valid character literal.\n");          return NULL;      }      for (size\_t i = 0; i <= stream\_size - 3; i++) {          char\* candidate = stream - i;          if (candidate[-2] == '\'' && candidate[0] == '\'' && candidate[-1] != '\'') {              char\* ch = malloc(sizeof(char));              if (!ch) {                  perror("[ERROR] malloc failed");                  exit(EXIT\_FAILURE);              }              \*ch = candidate[-1];              return ch;          }      }      return NULL;  }  void\* getValueForTokenType(char\* stream, Terminals type, int streamSize){      if(type != IDENTIFIER\_TOKEN && type != DIGIT\_TOKEN && type != CHAR\_LITERAL\_TOKEN){          return NULL;      }      if(type == DIGIT\_TOKEN){          return extractNumberFromStream(stream, streamSize);      }      if(type == IDENTIFIER\_TOKEN){          return extractIdentifierFromStream(stream, streamSize);      }      if(type == CHAR\_LITERAL\_TOKEN){          //printf("am i here?");          return extractCharacterFromStream(stream, streamSize);      }  }  void stepStreamUntilNewToken(char\*\* stream) {      // This function steps the stream until it reaches a delimiter character.      // It updates \*stream to point to the first character after the delimiter.      //printf("we are currently at point: %s\n", \*stream);      // Define the delimiter characters      char\* delis = "(),;{}<>=!&\*+-/%|' ";      int c  = 0;      // Loop through the stream until we find a delimiter      while (\*\*stream != '\0') {          // Check if the current character is a delimiter          for (int j = 0; delis[j] != '\0'; j++) {              if (\*\*stream == delis[j]) {                    return;              }          }            (\*stream)++;          c++;      }    }  char\* Lexer\_PrepareErrorMessage(int val, char ch){      // define local variale      char buffer[100];      // this function creates the different error messages in the lexical analysis      if(val == 90){          return \_strdup("'|' is not a valid operator; did you mean '||'?");        }      else if(val == 89){          return \_strdup("'&' is not a valid operator; did you mean '&&'?");        }      else if(val == 26){          return \_strdup("invalid identifier; identifiers can't begin with a digit");        }      else if(val == 29){          snprintf(buffer, sizeof(buffer), "unrecognized character: %c", ch);          return \_strdup(buffer);        }      else if(val == 25){          return \_strdup("invalid numeric literal; decimal points are not allowed");      }  }  Token\* getTokens(FILE\* fp){      // define variables      int Tokenslen = 0;      int currentState = 0;      int currentLine = 1;      int currentCharacterInLine = 0;      int lengthOfCurrentToken = 0;      char\* buffer = NULL;      char\* tempBuffer = NULL;      StateType currentStateType;      int currentTokentype;      void\* currentTokenValue;      Act currentAct;      // in panic mode?      int panicMode = 0;      // this variable hold the pointer into the error message      char\* errorMsg;      // initiate the global storage      SourceCodeStorage = lexer\_CreateNewCodeStorage();      // create the dfa      Lexer\_dfa \*dfa = init\_dfa();      init\_lexer\_dfa(dfa);      // create the array of tokens      Token \*tokenArray = NULL;        // looping until we reed all data      while (!feof(fp))      {          // getting the chatractes int he current line          buffer = getNextLine(fp);          // inserting the current line into the storage          lexer\_AddNewLineIntoStorage(buffer);          // setting the stream pointer that we use to iterate the chracters in the currnent line          tempBuffer = buffer;          // setting the character counter in the current line to be zero          currentCharacterInLine = 0;            //printf("the char is: %c\n", \*tempBuffer);          while (tempBuffer != NULL && \*tempBuffer != '\0')          {              //printf("the ascii of the current char is: %d   ", \*tempBuffer);              // checking if current character is lower then ascii 32              if(\*tempBuffer < getSymbolFromIndex(0) || \*tempBuffer > getSymbolFromIndex(NUMBER\_OF\_CHARACTERS - 1)){                  //printf("ma i need to be here?\n");                  //continue;                  \*tempBuffer = ' ';              }              else{                  //printf("\n");              }              //printf("CURRENT STATE: %d  current char: %c, ascii value: %d  \n", currentState, \*tempBuffer, \*tempBuffer);              //printf("\_%s\_",tempBuffer);              //printf("\n");                // checking if the current state isn't final              currentStateType = getStateType(dfa, currentState);              //printf("the sttype is: %d\n", currentStateType.stateType);              // checking the type of the current state              if(currentStateType.stateType == CONTINUE\_STATE){                  // getting the act in the current index                  currentAct = getAct(dfa, currentState, \*tempBuffer);                  currentState = currentAct.nextState;                  //printf("NEXT STATE: %d\n\n", currentState);                  // stepping the str pointer                  tempBuffer++;                  if(currentState == -1){printf("how am i here?"); exit(1);}                  // checking we are not on space character                  if(\*tempBuffer > 32){                      //printf("increasing the counter with the current char is: %c in ascii: %d\n", \*tempBuffer, \*tempBuffer);                      // increasing the token length counter                      lengthOfCurrentToken++;                  }                }              else if(currentStateType.stateType == FINAL\_STATE){                  //printf("starting to create new token in the array");                  // creating new token                  currentTokentype = getTokenType(currentState);                    // getting the value of that token, if not identifer, digit, or charcter, it will be null                  currentTokenValue = getValueForTokenType(tempBuffer,currentTokentype, currentCharacterInLine);                  if(currentLine == - 1){                      printf("COMPILER ERROR");                      exit(EXIT\_FAILURE);                  }                  if(currentCharacterInLine == -1){                      printf("COMPILER ERROR");                      exit(EXIT\_FAILURE);                  }                  //int typebla = getTokenType(currentState);                  //if(typebla < 0 || typebla > 36) exit(1);                  // creating a new token in the dynamic array with his attributes                  addNewToken(&tokenArray, &Tokenslen, getTokenType(currentState),                                                   currentTokenValue, currentLine,                                                   currentCharacterInLine - lengthOfCurrentToken); //// -lengthOfCurrentToken                    // chrcking if to step back                  if(toStepBack(currentState)){                      //printf("\nstepped back\n");                      tempBuffer -= 1;                  }                    // reseting the state to start state                  currentState = 0;                  // reseting the current token length                  lengthOfCurrentToken = 0;              }              else if(currentStateType.stateType == ERROR\_STATE){                  //printf("i am here, this is very wierd!");                  // handling lexical error while keeping the lexer working                  //printf("detected an error state in state: %d", currentState);                  // setting panic mode to 1                  panicMode = 1;                  // outputing the base error                  ErrorReporting\_ReportError\_Base(ERROR\_ON\_LEXER\_PHASE, currentLine, currentCharacterInLine - lengthOfCurrentToken);                  // preparing the error message                  errorMsg = Lexer\_PrepareErrorMessage(currentState, \*(buffer + currentCharacterInLine - lengthOfCurrentToken));                  // outputing the error msg                  ErrorReporting\_ReportError(errorMsg);                  // freeing the error message from the heap                  free(errorMsg);                    // reseting the current state back to the inital position                  currentState = 0;                  // reseting the counter of the token length                  lengthOfCurrentToken = 0;                  //printf("calling the maniac function!\n");                  // skipping processing the errored token                  stepStreamUntilNewToken(&tempBuffer);              }              // increasing the counter of chatacter in line              currentCharacterInLine++;          }          // freeing the bufer          if(buffer != NULL) free(buffer);            // STEPPING THE LINE COUTNER          currentLine++;      }        // adding another token indicating end of tokesn      addNewToken(&tokenArray, &Tokenslen, END\_OF\_TOKENS\_TOKEN, NULL, -1, -1);      if(panicMode){          free(dfa);          free(tokenArray);          return NULL;      }      //////tokenArray[Tokenslen - 1].type = END\_OF\_TOKENS\_TOKEN;      //printf("freeing the dfa memory at: %d\n", dfa);      free(dfa);        return tokenArray;  }  void lexer\_AddNewLineIntoStorage(char\* line){      // this function save the current line into the following line in the storage      if(SourceCodeStorage == NULL){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      if(SourceCodeStorage->number\_of\_lines == 0){          SourceCodeStorage->lines = (char\*\*)malloc(sizeof(char\*) \* 1);          if(SourceCodeStorage->lines == NULL){              printf("MALLOC ERROR");              exit(EXIT\_FAILURE);          }          SourceCodeStorage->number\_of\_lines = 1;      }      else{          // increasing the memory size          SourceCodeStorage->lines = (char\*\*)realloc(SourceCodeStorage->lines, sizeof(char\*) \* ++SourceCodeStorage->number\_of\_lines);      }        // saving the new line      SourceCodeStorage->lines[SourceCodeStorage->number\_of\_lines - 1] = \_strdup(line);    }  LexerSourceCodeStorage\* lexer\_CreateNewCodeStorage(){      // this function creates a new struct in type LexerSourceCodeStorage and return its memory address      // define variables      LexerSourceCodeStorage\* storage = (LexerSourceCodeStorage\*)malloc(sizeof(LexerSourceCodeStorage));      if(storage == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }      storage->number\_of\_lines = 0;      storage->lines = NULL;      return storage;  }  char\* getSourceCodeLine(int line){      // this function responsible for returning the corresponding souce code line      // checking that we arent accessing a non existing line      if(line > SourceCodeStorage->number\_of\_lines || line <= 0){          return NULL;      }      // returning the wanted line that was stored in the global struct      return SourceCodeStorage->lines[line - 1];  }  void SetsourceCodeFileName(char\* fname){      SOURCE\_CODE\_FILE = \_strdup(fname);  }  char\* getSourceCodeFileName(){      return SOURCE\_CODE\_FILE;  } |

## Parser

### ParserStack.h

נמצא תחת Parser.h עקב יציבות תלויות.

### ParserStack.c

|  |
| --- |
| //#include"ParserStack.h"  #include"../Parser.h"  // Create a new stack  Parser\_Stack\* parser\_stack\_createStack() {        Parser\_Stack \*stack = (Parser\_Stack\*) malloc(sizeof(Parser\_Stack));        if (!stack) {          printf("Memory allocation failed! one\n");          exit(EXIT\_FAILURE);      }      stack->arr = NULL;  // Initially NULL      stack->size = 0;    // Stack size      stack->memorySize = 0;  // Memory allocated for stack elements      return stack;  }  // Push an element onto the stack  void parser\_stack\_push(Parser\_Stack \*stack, Stack\_Element \*element) {      // Check if the stack needs more memory (increase memory size)      if (stack->size >= stack->memorySize) {          //printf("the stack size: %d the memory size: %d\n    ", stack->size, stack->memorySize);            stack->memorySize = stack->memorySize + 2;          stack->arr = (Stack\_Element\*\*) realloc(stack->arr, stack->memorySize \* sizeof(Stack\_Element\*));          if (!stack->arr) {              printf("Memory allocation failed! two\n");              exit(EXIT\_FAILURE);          }      }        // Add the new element at the top      stack->arr[stack->size] = element;      stack->size += 1;  }  // Pop an element from the stack  Stack\_Element\* parser\_stack\_pop(Parser\_Stack \*stack) {      if (stack->size == 0) {          printf("Error: Stack underflow!\n");          exit(EXIT\_FAILURE);      }      stack->size -= 1;      Stack\_Element\* tmp = stack->arr[stack->size];      stack->arr[stack->size] = NULL;      return tmp;  }  // Get the top element of the stack without removing it  Stack\_Element\* parser\_stack\_top(Parser\_Stack stack) {      if (stack.size == 0) {          printf("Error: Stack is empty!\n");          exit(EXIT\_FAILURE);      }      return (stack.arr[stack.size - 1]);  }  // Check if the stack is empty  int parser\_stack\_isEmpty(Parser\_Stack stack) {      return stack.size == 0;  }  // Destroy the stack and free memory  void parser\_stack\_destroyStack(Parser\_Stack \*stack) {      // Free each element      for (int i = 0; i < stack->size; i++) {          free(stack->arr[i]);          stack->arr[i] = NULL;      }      free(stack->arr);  // Free the array of elements      stack->arr = NULL;      free(stack);  // Free the stack structure itself      stack = NULL;  } |

### ParserTable.h

נמצא תחת Parser.h עקב יציבות תלויות.

### ParserTable.c

|  |
| --- |
| #include"../Parser.h"  int get\_index\_from\_terminal\_parser(Terminals TokenKind){      // as long as the terminals enum is in the right order, we can simply return the intefer value of that enum      return (int)TokenKind;  }  int get\_index\_from\_Non\_terminal\_parser(Non\_Terminals NonTerminalKind){      // as long as the non terminals enum is in the right order, we can simply return the intefer value of that enum      return (int)NonTerminalKind;  }  void set\_parsr\_action(Parser\* parser, int state, Terminals terminal, Action action){      parser->parse\_table.ActionMatrix[state][get\_index\_from\_terminal\_parser(terminal)] = action;  }  void set\_parser\_goto(Parser\* parser, int state, Non\_Terminals nonTerminal, int nextState){      parser->parse\_table.GotoMatrix[state][get\_index\_from\_Non\_terminal\_parser(nonTerminal)] = nextState;  }  void init\_action\_table(Parser\* parser){      // iniatating the action table of the parser according the the lr shift recude automata      int state = 0;      state = 0;      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      state = 1;      set\_parsr\_action(parser, state, DOLLAR\_TOKEN, (Action){ACCEPT,-1});      state = 2;      set\_parsr\_action(parser, state, DOLLAR\_TOKEN, (Action){REDUCE,1});      state = 3;      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      set\_parsr\_action(parser, state, DOLLAR\_TOKEN, (Action){REDUCE,3});      state = 4;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,9});      state = 5;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,23});      state = 6;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,24});      state = 7;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,25});      state = 8;      set\_parsr\_action(parser, state, DOLLAR\_TOKEN, (Action){REDUCE,2});      state = 9;      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,10});      state = 10;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){SHIFT,12});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      state = 11;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){SHIFT,15});      state = 12;      set\_parsr\_action(parser, state, OPEN\_BRACE\_TOKEN, (Action){SHIFT,16});      state = 13;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,6});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){SHIFT,17});      state = 14;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,18});      state = 15;      set\_parsr\_action(parser, state, OPEN\_BRACE\_TOKEN, (Action){SHIFT,19});      state = 16;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,31});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){SHIFT,37});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){SHIFT,38});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){SHIFT,33});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){SHIFT,34});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){SHIFT,35});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 17;      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      state = 18;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,8});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,8});      state = 19;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,31});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){SHIFT,37});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){SHIFT,38});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){SHIFT,33});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){SHIFT,34});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){SHIFT,35});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 20;      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){SHIFT,53});      state = 21;      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,9});      state = 22;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,31});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,10});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){SHIFT,37});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){SHIFT,38});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){SHIFT,33});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){SHIFT,34});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){SHIFT,35});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 23;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){SHIFT,55});      state = 24;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){SHIFT,56});      state = 25;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){SHIFT,57});      state = 26;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,15});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,15});      state = 27;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,16});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,16});      state = 28;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){SHIFT,58});      state = 29;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){SHIFT,59});      state = 30;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){SHIFT,60});      state = 31;      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,62});      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, ASSIGNMENT\_TOKEN, (Action){SHIFT,61});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,60});      state = 32;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,63});      state = 33;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,28});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 34;      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,66});      state = 35;      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,67});      state = 36;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,39});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,39});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,39});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){SHIFT,68});      state = 37;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,69});      state = 38;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 39;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,40});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,40});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,40});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,40});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){SHIFT,71});      state = 40;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,42});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,42});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,42});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,42});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,42});      state = 41;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,44});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,44});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,44});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,44});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,44});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){SHIFT,72});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){SHIFT,73});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){SHIFT,74});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){SHIFT,75});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){SHIFT,76});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){SHIFT,77});      state = 42;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,51});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){SHIFT,78});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,79});      state = 43;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,54});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){SHIFT,80});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){SHIFT,81});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){SHIFT,82});      state = 44;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 45;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,59});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,59});      state = 46;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,61});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,61});      state = 47;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 48;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 49;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,64});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,64});      state = 50;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,65});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,65});      state = 51;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,7});      state = 52;      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){SHIFT,86});      state = 53;      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,5});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,5});      set\_parsr\_action(parser, state, DOLLAR\_TOKEN, (Action){REDUCE,5});      state = 54;      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,11});      state = 55;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,12});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,12});      state = 56;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,13});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,13});      state = 57;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,14});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,14});      state = 58;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,17});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,17});      state = 59;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,18});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,18});      state = 60;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,19});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,19});      state = 61;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 62;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){SHIFT,89});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 63;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,22});      state = 64;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,27});      state = 65;      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,62});      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,60});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,60});      state = 66;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 67;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 68;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 69;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,20});      state = 70;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,21});      state = 71;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 72;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 73;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 74;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 75;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 76;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 77;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 78;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 79;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 80;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 81;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});    state = 82;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 83;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){SHIFT,107});      state = 84;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,62});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,62});      state = 85;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,63});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,63});      state = 86;      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,4});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,4});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,4});      set\_parsr\_action(parser, state, DOLLAR\_TOKEN, (Action){REDUCE,4});      state = 87;      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,26});      state = 88;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){SHIFT,108});      state = 89;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,30});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,30});      state = 90;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,31});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){SHIFT,109});      state = 91;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,33});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,33});      state = 92;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){SHIFT,110});      state = 93;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){SHIFT,111});      state = 94;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,41});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,41});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,41});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,41});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){SHIFT,71});      state = 95;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,43});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,43});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,43});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,43});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,43});      state = 96;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,45});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,45});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,45});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,45});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,45});      state = 97;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,46});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,46});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,46});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,46});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,46});      state = 98;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,47});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,47});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,47});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,47});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,47});      state = 99;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,48});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,48});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,48});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,48});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,48});      state = 100;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,49});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,49});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,49});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,49});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,49});      state = 101;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,50});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,50});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,50});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,50});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,50});      state = 102;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,52});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,52});      state = 103;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,53});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,53});      state = 104;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,55});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,55});      state = 105;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,56});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,56});      state = 106;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,57});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,57});      state = 107;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,58});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,58});      state = 108;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, COMMA\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, SEMICOLON\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, LOGICAL\_OR\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, LOGICAL\_AND\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, EQUAL\_TO\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, NOT\_EQUAL\_TO\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, LESS\_THEN\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, GREATER\_THEN\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, PLUS\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, MULTIPLY\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, DIVIDE\_TOKEN, (Action){REDUCE,29});      set\_parsr\_action(parser, state, MODULO\_TOKEN, (Action){REDUCE,29});      state = 109;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,65});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 110;      set\_parsr\_action(parser, state, OPEN\_BRACE\_TOKEN, (Action){SHIFT,113});      state = 111;      set\_parsr\_action(parser, state, OPEN\_BRACE\_TOKEN, (Action){SHIFT,114});      state = 112;      set\_parsr\_action(parser, state, CLOSE\_PARENTHESES\_TOKEN, (Action){REDUCE,32});      state = 113;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,31});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){SHIFT,37});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){SHIFT,38});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){SHIFT,33});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){SHIFT,34});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){SHIFT,35});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 114;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,31});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){SHIFT,37});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){SHIFT,38});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){SHIFT,33});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){SHIFT,34});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){SHIFT,35});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 115;      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){SHIFT,117});      state = 116;      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){SHIFT,118});      state = 117;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, ELSE\_TOKEN, (Action){SHIFT,120});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,35});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,35});      state = 118;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,38});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,38});      state = 119;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,34});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,34});      state = 120;      set\_parsr\_action(parser, state, OPEN\_BRACE\_TOKEN, (Action){SHIFT,122});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){SHIFT,34});      state = 121;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,36});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,36});      state = 122;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){SHIFT,31});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){SHIFT,44});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){SHIFT,37});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){SHIFT,38});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){SHIFT,5});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){SHIFT,6});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){SHIFT,7});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){SHIFT,33});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){SHIFT,34});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){SHIFT,35});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){SHIFT,47});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){SHIFT,48});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){SHIFT,49});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){SHIFT,50});      state = 123;      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){SHIFT,124});      state = 124;      set\_parsr\_action(parser, state, IDENTIFIER\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, OPEN\_PARENTHESES\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, CLOSE\_BRACE\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, GETCHAR\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, PUTCHAR\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, INT\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, CHAR\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, VOID\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, RETURN\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, IF\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, WHILE\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, MINUS\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, LOGICAL\_NOT\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, DIGIT\_TOKEN, (Action){REDUCE,37});      set\_parsr\_action(parser, state, CHAR\_LITERAL\_TOKEN, (Action){REDUCE,37});  }  void init\_goto\_table(Parser\* parser){      int state = 0;      state = 0;      set\_parser\_goto(parser, state, NON\_TERMINAL\_PROGRAM, 1);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTIONS, 2);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION, 3);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 4);      state = 1;      state = 2;      state = 3;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTIONS, 8);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION, 3);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 4);      state = 4;      state = 5;      state = 6;      state = 7;      state = 8;      state = 9;      state = 10;      set\_parser\_goto(parser, state, NON\_TERMINAL\_PARAMETERS, 11);      set\_parser\_goto(parser, state, NON\_TERMINAL\_PARAMETER, 13);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 14);      state = 11;      state = 12;      state = 13;      state = 14;      state = 15;      state = 16;      set\_parser\_goto(parser, state, NON\_TERMINAL\_BODY, 20);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENTS, 21);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENT, 22);      set\_parser\_goto(parser, state, NON\_TERMINAL\_GETCHAR, 29);      set\_parser\_goto(parser, state, NON\_TERMINAL\_PUTCHAR, 30);      set\_parser\_goto(parser, state, NON\_TERMINAL\_DECLARATION, 24);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 32);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ASSIGNMENT, 23);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RETURN, 25);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_IF\_STATEMENT, 26);      set\_parser\_goto(parser, state, NON\_TERMINAL\_WHILE\_LOOP, 27);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 28);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 17;      set\_parser\_goto(parser, state, NON\_TERMINAL\_PARAMETERS, 51);      set\_parser\_goto(parser, state, NON\_TERMINAL\_PARAMETER, 13);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 14);      state = 18;      state = 19;      set\_parser\_goto(parser, state, NON\_TERMINAL\_BODY, 52);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENTS, 21);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENT, 22);      set\_parser\_goto(parser, state, NON\_TERMINAL\_GETCHAR, 29);      set\_parser\_goto(parser, state, NON\_TERMINAL\_PUTCHAR, 30);      set\_parser\_goto(parser, state, NON\_TERMINAL\_DECLARATION, 24);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 32);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ASSIGNMENT, 23);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RETURN, 25);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_IF\_STATEMENT, 26);      set\_parser\_goto(parser, state, NON\_TERMINAL\_WHILE\_LOOP, 27);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 28);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 20;      state = 21;      state = 22;      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENTS, 54);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENT, 22);      set\_parser\_goto(parser, state, NON\_TERMINAL\_GETCHAR, 29);      set\_parser\_goto(parser, state, NON\_TERMINAL\_PUTCHAR, 30);      set\_parser\_goto(parser, state, NON\_TERMINAL\_DECLARATION, 24);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 32);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ASSIGNMENT, 23);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RETURN, 25);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_IF\_STATEMENT, 26);      set\_parser\_goto(parser, state, NON\_TERMINAL\_WHILE\_LOOP, 27);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 28);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 23;      state = 24;      state = 25;      state = 26;      state = 27;      state = 28;      state = 29;      state = 30;      state = 31;      state = 32;      state = 33;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 64);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 34;      state = 35;      state = 36;      state = 37;      state = 38;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 70);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 39;      state = 40;      state = 41;      state = 42;      state = 43;      state = 44;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 83);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 45;      state = 46;      state = 47;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 84);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 48;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 85);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 49;      state = 50;      state = 51;      state = 52;      state = 53;      state = 54;      state = 55;      state = 56;      state = 57;      state = 58;      state = 59;      state = 60;      state = 61;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 87);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 62;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ARGUMENTS, 88);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ARGUMENT, 90);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 91);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 63;      state = 64;      state = 65;      state = 66;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 92);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 67;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 93);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 68;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 94);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 69;      state = 70;      state = 71;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 95);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 72;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 96);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 73;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 97);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 74;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 98);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 75;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 99);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 76;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 100);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 77;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 101);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 78;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 102);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 79;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 103);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 80;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 104);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 81;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 105);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 82;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 106);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 83;      state = 84;      state = 85;      state = 86;      state = 87;      state = 88;      state = 89;      state = 90;      state = 91;      state = 92;      state = 93;      state = 94;      state = 95;      state = 96;      state = 97;      state = 98;      state = 99;      state = 100;      state = 101;      state = 102;      state = 103;      state = 104;      state = 105;      state = 106;      state = 107;      state = 108;      state = 109;      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ARGUMENTS, 112);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ARGUMENT, 90);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 91);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 110;      state = 111;      state = 112;      state = 113;      set\_parser\_goto(parser, state, NON\_TERMINAL\_BODY, 115);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENTS, 21);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENT, 22);      set\_parser\_goto(parser, state, NON\_TERMINAL\_GETCHAR, 29);      set\_parser\_goto(parser, state, NON\_TERMINAL\_PUTCHAR, 30);      set\_parser\_goto(parser, state, NON\_TERMINAL\_DECLARATION, 24);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 32);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ASSIGNMENT, 23);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RETURN, 25);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_IF\_STATEMENT, 26);      set\_parser\_goto(parser, state, NON\_TERMINAL\_WHILE\_LOOP, 27);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 28);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 114;      set\_parser\_goto(parser, state, NON\_TERMINAL\_BODY, 116);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENTS, 21);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENT, 22);      set\_parser\_goto(parser, state, NON\_TERMINAL\_GETCHAR, 29);      set\_parser\_goto(parser, state, NON\_TERMINAL\_PUTCHAR, 30);      set\_parser\_goto(parser, state, NON\_TERMINAL\_DECLARATION, 24);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 32);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ASSIGNMENT, 23);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RETURN, 25);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_IF\_STATEMENT, 26);      set\_parser\_goto(parser, state, NON\_TERMINAL\_WHILE\_LOOP, 27);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 28);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 115;      state = 116;      state = 117;      set\_parser\_goto(parser, state, NON\_TERMINAL\_ELSE\_IF\_OR\_ELSE, 119);      state = 118;      state = 119;      state = 120;      set\_parser\_goto(parser, state, NON\_TERMINAL\_IF\_STATEMENT, 121);      state = 121;      state = 122;      set\_parser\_goto(parser, state, NON\_TERMINAL\_BODY, 123);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENTS, 21);      set\_parser\_goto(parser, state, NON\_TERMINAL\_STATEMENT, 22);      set\_parser\_goto(parser, state, NON\_TERMINAL\_GETCHAR, 29);      set\_parser\_goto(parser, state, NON\_TERMINAL\_PUTCHAR, 30);      set\_parser\_goto(parser, state, NON\_TERMINAL\_DECLARATION, 24);      set\_parser\_goto(parser, state, NON\_TERMINAL\_TYPE, 32);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ASSIGNMENT, 23);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RETURN, 25);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FUNCTION\_CALL, 45);      set\_parser\_goto(parser, state, NON\_TERMINAL\_IF\_STATEMENT, 26);      set\_parser\_goto(parser, state, NON\_TERMINAL\_WHILE\_LOOP, 27);      set\_parser\_goto(parser, state, NON\_TERMINAL\_EXPRESSION, 28);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_OR\_TERM, 36);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LOGICAL\_AND\_TERM, 39);      set\_parser\_goto(parser, state, NON\_TERMINAL\_RELATIONAL\_TERM, 40);      set\_parser\_goto(parser, state, NON\_TERMINAL\_ADDITIVE\_TERM, 41);      set\_parser\_goto(parser, state, NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 42);      set\_parser\_goto(parser, state, NON\_TERMINAL\_FACTOR, 43);      set\_parser\_goto(parser, state, NON\_TERMINAL\_LITERAL, 46);      state = 123;      state = 124;  }  void init\_Production\_Rules(Parser\* parser){      // intiating the parser production rules array      int i = 0;      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_THE\_PROGRAM, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_PROGRAM, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FUNCTIONS, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FUNCTIONS, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FUNCTION, 8};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FUNCTION, 7};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_PARAMETERS, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_PARAMETERS, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_PARAMETER, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_BODY, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENTS, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENTS, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENT, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENT, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENT, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENT, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENT, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENT, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENT, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_STATEMENT, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_GETCHAR, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_PUTCHAR, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_DECLARATION, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_TYPE, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_TYPE, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_TYPE, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_ASSIGNMENT, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_RETURN, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_RETURN, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FUNCTION\_CALL, 4};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FUNCTION\_CALL, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_ARGUMENTS, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_ARGUMENTS, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_ARGUMENT, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_IF\_STATEMENT, 8};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_IF\_STATEMENT, 7};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_ELSE\_IF\_OR\_ELSE, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_ELSE\_IF\_OR\_ELSE, 4};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_WHILE\_LOOP, 7};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_EXPRESSION, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_LOGICAL\_OR\_TERM, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_LOGICAL\_OR\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_LOGICAL\_AND\_TERM, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_LOGICAL\_AND\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_RELATIONAL\_TERM, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_RELATIONAL\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_RELATIONAL\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_RELATIONAL\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_RELATIONAL\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_RELATIONAL\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_RELATIONAL\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_ADDITIVE\_TERM, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_ADDITIVE\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_ADDITIVE\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_MULTIPLICATIVE\_TERM, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FACTOR, 3};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FACTOR, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FACTOR, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FACTOR, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FACTOR, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_FACTOR, 2};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_LITERAL, 1};      parser->Production\_Rules[i++] = (Production\_Rule){NON\_TERMINAL\_LITERAL, 1};  } |

### ParserTree.h

נמצא תחת Parser.h עקב יציבות תלויות.

### ParserTree.c

|  |
| --- |
| #include"../Parser.h"  void parser\_treeNode\_addChild(TreeNode \*paraent, TreeNode \*child, int numberOfChildren){      //printf("adding children in place: %d out of %d\n",numberOfChildren - paraent->numberOfChildrens - 1, numberOfChildren );      int index = numberOfChildren - paraent->numberOfChildrens - 1;      paraent->childrens[index] = child;      paraent->numberOfChildrens++;  }  TreeNode\* parser\_treeNode\_create\_TreeNode(){      // allocating new memory      TreeNode\* node = (TreeNode\*)malloc(sizeof(TreeNode));        // checking malloc sucess      if(node == NULL){          printf("MALLOC ERROR");          //return NULL;          exit(1);      }        // setting values to diffault      node->childrens = NULL;      node->numberOfChildrens = 0;      node->value = NULL;      node->scope = NULL;        // return new node      return node;  }  void parser\_treeNode\_set\_type(TreeNode\* node, TreeNodeType treeType){      node->node\_type = treeType;  }  void parser\_treeNode\_set\_terminal(TreeNode\* node, Terminals terminal){      // setting the type of terminal to the tree node      // assuming the type is terminal      node->Kind.TerminalType = terminal;  }  void parser\_treeNode\_set\_non\_terminal(TreeNode\* node, Non\_Terminals nonTerminal){      // setting the type of the non-terminal      // assuming given tree is non terminal type      node->Kind.nonTerminalType = nonTerminal;  } |

### Parser.h

|  |
| --- |
| #pragma once  #include <windows.h>  #include"../Token/Token.h"  #include"../ErrorReporting/ErrorReporting.h"  #include<stdlib.h>  // Forward declarations to break dependency loops  // ------------------------------------  // Parser.h structs and costants  // ------------------------------------  #define PARSER\_NUMBER\_OF\_STATES 134  #define PARSER\_NUMBER\_OF\_NON\_TERMINALS 40  #define PARSER\_NUMBER\_OF\_TERMINALS 40  typedef enum \_non\_terminals {      NON\_TERMINAL\_THE\_PROGRAM,NON\_TERMINAL\_PROGRAM,NON\_TERMINAL\_FUNCTIONS,      NON\_TERMINAL\_FUNCTION,NON\_TERMINAL\_ARGUMENTS,NON\_TERMINAL\_ARGUMENT,      NON\_TERMINAL\_PARAMETERS,NON\_TERMINAL\_PARAMETER,NON\_TERMINAL\_BODY,      NON\_TERMINAL\_STATEMENTS, NON\_TERMINAL\_STATEMENT,NON\_TERMINAL\_GETCHAR,      NON\_TERMINAL\_PUTCHAR,NON\_TERMINAL\_DECLARATION,NON\_TERMINAL\_TYPE,      NON\_TERMINAL\_ASSIGNMENT,NON\_TERMINAL\_RETURN,NON\_TERMINAL\_FUNCTION\_CALL,      NON\_TERMINAL\_IF\_STATEMENT,NON\_TERMINAL\_ELSE\_IF\_OR\_ELSE,      NON\_TERMINAL\_WHILE\_LOOP,NON\_TERMINAL\_EXPRESSION,      NON\_TERMINAL\_LOGICAL\_OR\_TERM,NON\_TERMINAL\_LOGICAL\_AND\_TERM,      NON\_TERMINAL\_RELATIONAL\_TERM,NON\_TERMINAL\_ADDITIVE\_TERM,      NON\_TERMINAL\_MULTIPLICATIVE\_TERM,NON\_TERMINAL\_FACTOR,NON\_TERMINAL\_LITERAL  } Non\_Terminals;  // ------------------------------------  // ParserTable.h structs and costants  // ------------------------------------  typedef enum Action\_type { SHIFT, REDUCE, ACCEPT, ERROR\_ACTION } ActionType;  typedef struct \_Action {      ActionType typeOfAction;      int rule\_or\_state;  } Action;  typedef struct \_parse\_table {      Action ActionMatrix[PARSER\_NUMBER\_OF\_STATES][PARSER\_NUMBER\_OF\_TERMINALS];      int GotoMatrix[PARSER\_NUMBER\_OF\_STATES][PARSER\_NUMBER\_OF\_NON\_TERMINALS];  } parse\_table;  // productio rules  typedef struct \_Production\_Rule {      Non\_Terminals Production\_Rule\_Type;      int numberOfRightProductions;  } Production\_Rule;  typedef struct \_parser {      parse\_table parse\_table;      Production\_Rule Production\_Rules[PARSER\_NUMBER\_OF\_NON\_TERMINALS];  } Parser;  // ------------------------------------  // ParserTree.h structs and costants  // ------------------------------------  typedef enum \_treeNodeType {      TERMINAL,      NON\_TERMINAL  } TreeNodeType;  typedef struct \_TreeNode {      // the type of the node      TreeNodeType node\_type;      union parser {          Non\_Terminals nonTerminalType;          Terminals TerminalType;      } Kind;        void\* value; // for int, char, identifier terminals      // for reporting an error      int SourceCodeLine;  // the line of the error      int CharacterInSourceCodeLine; // the character of the error      // pointer to childrens      struct \_TreeNode\*\* childrens;      // count the number of childrens      int numberOfChildrens;      // point into the current scope      void\* scope;  } TreeNode;  // ------------------------------------  // ParserStack.h structs and costants  // ------------------------------------  typedef struct \_Stack\_Element {      int state;      TreeNode \*node;  } Stack\_Element;  typedef struct \_Parser\_Stack {      Stack\_Element \*\*arr;      int size;      int memorySize;  } Parser\_Stack;    // Parser.h function prototypes  Parser\* parser\_init();  Stack\_Element\* create\_stack\_element(TreeNode\* tree, int state);  Action parser\_get\_action(Parser\* parser, int CurrentState, Terminals CurrentTerminal);  int parser\_get\_goto(Parser\* parser, int CurrentState, Non\_Terminals CnonTerminal);  void parser\_add\_token\_Attr\_to\_treeNode(TreeNode\* tree, Token\* token);  int should\_add\_to\_abstact\_syntax\_tree(Terminals token);  const char\* actionTypeToString(ActionType type);  void convertTokenFormat(Token\* tokens);  TreeNode\* create\_abstact\_syntax\_tree(Parser\* parser, Token\* tokens);  int countMagnitudedElementsInTheStack(Parser\_Stack \*stack, int num);  void freeAst(TreeNode\* root);  // ------------------------------------  // ParserTree.h contents  // ------------------------------------  // ParserTree function prototypes  void parser\_treeNode\_addChild(TreeNode \*, TreeNode \*, int);  TreeNode\* parser\_treeNode\_create\_TreeNode();  void parser\_treeNode\_set\_type(TreeNode\*, TreeNodeType);  void parser\_treeNode\_set\_terminal(TreeNode\* , Terminals );  void parser\_treeNode\_set\_non\_terminal(TreeNode\*, Non\_Terminals);  // ------------------------------------  // ParserStack.h contents  // ------------------------------------  // ParserStack function prototypes  Parser\_Stack\* parser\_stack\_createStack();  void parser\_stack\_push(Parser\_Stack \*, Stack\_Element \*);  Stack\_Element\* parser\_stack\_pop(Parser\_Stack \*stack);  Stack\_Element\* parser\_stack\_top(Parser\_Stack stack);  int parser\_stack\_isEmpty(Parser\_Stack stack);  void parser\_stack\_destroyStack(Parser\_Stack \*stack);  // parserTable functions prototype  int get\_index\_from\_terminal\_parser(Terminals);  int get\_index\_from\_Non\_terminal\_parser(Non\_Terminals);  void set\_parsr\_action(Parser\*, int, Terminals, Action);  void set\_parser\_goto(Parser\*, int, Non\_Terminals, int);  void init\_action\_table(Parser\*);  void init\_goto\_table(Parser\*);  void init\_Production\_Rules(Parser\*); |

### Parser.c

|  |
| --- |
| #include"Parser.h"  Parser\* parser\_init() {      // Determine the size of the Parser object      size\_t size = sizeof(Parser); // Adjust the size to match your Parser structure      // Use VirtualAlloc to allocate the memory for the parser      Parser\* parser = (Parser\*)VirtualAlloc(NULL, size, MEM\_COMMIT | MEM\_RESERVE, PAGE\_READWRITE);        if (parser == NULL) {          // If allocation fails, print error and exit          printf("VirtualAlloc failed with error code: %lu\n", GetLastError());          exit(1);      }      if (!HeapValidate(GetProcessHeap(), 0, NULL)) {          printf("Heap is corrupted!\n");          exit(10);      }        // Set the allocated memory to zero      memset(parser, 0, size);        // Debugging: Print out the size of the parser object      //printf("The size of the parser is: %zu bytes\n", sizeof(Parser));        // Return the pointer to the allocated memory for the parser object      return parser;  }  Stack\_Element\* create\_stack\_element(TreeNode\* tree, int state){        Stack\_Element\* newElement = (Stack\_Element\*)malloc(sizeof(Stack\_Element));      if(newElement == NULL){          printf("MALLOC ERROR");          exit(1);      }      newElement->state = state;      newElement->node = tree;      return newElement;  }  Action parser\_get\_action(Parser\* parser, int CurrentState, Terminals CurrentTerminal){      return parser->parse\_table.ActionMatrix[CurrentState][get\_index\_from\_terminal\_parser(CurrentTerminal)];  }  int parser\_get\_goto(Parser\* parser, int CurrentState, Non\_Terminals CnonTerminal){      return parser->parse\_table.GotoMatrix[CurrentState][get\_index\_from\_Non\_terminal\_parser(CnonTerminal)];  }  void parser\_add\_token\_Attr\_to\_treeNode(TreeNode\* tree, Token\* token){      parser\_treeNode\_set\_type(tree, TERMINAL);      parser\_treeNode\_set\_terminal(tree, (\*token).type);      if((\*token).type == DIGIT\_TOKEN){          int\* iPtr = (int\*)malloc(sizeof(int));          if(iPtr == NULL){              perror("[ERROR]");              exit(EXIT\_FAILURE);          }          \*iPtr = token->value.integer;          tree->value = (void\*)(iPtr);      }      else if((\*token).type == CHAR\_LITERAL\_TOKEN){          char\* cPtr =  (char\*)malloc(sizeof(char));          if(cPtr == NULL){              perror("[ERROR]");              exit(EXIT\_FAILURE);          }          \*cPtr = token->value.ch;          tree->value = (void\*)(cPtr);      }      else if((\*token).type == IDENTIFIER\_TOKEN){          tree->value = (void\*)((token->value.identifer));      }      // adding the token line and char in line for reporting errros      tree->SourceCodeLine = token->SourceCodeLine;      tree->CharacterInSourceCodeLine = token->CharacterInSourceCodeLine;      //printf("so far, so good...");  }  int should\_add\_to\_abstact\_syntax\_tree(Terminals token) {      // Tokens that have semantic meaning and should be added to the stack      switch (token) {          case IDENTIFIER\_TOKEN:          case DIGIT\_TOKEN:          case CHAR\_LITERAL\_TOKEN:          //case GETCHAR\_TOKEN:          //case PUTCHAR\_TOKEN:          case INT\_TOKEN:          case CHAR\_TOKEN:          case VOID\_TOKEN:          //case RETURN\_TOKEN:          //case IF\_TOKEN:          case ELSE\_TOKEN:          //case FOR\_TOKEN:          //case WHILE\_TOKEN:          case LOGICAL\_OR\_TOKEN:          case LOGICAL\_AND\_TOKEN:          case EQUAL\_TO\_TOKEN:          case NOT\_EQUAL\_TO\_TOKEN:          case LESS\_THEN\_TOKEN:          case GREATER\_THEN\_TOKEN:          case LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN:          case GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN:          case PLUS\_TOKEN:          case MINUS\_TOKEN:          case MULTIPLY\_TOKEN:          case DIVIDE\_TOKEN:          case MODULO\_TOKEN:          case LOGICAL\_NOT\_TOKEN:              return 1; // Add to stack          default:              return 0; // Do not add to stack      }  }  const char\* actionTypeToString(ActionType type) {      switch (type) {          case SHIFT:            return "SHIFT";          case REDUCE:           return "REDUCE";          case ACCEPT: return "ACCEPT";            default: return "UNKNOWN";      }  }  int isValidSyncronizationToken(Token\* token) {      // this function return true if the given token can be used      // as a sybcronization token while doing error recovrry      if (token == NULL) {          return 0; // Cannot synchronize on NULL      }      switch (token->type) {          // --- Reliable tokens,          case SEMICOLON\_TOKEN:          case CLOSE\_PARENTHESES\_TOKEN:          case CLOSE\_BRACE\_TOKEN:          case IF\_TOKEN:          case ELSE\_TOKEN:          case WHILE\_TOKEN:          case RETURN\_TOKEN:          case GETCHAR\_TOKEN:          case PUTCHAR\_TOKEN:          case INT\_TOKEN:          case CHAR\_TOKEN:          case VOID\_TOKEN:              return 1;          default:              // Any other token is not considered a primary synchronization point.              return 0;      }  }  Action panicMode\_Adjust\_Parser\_Stack(Parser\* parser, Parser\_Stack\* stack, Token\* token){      // This function is used when entered panic mode.      // It adjusts the stack by popping elements until it finds a state      // that can perform a valid action (usually shift) on the synchronization token.      Action RecoveryAction;      int CurrentState;      Stack\_Element\* tmpStackElement;        // Loop until we find a state that can handle the synchronization token      // or the stack becomes empty.      while (!parser\_stack\_isEmpty(\*stack)) {          CurrentState = parser\_stack\_top(\*stack)->state;          RecoveryAction = parser\_get\_action(parser, CurrentState, token->type);          //printf("looping..");          // Check if the current state on top of the stack can handle          // the synchronization token (i.e., the action is NOT an error action).          // An error action is typically represented by { type=0, rule\_or\_state=0 }          // or some specific ERROR type if you defined one.          if ((RecoveryAction.typeOfAction == SHIFT && RecoveryAction.rule\_or\_state != 0) || RecoveryAction.typeOfAction == REDUCE){                // Found a state that can handle the synchronization token.              // Return the state we need to shift \*to\* based on the recovery action.              // For panic mode recovery, we usually expect to SHIFT the sync token.                return RecoveryAction;          }          // If the current state cannot handle the sync token, pop it.          //printf("Panic Mode: Popping state %d from stack.\n", CurrentState); // Debugging          tmpStackElement = parser\_stack\_pop(stack);          //printf("poped element from the stack\n");          //printf("the element is: %s\n", (tmpStackElement->node->node\_type == TERMINAL) ?"TERMINAL": "NON TERMINAL");          //printf("the value is: %d",(tmpStackElement->node->node\_type == TERMINAL) ? tmpStackElement->node->Kind.TerminalType : tmpStackElement->node->Kind.nonTerminalType);          if (tmpStackElement) {               // Free the associated AST node if it exists               // Avoid freeing the initial NULL node pushed at state 0              if(tmpStackElement->node != NULL) {                    free(tmpStackElement->node->value); // Free value if allocated                  free(tmpStackElement->node->childrens); // Free children array if allocated                  free(tmpStackElement->node); // Free the node itself              }              free(tmpStackElement); // Free the stack element wrapper          }      }      // If the stack becomes empty, we couldn't find a recovery state.      //printf("Panic Mode Recovery Failed: Stack emptied.\n");      return (Action){-1, -1}; // Indicate failure to recover  }  void panicMode\_Skip\_Unsynchronization\_Points(Token\*\* tokens){      // this function is used in panic mode to skipp unsynchronize points      // firstly, discarding the first token, which causes the eror      (\*tokens)++;        // skipping tokens      while((\*tokens)->type != DOLLAR\_TOKEN && !isValidSyncronizationToken(\*tokens)){          (\*tokens)++;      }    }  void convertTokenFormat(Token\* tokens){      while(tokens[0].type != END\_OF\_TOKENS\_TOKEN) tokens++;      tokens[0].type = DOLLAR\_TOKEN;  }  void parser\_shift(Parser\_Stack\* stack, Token\* token, int state){      // this function responsible for shifting action in the parsing algorithm      // Shift action: create new tree from the token and push it onto the stack      // define local variables      TreeNode\* tempTreeNode;      Stack\_Element\* stack\_element;        // creating new tree node      tempTreeNode = parser\_treeNode\_create\_TreeNode();        // adding right attributes for the new node      parser\_add\_token\_Attr\_to\_treeNode(tempTreeNode, token);        // creating new stack element with the new tree node and the corresponding valu      stack\_element = create\_stack\_element(tempTreeNode, state);        // pusing the new element to the stack      parser\_stack\_push(stack, stack\_element);  }  void parser\_reduce(Parser\* parser, Parser\_Stack\* stack, int rule){      // this function is preforming the reduce action in the parsing alorithm      // Reduce action: create new non-terminal node and reduce the stack elements      // define local variables      TreeNode\* tempTreeNode;      TreeNode\* lastSon;      Stack\_Element\* tempStackElement;      int numerOfProductions;      int numerOfProductionsInAst;      int i;      int nextState;        tempTreeNode = parser\_treeNode\_create\_TreeNode();        parser\_treeNode\_set\_type(tempTreeNode, NON\_TERMINAL);        numerOfProductions = parser->Production\_Rules[rule].numberOfRightProductions;      parser\_treeNode\_set\_non\_terminal(tempTreeNode, parser->Production\_Rules[rule].Production\_Rule\_Type);        // checking how many childes should be in the tree      numerOfProductionsInAst = countMagnitudedElementsInTheStack(stack, numerOfProductions);        // Allocate memory for children of the new tree      tempTreeNode->childrens = (TreeNode\*\*)malloc(sizeof(TreeNode\*) \* numerOfProductionsInAst);      if(tempTreeNode->childrens == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }      for (i = 0; i < numerOfProductions; i++) {            tempStackElement = parser\_stack\_pop(stack);          if(tempStackElement == NULL){              printf("COMPILER ERROR");              exit(EXIT\_FAILURE);          }            // checking if element has magnitude for the AST          if (!((tempStackElement->node)->node\_type == TERMINAL && !should\_add\_to\_abstact\_syntax\_tree((tempStackElement->node)->Kind.TerminalType))) {              // adding the element tree node as child of the new node              parser\_treeNode\_addChild(tempTreeNode, tempStackElement->node, numerOfProductionsInAst);              // saving the last sone that were added as a child in the ast              lastSon = tempStackElement->node;          }            // Free the stack element after processing,          free(tempStackElement);      }      tempTreeNode->SourceCodeLine = lastSon->SourceCodeLine;      tempTreeNode->CharacterInSourceCodeLine = lastSon->CharacterInSourceCodeLine;      // Go to next state using the "goto" table      nextState = parser\_get\_goto(parser, (\*parser\_stack\_top(\*stack)).state, tempTreeNode->Kind.nonTerminalType);      tempStackElement = create\_stack\_element(tempTreeNode, nextState);        // pusing the nre element to the stack      parser\_stack\_push(stack, tempStackElement);  }  TreeNode\* parser\_accept(Parser\_Stack\* stack){      // Accept action: pop the top of the stack and return the tree node      // define local variables      Stack\_Element\* tempStackElement;      TreeNode\* tempTreeNode;      // poping the top element, this element containg the souce code full AST      tempStackElement = parser\_stack\_pop(stack);      tempTreeNode = tempStackElement->node;      free(tempStackElement);  // Free the stack element after processing        // destroying the stack      parser\_stack\_destroyStack(stack);      // returning the ast top node      return tempTreeNode;  }  void parser\_print\_parser\_stack(Parser\_Stack\* stack){      printf("Stack: ");      Parser\_Stack \*tmp\_stack = stack;      int bla = 0;      printf("[ ");      while(bla < stack->size){          printf(" %d,", stack->arr[bla]->state);          bla++;      }      printf("]\n");  }  void parser\_recovery\_error(Parser\* parser, Parser\_Stack\* stack, Token\*\* tokens){      // define local variables      Action recoveryAction; // THIS WILL BE THE FIRST VALID ACTION WHICH WILL BE USED FOR THE RECOVERY      //printf("Entering Panic Mode Recovery...\n"); // Debugging      // Skip input tokens until a synchronization token or EOF is found.      panicMode\_Skip\_Unsynchronization\_Points(tokens);      // Check if we hit the end of the input stream.      if((\*tokens)->type == DOLLAR\_TOKEN){          // Cannot recover if the sync point is the end of the file.          // The error likely involves expecting something before EOF.          printf("Syntax Error: Unexpected end of file during error recovery.\n");          // Depending on requirements, you might exit, return NULL, or try a final reduction.          exit(EXIT\_FAILURE); // Simplest approach for now      }      //printf("the stack before:\n");      //parser\_print\_parser\_stack(stack);      // Adjust the parser stack to find a state that can handle the sync token.      recoveryAction = panicMode\_Adjust\_Parser\_Stack(parser, stack, \*tokens);      //printf("\nthe stack after:\n");      //parser\_print\_parser\_stack(stack);      // 4. Check if stack adjustment was successful.      if(recoveryAction.rule\_or\_state == -1 && recoveryAction.typeOfAction == -1){          // Stack adjustment failed (e.g., stack became empty). Recovery failed.          printf("Fatal Error: Cannot recover from syntax error near line %d. Stack adjustment failed.\n", (\*tokens)->SourceCodeLine);          // Report error more formally if needed          exit(EXIT\_FAILURE); // Cannot continue parsing      }      // 5. Perform the recovery action: Shift the synchronization token, or reduce it      if(recoveryAction.typeOfAction == SHIFT){          //printf("Recovered: Shifting token "); // Debugging          //printTokenTypeName((\*tokens)->type);          //printf(" into state %d\n", recoveryAction.rule\_or\_state);          parser\_shift(stack, \*tokens, recoveryAction.rule\_or\_state);          // getting the next token          (\*tokens)++;      }      else if(recoveryAction.typeOfAction == REDUCE){          //printf("Recovered: reduce token\n");          parser\_reduce(parser, stack, recoveryAction.rule\_or\_state);      }        //printf("Exiting Panic Mode Recovery.\n"); // Debugging    }  char\* Parser\_PrepareErrorMessage(Parser\* parser, int state ,Token\* invalidToken){      // define local variables      // temporery buffers      char msg[1000];      char msg2[1000];      // integer for iteration      int i;      // tell if we are in the first token      int isFirstValidToken = 1;      // temp structures      Action tmpAction;      Token tmpToken;      // getting the name of the invalid token      char \*token\_name = Lexer\_GetTokenName(invalidToken);      //printf("token is %s\n", (token\_name == NULL)?"NULL":"NOT NULL");        // setting the message with using convinient format      sprintf(msg, "unexpected token: '%s', expecting: ", token\_name);      //looking for all valid tokens at current state      for(i = 0 ; i < END\_OF\_TOKENS\_TOKEN; i++){          // getting the action of the current index          tmpAction = parser->parse\_table.ActionMatrix[state][i];          // checking if the current index(which represen a terminal) is valid          if((tmpAction.typeOfAction == SHIFT && tmpAction.rule\_or\_state != 0) || tmpAction.typeOfAction == REDUCE){              // the current index represent a valid token!              // adding the token into the error message as vaild possible token              // setting the temp token type              tmpToken.type = i;              // getting the token name              token\_name = Lexer\_GetTokenName(&tmpToken);              // checking if this is the first iteration              if(isFirstValidToken){                  sprintf(msg2, "'%s'", token\_name);                  isFirstValidToken = 0;              }              else{                  sprintf(msg2, " or '%s'", token\_name);              }                // concating the messages              strcat(msg, msg2);            }      }      // returning an dynamic memory version of the buffer msg      return \_strdup(msg);  }  TreeNode\* create\_abstact\_syntax\_tree(Parser\* parser, Token\* tokens){      // ... (setup code remains the same) ...      Parser\_Stack \*stack = parser\_stack\_createStack();      int CurrentState;      Action CurrentAction;      int panicMode = 0;      char\* errorMsg; // hold the error message before outputing      convertTokenFormat(tokens);      Stack\_Element\* initialElement = create\_stack\_element(NULL, 0); // State 0, no node      parser\_stack\_push(stack, initialElement);      while (1) {          if (parser\_stack\_isEmpty(\*stack)) {              printf("Fatal Error: Parser stack became empty unexpectedly.\n");              // Handle this error case, maybe return NULL or exit              // This shouldn't happen in a correct SLR(1) implementation unless recovery fails badly              exit(EXIT\_FAILURE);          }          CurrentState = parser\_stack\_top(\*stack)->state;          // Debugging: Print stack and current token          // parser\_print\_parser\_stack(stack);          //printf("Current Token: "); printTokenTypeName(tokens[0].type); printf("\n");          CurrentAction = parser\_get\_action(parser, CurrentState, tokens[0].type);          // Check for Syntax Error (Invalid entry in Action Table)          // Assuming Error is marked by type=0, rule\_or\_state=0          if(CurrentAction.typeOfAction == 0 && CurrentAction.rule\_or\_state == 0){              // --- Syntax Error Detected ---              panicMode = 1; // Set flag              // outputing the base error message              ErrorReporting\_ReportError\_Base(ERROR\_ON\_PARSER\_PHASE,                  tokens[0].SourceCodeLine,                  tokens[0].CharacterInSourceCodeLine);                // getting the detailed error messge              errorMsg = Parser\_PrepareErrorMessage(parser, CurrentState, tokens);                // outputing the error message              ErrorReporting\_ReportError(errorMsg);                // freeing the error msg              free(errorMsg);              // Attempt Panic Mode Recovery              parser\_recovery\_error(parser, stack, &tokens);              // After recovery, the stack and current token are adjusted.              // Continue the loop to process the (potentially new) current token              // with the state now on top of the stack.          }          else if (CurrentAction.typeOfAction == SHIFT) {              parser\_shift(stack, tokens, CurrentAction.rule\_or\_state);              tokens++; // Consume token only on successful shift          }          else if (CurrentAction.typeOfAction == REDUCE) {              parser\_reduce(parser, stack, CurrentAction.rule\_or\_state);              // Do NOT advance token on reduce          }          else if (CurrentAction.typeOfAction == ACCEPT) {              // Check if the ACCEPT happens on the DOLLAR\_TOKEN              if (tokens[0].type == DOLLAR\_TOKEN) {                  if(panicMode){                      freeAst(parser\_accept(stack));                      return NULL;                  }                  else{                      //printf("Parse Successful (ACCEPT)\n");                      return parser\_accept(stack);                  }                } else {                   // Should not ACCEPT before reaching end of input ($)                  ErrorReporting\_ReportError\_Base(ERROR\_ON\_PARSER\_PHASE,                                             tokens[0].SourceCodeLine,                                             tokens[0].CharacterInSourceCodeLine); // Error: ACCEPT before $                  printf("Error: ACCEPT action occurred before end of input ($).\n");                  // Attempt recovery? Or treat as fatal? Let's try recovery.                  parser\_recovery\_error(parser, stack, &tokens);                  continue;              }          }        }      // Should not reach here if ACCEPT works correctly      return NULL;  }  int countMagnitudedElementsInTheStack(Parser\_Stack \*stack, int num){      // checking the amount of magnituded elemtns in the first num elenents      int c = 0;      int i;      Stack\_Element \*tempElem = NULL;      // creating new temp stack      Parser\_Stack \*temp = parser\_stack\_createStack();      // making sure num lower then the stack elements      if(stack->size < num){          printf("this is good??? found new problem");          exit(1);      }      for(i = 0; i < num; i++){          tempElem = parser\_stack\_pop(stack);          parser\_stack\_push(temp, tempElem);          // check if not an inportent node and can be not add          if ((tempElem->node)->node\_type == TERMINAL && !should\_add\_to\_abstact\_syntax\_tree((tempElem->node)->Kind.TerminalType)){              continue;          }          // increase the counter          c += 1;      }      // returning the element to the intiall stack      for(i = 0; i < num; i++){          tempElem = parser\_stack\_pop(temp);          parser\_stack\_push(stack, tempElem);      }      parser\_stack\_destroyStack(temp);      return c;  }  void freeAst(TreeNode\* root){        // function for recursivly freeing the ast nodes      if(root == NULL) return;      for(int i = 0 ; i < root->numberOfChildrens; i++){          freeAst(root->childrens[i]);      }        // freeing this node and its dynamic attributes      // freeing the array of node's pointers      free(root->childrens);        // freeing the node value(if not null)      if(root->value != NULL){          free(root->value);      }        // freeing the node itself      free(root);    } |

## Semantic

### Scope.h

|  |
| --- |
| #pragma once  #include"../SymbolTable/SymbolTable.h"  #include"../../Parser/Parser.h"  typedef struct Scope  {      // pointer into the current scope symbol table      SymbolTable\* symbol\_table;      // number of children      int number\_of\_childrens;      // pointer to childrens      struct Scope\*\* childrens;      // pointer to scope's parent      struct Scope\* parent;      // this is a pointer to prototype of the current function      Symbol\_Table\_Function\_Entry\* functionPrototype;      // count the number of declared variables(not matter if char or int)      int variables\_c;  }Scope;  void set\_scope\_function\_prototype(Scope\* scope, Symbol\_Table\_Function\_Entry\* func\_proto);  Symbol\_Table\_Function\_Entry\* get\_scope\_function\_prototype(Scope\* scope);  Function\_Parameter\* create\_Functionn\_Params(TreeNode\* AST, int max, int\* more\_them\_max);  Function\_Parameter\* create\_Functionn\_Call\_Args(TreeNode\* AST);  Scope\* create\_global\_scope();  Scope\* create\_local\_scope();  char\* extract\_variable\_name(TreeNode\* node);  Variable\_And\_Return\_Type extract\_variable\_type(TreeNode\* node);  Variable\_And\_Return\_Type extract\_function\_return\_type(TreeNode\* AST); |

### Scope.c

|  |
| --- |
| #include"Scope.h"  void set\_scope\_function\_prototype(Scope\* scope, Symbol\_Table\_Function\_Entry\* func\_proto){      // this function set the pointer to the record which there is the prototype of the function      // setting      scope->functionPrototype = func\_proto;  }  Symbol\_Table\_Function\_Entry\* get\_scope\_function\_prototype(Scope\* scope){      // return the pointer to the record which there is the prototype of the function      return scope->functionPrototype;  }  Function\_Parameter\* create\_Functionn\_Params(TreeNode\* AST, int max, int\* more\_them\_max){      // define local variables      TreeNode\* tmpTreeNode = AST;      Function\_Parameter\* ParamsHead = NULL;      Function\_Parameter\* ParamsTmp = NULL;      Variable\_And\_Return\_Type tmpType;      int count = 0;      char\* errorMsg;      // firstly, making sure that the ast node is type PARAMETERS      if(AST == NULL || AST->node\_type != NON\_TERMINAL|| AST->numberOfChildrens == 0 || AST->Kind.nonTerminalType != NON\_TERMINAL\_PARAMETERS){          printf("COMPILER ERRORN!\n");          exit(EXIT\_FAILURE);      }      while (tmpTreeNode!= NULL)      {          // increasing the parmas counter          count += 1;            if(tmpTreeNode->childrens[0]->node\_type != NON\_TERMINAL || tmpTreeNode->childrens[0]->Kind.nonTerminalType != NON\_TERMINAL\_PARAMETER || ((tmpTreeNode->childrens[0])->childrens[0])->node\_type != NON\_TERMINAL ||              ((tmpTreeNode->childrens[0])->childrens[0])->Kind.nonTerminalType != NON\_TERMINAL\_TYPE){              printf("COMPILER ERRORO!\n");              exit(EXIT\_FAILURE);          }          // processing the max first parameters          if(count <= max){                // GETTING THE TYPE OF THE CURRENT PARAMETER(here we dont address its name)              tmpType = extract\_variable\_type((tmpTreeNode->childrens[0])->childrens[0]);                // creaitng new paraneter strcut and adding it to the linked list                // adding to the linked list              if(ParamsHead == NULL){                  ParamsHead = create\_parameter(tmpType);                  ParamsTmp = ParamsHead;              }              else{                  ParamsTmp->next\_param = create\_parameter(tmpType);                  ParamsTmp = ParamsTmp->next\_param;              }            }          // checking if we have only one child(meaning we are at the last iteration)          if(tmpTreeNode->numberOfChildrens == 1){                // checking if the counter greater them max              if(count > max){                    // setting the flag that indicate that we overriden the max allowed params                  \*more\_them\_max = count;              }              // returning the head of the linked params list                return ParamsHead;          }          // stepping the tmpASTnode          tmpTreeNode = tmpTreeNode->childrens[1];      }      // shoud never be here      printf("COMPILER ERROR");      exit(EXIT\_FAILURE);  }  Scope\* create\_global\_scope(){      // creating a glocal scope object      Scope\* newScope = (Scope\*)malloc(sizeof(Scope));      SymbolTable\* newSymbolTable;      if(newScope == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }       // setting the diafult attributes for the scope     newScope->parent = NULL;     newScope->childrens = NULL;     newScope->number\_of\_childrens = 0;     newScope->symbol\_table = create\_symbol\_table(GLOBAL\_SYMBOL\_TABLE);     newScope->variables\_c = 0;      return newScope;  }  Scope\* create\_local\_scope(){      Scope\* newScope = (Scope\*)malloc(sizeof(Scope));      if(newScope == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }        // setting the diafult attributes for the scope      newScope->parent = NULL;      newScope->childrens = NULL;      newScope->number\_of\_childrens = 0;      newScope->symbol\_table = create\_symbol\_table(LOCAL\_SYMBOL\_TABLE);      newScope->variables\_c = 0;      return newScope;  }  char\* extract\_variable\_name(TreeNode\* node) {      if (node == NULL || node->node\_type != TERMINAL || node->Kind.TerminalType != IDENTIFIER\_TOKEN || node->value == NULL) {          printf("Invalid variable name!\n");          exit(1);      }      return node->value;  }  Variable\_And\_Return\_Type extract\_variable\_type(TreeNode\* node) {      if (node == NULL || node->numberOfChildrens != 1 || node->node\_type == TERMINAL || node->Kind.nonTerminalType != NON\_TERMINAL\_TYPE) {          printf("Invalid type declaration!\n");          exit(1);      }      TreeNode\* typeNode = node->childrens[0];      if (typeNode == NULL || typeNode->node\_type != TERMINAL) {          printf("Unknown situation!\n");          exit(1);      }      switch (typeNode->Kind.TerminalType) {          case INT\_TOKEN: return VARIABLE\_AND\_RETURN\_TYPE\_INT;          case CHAR\_TOKEN: return VARIABLE\_AND\_RETURN\_TYPE\_CHAR;          case VOID\_TOKEN: return VARIABLE\_AND\_RETURN\_TYPE\_VOID;          default:              printf("Unknown type!\n");              exit(1);      }  }  Variable\_And\_Return\_Type extract\_function\_return\_type(TreeNode\* AST){      // define local variables      TreeNode\* tmp  = NULL;      // the current node shuld be non terminal type, checking it      if(AST == NULL || AST->node\_type != NON\_TERMINAL|| AST->numberOfChildrens != 1 || AST->Kind.nonTerminalType != NON\_TERMINAL\_TYPE){          printf("COMPILER ERRORL!\n");          exit(EXIT\_FAILURE);      }      tmp = AST->childrens[0];      if(tmp->Kind.TerminalType == INT\_TOKEN){          return VARIABLE\_AND\_RETURN\_TYPE\_INT;      }      if(tmp->Kind.TerminalType == CHAR\_TOKEN){          return VARIABLE\_AND\_RETURN\_TYPE\_CHAR;      }      if(tmp->Kind.TerminalType == VOID\_TOKEN){          return VARIABLE\_AND\_RETURN\_TYPE\_VOID;      }      // IF here, there is an compiler error      printf("COMPILER ERRORM!\n");          exit(EXIT\_FAILURE);  } |

### SymbolTable.h

|  |
| --- |
| #pragma once  #include<stdio.h>  #include<stdlib.h>  #include<string.h>  typedef enum{      VARIABLE\_AND\_RETURN\_TYPE\_CHAR,      VARIABLE\_AND\_RETURN\_TYPE\_INT,      VARIABLE\_AND\_RETURN\_TYPE\_VOID  }Variable\_And\_Return\_Type;  typedef enum {      VARIABLE\_PARAM,      VARIABLE\_LOCAL  } Variable\_Origin;  typedef enum{      LOCAL\_SYMBOL\_TABLE,      GLOBAL\_SYMBOL\_TABLE  }Symbol\_Table\_Type;  typedef struct Function\_Parameter{      // the type of the variable expected for the corresponding function      Variable\_And\_Return\_Type variable\_type;      // pointer into the next parameter in the linked list      struct Function\_Parameter\* next\_param;  }Function\_Parameter;  typedef struct Symbol\_Table\_Function\_Entry{      // the name of the function stored in the symbol table      char\* name;      // the return type of the function      Variable\_And\_Return\_Type return\_type;      // the amount of parameters its gets      int number\_of\_parameters;      // pointer into linked list of the parameters in order      Function\_Parameter\* params;      // pointer into the next entry in the symbol table linked list      struct Symbol\_Table\_Function\_Entry\* next\_entry;      // pointer into the previes entry in the symbol table linked list      struct Symbol\_Table\_Function\_Entry\* prev\_entry;      // the index in the gloval scope children where the function scope in      int function\_index;      // the name of the label which we uses for returning from the function      char\* function\_return\_label;      // dynamic counter on how many variables been declared so far      // used while processing code generation      int declared\_vars;  }Symbol\_Table\_Function\_Entry;  typedef struct Symbol\_Table\_Variable\_Entry{      // the name of the variable      char\* name;      // the type of the variable      Variable\_And\_Return\_Type variable\_type;      // the origin of the varible, was declared or was provided as a parameter, used in code generation      Variable\_Origin variable\_origin;      // the possititon relate to bp, used in code generation to acess the memory address of that variable      int index\_bp\_relative;      // the position of precedence of all local/params variables      int variable\_position;      // pointer into the next entry in the symbol table linked list      struct Symbol\_Table\_Variable\_Entry\* next\_entry;  }Symbol\_Table\_Variable\_Entry;  typedef struct SymbolTable  {      // hold two types of symbol types entrires      // one is for global scope only, the second for all non\_global symbol table      // there is no case where poth pointers will not be null      Symbol\_Table\_Function\_Entry\* Function\_Entry;      Symbol\_Table\_Variable\_Entry\* Variable\_Entry;        // hold the number of records in the table      int number\_of\_records;      Symbol\_Table\_Type symbol\_table\_type;  }SymbolTable;  SymbolTable\* create\_symbol\_table(Symbol\_Table\_Type type);  Symbol\_Table\_Function\_Entry\* add\_function\_record\_into\_global\_symbol\_table(SymbolTable\* symbol\_table, Variable\_And\_Return\_Type return\_type, char\* func\_name, Function\_Parameter\* params, int number\_of\_params);  void add\_variable\_record\_into\_local\_symbol\_table(SymbolTable\* symbol\_table, Variable\_And\_Return\_Type var\_type, char\* var\_name, Variable\_Origin var\_origin);  int isVariableExistsInTable(SymbolTable\* table, char\* var\_name);  int isFunctionExistsInTable(SymbolTable\* table, char\* var\_name);  Variable\_And\_Return\_Type get\_Varaible\_Type\_From\_Table(SymbolTable\* table, char\* var\_name);  void print\_symbol\_table(SymbolTable\* table);  Function\_Parameter\* create\_parameter(Variable\_And\_Return\_Type type);  int count\_number\_of\_params(Function\_Parameter\* param);  void freeSymbolTable(SymbolTable\* symbolTable);  void freeSymbolTableFunctionEntries(Symbol\_Table\_Function\_Entry\* func\_entry);  void freeSymbolTableVariableEntires(Symbol\_Table\_Variable\_Entry\* var\_entry); |

### SymbolTable.c

|  |
| --- |
| #include"SymbolTable.h"  Function\_Parameter\* create\_parameter(Variable\_And\_Return\_Type type){      // allocating new memory for the new parameter      Function\_Parameter\* newParam = (Function\_Parameter\*)malloc(sizeof(Function\_Parameter));      if(newParam == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }      // setting the intiall values      newParam->next\_param = NULL;      newParam->variable\_type = type;      // returning the new parameter      return newParam;  }  SymbolTable\* create\_symbol\_table(Symbol\_Table\_Type type){      SymbolTable\* newTable = (SymbolTable\*)malloc(sizeof(SymbolTable));      if(newTable == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }      // setting the intial attrubutes values      newTable->symbol\_table\_type = type;      newTable->number\_of\_records = 0;      newTable->Variable\_Entry = NULL;      newTable->Function\_Entry = NULL;      return newTable;  }  Symbol\_Table\_Function\_Entry\* add\_function\_record\_into\_global\_symbol\_table(SymbolTable\* symbol\_table, Variable\_And\_Return\_Type return\_type, char\* func\_name, Function\_Parameter\* params, int number\_of\_params){      // define local variables      Symbol\_Table\_Function\_Entry\* tmp = NULL;      // making sure that the symbol table is the global one      if(symbol\_table->symbol\_table\_type != GLOBAL\_SYMBOL\_TABLE){          printf("COMPILER ERROR!\_130\n");          exit(EXIT\_FAILURE);      }      // allocating  new memory for the new symbol table entry      Symbol\_Table\_Function\_Entry\* NewEntry = (Symbol\_Table\_Function\_Entry\*)malloc(sizeof(Symbol\_Table\_Function\_Entry));      if(NewEntry == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }      NewEntry->name = strdup(func\_name);      NewEntry->return\_type = return\_type;      NewEntry->next\_entry = NULL;      NewEntry->params = params;      NewEntry->number\_of\_parameters = number\_of\_params;      NewEntry->function\_return\_label = NULL;      NewEntry->declared\_vars = 0;        // increasing the counter of records in the gloval scope      (symbol\_table->number\_of\_records)++;      // setting the function index      NewEntry->function\_index = symbol\_table->number\_of\_records - 1;      // handling first entry cace      if(symbol\_table->Function\_Entry == NULL){          // setting the new function entry as the head of the linked list          symbol\_table->Function\_Entry = NewEntry;            // setting the privies node to be null          NewEntry->prev\_entry = NULL;          return NewEntry;      }      // going to the last record in the linked list      tmp = symbol\_table->Function\_Entry;      while (tmp->next\_entry != NULL) tmp = tmp->next\_entry;        // add the new record at the end of the list      tmp->next\_entry = NewEntry;      // setting the privies of the new entry to be tmp      NewEntry->prev\_entry = tmp;        // RETURNING THE NEW ENTRY      return NewEntry;  }  void add\_variable\_record\_into\_local\_symbol\_table(SymbolTable\* symbol\_table, Variable\_And\_Return\_Type var\_type, char\* var\_name, Variable\_Origin var\_origin){      Symbol\_Table\_Variable\_Entry\* tmp;      int prev\_vars\_c = 0;      //printf("adding the variable: %s into the symbol table\n", var\_name);      //printf("\n");      //print\_symbol\_table(symbol\_table);      //printf("\n");      // Creating new record      Symbol\_Table\_Variable\_Entry\* NewEntry = (Symbol\_Table\_Variable\_Entry\*)malloc(sizeof(Symbol\_Table\_Variable\_Entry));      if(NewEntry == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }      NewEntry->name = \_strdup(var\_name);  //Copy name      NewEntry->variable\_type = var\_type;      NewEntry->variable\_origin = var\_origin;      NewEntry->next\_entry = NULL;        // Handle first entry case      if (symbol\_table->Variable\_Entry == NULL) {          symbol\_table->Variable\_Entry = NewEntry;          }      else{              // Traverse to last node and count the amount of variables previesly declared          // at this symbol table in the same way(local var/ param var)          tmp = symbol\_table->Variable\_Entry;          while (tmp->next\_entry != NULL){              // checking if the origin match the new var origin              if(tmp->variable\_origin == var\_origin){                  // increase the counter                  prev\_vars\_c++;              }              // stepping the iterator              tmp = tmp->next\_entry;          }          // checking last time          if(tmp != NULL && tmp->variable\_origin == var\_origin){              prev\_vars\_c++;          }          // Adding the new entry at end          tmp->next\_entry = NewEntry;      }      // setting the position of the new entry      NewEntry->variable\_position = prev\_vars\_c;      //print\_symbol\_table(symbol\_table);      //fflush(stdout);  }  int isVariableExistsInTable(SymbolTable\* table, char\* var\_name){      if(table == NULL){          printf("COMPILER ERROR!\_140\n");          exit(EXIT\_FAILURE);      }      //print\_symbol\_table(table);      // checking if the name of the variable exists in this symbol table      Symbol\_Table\_Variable\_Entry\* tmp = table->Variable\_Entry;      // looping for every entry in the current scope      while(tmp != NULL){            // checking if the current record match the name          if(strcmp(var\_name, tmp->name) == 0){              return 1;          }            // stepping the tmp to the next node in the linked list          tmp = tmp->next\_entry;      }      // returning false      return 0;  }  int isFunctionExistsInTable(SymbolTable\* table, char\* var\_name){      // this function checks if there is function entry in the symbol table      if(table == NULL){          printf("COMPILER ERROR!\_150\n");          exit(EXIT\_FAILURE);      }      // checking if the name of the variable exists in this symbol table      // getting the first entry      Symbol\_Table\_Function\_Entry\* tmp = table->Function\_Entry;      // looping for every entry in the current scope      while(tmp != NULL){            // checking if the current record match the name          if(strcmp(var\_name, tmp->name) == 0){              return 1;          }            // stepping the tmp to the next node in the linked list          tmp = tmp->next\_entry;      }      // returning false      return 0;  }  Variable\_And\_Return\_Type get\_Varaible\_Type\_From\_Table(SymbolTable\* table, char\* var\_name){      // this function assume that the variable do exists in the symbol table      Symbol\_Table\_Variable\_Entry\* tmp = table->Variable\_Entry;      while(tmp != NULL){            // checking if the current record match the name          if(strcmp(var\_name, tmp->name) == 0){                return tmp->variable\_type;          }          // stepping the tmp to the next node in the linked list          tmp = tmp->next\_entry;      }      // if in here, name isnt exists, return null      return -1;      //printf("COMPILER ERROR!\_160\n");      //exit(EXIT\_FAILURE);  }  void print\_symbol\_table(SymbolTable\* table) {      if (table == NULL) {          printf("Symbol Table is NULL\n");          return;      }      printf("=== Symbol Table (%s) ===\n",             table->symbol\_table\_type == GLOBAL\_SYMBOL\_TABLE ? "GLOBAL" : "LOCAL");        if (table->symbol\_table\_type == GLOBAL\_SYMBOL\_TABLE) {          // Global symbol table holds only functions          Symbol\_Table\_Function\_Entry\* func\_entry = table->Function\_Entry;          while (func\_entry != NULL) {              printf("Function: %s, Return Type: %d, Parameters: %d\n",                     func\_entry->name, func\_entry->return\_type, func\_entry->number\_of\_parameters);                Function\_Parameter\* param = func\_entry->params;              printf("  Parameters: ");              while (param != NULL) {                  printf("%d ", param->variable\_type);                  param = param->next\_param;              }              printf("\n");              func\_entry = func\_entry->next\_entry;          }      } else {          // Local symbol table holds only variables          Symbol\_Table\_Variable\_Entry\* var\_entry = table->Variable\_Entry;          while (var\_entry != NULL) {              printf("Variable: %s, Type: %d, position: %d\n", var\_entry->name, var\_entry->variable\_type, var\_entry->variable\_position);              var\_entry = var\_entry->next\_entry;          }      }    }    void freeSymbolTable(SymbolTable\* symbolTable) {      if (symbolTable == NULL) return;        freeSymbolTableFunctionEntries(symbolTable->Function\_Entry);        freeSymbolTableVariableEntires(symbolTable->Variable\_Entry);        free(symbolTable);    }  void freeSymbolTableFunctionEntries(Symbol\_Table\_Function\_Entry\* func\_entry) {      if (func\_entry == NULL) return;      freeSymbolTableFunctionEntries(func\_entry->next\_entry);        free(func\_entry->name);        free(func\_entry);    }  void freeSymbolTableVariableEntires(Symbol\_Table\_Variable\_Entry\* var\_entry) {      if (var\_entry == NULL) return;        freeSymbolTableVariableEntires(var\_entry->next\_entry);      free(var\_entry->name);      free(var\_entry);  } |

### ScopeTree.h

|  |
| --- |
| #include"../SymbolTable/SymbolTable.h"  #include"../../Parser/Parser.h"  #include"Scope.h"  #include"../../ErrorReporting/ErrorReporting.h"  // Function declarations  void process\_program(TreeNode\* AST, Scope\*\* scope);  void process\_functions(TreeNode\* AST, Scope\* scope);  void process\_function\_declaration(TreeNode\* AST, Scope\* global\_scope);  void process\_body(TreeNode\* AST, Scope\* parentScope);  void process\_statement(TreeNode\* AST, Scope\* scope);  void process\_declaration(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_assignment(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_expression(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_logical\_or(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_logical\_and(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_relational\_term(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_additive\_term(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_multiplicative\_term(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_factor(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_literal(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_variable\_identifier(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type process\_function\_call(TreeNode\* AST, Scope\* currentScope);  void check\_Function\_Call\_Arguments(TreeNode\* AST, Scope\* scope, Function\_Parameter\* cur\_param, char\*);  void process\_if(TreeNode\* AST, Scope\* scope);  void process\_while(TreeNode\* AST, Scope\* scope);  void process\_else\_if\_or\_else(TreeNode\* AST, Scope\* scope);  void process\_putchar(TreeNode\* AST, Scope\* scope);  void process\_getchar(TreeNode\* AST, Scope\* scope);  void process\_return(TreeNode\* AST, Scope\* currentScope);  Variable\_And\_Return\_Type variable\_declaration\_lookup(Scope\* scope, char\* var\_name);  void process\_statements(TreeNode\* AST, Scope\* scope);  void freeScopeTree(Scope\* scope);  char\* extract\_variable\_name\_lookdown(TreeNode\* AST); |

### ScopeTree.c

|  |
| --- |
| #include"ScopeTree.h"  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  // define global variable that tell if we are in an panic mode  int panicMode = 0;  char\* Semantic\_PrepareErrorMessage(int code, char\* str, int index, char\* str2) {      char buffer[256];      switch (code) {          case 1:              snprintf(buffer, sizeof(buffer), "variable '%s' cannot be declared with type 'void'", str);              break;          case 2:              snprintf(buffer, sizeof(buffer), "redeclaration of variable name '%s' in the same scope", str);              break;          case 3:              snprintf(buffer, sizeof(buffer), "function parameter '%s' cannot have type 'void'", str);              break;          case 4:              snprintf(buffer, sizeof(buffer), "call to undefined function '%s'", str);              break;          case 5:              snprintf(buffer, sizeof(buffer), "calling function '%s' without required arguments", str);              break;          case 6:              snprintf(buffer, sizeof(buffer), "too many arguments in call to function '%s'", str);              break;          case 7:              snprintf(buffer, sizeof(buffer), "too few arguments in call to function '%s'", str);              break;          case 8:              snprintf(buffer, sizeof(buffer), "invalid argument type at index %d in function call '%s'", index, str);              break;          case 9:              snprintf(buffer, sizeof(buffer), "trying to use an undeclared variable '%s'", str);              break;          case 10:              snprintf(buffer, sizeof(buffer), "invalid return type according to function declaration");              break;          case 11:              snprintf(buffer, sizeof(buffer), "function '%s' already exists", str);              break;          case 12:              snprintf(buffer, sizeof(buffer), "duplicate declration of parameter name: '%s' in index: %d in function call '%s'", str, index, str2);              break;          case 13:              snprintf(buffer, sizeof(buffer), "no main function detected '%s'", str);              break;          case 14:              snprintf(buffer, sizeof(buffer), "trying to use function: '%s' inside expression, but function return 'void'", str);              break;          case 15:              snprintf(buffer, sizeof(buffer), "function '%s' has %d parameters, but the maximun allowed is 4", str, index);              break;          case 16:              snprintf(buffer, sizeof(buffer), "can't assign variable '%s' with expression evaluate type void", str);              break;          default:              snprintf(buffer, sizeof(buffer), "unknown semantic error");      }      return \_strdup(buffer); // returning the new buffer content in dynamic memory  }  void Semantic\_LookForMainFunction(SymbolTable\* symbolTable){      // this function checks for the presidence of the function with the saed name 'main'.      // define varialbes      Symbol\_Table\_Function\_Entry\* entry;      int mainFound = 0; //is main found?      char\* errorMsg;        // setting the entry to the first entry in the symbol table      entry = symbolTable->Function\_Entry;      // looping untill the current entry is null      while (entry != NULL)      {            // checking for the name 'main'          if(strcmp(entry->name, "main") == 0){              // setting the flag to found              mainFound = 1;          }            // stepping the node into the next entry          entry = entry->next\_entry;      }        // checking if the flag is off      if(!mainFound){            // print the base error message          ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, 0,0);          // gett the error details          errorMsg = Semantic\_PrepareErrorMessage(13, "main", -1, NULL);          // outputting the error message          ErrorReporting\_ReportError(errorMsg);          // free the message          free(errorMsg);          // setting panic mode to 1          panicMode = 1;      }    }  void process\_program(TreeNode\* AST, Scope\*\* scope) {      // creating the global scope      \*scope = create\_global\_scope();      // calling process functions      process\_functions(AST->childrens[0], \*scope);        // looking for main      Semantic\_LookForMainFunction((\*scope)->symbol\_table);        // checking if we went into panic mode      if(panicMode == 1){          // freeing the scope tree            freeScopeTree(\*scope);            \*scope = NULL;      }  }  void process\_functions(TreeNode\* AST, Scope\*scope){      // the current node shuld be non terminal type, checking it      if(AST == NULL || AST->node\_type != NON\_TERMINAL|| AST->numberOfChildrens == 0 || AST->Kind.nonTerminalType != NON\_TERMINAL\_FUNCTIONS){          printf("COMPILER ERRORL\_12!\n");          exit(EXIT\_FAILURE);      }      if(AST->numberOfChildrens == 1){          process\_function\_declaration(AST->childrens[0], scope);      }      if(AST->numberOfChildrens == 2){          process\_function\_declaration(AST->childrens[0], scope);          process\_functions(AST->childrens[1], scope);      }  }  void process\_body(TreeNode\* AST, Scope\* scope) {      // checking that the current ast node is non terminal body      if(AST == NULL || AST->numberOfChildrens != 1 || AST->node\_type != NON\_TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_BODY){          printf("COMPILER ERRORL\_14!\n");          exit(EXIT\_FAILURE);      }      if(scope == NULL){          printf("SEMNATIC GOT NULL SCOPE ERROR");          exit(EXIT\_FAILURE);      }        // saving the scope of the body      AST->scope = (void\*)scope;      // calling process statemets      process\_statements(AST->childrens[0], scope);    }  void process\_statements(TreeNode\* AST, Scope\* scope){      // firstly, checking that the current node is non terminal statements      if(AST == NULL || AST->numberOfChildrens == 0 || AST->node\_type != NON\_TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_STATEMENTS){          printf("COMPILER ERRORL\_13!\n");          exit(EXIT\_FAILURE);      }      // cheking the number of childrens      if(AST->numberOfChildrens == 1){          // calling process statement          return process\_statement(AST->childrens[0], scope);      }      if(AST->numberOfChildrens == 2){          // calling process statement          process\_statement(AST->childrens[0], scope);          // calling process statements          process\_statements(AST->childrens[1], scope);      }  }  void process\_statement(TreeNode\* AST, Scope\* scope){      // processing statement      if(AST == NULL || AST->numberOfChildrens != 1 || AST->node\_type != NON\_TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_STATEMENT){          printf("COMPILER ERRORL\_1!\n");          exit(EXIT\_FAILURE);      }      if(AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_ASSIGNMENT){          process\_assignment(AST->childrens[0], scope);          return;      }      if(AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_DECLARATION){          return process\_declaration(AST->childrens[0], scope);      }      if(AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_RETURN){          return process\_return(AST->childrens[0], scope);        }      if(AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_FUNCTION\_CALL){          process\_function\_call(AST->childrens[0], scope);          return;      }      if(AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_IF\_STATEMENT){          return process\_if(AST->childrens[0], scope);      }      if(AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_WHILE\_LOOP){          return process\_while(AST->childrens[0], scope);      }      if(AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_EXPRESSION){          process\_expression(AST->childrens[0], scope);          return;      }      if(AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_PUTCHAR){          return process\_putchar(AST->childrens[0], scope);      }      if(AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_GETCHAR){          return process\_getchar(AST->childrens[0], scope);      }  }  void process\_declaration(TreeNode\* AST, Scope\* currentScope) {      // declare local variable      char\* errorMsg;        if (AST->numberOfChildrens != 2) {          printf("Unexpected declaration structure!\n");          exit(1);      }      char\* varName = extract\_variable\_name(AST->childrens[1]);      Variable\_And\_Return\_Type varType = extract\_variable\_type(AST->childrens[0]);      // checking if the code attempted to declare a variable with type void      if(varType == VARIABLE\_AND\_RETURN\_TYPE\_VOID){          // in this position, code attempted to declare a variable with type void          // print the base error message          ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);          // gett the error details          errorMsg = Semantic\_PrepareErrorMessage(1, varName, -1, NULL);          // outputting the error message          ErrorReporting\_ReportError(errorMsg);          // free the message          free(errorMsg);          // setting panic mode to 1          panicMode = 1;          //return;          // setting the type to int for avoiding more errors relating to the void problem, since we have already reported that          varType = VARIABLE\_AND\_RETURN\_TYPE\_INT;      }      // check if variable already exists in the scope's symbol table      if(isVariableExistsInTable(currentScope->symbol\_table, varName)){          // deplicated declaration of variable in the same scope error          ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[1]->SourceCodeLine, AST->childrens[1]->CharacterInSourceCodeLine);            // gett the error details          errorMsg = Semantic\_PrepareErrorMessage(2, varName, -1, NULL);          // outputting the error message          ErrorReporting\_ReportError(errorMsg);          // free the message          free(errorMsg);            // setting panic mode to 1          panicMode = 1;          return;      }      // adding the new variable into the scope's symbol table      add\_variable\_record\_into\_local\_symbol\_table(currentScope->symbol\_table, varType, varName, VARIABLE\_LOCAL);      // increasing the counter in the current scoope      currentScope->variables\_c++;  }  void process\_if(TreeNode\* AST, Scope\* scope){      // define local variables      Scope\* inner\_scope;      // making sure that we are on an if node      if(AST == NULL || AST->numberOfChildrens == 0 || AST->node\_type != NON\_TERMINAL|| AST->Kind.nonTerminalType != NON\_TERMINAL\_IF\_STATEMENT ){          printf("COMPILER ERRORL");          exit(EXIT\_FAILURE);      }      // creating new scope for the inside of the if      inner\_scope = create\_local\_scope();      // setting scope's parent      inner\_scope->parent = scope;      // setting the pointer to the current function prototype      set\_scope\_function\_prototype(inner\_scope, get\_scope\_function\_prototype(scope));      // increasing the size of the parent childrens array      scope->childrens = (Scope\*\*)realloc(scope->childrens, sizeof(Scope) \* ++scope->number\_of\_childrens);      if(scope->childrens == NULL){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      // setting the son to be in the children array of the parent      scope->childrens[scope->number\_of\_childrens - 1] = inner\_scope;      if(AST->numberOfChildrens == 2){          // check valid expression          process\_expression(AST->childrens[0], scope);          // handling new if body          process\_body(AST->childrens[1], inner\_scope);          return;      }      if(AST->numberOfChildrens == 3){          // check valid expression          process\_expression(AST->childrens[0], scope);          // handling new if body          process\_body(AST->childrens[1], inner\_scope);          // handling else\_if\_of else          process\_else\_if\_or\_else(AST->childrens[2], scope); // not exists      }  }  void process\_while(TreeNode\* AST, Scope\* scope){      // define variables      Scope\* inner\_scope;      //printf("num: %d\n", AST->numberOfChildrens);      // checking we are in non terminal node type while      if(AST == NULL || AST->numberOfChildrens != 2 || AST->node\_type != NON\_TERMINAL|| AST->Kind.nonTerminalType != NON\_TERMINAL\_WHILE\_LOOP){          printf("COMPILER ERRORL");          exit(EXIT\_FAILURE);      }      // checking valid expression      process\_expression(AST->childrens[0], scope);      // creating new scope for the inside of the if      inner\_scope = create\_local\_scope();      // setting scope's parent      inner\_scope->parent = scope;      // setting the pointer to the current function prototype      set\_scope\_function\_prototype(inner\_scope, get\_scope\_function\_prototype(scope));      // increasing the size of the parent childrens array      scope->childrens = (Scope\*\*)realloc(scope->childrens, sizeof(Scope) \* ++scope->number\_of\_childrens);      if(scope->childrens == NULL){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      // setting the son to be in the children array of the parent      scope->childrens[scope->number\_of\_childrens - 1] = inner\_scope;      // handling creating new body      process\_body(AST->childrens[1], inner\_scope);  }  void process\_else\_if\_or\_else(TreeNode\* AST, Scope\* scope){      // define varaibles      Scope\* inner\_scope;      // making sure we are in else\_if\_or\_else non terminal      if(AST == NULL || AST->numberOfChildrens != 2 || AST->node\_type != NON\_TERMINAL|| AST->Kind.nonTerminalType != NON\_TERMINAL\_ELSE\_IF\_OR\_ELSE){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      if(AST->childrens[0]->node\_type != TERMINAL || AST->childrens[0]->Kind.TerminalType != ELSE\_TOKEN || AST->childrens[1]->node\_type != NON\_TERMINAL){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      if(AST->childrens[1]->Kind.nonTerminalType == NON\_TERMINAL\_IF\_STATEMENT){          // process if in else          return process\_if(AST->childrens[1], scope);      }      if(AST->childrens[1]->Kind.nonTerminalType == NON\_TERMINAL\_BODY){          // creating new scope for the inside of the if          inner\_scope = create\_local\_scope();          // setting scope's parent          inner\_scope->parent = scope;          // setting the pointer to the current function prototype          set\_scope\_function\_prototype(inner\_scope, get\_scope\_function\_prototype(scope));          // increasing the size of the parent childrens array          scope->childrens = (Scope\*\*)realloc(scope->childrens, sizeof(Scope) \* ++scope->number\_of\_childrens);          if(scope->childrens == NULL){              printf("COMPILER ERROR");              exit(EXIT\_FAILURE);          }          // setting the son to be in the children array of the parent          scope->childrens[scope->number\_of\_childrens - 1] = inner\_scope;          // processing that body          process\_body(AST->childrens[1], inner\_scope);      }  }  void process\_getchar(TreeNode\* AST, Scope\* scope){      // define local variables      Variable\_And\_Return\_Type VarType;      // firstly, making sure we are on an getchar non terminal      if(AST == NULL || AST->numberOfChildrens != 1 || AST->node\_type != NON\_TERMINAL|| AST->Kind.nonTerminalType != NON\_TERMINAL\_GETCHAR      || AST->childrens[0]->node\_type != TERMINAL || AST->childrens[0]->Kind.nonTerminalType != IDENTIFIER\_TOKEN){          printf("COMPILER ERRORL\_9!\n");          exit(EXIT\_FAILURE);      }      // checking the type of the identifier      VarType = process\_variable\_identifier(AST->childrens[0], scope);       return;  }  void process\_putchar(TreeNode\* AST, Scope\* scope){      // define local variables      Variable\_And\_Return\_Type expressionType;      // firstly, making sure we are on an putchar non terminal      if(AST == NULL || AST->numberOfChildrens != 1 || AST->node\_type != NON\_TERMINAL|| AST->Kind.nonTerminalType != NON\_TERMINAL\_PUTCHAR){          printf("COMPILER ERRORL\_10!\n");          exit(EXIT\_FAILURE);      }      // checking valid expression      expressionType = process\_expression(AST->childrens[0], scope);        return;  }  void process\_function\_declaration(TreeNode\* AST, Scope\* global\_scope){      // the function responsible for creating the new function prototype in the global scope      // and also create new scope for the function with the params      // define local variables      Function\_Parameter\* params\_head = NULL;      char\* func\_name = NULL;      Variable\_And\_Return\_Type return\_type;      Symbol\_Table\_Function\_Entry\* newEntry;      Scope\* newScope;      TreeNode\* tmpTreeNodeParams;      TreeNode\* tmpParam;      char\* tmp\_param\_name;      Variable\_And\_Return\_Type tmp\_param\_type;      int c = 0;      int paramIndex = 0;      char\* errorMsg;      //int errorFlag = 0;      // make sure we are in an function node      if(AST == NULL|| AST->node\_type != NON\_TERMINAL || (AST->numberOfChildrens != 4 && AST->numberOfChildrens != 3) || AST->Kind.nonTerminalType != NON\_TERMINAL\_FUNCTION){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      // get the function name      if(AST->childrens[0]->node\_type != NON\_TERMINAL || AST->childrens[0]->Kind.nonTerminalType != NON\_TERMINAL\_TYPE || AST->childrens[0]->numberOfChildrens != 1 || AST->childrens[0]->childrens[0]->node\_type != TERMINAL){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }        // getting the function name      func\_name = extract\_variable\_name(AST->childrens[1]);      // check if the function name already exists      if(isFunctionExistsInTable(global\_scope->symbol\_table, func\_name)){          // in here, there is a duplicate of function declaration          ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, tmpParam->childrens[0]->SourceCodeLine, tmpParam->childrens[0]->CharacterInSourceCodeLine);            // gett the error details          errorMsg = Semantic\_PrepareErrorMessage(11, func\_name, -1, NULL);          // outputting the error message          ErrorReporting\_ReportError(errorMsg);          // free the message          free(errorMsg);            // setting panic mode to 1          panicMode = 1;          return;      }        // getting the function return type      return\_type = extract\_function\_return\_type(AST->childrens[0]);        // creating new local scope for that function      newScope = create\_local\_scope();      // setting the new scope parent      newScope->parent = global\_scope;        // check if there is parameters for this function      if(AST->numberOfChildrens == 4){            //create the params list          params\_head = create\_Functionn\_Params(AST->childrens[2], 4, &c);            // checking if overriden the maxmum parameters          if(c > 4){              // report an error                ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->SourceCodeLine, AST->CharacterInSourceCodeLine); //15                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(15, func\_name, c, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;              // setting the number of parameter to 4 and continue in panic mode              c = 4;          }            //// counting the amount of the parameters in the list, if bigger them 4, it il be 4 in panic mode          /////c = count\_number\_of\_params(params\_head);          // now add the parameters which we have got to add to the function scope as paraneters          tmpTreeNodeParams = AST->childrens[2];            while (tmpTreeNodeParams != NULL)          {                tmpParam = tmpTreeNodeParams->childrens[0];                if(tmpParam == NULL || tmpParam->numberOfChildrens != 2 ||              tmpParam->node\_type != NON\_TERMINAL || tmpParam->Kind.nonTerminalType != NON\_TERMINAL\_PARAMETER){                  printf("COMPILER ERROR");                  exit(EXIT\_FAILURE);              }                  tmp\_param\_name = extract\_variable\_name(tmpParam->childrens[1]);              tmp\_param\_type = extract\_variable\_type(tmpParam->childrens[0]);              // handling the case where there is parameter with type void              if(tmp\_param\_type == VARIABLE\_AND\_RETURN\_TYPE\_VOID){                  // in this position, source code contains an parameter variable with type void                  ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, tmpParam->childrens[0]->SourceCodeLine, tmpParam->childrens[0]->CharacterInSourceCodeLine); //15                    // gett the error details                  errorMsg = Semantic\_PrepareErrorMessage(3, tmp\_param\_name, -1, NULL);                  // outputting the error message                  ErrorReporting\_ReportError(errorMsg);                  // free the message                  free(errorMsg);                  // setting panic mode to 1                  panicMode = 1;                    //return;              }              // check the case if the parameter name already exists(in case where multiple parameters with the same name)              if(isVariableExistsInTable(newScope->symbol\_table, tmp\_param\_name)){                  // in this position, source code has two(or possibly more) parameters with the same name                  ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, tmpParam->childrens[1]->SourceCodeLine, tmpParam->childrens[1]->CharacterInSourceCodeLine); //15                    // gett the error details                  errorMsg = Semantic\_PrepareErrorMessage(12, tmp\_param\_name, paramIndex, func\_name);                  // outputting the error message                  ErrorReporting\_ReportError(errorMsg);                  // free the message                  free(errorMsg);                  // setting panic mode to 1                  panicMode = 1;                    //return;              }              // adding a new record to the symbol tree              add\_variable\_record\_into\_local\_symbol\_table(newScope->symbol\_table, tmp\_param\_type, tmp\_param\_name, VARIABLE\_PARAM);              //printf("()");              if(tmpTreeNodeParams->numberOfChildrens == 1 || tmpTreeNodeParams->childrens == NULL){                  // if reached the end of the tree of params                  break;              }              // else, continue              tmpTreeNodeParams = tmpTreeNodeParams->childrens[1];              // indecreasing the counter of params              paramIndex++;          }      }        // adding the new function record to the global scope      newEntry = add\_function\_record\_into\_global\_symbol\_table(global\_scope->symbol\_table, return\_type, func\_name, params\_head, c);        // setting this entry to be the pointer to the function prototype in the newScope scope      set\_scope\_function\_prototype(newScope, newEntry);      // setting the new scope as the children of the global      // increasing the size of pointer to the children which is on the global scope      //global\_scope->childrens = (Scope\*\*)realloc(global\_scope->childrens, sizeof(Scope\*) \*global\_scope->symbol\_table->number\_of\_records);      global\_scope->childrens = (Scope\*\*)realloc(global\_scope->childrens, sizeof(Scope\*) \*(++global\_scope->number\_of\_childrens));      if(global\_scope->childrens == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }      if(global\_scope->number\_of\_childrens - 1 != newEntry->function\_index){          printf("need to think");          exit(0);      }        // adding the children in the right place      global\_scope->childrens[newEntry->function\_index] = newScope;        // calling process body for the functions body      if(AST->numberOfChildrens == 4){          process\_body(AST->childrens[3], newScope);      }      else if(AST->numberOfChildrens == 3){          process\_body(AST->childrens[2], newScope);      }  }  Variable\_And\_Return\_Type process\_expression(TreeNode\* AST, Scope\* currentScope){      // checking if current node isnt an expression, yield an error      if(AST == NULL || AST->numberOfChildrens != 1|| AST->node\_type != NON\_TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_EXPRESSION){          printf("COMPILER ERROR1!\n");          exit(EXIT\_FAILURE);      }        // calling handle logical or function      return process\_logical\_or(AST->childrens[0], currentScope);  }  Variable\_And\_Return\_Type process\_logical\_or(TreeNode\* AST, Scope\* currentScope){      // define local variables      Variable\_And\_Return\_Type left;      Variable\_And\_Return\_Type right;      char\* errorMsg;      // checking if current node isnt an logical or      if(AST == NULL || AST->numberOfChildrens == 0 || AST->node\_type == TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_LOGICAL\_OR\_TERM){          printf("COMPILER ERROR2!\n");          exit(EXIT\_FAILURE);      }      // checking if this node has only one child, if yes, this node can be reduce to just logical and      if(AST->numberOfChildrens == 1){          process\_logical\_and(AST->childrens[0], currentScope);      }      else if(AST->numberOfChildrens == 3){          if(AST->childrens[1]->node\_type != TERMINAL || AST->childrens[1]->Kind.TerminalType != LOGICAL\_OR\_TOKEN){              printf("COMPILER ERROR3!\n");              exit(EXIT\_FAILURE);          }          // checking both sides has the same type          left = process\_logical\_or(AST->childrens[0], currentScope);          right = process\_logical\_and(AST->childrens[2], currentScope);            // checking to see that none are type void(cant have void inside expression)          if(left == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }          if(right == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }          return VARIABLE\_AND\_RETURN\_TYPE\_INT;      }  }  Variable\_And\_Return\_Type process\_logical\_and(TreeNode\* AST, Scope\* currentScope){      // define local variables      Variable\_And\_Return\_Type left;      Variable\_And\_Return\_Type right;      char\* errorMsg;      if(AST == NULL || AST->numberOfChildrens == 0 || AST->node\_type != NON\_TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_LOGICAL\_AND\_TERM){          printf("COMPILER ERROR4!\n");          exit(EXIT\_FAILURE);      }      if(AST->numberOfChildrens == 1){          return process\_relational\_term(AST->childrens[0], currentScope);      }      // checking both sides to see if they have a matching types      if(AST->numberOfChildrens == 3){          if(AST->childrens[1]->node\_type != TERMINAL || AST->childrens[1]->Kind.TerminalType != LOGICAL\_AND\_TOKEN){              printf("COMPILER ERROR5!\n");              exit(EXIT\_FAILURE);          }          left = process\_logical\_and(AST->childrens[0], currentScope);          right = process\_relational\_term(AST->childrens[2], currentScope);              // checking to see that none are type void(cant have void inside expression)          if(left == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }          if(right == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }          return VARIABLE\_AND\_RETURN\_TYPE\_INT;      }  }  Variable\_And\_Return\_Type process\_relational\_term(TreeNode\* AST, Scope\* currentScope){      // define local variables      Variable\_And\_Return\_Type left;      Variable\_And\_Return\_Type right;      char\* errorMsg;      if(AST ==NULL || AST->numberOfChildrens == 0 || AST->node\_type == TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_RELATIONAL\_TERM){          printf("COMPILER ERROR6!\n");          exit(EXIT\_FAILURE);      }      if(AST->numberOfChildrens == 1){          return process\_additive\_term(AST->childrens[0], currentScope);      }      if(AST->numberOfChildrens == 3){          if(AST->childrens[1]->node\_type != TERMINAL || (AST->childrens[1]->Kind.TerminalType != EQUAL\_TO\_TOKEN && AST->childrens[1]->Kind.TerminalType != NOT\_EQUAL\_TO\_TOKEN              && AST->childrens[1]->Kind.TerminalType != LESS\_THEN\_TOKEN && AST->childrens[1]->Kind.TerminalType != GREATER\_THEN\_TOKEN && AST->childrens[1]->Kind.TerminalType != LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN              && AST->childrens[1]->Kind.TerminalType != GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN)){                printf("COMPILER ERROR7!\n");              exit(EXIT\_FAILURE);          }            // checking ig both side of the retional term are the same type          left = process\_additive\_term(AST->childrens[0], currentScope);          right = process\_additive\_term(AST->childrens[2], currentScope);            // checking to see that none are type void(cant have void inside expression)          if(left == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }          if(right == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }            return VARIABLE\_AND\_RETURN\_TYPE\_INT;      }  }  Variable\_And\_Return\_Type process\_additive\_term(TreeNode\* AST, Scope\* currentScope){      // define local variables      Variable\_And\_Return\_Type left;      Variable\_And\_Return\_Type right;      char\* errorMsg;      if(AST ==NULL || AST->numberOfChildrens == 0 || AST->node\_type == TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_ADDITIVE\_TERM){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      if(AST->numberOfChildrens == 1){          return process\_multiplicative\_term(AST->childrens[0], currentScope);      }      if(AST->numberOfChildrens == 3){          if(AST->childrens[1]->node\_type != TERMINAL || (AST->childrens[1]->Kind.TerminalType != PLUS\_TOKEN && AST->childrens[1]->Kind.TerminalType != MINUS\_TOKEN)){              printf("COMPILER ERROR");              exit(EXIT\_FAILURE);          }          // checking if both sides of the additiive expresion are type matching          left = process\_multiplicative\_term(AST->childrens[0], currentScope);          right = process\_additive\_term(AST->childrens[2], currentScope);            // checking to see that none are type void(cant have void inside expression)          if(left == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }          if(right == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }          return VARIABLE\_AND\_RETURN\_TYPE\_INT;      }  }  Variable\_And\_Return\_Type process\_multiplicative\_term(TreeNode\* AST, Scope\* currentScope){      // define local variables      Variable\_And\_Return\_Type left;      Variable\_And\_Return\_Type right;      char\* errorMsg;      if(AST ==NULL || AST->numberOfChildrens == 0 || AST->node\_type == TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_MULTIPLICATIVE\_TERM){          printf("COMPILER ERROR10!\n");          exit(EXIT\_FAILURE);      }        if(AST->numberOfChildrens == 1){          return process\_factor(AST->childrens[0], currentScope);      }        if(AST->numberOfChildrens == 3){          if(AST->childrens[1]->node\_type != TERMINAL || (AST->childrens[1]->Kind.TerminalType != MULTIPLY\_TOKEN && AST->childrens[1]->Kind.TerminalType != DIVIDE\_TOKEN          && AST->childrens[1]->Kind.TerminalType != MODULO\_TOKEN)){              printf("COMPILER ERROR");              exit(EXIT\_FAILURE);          }            // checking if both sides of the additiive expresion are type matching          left = process\_factor(AST->childrens[0], currentScope);            right = process\_multiplicative\_term(AST->childrens[2], currentScope);            // checking to see that none are type void(cant have void inside expression)          if(left == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }          if(right == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // ALLERT ERROR              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(14, extract\_variable\_name\_lookdown(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;          }          return VARIABLE\_AND\_RETURN\_TYPE\_INT;      }    }  Variable\_And\_Return\_Type process\_factor(TreeNode\* AST, Scope\* currentScope){      // define local variables      Variable\_And\_Return\_Type retType;      char\* errorMsg; // IN CASE OF AN ERROR        if(AST == NULL || AST->node\_type == TERMINAL || AST->numberOfChildrens == 0){          printf("COMPILER ERROR12!\n");          exit(EXIT\_FAILURE);      }        // checking the amount of childres      if(AST->numberOfChildrens == 1){            // checking if it is an expression          if(AST->childrens[0]->node\_type == NON\_TERMINAL && AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_EXPRESSION){              // calling handle expression              return process\_expression(AST->childrens[0], currentScope);          }            // checking if it is an function call          if(AST->childrens[0]->node\_type == NON\_TERMINAL && AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_FUNCTION\_CALL){              // calling handle expression                retType = process\_function\_call(AST->childrens[0], currentScope);                return retType;            }            // CHECKINg if it is an literal value          if(AST->childrens[0]->node\_type == NON\_TERMINAL && AST->childrens[0]->Kind.nonTerminalType == NON\_TERMINAL\_LITERAL){              // CALLING HANDLING LITERAL EXPRESSION               return process\_literal(AST->childrens[0], currentScope);          }          if(AST->childrens[0]->node\_type == TERMINAL && AST->childrens[0]->Kind.TerminalType == IDENTIFIER\_TOKEN){                return process\_variable\_identifier(AST->childrens[0], currentScope);          }      }      // IF HAS TWO RHS      if(AST->numberOfChildrens == 2){          // can be - or ! in first place, after factor          // calling process factor recursivly          return process\_factor(AST->childrens[1], currentScope);      }  }  Variable\_And\_Return\_Type process\_function\_call(TreeNode\* AST, Scope\* currentScope){      // define local variables      char\* func\_name;      char\* errorMsg;      Symbol\_Table\_Function\_Entry\* tmpFunc;      Function\_Parameter\* func\_Params;      Variable\_And\_Return\_Type funcRetType;      int funcExists = 0;      // firstly, checking that the current ast node is fnuction call      if(AST == NULL || (AST->numberOfChildrens != 1 && AST->numberOfChildrens != 2) || AST->node\_type != NON\_TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_FUNCTION\_CALL){          printf("COMPILER ERROR!&()\n");          exit(EXIT\_FAILURE);      }      // check that there is a function declared with the same name      func\_name = extract\_variable\_name(AST->childrens[0]);      // checking to see if it exists      tmpFunc = currentScope->functionPrototype;      // looping from the current node till the beginning      while (tmpFunc)      {          // checking if the current function prototype is the wanted one          if(strcmp(tmpFunc->name, func\_name) == 0){              // setting is exists to be true              funcExists = 1;              func\_Params = tmpFunc->params;              funcRetType = tmpFunc->return\_type;          }          // iterating to the prev entry          tmpFunc = tmpFunc->prev\_entry;      }      if(!funcExists){          // in this position, the source code called an unknown or defined later function          ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine); // 10            // gett the error details          errorMsg = Semantic\_PrepareErrorMessage(4, func\_name, -1, NULL);          // outputting the error message          ErrorReporting\_ReportError(errorMsg);          // free the message          free(errorMsg);          // setting panic mode to 1          panicMode = 1;            return -1;      }      // checking if we have arguments      if(AST->numberOfChildrens == 2){          // checking that the arguments that were given matching the parameters and the types          // calling check function call arguments          check\_Function\_Call\_Arguments(AST->childrens[1], currentScope, func\_Params, func\_name);      }      else if(AST->numberOfChildrens == 1 && func\_Params != NULL){          // in this position, source code called a function without providing any arguments          //printf("\ndetected an semantic error, need to handle in error handling\ncase: 17");          //exit(EXIT\_FAILURE);          // TODO, MAKE THIS NOT EXIT BUT TO BE IN PANIC MODE FOR THAT PHASE          ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine); // 17            // gett the error details          errorMsg = Semantic\_PrepareErrorMessage(5, func\_name, -1, NULL);          // outputting the error message          ErrorReporting\_ReportError(errorMsg);          // free the message          free(errorMsg);            // setting panic mode to 1          panicMode = 1;          return -1;      }      // returning the return type of the called function      return funcRetType;    }  void check\_Function\_Call\_Arguments(TreeNode\* AST, Scope\* scope, Function\_Parameter\* cur\_param, char\* func\_name){      // define local variable      Variable\_And\_Return\_Type argumentType;      char\* errorMsg;      int static index = 0;      // firstly, making sure that the ast node is a non terminal arguments type      if(AST == NULL || AST->node\_type != NON\_TERMINAL || (AST->numberOfChildrens != 2 && AST->numberOfChildrens != 1)|| AST->Kind.nonTerminalType != NON\_TERMINAL\_ARGUMENTS){          printf("COMPILER ERROR!\n");          exit(EXIT\_FAILURE);      }      // checking if arguments dismatch parametes      if(cur\_param->next\_param == NULL && AST->numberOfChildrens == 2){          // in this position, the source code called a function with more arguments as expected          ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine); // 26            // gett the error details          errorMsg = Semantic\_PrepareErrorMessage(6, func\_name, -1, NULL);          // outputting the error message          ErrorReporting\_ReportError(errorMsg);          // free the message          free(errorMsg);          // setting panic mode to 1          panicMode = 1;      }      else if(AST->numberOfChildrens == 1 && cur\_param->next\_param != NULL){          // in this position, the cource code called a function with less arguments as expected          ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine);            // gett the error details          errorMsg = Semantic\_PrepareErrorMessage(7, func\_name, -1, NULL);          // outputting the error message          ErrorReporting\_ReportError(errorMsg);          // free the message          free(errorMsg);          // setting panic mode to 1          panicMode = 1;      }        // processing and getting the type of the      argumentType = process\_expression((AST->childrens[0])->childrens[0], scope);        if(argumentType != cur\_param->variable\_type){          // in this position, the argument in index index is type incorrect according to the function prototype          ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, (AST->childrens[0])->childrens[0]->SourceCodeLine, (AST->childrens[0])->childrens[0]->CharacterInSourceCodeLine);            // gett the error details          errorMsg = Semantic\_PrepareErrorMessage(8, func\_name, index, NULL);          // outputting the error message          ErrorReporting\_ReportError(errorMsg);          // free the message          free(errorMsg);          // setting panic mode to 1          panicMode = 1;            return;      }        // if not at the last argument, call this function recursivly, checking on the next argument      if(AST->numberOfChildrens == 2 && cur\_param->next\_param != NULL){          check\_Function\_Call\_Arguments(AST->childrens[1], scope, cur\_param->next\_param, func\_name);      }  }  Variable\_And\_Return\_Type process\_variable\_identifier(TreeNode\* AST, Scope\* currentScope){      // define local variables      Variable\_And\_Return\_Type varType;      char\* errorMsg;        // looking in the symbol tables for the defination of the identifier, and its type      // if we went able to find the declaraiton, meanign the code trying to use undeclared variable, and the compiler will yield an error      if(AST == NULL || AST->node\_type != TERMINAL || AST->Kind.TerminalType != IDENTIFIER\_TOKEN || AST->value == NULL){          printf("COMPILER ERROR13!\n");          exit(EXIT\_FAILURE);      }        // looking for the type of variable, if not exists, the called function will handle the error      varType =  variable\_declaration\_lookup(currentScope, (char\*)(AST->value));        if(varType != -1) return varType;      // in this position, variable dont exists, report and error and entering panic mode      ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->SourceCodeLine, AST->CharacterInSourceCodeLine);        // gett the error details      errorMsg = Semantic\_PrepareErrorMessage(9, (char\*)(AST->value), -1, NULL);      // outputting the error message      ErrorReporting\_ReportError(errorMsg);      // free the message      free(errorMsg);      // setting panic mode to 1      panicMode = 1;      return -1;  }  Variable\_And\_Return\_Type process\_literal(TreeNode\* AST, Scope\* currentScope){      // cheking valid ast position      if(AST ==NULL || AST->numberOfChildrens != 1 || AST->node\_type != NON\_TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_LITERAL){          printf("COMPILER ERROR14!\n");          exit(EXIT\_FAILURE);      }        if(AST->childrens[0]->node\_type != TERMINAL || (AST->childrens[0]->Kind.TerminalType != CHAR\_LITERAL\_TOKEN && AST->childrens[0]->Kind.TerminalType != DIGIT\_TOKEN)) {          printf("COMPILER ERROR15!\n");          exit(EXIT\_FAILURE);      }        // RETURNIGN THE TYPE THIS LITERAL HOLDS      if(AST->childrens[0]->Kind.TerminalType == CHAR\_LITERAL\_TOKEN){            return VARIABLE\_AND\_RETURN\_TYPE\_CHAR;      }      else{            return VARIABLE\_AND\_RETURN\_TYPE\_INT;      }  }  Variable\_And\_Return\_Type process\_assignment(TreeNode\* AST, Scope\* currentScope){      // define local variables      Variable\_And\_Return\_Type left;      Variable\_And\_Return\_Type right;      char\* errorMsg;      // checking valid ast position      if(AST ==NULL || AST->numberOfChildrens != 2 || AST->node\_type != NON\_TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_ASSIGNMENT){          printf("COMPILER ERROR15!\n");          exit(EXIT\_FAILURE);      }      // processing left and right      left = process\_variable\_identifier(AST->childrens[0], currentScope);      right = process\_expression(AST->childrens[1], currentScope);        // checking if the identifier is int      if(left == VARIABLE\_AND\_RETURN\_TYPE\_INT){            // checking if right is int          if(right == VARIABLE\_AND\_RETURN\_TYPE\_INT) return right;          // checking if right type char          if(right == VARIABLE\_AND\_RETURN\_TYPE\_CHAR){              // in this position the compiler will promote the unsign char into integer                //todo, print a warning saying we promote the char to int              printf("converting unsign character to be type int");              // returning int type              return left;          }          if(right == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // canot assign variable with value type void              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine); //14                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(16, extract\_variable\_name(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;                return VARIABLE\_AND\_RETURN\_TYPE\_INT;          }      }      if(left == VARIABLE\_AND\_RETURN\_TYPE\_CHAR){          // checking if the right side is character          if(right == VARIABLE\_AND\_RETURN\_TYPE\_CHAR) return right;          if(right == VARIABLE\_AND\_RETURN\_TYPE\_INT){              return VARIABLE\_AND\_RETURN\_TYPE\_INT;          }          if(right == VARIABLE\_AND\_RETURN\_TYPE\_VOID){              // canot assign variable with value type void              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine); //14                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(16, extract\_variable\_name(AST->childrens[0]), -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;                return VARIABLE\_AND\_RETURN\_TYPE\_CHAR;          }      }  }  void process\_return(TreeNode\* AST, Scope\* currentScope){      // define local variables      Variable\_And\_Return\_Type retType;      char\* errorMsg;      // firstly, cheeck that the current node in th ast is non terminal return type      if(AST == NULL || (AST->numberOfChildrens != 0 && AST->numberOfChildrens != 1) || AST->node\_type != NON\_TERMINAL|| AST->Kind.nonTerminalType != NON\_TERMINAL\_RETURN){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      // checking if we return an expresion      if(AST->numberOfChildrens == 1){          // getting the valeu we are getting from the return expression          retType = process\_expression(AST->childrens[0], currentScope);          // checking if we returning the declared type          if((currentScope->functionPrototype)->return\_type != retType){              // in this position, the type we are returning isnt equal the declared returned type              ErrorReporting\_ReportError\_Base(ERROR\_ON\_SEMANTIC\_PHASE, AST->childrens[0]->SourceCodeLine, AST->childrens[0]->CharacterInSourceCodeLine); //14                // gett the error details              errorMsg = Semantic\_PrepareErrorMessage(10, NULL, -1, NULL);              // outputting the error message              ErrorReporting\_ReportError(errorMsg);              // free the message              free(errorMsg);              // setting panic mode to 1              panicMode = 1;                return;          }      }    }  Variable\_And\_Return\_Type variable\_declaration\_lookup(Scope\* scope, char\* var\_name){        // looking for the declaration of the variable name in the current scope, or parents one      Scope\* tmp = scope;      while(tmp->symbol\_table->symbol\_table\_type != GLOBAL\_SYMBOL\_TABLE){            // checking if the name exists in the current scope          if(isVariableExistsInTable(tmp->symbol\_table, var\_name)){              //printf("\nman\n");              return get\_Varaible\_Type\_From\_Table(tmp->symbol\_table, var\_name);          }            // going to the parent scope          tmp = tmp->parent;      }      // if we are here, we didnt found the variabel, return -1      return -1;  }  char\* extract\_variable\_name\_lookdown(TreeNode\* AST){      // THIS FUNCTION GOES DOWN THE TREE UNTILL REACHING AN IDENTIFIER, OR FUNCTION CALL      // THIS FUNCTION ASSUME THAT THERE IS ONE CHILD EXACTLY EVERY ITERATION      // GO DOWN      while (AST->numberOfChildrens == 1)      {          AST = AST->childrens[0];      }      // CHECK THE TYPE      if(AST->node\_type == NON\_TERMINAL && AST->Kind.nonTerminalType == NON\_TERMINAL\_FUNCTION\_CALL){          // RETURN FUNCTION NAME          return extract\_variable\_name(AST->childrens[0]);      }      if(AST->node\_type == TERMINAL && AST->Kind.TerminalType == IDENTIFIER\_TOKEN){          // IDENTIFIER TOKEN          return extract\_variable\_name(AST);      }      // in this position, we called this function in an invalid position, compiler error      printf("COMPILER ERROR");      exit(EXIT\_FAILURE);  }  void freeScopeTree(Scope\* scope){        if(scope == NULL) return;      for(int i = 0; i < scope->number\_of\_childrens; i++){          freeScopeTree(scope->childrens[i]);      }        free(scope->childrens);        freeSymbolTable(scope->symbol\_table);        free(scope);    } |

## Code Generator

### CodeGenerator\_Base.h

|  |
| --- |
| #define LOCAL\_VAR\_SLOT\_SIZE 8  #define LABEL\_BASE\_NAME "Label"  #define LABEL\_BASE\_LENGTH 6  // SPECIAL REGISTERS  #define RAX "RAX"  #define RBX "RBX"  #define RCX "RCX"  #define RDX "RDX"  #define RSI "RSI"  #define RDI "RDI"  #define RBP "RBP"  #define RSP "RSP"  // GLOBAL USE REGISTERS  #define R8 "R8"  #define R9 "R9"  #define R10 "R10"  #define R11 "R11"  #define R12 "R12"  #define R13 "R13"  #define R14 "R14"  #define R15 "R15"  // LOW BYTE REGISTERS  #define AL  "AL"  #define BL  "BL"  #define CL  "CL"  #define DL  "DL"  #define SIL "SIL"  #define DIL "DIL"  #define BPL "BPL"  #define SPL "SPL"  #define R8B  "R8B"  #define R9B  "R9B"  #define R10B "R10B"  #define R11B "R11B"  #define R12B "R12B"  #define R13B "R13B"  #define R14B "R14B"  #define R15B "R15B"  #define SPECIAL\_REGISTERS\_NUM 8  #define GLOBAL\_USE\_REGISTERS\_NUM 8 |

### CodeGenerator.h

|  |
| --- |
| #include"../Parser/Parser.h"  #include"../Semantic/Scope/Scope.h"  #include"../Semantic/SymbolTable/SymbolTable.h"  #include"CodeGenerator\_Base.h"  typedef struct Register  {      // the name of the register, r11, rax, etc      char\* registerName;      // is this register is currently being use?      int inuse;  }Register;  typedef struct RegisterPool  {      // pointer into  register      Register\* registers[GLOBAL\_USE\_REGISTERS\_NUM];      // the number of registers      int number\_of\_registers;  }RegisterPool;  // Function declarations  void CodeGenerator\_InititeRegistersPool();  Register\* allocate\_register();  void free\_register(Register\* reg);  char\* CodeGenerator\_Create\_Label();  int CodeGenerator\_GetTypeSize(TreeNode\* AST);  int CodeGenerator\_GetNumberOfVariable(Scope\* scope);  Symbol\_Table\_Function\_Entry\* CodeGenerator\_FetchFunctionEntry(SymbolTable\* globalSymbolTable, char\* func\_name);  Symbol\_Table\_Variable\_Entry\* CodeGenerator\_FetchVariableEntry(Scope\* scope, char\* var\_name);  void CodeGenerator\_Generate(TreeNode\* AST, Scope\* scope, FILE\* dest\_file);  void CodeGenerator\_GenerateProgram(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GenerateFunctions();  void CodeGenerator\_GenerateFunction(TreeNode\* fnNode, Scope\* globalScope);  void CodeGenerator\_GenerateBody(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GenerateStatements(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GenerateStatement(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GenerateDeclaration(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GenerateAssignment(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateExpression(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateLogicalOr(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateLogicalAnd(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateRelationalTerm(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateAdditiveTerm(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateMultiplicativeTerm(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateFactor(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateFunctionCall(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateLiteral(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateIdentifier(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GenerateIf(TreeNode\* AST, Scope\* scope);  Register\* CodeGenerator\_GenerateGetchar(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GeneratePutchar(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GenerateReturn(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GenerateWhile(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_GenerateFunctionCall\_SetParamsRegisters(TreeNode\* AST, Scope\* scope);  void CodeGenerator\_PrintHeader();  void CodeGenerator\_write\_asm\_functions\_to\_file();  char\* CodeGenerator\_GetParameterRegister(int index);  void CodeGenerator\_PushParamRegis();  void CodeGenerator\_PopParamRegis(); |

### CodeGenerator.c

|  |
| --- |
| #include"CodeGenerator.h"  // global variable, registers pool  RegisterPool registersPool;  FILE\* OUTPUT\_FILE;  Register\* CodeGenerator\_CreateRegister(char\* name, int inuse){      Register\* reg = (Register\*)malloc(sizeof(Register));      if(reg == NULL){          printf("MALLOC ERROR");          exit(EXIT\_FAILURE);      }      reg->inuse = inuse;      reg->registerName = \_strdup(name);  }  void CodeGenerator\_InititeRegistersPool(){      // define local variables      int reg = 0;        // setting the registers pool attributes      registersPool.number\_of\_registers = GLOBAL\_USE\_REGISTERS\_NUM;      // setting the global registers in the pool      registersPool.registers[reg++] =  CodeGenerator\_CreateRegister(R8, 0);      registersPool.registers[reg++] =  CodeGenerator\_CreateRegister(R9, 0);      registersPool.registers[reg++] =  CodeGenerator\_CreateRegister(R10, 0);      registersPool.registers[reg++] =  CodeGenerator\_CreateRegister(R11, 0);      registersPool.registers[reg++] =  CodeGenerator\_CreateRegister(R12, 0);      registersPool.registers[reg++] =  CodeGenerator\_CreateRegister(R13, 0);      registersPool.registers[reg++] =  CodeGenerator\_CreateRegister(R14, 0);      registersPool.registers[reg++] =  CodeGenerator\_CreateRegister(R15, 0);  }  Register\* allocate\_register() {      for (int i = registersPool.number\_of\_registers - 1; i >= 0; --i) {          if (!registersPool.registers[i]->inuse) {              registersPool.registers[i]->inuse = 1;              return registersPool.registers[i];          }      }      printf("COMPILER ERROR: No available registers\n");      return NULL;  }  void free\_register(Register\* reg) {      reg->inuse = 0;  }  char\* CodeGenerator\_Create\_Label(){      // this function generate unique label names      // define static variable      static int label\_c = 1;      // define local variables      char buf[15];      sprintf(buf, "%s%d", LABEL\_BASE\_NAME, label\_c++);      return \_strdup(buf);  }  int CodeGenerator\_GetTypeSize(TreeNode\* AST){      // checking that the current ast node is terminal      if(AST->node\_type == NON\_TERMINAL){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      // cheking if int      if(AST->Kind.TerminalType == INT\_TOKEN){          return 8; // 8 \*8 = 64 bits      }      // cheking if char      if(AST->Kind.TerminalType == CHAR\_TOKEN){          return 1; // 1 \* 8 = 8  bits      }      // if here, error      printf("COMPILER ERROR");      exit(EXIT\_FAILURE);  }  char\* CodeGenerator\_GetRegisterLowestByteRegister(const char\* reg) {      if (strcmp(reg, RAX) == 0) return \_strdup(AL);      if (strcmp(reg, RBX) == 0) return \_strdup(BL);      if (strcmp(reg, RCX) == 0) return \_strdup(CL);      if (strcmp(reg, RDX) == 0) return \_strdup(DL);      if (strcmp(reg, RSI) == 0) return \_strdup(SIL);      if (strcmp(reg, RDI) == 0) return \_strdup(DIL);      if (strcmp(reg, RBP) == 0) return \_strdup(BPL);      if (strcmp(reg, RSP) == 0) return \_strdup(SPL);      if (strcmp(reg, R8)  == 0) return \_strdup(R8B);      if (strcmp(reg, R9)  == 0) return \_strdup(R9B);      if (strcmp(reg, R10) == 0) return \_strdup(R10B);      if (strcmp(reg, R11) == 0) return \_strdup(R11B);      if (strcmp(reg, R12) == 0) return \_strdup(R12B);      if (strcmp(reg, R13) == 0) return \_strdup(R13B);      if (strcmp(reg, R14) == 0) return \_strdup(R14B);      if (strcmp(reg, R15) == 0) return \_strdup(R15B);      return NULL; // Unknown register  }  int CodeGenerator\_GetNumberOfVariable(Scope\* scope){      // this function count and return the number of declared varialbles      // in this current scope and in all the inner scopes, not addressing variable type      // define loacal variables      int prev\_c = 0;      int i;      // counting the amount of declared variables in the inner scopes fiest      // if we are in a leaf scope, thats mean that the number of childrens is zero and this      // for loop will not call the function recursivly and behave as an exit condition      for(i = 0; i < scope->number\_of\_childrens; i++){          prev\_c += CodeGenerator\_GetNumberOfVariable(scope->childrens[i]);      }      // returning the prev count + the count in this scope      return prev\_c + scope->variables\_c;  }  Symbol\_Table\_Function\_Entry\* CodeGenerator\_FetchFunctionEntry(SymbolTable\* globalSymbolTable, char\* func\_name){      // this function returning the entry which beling to the function with the given name      // define local variables      Symbol\_Table\_Function\_Entry\* tmp;      // getting the first entry      tmp = globalSymbolTable->Function\_Entry;      // looping for all nodes in the list      while (tmp != NULL)      {          if(strcmp(tmp->name, func\_name) == 0){              // return the current entry              return tmp;          }          // iterating to the next entry in the list          tmp = tmp->next\_entry;      }      return NULL;    }  Symbol\_Table\_Variable\_Entry\* CodeGenerator\_FetchVariableEntry(Scope\* scope, char\* var\_name) {      Symbol\_Table\_Variable\_Entry\* tmp;      //printf("looking for name: %s\n", var\_name);      if(scope == NULL){          printf("COMPILER ERROR1");          exit(EXIT\_FAILURE);      }      if(scope->parent == NULL){          printf("COMPILER ERROR2");          exit(EXIT\_FAILURE);      }        // Walk up the scope hierarchy      while (scope != NULL && scope->symbol\_table != NULL) {          if (scope->symbol\_table->symbol\_table\_type == GLOBAL\_SYMBOL\_TABLE) {              break; // Stop at global scope          }            tmp = scope->symbol\_table->Variable\_Entry;          //print\_symbol\_table(scope->symbol\_table);          while (tmp != NULL) {                if (strcmp(tmp->name, var\_name) == 0) {                  //printf("found");                  return tmp; // Found              }              tmp = tmp->next\_entry;          }          // Move up to parent          scope = scope->parent;      }      return NULL; // Not found  }  void CodeGenerator\_PrintHeader() {      fprintf(OUTPUT\_FILE, "BITS 64          ; Specify 64-bit code generation\n");      fprintf(OUTPUT\_FILE, "DEFAULT REL      ; Use RIP-relative addressing by default\n");      fprintf(OUTPUT\_FILE, "global main\n");      fprintf(OUTPUT\_FILE, "extern GetStdHandle\n");      fprintf(OUTPUT\_FILE, "extern WriteConsoleA\n");      fprintf(OUTPUT\_FILE, "extern ReadConsoleA\n");      fprintf(OUTPUT\_FILE, "extern ExitProcess\n");      fprintf(OUTPUT\_FILE, "\n");      fprintf(OUTPUT\_FILE, "section .data\n");      fprintf(OUTPUT\_FILE, "CHAR\_BUF db 0\n");      fprintf(OUTPUT\_FILE, "STDOUT\_HANDLE dq 0\n");      fprintf(OUTPUT\_FILE, "N\_WRITTEN dq 0\n");      fprintf(OUTPUT\_FILE, "STDIN\_HANDLE dq 0  ; Declare stdin\_handle in the .data section\n");      fprintf(OUTPUT\_FILE, "N\_READ dq 0        ; Declare n\_read in the .data section\n");      fprintf(OUTPUT\_FILE, "\n");      fprintf(OUTPUT\_FILE, "section .text\n");  }  #include <stdio.h> // Assuming you have this included  // Assuming OUTPUT\_FILE is a valid FILE\* pointer opened for writing  extern FILE \*OUTPUT\_FILE;  void CodeGenerator\_write\_asm\_functions\_to\_file() {      // Ensure you are in the .text section before writing functions      // fprintf(OUTPUT\_FILE, "section .text\n"); // Uncomment if needed      fprintf(OUTPUT\_FILE, "; Writes a single character (passed in RCX) to the console.\n");      fprintf(OUTPUT\_FILE, "; Uses local stack space for buffer and bytes written variable.\n");      fprintf(OUTPUT\_FILE, "putchar:\n");      fprintf(OUTPUT\_FILE, "    ; ABI: Character expected in CL (lowest byte of RCX)\n");      fprintf(OUTPUT\_FILE, "    PUSH RBP\n");      fprintf(OUTPUT\_FILE, "    MOV RBP, RSP\n");      fprintf(OUTPUT\_FILE, "    ; Allocate 40 bytes: \n");      fprintf(OUTPUT\_FILE, "    ; - Aligns RSP: RBP is (Initial RSP - 8), so RBP - 40 = Initial RSP - 48 (Aligned)\n");      fprintf(OUTPUT\_FILE, "    ; - Provides 32 bytes shadow space for callees below entry RSP.\n");      fprintf(OUTPUT\_FILE, "    ; - Provides space for locals (1 byte char buffer, 8 byte QWORD target for N\_WRITTEN).\n");      fprintf(OUTPUT\_FILE, "    SUB RSP, 40\n");      fprintf(OUTPUT\_FILE, "    ; Stack frame layout relative to RBP:\n");      fprintf(OUTPUT\_FILE, "    ; [RBP - 8] .. [RBP - 40] = 32 bytes available below RBP for locals/padding.\n");      fprintf(OUTPUT\_FILE, "    ; Use [RBP - 16] for N\_WRITTEN target (needs 8 bytes).\n");      fprintf(OUTPUT\_FILE, "    ; Use [RBP - 24] for char buffer (needs 1 byte).\n");      fprintf(OUTPUT\_FILE, "    \n");      fprintf(OUTPUT\_FILE, "    ; Store the character (from CL) onto the STACK buffer\n");      fprintf(OUTPUT\_FILE, "    MOV BYTE [RBP - 24], CL\n");      fprintf(OUTPUT\_FILE, "    \n");      fprintf(OUTPUT\_FILE, "    ; Get STDOUT handle if not already obtained (still cache globally)\n");      fprintf(OUTPUT\_FILE, "    CMP QWORD [rel STDOUT\_HANDLE], 0\n"); // Use rel for global handle cache      fprintf(OUTPUT\_FILE, "    JNE .putchar\_have\_handle\_stack\n");   // Use unique label      fprintf(OUTPUT\_FILE, "    MOV RCX, -11               ; STD\_OUTPUT\_HANDLE\n");      fprintf(OUTPUT\_FILE, "    CALL GetStdHandle          ; Returns handle in RAX\n");      fprintf(OUTPUT\_FILE, "    ; Optional but Recommended: Check GetStdHandle for errors\n");      fprintf(OUTPUT\_FILE, "    ; Example: CMP RAX, -1 ; or CMP RAX, 0 depending on what errors to check\n");      fprintf(OUTPUT\_FILE, "    ;          JE .putchar\_error\_handler\n");      fprintf(OUTPUT\_FILE, "    MOV [rel STDOUT\_HANDLE], RAX\n");     // Store handle globally      fprintf(OUTPUT\_FILE, ".putchar\_have\_handle\_stack:\n");         // Unique label      fprintf(OUTPUT\_FILE, "    \n");      fprintf(OUTPUT\_FILE, "    ; Prepare arguments for WriteConsoleA using STACK locations\n");      fprintf(OUTPUT\_FILE, "    MOV RCX, [rel STDOUT\_HANDLE]    ; 1st: hConsoleOutput (from global cache)\n");      fprintf(OUTPUT\_FILE, "    LEA RDX, [RBP - 24]             ; 2nd: lpBuffer (address of char on STACK)\n");      fprintf(OUTPUT\_FILE, "    MOV R8, 1                       ; 3rd: nNumberOfCharsToWrite\n");      fprintf(OUTPUT\_FILE, "    LEA R9, [RBP - 16]             ; 4th: lpNumberOfCharsWritten (address on STACK for dummy output)\n");      fprintf(OUTPUT\_FILE, "    MOV QWORD [RSP+32], 0           ; 5th: lpReserved (NULL) - Positioned relative to current RSP (accesses caller's arg space)\n");      fprintf(OUTPUT\_FILE, "    CALL WriteConsoleA\n");      fprintf(OUTPUT\_FILE, "    ; Optional but Recommended: Check WriteConsoleA for errors (returns 0 on failure)\n");      fprintf(OUTPUT\_FILE, "    ; Example: TEST RAX, RAX\n");      fprintf(OUTPUT\_FILE, "    ;          JZ .putchar\_error\_handler\n");      fprintf(OUTPUT\_FILE, "    \n");      fprintf(OUTPUT\_FILE, "    ; CLEAN UP STACK AND RETURN using LEAVE\n");      fprintf(OUTPUT\_FILE, "    LEAVE                    ; Equivalent to: MOV RSP, RBP; POP RBP\n");      fprintf(OUTPUT\_FILE, "    RET\n");      fprintf(OUTPUT\_FILE, "\n");        // --- Write getchar function (Corrected for Win64) ---      // Start generating the assembly 'getchar' function      fprintf(OUTPUT\_FILE, "; --- getchar (Re-entrant version using stack) ---\n");      fprintf(OUTPUT\_FILE, "; Reads a single character from the console.\n");      fprintf(OUTPUT\_FILE, "; Takes NO parameters.\n"); // Explicitly state no params for assembly func      fprintf(OUTPUT\_FILE, "; Returns character in AL (zero-extended into RAX).\n");      fprintf(OUTPUT\_FILE, "; Uses local stack space for buffer and bytes read variable.\n");      fprintf(OUTPUT\_FILE, "getchar:\n"); // Label for the assembly function      // Standard Win64 function prologue      fprintf(OUTPUT\_FILE, "    PUSH RBP\n");      fprintf(OUTPUT\_FILE, "    MOV RBP, RSP\n");      // Allocate stack space: 40 bytes total below saved RBP.      // Ensures 16-byte alignment for RSP after allocation (RBP-40 is Initial\_RSP-48).      // Provides 32 bytes shadow space for API calls below entry RSP.      // Provides local space for 1-byte buffer and 8-byte QWORD target.      fprintf(OUTPUT\_FILE, "    SUB RSP, 40\n");      fprintf(OUTPUT\_FILE, "    ; Local stack layout: Use [RBP - 16] for N\_READ target, [RBP - 24] for char buffer.\n");      fprintf(OUTPUT\_FILE, "    \n");      // Get STDIN handle (using global cache)      fprintf(OUTPUT\_FILE, "    ; Get STDIN handle if not already obtained (still cache globally)\n");      fprintf(OUTPUT\_FILE, "    CMP QWORD [rel STDIN\_HANDLE], 0\n");      fprintf(OUTPUT\_FILE, "    JNE .getchar\_have\_handle\_stack\n");      fprintf(OUTPUT\_FILE, "    MOV RCX, -10               ; STD\_INPUT\_HANDLE (Arg 1 for GetStdHandle)\n");      fprintf(OUTPUT\_FILE, "    CALL GetStdHandle          ; Returns handle in RAX\n");      fprintf(OUTPUT\_FILE, "    ; Optional: Check GetStdHandle for errors\n");      fprintf(OUTPUT\_FILE, "    MOV [rel STDIN\_HANDLE], RAX\n");      fprintf(OUTPUT\_FILE, ".getchar\_have\_handle\_stack:\n");      fprintf(OUTPUT\_FILE, "    \n");      // Prepare arguments for ReadConsoleA using stack locations      fprintf(OUTPUT\_FILE, "    ; Prepare arguments for ReadConsoleA using STACK locations\n");      fprintf(OUTPUT\_FILE, "    MOV RCX, [rel STDIN\_HANDLE]     ; Arg 1: hConsoleInput (from global cache)\n");      fprintf(OUTPUT\_FILE, "    LEA RDX, [RBP - 24]             ; Arg 2: lpBuffer (address of char buffer on STACK)\n");      fprintf(OUTPUT\_FILE, "    MOV R8, 1                       ; Arg 3: nNumberOfCharsToRead\n");      fprintf(OUTPUT\_FILE, "    LEA R9, [RBP - 16]             ; Arg 4: lpNumberOfCharsRead (address on STACK)\n");      fprintf(OUTPUT\_FILE, "    MOV QWORD [RSP+32], 0           ; Arg 5: pInputControl (NULL) - Placed in caller's arg space\n");      // Call ReadConsoleA to get the character      fprintf(OUTPUT\_FILE, "    CALL ReadConsoleA\n");      fprintf(OUTPUT\_FILE, "    ; Optional: Check ReadConsoleA for errors (returns 0 on failure)\n");      fprintf(OUTPUT\_FILE, "    \n");      // Prepare return value      fprintf(OUTPUT\_FILE, "    ; Return character read from STACK buffer in AL (zero-extended to RAX)\n");      fprintf(OUTPUT\_FILE, "    MOVZX EAX, BYTE [RBP - 24]\n"); // Read from stack buffer, zero-extend into EAX/RAX      fprintf(OUTPUT\_FILE, "    \n");      // Standard Win64 function epilogue      fprintf(OUTPUT\_FILE, "    ; CLEAN UP STACK AND RETURN using LEAVE\n");      fprintf(OUTPUT\_FILE, "    LEAVE                    ; MOV RSP, RBP; POP RBP\n");      fprintf(OUTPUT\_FILE, "    RET\n"); // Return to caller, result is in RAX      fprintf(OUTPUT\_FILE, "\n");  }  void CodeGenerator\_Generate(TreeNode\* AST, Scope\* scope, FILE\* dest\_file){      // main function for generating the destination code      // first of all, set the global variable to teh given file      OUTPUT\_FILE = dest\_file;      // firstly, print the headers and text needed for nasm      CodeGenerator\_PrintHeader();      // print putchar and getchar io function helpers      CodeGenerator\_write\_asm\_functions\_to\_file();      fflush(OUTPUT\_FILE);      // intiialge the registers pool      CodeGenerator\_InititeRegistersPool();      // calling generating program      CodeGenerator\_GenerateProgram(AST, scope);  }  void CodeGenerator\_GenerateProgram(TreeNode\* AST, Scope\* scope){      CodeGenerator\_GenerateFunctions(AST->childrens[0], scope);  }  void CodeGenerator\_GenerateFunctions(TreeNode\* AST, Scope\* scope){      if(AST->numberOfChildrens == 1){          CodeGenerator\_GenerateFunction(AST->childrens[0], scope);      }      else if(AST->numberOfChildrens == 2){          CodeGenerator\_GenerateFunction(AST->childrens[0], scope);          CodeGenerator\_GenerateFunctions(AST->childrens[1], scope);      }  }  void CodeGenerator\_GenerateFunction(TreeNode\* fnNode, Scope\* globalScope) {      // define local variables      char \*func\_name;      Symbol\_Table\_Function\_Entry\* func\_entry;      int vars\_count;      char\* finishLabel;      // this varible hold the size of the amount of bytes we allocated in the stack for local variables      // this varible need to be divisable by 16      int stack\_sub;      // get the name of the function      func\_name = extract\_variable\_name(fnNode->childrens[1]);      // getting the function entry based on its name      func\_entry = CodeGenerator\_FetchFunctionEntry(globalScope->symbol\_table, func\_name);      // getting the amount of declaration in the whole function(this scope and all inner scopes)      vars\_count = CodeGenerator\_GetNumberOfVariable(globalScope->childrens[func\_entry->function\_index]);      // setting an label for jumping to the end of the function      // in case where the function uses the return function somehwere inside      finishLabel = CodeGenerator\_Create\_Label();      // settin the function return label      func\_entry->function\_return\_label = finishLabel;      // checking if this function has 3 or 4 parameters, need to mark the global regsiters as used      if(func\_entry->number\_of\_parameters >= 2){          // setting r8, which is the fith parameter as inuse          registersPool.registers[0]->inuse = 1;      }      if(func\_entry->number\_of\_parameters == 3){          // setting r9, which is the sixth parameter as inuse          registersPool.registers[1]->inuse = 1;      }      // calculating the subeed stack size      stack\_sub = LOCAL\_VAR\_SLOT\_SIZE \* vars\_count + 32;        // checking if stack sub isnt divisable by 16      if(stack\_sub % 16 != 0){          stack\_sub += LOCAL\_VAR\_SLOT\_SIZE;      }        if(func\_name == NULL || func\_entry == NULL){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      // 1) Label & prologue      fprintf(OUTPUT\_FILE, "%s:\n", func\_name);      fprintf(OUTPUT\_FILE, "    PUSH RBP\n");      fprintf(OUTPUT\_FILE, "    MOV RBP, RSP\n");      fprintf(OUTPUT\_FILE, "    SUB RSP, %d\n", stack\_sub);      fflush(OUTPUT\_FILE);      // 2) Body      TreeNode \*body = (fnNode->numberOfChildrens == 3)                        ? fnNode->childrens[2]                        : fnNode->childrens[3];      // generating the function body code      CodeGenerator\_GenerateBody(body, globalScope->childrens[func\_entry->function\_index]);      fprintf(OUTPUT\_FILE, "%s:\n", finishLabel);      // checking if main function      if(strcmp(func\_entry->name, "main") != 0){          fprintf(OUTPUT\_FILE, "    ADD RSP, %d\n", stack\_sub);            //fprintf(OUTPUT\_FILE, "    MOV RSP, RBP\n");            fprintf(OUTPUT\_FILE, "    pop RBP\n");          fprintf(OUTPUT\_FILE, "    ret\n\n");      }      else{          // if main function, exiting program properly and returning status 0            fprintf(OUTPUT\_FILE, "  MOV %s, 0\n", CodeGenerator\_GetParameterRegister(0));          fprintf(OUTPUT\_FILE, "  CALL ExitProcess\n");      }    }  void CodeGenerator\_GenerateBody(TreeNode\* AST, Scope\* scope){      if(AST == NULL || AST->numberOfChildrens != 1 || AST->node\_type != NON\_TERMINAL || AST->Kind.nonTerminalType != NON\_TERMINAL\_BODY){          printf("COMPILER ERRORL NO\n");          exit(EXIT\_FAILURE);      }      if(AST->scope == NULL){          printf("COMPILER ERROR YES");          exit(EXIT\_FAILURE);      }      CodeGenerator\_GenerateStatements(AST->childrens[0], (Scope\*)AST->scope);  }  void CodeGenerator\_GenerateStatements(TreeNode\* AST, Scope\* scope){      while(AST->numberOfChildrens == 2){          CodeGenerator\_GenerateStatement(AST->childrens[0], scope);          AST = AST->childrens[1];      }      CodeGenerator\_GenerateStatement(AST->childrens[0], scope);  }  void CodeGenerator\_GenerateStatement(TreeNode\* AST, Scope\* scope){      if(AST->node\_type != NON\_TERMINAL){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      switch (AST->childrens[0]->Kind.nonTerminalType)      {          case NON\_TERMINAL\_DECLARATION:              CodeGenerator\_GenerateDeclaration(AST->childrens[0], scope);              break;          case NON\_TERMINAL\_ASSIGNMENT:              CodeGenerator\_GenerateAssignment(AST->childrens[0], scope);              break;          case NON\_TERMINAL\_EXPRESSION:              CodeGenerator\_GenerateExpression(AST->childrens[0], scope);              break;          case NON\_TERMINAL\_IF\_STATEMENT:              //printf("problem start here=========\n");              CodeGenerator\_GenerateIf(AST->childrens[0], scope);              break;          case NON\_TERMINAL\_RETURN:              CodeGenerator\_GenerateReturn(AST->childrens[0], scope);              break;          case NON\_TERMINAL\_WHILE\_LOOP:              CodeGenerator\_GenerateWhile(AST->childrens[0], scope);              break;          case NON\_TERMINAL\_PUTCHAR:              CodeGenerator\_GeneratePutchar(AST->childrens[0], scope);              break;          case NON\_TERMINAL\_GETCHAR:              CodeGenerator\_GenerateGetchar(AST->childrens[0], scope);              break;          default:          break;      }  }  void CodeGenerator\_GenerateIf(TreeNode\* AST, Scope\* scope){      //printf("CodeGenerator\_GenerateIf1");      // define variables      char\* doneLabel = CodeGenerator\_Create\_Label();      char\* falseLabel = CodeGenerator\_Create\_Label();      Register\* reg;      //printf("before");      // GETTING THE REGISTER HOLDING THE EXPRESSION VALUE      reg = CodeGenerator\_GenerateExpression(AST->childrens[0], scope);      //printf("here?");      // comparing to zero      fprintf(OUTPUT\_FILE, "    CMP %s, 0\n", reg->registerName);      fprintf(OUTPUT\_FILE, "    JE %s\n", falseLabel);      // handling true case body      CodeGenerator\_GenerateBody(AST->childrens[1], scope);      fprintf(OUTPUT\_FILE, "    JMP %s\n", doneLabel);      fprintf(OUTPUT\_FILE, "%s:\n", falseLabel);      // handling else and else if      if(AST->numberOfChildrens == 3){          //printf("inside");          // handling case of else if, or else          // checking if it is else if or else          if(AST->childrens[2]->childrens[0]->node\_type == TERMINAL && AST->childrens[2]->childrens[0]->Kind.TerminalType == ELSE\_TOKEN &&              AST->childrens[2]->childrens[1]->node\_type == NON\_TERMINAL && AST->childrens[2]->childrens[1]->Kind.TerminalType == NON\_TERMINAL\_IF\_STATEMENT){              // WE ARE IN A ELSE IF              // calling this function recursivly              //printf("inside2");              CodeGenerator\_GenerateIf(AST->childrens[2]->childrens[1], scope);          }          else if(AST->childrens[2]->childrens[0]->node\_type == TERMINAL && AST->childrens[2]->childrens[0]->Kind.TerminalType == ELSE\_TOKEN &&              AST->childrens[2]->childrens[1]->node\_type == NON\_TERMINAL && AST->childrens[2]->childrens[1]->Kind.TerminalType == NON\_TERMINAL\_BODY){                  // WE ARE IN ELSE STATE                  //printf("heere");                  CodeGenerator\_GenerateBody(AST->childrens[2]->childrens[1], scope);          }          }      free\_register(reg);      fprintf(OUTPUT\_FILE, "%s:\n", doneLabel);  }  void CodeGenerator\_GenerateDeclaration(TreeNode\* AST, Scope\* scope){      // getting the size of the type of the new var      //int size = CodeGenerator\_GetTypeSize(AST->childrens[0]->childrens[0]);      int size = 8;      int flag = 0;      Scope\* func\_scope;      Symbol\_Table\_Variable\_Entry\* var\_entry;      Symbol\_Table\_Function\_Entry\* func\_entry;        // gettin the variable name      char\* var\_name = extract\_variable\_name(AST->childrens[1]);      // getting the function entry      func\_entry = scope->functionPrototype;        // getting the variable entry      var\_entry = CodeGenerator\_FetchVariableEntry(scope, var\_name);      // setting the bp relative of the variable      var\_entry->index\_bp\_relative = -size \*((func\_entry->declared\_vars++) + 1);  }  void CodeGenerator\_GenerateAssignment(TreeNode\* AST, Scope\* scope){      // define variables      Register\* reg;      char\* var\_name;      Symbol\_Table\_Variable\_Entry\* var\_entry;        // firstly, need to evaluate the right side expression      reg = CodeGenerator\_GenerateExpression(AST->childrens[1], scope);      //printf("alpha");      // getting the name of the variable      var\_name = extract\_variable\_name(AST->childrens[0]);        // getting the varaible entry from the scope table      var\_entry = CodeGenerator\_FetchVariableEntry(scope, var\_name);      fprintf(OUTPUT\_FILE, "    MOV [RBP %d], %s\n", var\_entry->index\_bp\_relative, reg->registerName);      // freeing the register      free\_register(reg);  }  Register\* CodeGenerator\_GenerateExpression(TreeNode\* AST, Scope\* scope){      Register\* temp =  CodeGenerator\_GenerateLogicalOr(AST->childrens[0], scope);      //printf("ret 1\n");      //if(temp == NULL) printf("null");      return temp;  }  Register\* CodeGenerator\_GenerateLogicalOr(TreeNode\* AST, Scope\* scope){      //printf("2");      // define local variables      Register\* regLeft = NULL;      Register\* regRight = NULL;      char\* trueLabel;      char\* falseLabel;      if(AST->numberOfChildrens == 1){          regLeft = CodeGenerator\_GenerateLogicalAnd(AST->childrens[0], scope);            //printf("ret 2\n");        }      if(AST->numberOfChildrens == 3){          trueLabel = CodeGenerator\_Create\_Label();          falseLabel = CodeGenerator\_Create\_Label();          // handing logical or          // evaluating logical and and get the return register          regLeft = CodeGenerator\_GenerateLogicalOr(AST->childrens[0], scope);          fprintf(OUTPUT\_FILE, "    CMP %s, 0\n", regLeft->registerName);          fprintf(OUTPUT\_FILE, "    JNE %s\n", trueLabel);          // evalutgin right side          regRight = CodeGenerator\_GenerateLogicalAnd(AST->childrens[2], scope);          // do the logical or operation          //fprintf(OUTPUT\_FILE, "    OR %s, %s\n", regLeft->registerName, regRight->registerName);            fprintf(OUTPUT\_FILE, "    CMP %s, 0\n", regRight->registerName);          fprintf(OUTPUT\_FILE, "    JNE %s\n", trueLabel);          fprintf(OUTPUT\_FILE, "    MOV %s, 0\n", regRight);          fprintf(OUTPUT\_FILE, "    JMP %s\n", falseLabel);          fprintf(OUTPUT\_FILE, "%s:\n", trueLabel);          fprintf(OUTPUT\_FILE, "    MOV %s, 1\n", regRight);          fprintf(OUTPUT\_FILE, "%s:\n", falseLabel);          // freeing the right register          free\_register(regRight);        }      //printf("ret 2\n");      // returning the left register      return regLeft;  }  Register\* CodeGenerator\_GenerateLogicalAnd(TreeNode\* AST, Scope\* scope){      //printf("3");      // define local variables      Register\* regLeft = NULL;      Register\* regRight = NULL;      char\* trueLabel;      char\* falseLabel;     if(AST->numberOfChildrens == 1){         regLeft = CodeGenerator\_GenerateRelationalTerm(AST->childrens[0], scope);           //printf("ret 3\n");      }     if(AST->numberOfChildrens == 3){          trueLabel = CodeGenerator\_Create\_Label();          falseLabel = CodeGenerator\_Create\_Label();         // handing logical or         // evaluating left side         regLeft = CodeGenerator\_GenerateLogicalAnd(AST->childrens[0], scope);         fprintf(OUTPUT\_FILE, "    CMP %s, 0\n", regLeft->registerName);         fprintf(OUTPUT\_FILE, "    JE %s\n", falseLabel);         // evalutgin right side         regRight = CodeGenerator\_GenerateRelationalTerm(AST->childrens[2], scope);         fprintf(OUTPUT\_FILE, "    CMP %s, 0\n", regRight->registerName);         fprintf(OUTPUT\_FILE, "    JE %s\n", falseLabel);         // do the logical and operation         //fprintf(OUTPUT\_FILE, "    AND %s, %s\n", regLeft->registerName, regRight->registerName);         fprintf(OUTPUT\_FILE, "    MOV %s, 1\n", regLeft->registerName);         fprintf(OUTPUT\_FILE, "    JMP %s\n", trueLabel);         fprintf(OUTPUT\_FILE, "%s:\n", falseLabel);         fprintf(OUTPUT\_FILE, "    MOV %s, 0\n",regLeft->registerName);         fprintf(OUTPUT\_FILE, "%s:\n", trueLabel);         // freeing the right register         free\_register(regRight);      }      //printf("ret 3\n");     // returning the left register     return regLeft;  }  Register\* CodeGenerator\_GenerateRelationalTerm(TreeNode\* AST, Scope\* scope){      //printf("4");      // define local variables      // registers      Register\* regLeft = NULL;      Register\* regRight = NULL;      // labels      char\* trueLabel;      char\* doneLabel;      if(AST->numberOfChildrens == 1){          regLeft = CodeGenerator\_GenerateAdditiveTerm(AST->childrens[0], scope);          //printf("ret 4\n");        }      if(AST->numberOfChildrens == 3){          //printf("i am do here!\n");          // evaluate left side          regLeft = CodeGenerator\_GenerateAdditiveTerm(AST->childrens[0], scope);          //printf("i am still here");          // evalaute right side          regRight = CodeGenerator\_GenerateAdditiveTerm(AST->childrens[2], scope);          //printf("still here");          // getting unique names for the labels          trueLabel = CodeGenerator\_Create\_Label();          doneLabel = CodeGenerator\_Create\_Label();          //printf("reg1 is: %s", (regLeft == NULL)? "NULL":"NOT NULL");          // printf("reg2 is: %s", (regRight == NULL)? "NULL":"NOT NULL");            //printf("reg1: %d\nreg2: \n", regLeft->registerKind, regRight->registerKind);          // comparing the two registers          fprintf(OUTPUT\_FILE, "    CMP %s, %s\n", regLeft->registerName, regRight->registerName);          // do the retional operation          switch (AST->childrens[1]->Kind.TerminalType)          {              case EQUAL\_TO\_TOKEN:                  fprintf(OUTPUT\_FILE, "    JE %s\n", trueLabel);                  break;              case NOT\_EQUAL\_TO\_TOKEN:                  fprintf(OUTPUT\_FILE, "    JNE %s\n", trueLabel);                  break;              case LESS\_THEN\_TOKEN:                  fprintf(OUTPUT\_FILE, "    JL %s\n", trueLabel);                  break;              case GREATER\_THEN\_TOKEN:                  fprintf(OUTPUT\_FILE, "    JG %s\n", trueLabel);                  break;              case LESS\_THEN\_OR\_EQUAL\_TO\_TOKEN:                  fprintf(OUTPUT\_FILE, "    JLE %s\n", trueLabel);                  break;              case GREATER\_THEN\_OR\_EQUAL\_TO\_TOKEN:                  fprintf(OUTPUT\_FILE, "    JGE %s\n", trueLabel);                  break;          }          fprintf(OUTPUT\_FILE, "    MOV %s, 0\n", regLeft->registerName);          fprintf(OUTPUT\_FILE, "    JMP %s\n", doneLabel);          fprintf(OUTPUT\_FILE, "%s:\n", trueLabel);          fprintf(OUTPUT\_FILE, "    MOV %s, 1\n", regLeft->registerName);          fprintf(OUTPUT\_FILE, "%s:\n", doneLabel);          // freeing dynamic strings          free(trueLabel);          free(doneLabel);          // freeing the right register          free\_register(regRight);      }      //printf("ret 4");      // returning the left register      return regLeft;  }  Register\* CodeGenerator\_GenerateAdditiveTerm(TreeNode\* AST, Scope\* scope){      //printf("5");      // define local variables      // registers      Register\* regLeft = NULL;      Register\* regRight = NULL;        if(AST->numberOfChildrens == 1){          regLeft = CodeGenerator\_GenerateMultiplicativeTerm(AST->childrens[0], scope);            //printf("ret 5\n");      }      if(AST->numberOfChildrens == 3){          // evaluate left side          regLeft = CodeGenerator\_GenerateMultiplicativeTerm(AST->childrens[0], scope);          // evalaute right side          regRight = CodeGenerator\_GenerateAdditiveTerm(AST->childrens[2], scope);          switch(AST->childrens[1]->Kind.TerminalType){              case PLUS\_TOKEN:                  fprintf(OUTPUT\_FILE, "    ADD %s, %s\n", regLeft->registerName, regRight->registerName);                  break;              case MINUS\_TOKEN:                  fprintf(OUTPUT\_FILE, "    SUB %s, %s\n", regLeft->registerName, regRight->registerName);                  break;          }          // free the right side register          free\_register(regRight);      }      //printf("ret 5\n");      // returning the left register      return regLeft;    }  Register\* CodeGenerator\_GenerateMultiplicativeTerm(TreeNode\* AST, Scope\* scope){      //printf("6");      // define local variables      // registers      Register\* regLeft = NULL;      Register\* regRight = NULL;      if(AST->numberOfChildrens == 1){          regLeft = CodeGenerator\_GenerateFactor(AST->childrens[0], scope);          //printf("ret 6\n");          //printf("%d", regLeft->registerKind);      }      if(AST->numberOfChildrens == 3){          // evaluate left side          regLeft = CodeGenerator\_GenerateFactor(AST->childrens[0], scope);          // evalaute right side          regRight = CodeGenerator\_GenerateMultiplicativeTerm(AST->childrens[2], scope);          switch(AST->childrens[1]->Kind.TerminalType){              case MULTIPLY\_TOKEN:                  fprintf(OUTPUT\_FILE, "    IMUL %s, %s\n", regLeft->registerName, regRight->registerName);                  break;              case DIVIDE\_TOKEN:                  fprintf(OUTPUT\_FILE, "    MOV RDX, 0\n");                  fprintf(OUTPUT\_FILE, "    MOV RAX, %s\n", regLeft->registerName);                  fprintf(OUTPUT\_FILE, "    IDIV %s\n", regRight->registerName);                  fprintf(OUTPUT\_FILE, "    MOV %s, RAX\n", regLeft->registerName);                  break;              case MODULO\_TOKEN:                  fprintf(OUTPUT\_FILE, "    MOV RDX, 0\n");                  fprintf(OUTPUT\_FILE, "    MOV RAX, %s\n", regLeft->registerName);                  fprintf(OUTPUT\_FILE, "    IDIV %s\n", regRight->registerName);                  fprintf(OUTPUT\_FILE, "    MOV %s, RDX\n", regLeft->registerName);                  break;          }          // freeing the right register          free\_register(regRight);      }          // returning the left register          return regLeft;    }  Register\* CodeGenerator\_GenerateFactor(TreeNode\* AST, Scope\* scope){      //printf("7");      // define variables      Register\* reg = NULL;      char\* trueLabel;      char\* doneLabel;      //printf("z");      if(AST->numberOfChildrens == 1){          //printf("x");          if(AST->childrens[0]->node\_type == NON\_TERMINAL){              switch (AST->childrens[0]->Kind.nonTerminalType)              {                  case NON\_TERMINAL\_EXPRESSION:                      //printf("c");                      reg =CodeGenerator\_GenerateExpression(AST->childrens[0], scope);                      break;                  case NON\_TERMINAL\_FUNCTION\_CALL:                      //printf("v");                      reg = CodeGenerator\_GenerateFunctionCall(AST->childrens[0], scope);                      break;                  case NON\_TERMINAL\_LITERAL:                      //printf("b");                      reg = CodeGenerator\_GenerateLiteral(AST->childrens[0], scope);                      break;                }          }          else if(AST->childrens[0]->node\_type == TERMINAL){              //printf("n");              if(AST->childrens[0]->Kind.TerminalType == IDENTIFIER\_TOKEN){                  //printf("m");                  reg = CodeGenerator\_GenerateIdentifier(AST->childrens[0], scope);              }          }      }      if(AST->numberOfChildrens == 2){          reg = CodeGenerator\_GenerateFactor(AST->childrens[1], scope);          if(AST->childrens[0]->Kind.TerminalType == MINUS\_TOKEN){              fprintf(OUTPUT\_FILE, "    NEG %s\n", reg->registerName);          }          if(AST->childrens[0]->Kind.TerminalType == LOGICAL\_NOT\_TOKEN){              trueLabel = CodeGenerator\_Create\_Label();              doneLabel = CodeGenerator\_Create\_Label();              fprintf(OUTPUT\_FILE, "    CMP %s, 0\n", reg->registerName);              fprintf(OUTPUT\_FILE, "    JE %s\n", trueLabel);              fprintf(OUTPUT\_FILE, "    MOV %s, 0\n", reg->registerName);              fprintf(OUTPUT\_FILE, "    JMP %s\n", doneLabel);              fprintf(OUTPUT\_FILE, "%s:\n", trueLabel);              fprintf(OUTPUT\_FILE, "    MOV %s, 1\n", reg->registerName);              fprintf(OUTPUT\_FILE, "%s:\n", doneLabel);          }        }      //printf("try %d", reg->registerKind);        return reg;  }  char CodeGenerator\_PushGeneralRegisters(int\* c) {      // this function pushes all the general-purpose registers (r8 - r15) that are currently in use      char used\_mask = 0; // 8-bit mask representing which registers are in use (r8 = LSB, r15 = MSB)      \*c = 0;        int i;      for(i = 0; i < registersPool.number\_of\_registers; i++) {          if(registersPool.registers[i]->inuse) {              // Set the corresponding bit in the mask              used\_mask |= (1 << i);              // Emit the PUSH instruction for this register              fprintf(OUTPUT\_FILE, "    PUSH %s\n", registersPool.registers[i]->registerName);              // setting the register as unuse, since we saving it in the stack              registersPool.registers[i]->inuse = 0;              // increasing the counter              \*c += 1;            }      }      return used\_mask;  }  void CodeGenerator\_PopGeneralRegisters(char used\_mask) {      // this function pops all the general-purpose registers (r8 - r15) that were previously pushed,      // in the reverse order of pushing (from r15 to r8)      for(int i = 7; i >= 0; i--) {          if((used\_mask >> i) & 1) {              // Retrieve the register name from the pool and emit POP              fprintf(OUTPUT\_FILE, "    POP %s\n", registersPool.registers[i]->registerName);              // setting the register as used              registersPool.registers[i]->inuse = 1;          }      }  }  Register\* CodeGenerator\_GenerateFunctionCall(TreeNode\* AST, Scope\* scope){      // define variables      char pushed\_regs;      int pushed\_c;      char\* func\_name;      Symbol\_Table\_Function\_Entry\* func\_entry;      //SymbolTable\* g\_symbol\_table;      Scope\* tmp;      Register\* reg = NULL;        // getting the name of the function      func\_name = extract\_variable\_name(AST->childrens[0]);        // getting the global symbol table        tmp = scope;      while (tmp->symbol\_table->symbol\_table\_type != GLOBAL\_SYMBOL\_TABLE)      {          tmp = tmp->parent;      }      //g\_symbol\_table = tmp->symbol\_table;        // getting the function entry      func\_entry = CodeGenerator\_FetchFunctionEntry(tmp->symbol\_table, func\_name);      // pushing the registers      pushed\_regs = CodeGenerator\_PushGeneralRegisters(&pushed\_c);      //CodeGenerator\_PushAllRegs();      CodeGenerator\_PushParamRegis();        // checking if count is off      if(pushed\_c % 2 == 1){          // keep the stack 16 bytes alignment          fprintf(OUTPUT\_FILE, "    SUB RSP, 8\n");      }      // checking if there is params      if(AST->numberOfChildrens == 2){          // setting the parameters registers          CodeGenerator\_GenerateFunctionCall\_SetParamsRegisters(AST->childrens[1], scope);      }      fprintf(OUTPUT\_FILE, "    CALL %s\n", func\_name);          if(pushed\_c % 2 == 1){          // keep the stack 16 bytes alignment          fprintf(OUTPUT\_FILE, "    ADD RSP, 8\n");      }        CodeGenerator\_PopParamRegis();      CodeGenerator\_PopGeneralRegisters(pushed\_regs);      // checking if the function return a value      if(func\_entry->return\_type != VARIABLE\_AND\_RETURN\_TYPE\_VOID){          //printf("here10");          // returning the return value of the function          reg = allocate\_register();            fprintf(OUTPUT\_FILE, "    MOV %s, RAX\n", reg->registerName);      }      return reg;  }  char\* CodeGenerator\_GetParameterRegister(int index){      // this function return the regiter that ack as parameter holder for the [index] place        // cheking that the index is lower them the maximun amount of arguments      if(index >= 4){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      // handling cases      switch (index)      {      case 0:          return RCX;      case 1:          return RDX;      case 2:          return R8;      case 3:          return R9;        }      return NULL;  }  void CodeGenerator\_GenerateFunctionCall\_SetParamsRegisters(TreeNode\* AST, Scope\* scope){      // define local variables      int index = 0;      TreeNode\* tmp;      Register\* reg;      do      {          // gettin ghe current argument          tmp = AST->childrens[0];          // evaluating the argument expression          reg = CodeGenerator\_GenerateExpression(tmp->childrens[0], scope);          // getting the register who hold the parameter in the [index] plase          fprintf(OUTPUT\_FILE, "    MOV %s, %s\n", CodeGenerator\_GetParameterRegister(index++), reg->registerName);          // if there is anoter argument, move to it          if(AST->numberOfChildrens == 2){              // iterating to the next argument              AST = AST->childrens[1];          }          free\_register(reg);      }while(AST->numberOfChildrens == 2);    }  Register\* CodeGenerator\_GenerateIdentifier(TreeNode\* AST, Scope\* scope){      //printf("8");      // define varilabes      Register\* reg = NULL;        char\* var\_name;      char\* lowReg1;      char\* lowReg2;      Symbol\_Table\_Variable\_Entry\* var\_entry;      // getting free register      reg = allocate\_register();      // getting the variable name      var\_name = extract\_variable\_name(AST);      // getting the entry of that variable      var\_entry = CodeGenerator\_FetchVariableEntry(scope, var\_name);      // checking if the identifier is from parameter, or localy declared      if(var\_entry->variable\_origin == VARIABLE\_LOCAL){          // THIS IS LOCAL DECLARED VARIABLE, ACESSING VIA STACK          // checking if it is type char or type int          if(var\_entry->variable\_type == VARIABLE\_AND\_RETURN\_TYPE\_INT){              fprintf(OUTPUT\_FILE, "    MOV %s, [RBP %d]\n", reg->registerName, var\_entry->index\_bp\_relative);          }          else if(var\_entry->variable\_type == VARIABLE\_AND\_RETURN\_TYPE\_CHAR){              // in here, the type is char, so we only care about the lowerest 8 bits              // getting the name of the low 8 bits register              CodeGenerator\_GetRegisterLowestByteRegister(reg->registerName);              //fprintf(OUTPUT\_FILE, "    XOR %s, %s\n", reg->registerName, reg->registerName);              //fprintf(OUTPUT\_FILE, "    MOV %s, byte [RBP + %d]\n",              //     CodeGenerator\_GetRegisterLowestByteRegister(reg->registerName), var\_entry->index\_bp\_relative);              fprintf(OUTPUT\_FILE, "    MOVZX %s, BYTE [RBP %d]\n", reg->registerName, var\_entry->index\_bp\_relative);            }      }      else if(var\_entry->variable\_origin == VARIABLE\_PARAM){          // this is variable that the function got as a parameter, acessing it via the resired register          // checking the type of the param          if(var\_entry->variable\_type == VARIABLE\_AND\_RETURN\_TYPE\_INT){              fprintf(OUTPUT\_FILE, "    MOV %s, %s\n", reg->registerName, CodeGenerator\_GetParameterRegister(var\_entry->variable\_position));          }          else if(var\_entry->variable\_type == VARIABLE\_AND\_RETURN\_TYPE\_CHAR){                fprintf(OUTPUT\_FILE, "    XOR %s, %s\n", reg->registerName, reg->registerName);              fprintf(OUTPUT\_FILE, "    MOV %s, %s\n", CodeGenerator\_GetRegisterLowestByteRegister(reg->registerName),              CodeGenerator\_GetRegisterLowestByteRegister(CodeGenerator\_GetParameterRegister(var\_entry->variable\_position)));          }        }      // returnign the register      return reg;  }  Register\* CodeGenerator\_GenerateLiteral(TreeNode\* AST, Scope\* scope){      // define local variables      Register\* reg = NULL ;      int val;      // getting an unused register      reg = allocate\_register();      if(AST->childrens[0]->Kind.TerminalType == DIGIT\_TOKEN){          val = \*(int\*)AST->childrens[0]->value;        }      if(AST->childrens[0]->Kind.TerminalType == CHAR\_LITERAL\_TOKEN){          val = (int)\*(char\*)AST->childrens[0]->value;      }      fprintf(OUTPUT\_FILE, "    MOV %s, %d\n", reg->registerName, val);        // returning the register      return reg;  }  Register\* CodeGenerator\_GenerateGetchar(TreeNode\* AST, Scope\* scope){      // define local variables      Symbol\_Table\_Variable\_Entry\* var\_entry;      char\* var\_name;      Register\* reg;      char regis;      int pushed\_c;      // getting the name of the variable who will hold the goten char      var\_name = extract\_variable\_name(AST->childrens[0]);      // getting the variable entry      var\_entry = CodeGenerator\_FetchVariableEntry(scope, var\_name);      // saving registers before calling putchar function      regis = CodeGenerator\_PushGeneralRegisters(&pushed\_c);      //CodeGenerator\_PushAllRegs();      CodeGenerator\_PushParamRegis();        // checking if count is off      if(pushed\_c % 2 == 1){          // keep the stack 16 bytes alignment          fprintf(OUTPUT\_FILE, "    SUB RSP, 8\n");      }      // getting a free used register      reg = allocate\_register();      //fprintf(OUTPUT\_FILE, "    SUB RSP, 32\n"); // shadow spacing for win64 convention      fprintf(OUTPUT\_FILE, "    CALL getchar\n");      //fprintf(OUTPUT\_FILE, "    ADD RSP, 32\n"); // shadow spacing for win64 convention      //fprintf(OUTPUT\_FILE, "    MOV %s, RAX\n", reg->registerName);      fprintf(OUTPUT\_FILE, "    MOV [RBP %d], %s\n", var\_entry->index\_bp\_relative, RAX);      // checking if count is off      if(pushed\_c % 2 == 1){          // keep the stack 16 bytes alignment          fprintf(OUTPUT\_FILE, "    ADD RSP, 8\n");      }      // saving th registers      CodeGenerator\_PopParamRegis();      CodeGenerator\_PopGeneralRegisters(regis);      // returning the register with the input ascii value      return reg;    }  void CodeGenerator\_GeneratePutchar(TreeNode\* AST, Scope\* scope){      // define loval variables      Register\* reg;      char regis;      int pushed\_c;      // saving registers before calling putchar function      regis = CodeGenerator\_PushGeneralRegisters(&pushed\_c);      //CodeGenerator\_PushAllRegs();      CodeGenerator\_PushParamRegis();        // checking if count is off      if(pushed\_c % 2 == 1){          // keep the stack 16 bytes alignment          fprintf(OUTPUT\_FILE, "    SUB RSP, 8\n");      }      // evaluating the expresstion to print(lowerst 8 bits);      reg = CodeGenerator\_GenerateExpression(AST->childrens[0], scope);      fprintf(OUTPUT\_FILE, "    MOV %s, %s\n",CodeGenerator\_GetParameterRegister(0), reg->registerName);      //fprintf(OUTPUT\_FILE, "    SUB RSP, 32\n"); // shadow spacing for win64 convention      fprintf(OUTPUT\_FILE, "    CALL putchar\n");      //fprintf(OUTPUT\_FILE, "    ADD RSP, 32\n"); // shadow spacing for win64 convention      // checking if count is off      if(pushed\_c % 2 == 1){          // keep the stack 16 bytes alignment          fprintf(OUTPUT\_FILE, "    ADD RSP, 8\n");      }      //fprintf(OUTPUT\_FILE, "    ADD RSP, 64\n");      // freeing the inuse register      free\_register(reg);      // saving th registers      CodeGenerator\_PopParamRegis();      CodeGenerator\_PopGeneralRegisters(regis);  }  void CodeGenerator\_GenerateReturn(TreeNode\* AST, Scope\* scope){      // local variable      Register\* reg;      Symbol\_Table\_Function\_Entry\* func\_entry;        // getting the entry of the current function      func\_entry = scope->functionPrototype;        // checking if in here we not retuning any value(returning void)      if(AST->numberOfChildrens == 0){          fprintf(OUTPUT\_FILE, "    JMP %s\n", func\_entry->function\_return\_label);      }      else{          // evaluating the return expression and getting it in the gotten register          reg =  CodeGenerator\_GenerateExpression(AST->childrens[0], scope);          fprintf(OUTPUT\_FILE, "    MOV RAX, %s\n", reg->registerName);          fprintf(OUTPUT\_FILE, "    JMP %s\n", func\_entry->function\_return\_label);          free\_register(reg);      }  }  void CodeGenerator\_GenerateWhile(TreeNode\* AST, Scope\* scope){      // define local variables      char\* whileLabel = CodeGenerator\_Create\_Label();      char\* doneLabel = CodeGenerator\_Create\_Label();      Register\* reg;      // start of the while loop label      fprintf(OUTPUT\_FILE, "%s:\n", whileLabel);      // evaluate the while expression      reg = CodeGenerator\_GenerateExpression(AST->childrens[0], scope);      // checking if the expression ivaluate true value      fprintf(OUTPUT\_FILE, "    CMP %s, 0\n", reg->registerName);      fprintf(OUTPUT\_FILE, "    JE %s\n", doneLabel);      // free reg      free\_register(reg);      // here is the while scope      CodeGenerator\_GenerateBody(AST->childrens[1], scope);      // jumping to the begginging of the while loop      fprintf(OUTPUT\_FILE, "    JMP %s\n", whileLabel);      // the done label      fprintf(OUTPUT\_FILE, "%s:\n", doneLabel);  }  void CodeGenerator\_PushParamRegis() {      // Emit PUSH for each GP register except RSP      fprintf(OUTPUT\_FILE,"    PUSH %s\n", RCX);      fprintf(OUTPUT\_FILE,"    PUSH %s\n", RDX);      fprintf(OUTPUT\_FILE,"    PUSH %s\n", R8);      fprintf(OUTPUT\_FILE,"    PUSH %s\n", R9);  }  void CodeGenerator\_PopParamRegis() {      // Emit PUSH for each GP register except RSP      fprintf(OUTPUT\_FILE,"    POP %s\n", R9);      fprintf(OUTPUT\_FILE,"    POP %s\n", R8);      fprintf(OUTPUT\_FILE,"    POP %s\n", RDX);      fprintf(OUTPUT\_FILE,"    POP %s\n", RCX);  } |

## ErrorReporting

### ErrorReporting.h

|  |
| --- |
| #pragma once  #include<stdio.h>  #include"../Lexer/Lexer.h"  // define colors for printing  // red for errors  #define Red "\033[0;31m"  // black for normal printing  #define White  "\033[0;37m"  // yelow for warnings  #define Yellow "\033[0;33m"  // purpule for printing types  #define Purple "\033[0;35m"  // cyan for printing file name  #define Cyan "\033[0;36m"  typedef enum ErrorPhase  {      ERROR\_ON\_LEXER\_PHASE,      ERROR\_ON\_PARSER\_PHASE,      ERROR\_ON\_SEMANTIC\_PHASE  }ErrorPhaseType;  // Function declarations  void ErrorReporting\_Switch\_To\_Red();  void ErrorReporting\_Switch\_To\_White();  void ErrorReporting\_Switch\_To\_Yellow();  void ErrroReporting\_Switch\_To\_Purpule();  void ErrorReporting\_PrintSourceCodeLine(int line);  void ErrorReporting\_PrintErrorPhase(ErrorPhaseType type);  void ErrorReporting\_ReportError\_Base(ErrorPhaseType errorPhaseType, int line, int char\_in\_line);  void ErrorReporting\_ReportError(char\*); |

### ErrorReporting.c

|  |
| --- |
| #include"ErrorReporting.h"    void ErrorReporting\_Switch\_To\_Red(){      printf(Red);  }  void ErrorReporting\_Switch\_To\_White(){      printf(White);  }  void ErrorReporting\_Switch\_To\_Yellow(){      printf(Yellow);  }  void ErrroReporting\_Switch\_To\_Purpule(){      printf(Purple);  }  void ErrroReporting\_Switch\_To\_Cyan(){      printf(Cyan);  }  void ErrorReporting\_PrintSourceCodeLine(int line){      // this function prints the wanted line in the source code      // define local variables      char\* line\_p;        // getting the wanted line      line\_p = getSourceCodeLine(line);      // printing the line      fputs(line\_p, stdout);  // using fputs to avoid formating printing, like if the character % appear, etc.  }  void ErrorReporting\_PrintErrorPhase(ErrorPhaseType type){      if(type == ERROR\_ON\_LEXER\_PHASE){          printf("Lexer error:");      }      else if(type == ERROR\_ON\_PARSER\_PHASE){          printf("Parser error:");      }      else if(type == ERROR\_ON\_SEMANTIC\_PHASE){          printf("Semantic error:");      }  }  void ErrorReporting\_ReportError\_Base(ErrorPhaseType errorPhaseType, int line, int char\_in\_line){      // going line down      printf("\n");      // print the phase of error      ErrroReporting\_Switch\_To\_Purpule();      ErrorReporting\_PrintErrorPhase(errorPhaseType);      // printing file name      printf(" in file: ");      // switch to red color      ErrorReporting\_Switch\_To\_Red();      printf("\"");      // switch to blue color      ErrroReporting\_Switch\_To\_Cyan();      //printf("yourfile.txt");      printf("%s", getSourceCodeFileName());      // switch back to red      ErrorReporting\_Switch\_To\_Red();      printf("\"");      // print the base error format      ErrorReporting\_Switch\_To\_Red();      printf(" on line %d:%d: \n", line, char\_in\_line);      // print the source code line      ErrorReporting\_PrintSourceCodeLine(line);      // returning to white color      ErrorReporting\_Switch\_To\_White();  }  void ErrorReporting\_ReportError(char\* errorMsg){      // switch to red color      ErrorReporting\_Switch\_To\_Red();        // outputing the error message      printf(errorMsg);      // switching back to white      ErrorReporting\_Switch\_To\_White();        printf("\n"); // go line down    } |

## Compiler.h

|  |
| --- |
| #include"Parser/Parser.h"  #include"Token/Token.h"  #include"Lexer/Lexer.h"  #include"Semantic/Scope/Scope.h"  #include"Semantic/SymbolTable/SymbolTable.h"  #include"Semantic/Scope/ScopeTree.h"  #include"CodeGenerator/CodeGenerator.h"  #include<stdio.h>  #include<stdlib.h>  #include<windows.h>  #define KERNEL32\_FILE\_PATH "C:\\Program Files (x86)\\Windows Kits\\10\\Lib\\10.0.26100.0\\um\\x64\\kernel32.lib"  #define NASM\_FILE\_PATH "nasm"  #define LINKER\_FILE\_PSTH "link" |

## Compiler.c

|  |
| --- |
| #include"Compiler.h"  /\*  void printTokenTypeNameVersionForWEB(int token) {      // Static array of token names, matching the updated TokenType enum      static const char\* tokenNames[] = {          "identifier",          "(",          ")",          "{",          "}",          //"ESSILON\_TOKEN", // No corresponding name provided, keeping original          ",",          ";",          "getchar",          "putchar",          "int",          "char",          "void",          "=",          "return",          "if",          "else",          //"for",          "while",          "||",          "&&",          "==",          "!=",          "<",          ">",          "<=",          ">=",          "+",          "-",          "\*",          "/",          "%",          "!",          "int\_literal",          "char\_literal",          "$",          "ERROR\_TOKEN", // No corresponding name provided, keeping original          "END\_OF\_TOKENS\_TOKEN" // No corresponding name provided, keeping original      };      // Ensure the token is within the valid range of the enum values      if (token >= 0 && token < sizeof(tokenNames) / sizeof(tokenNames[0])) {          printf("%s ", tokenNames[token]);      } else {          printf("Unknown token\n");      }  }\*/  // Recursive function to print the AST  void printAST(TreeNode \*node, int indent) {      if (node == NULL) return;      // Print indentation dynamically      for (int i = 0; i < indent; i++) {          printf("\_\_"); // Indentation for clarity      }      printf("Node: "); // Indicating a new node      // Print the node value      if (node->node\_type == TERMINAL) {          printf("[TERMINAL] ");          // Debug print before calling printTokenTypeName          printf("Token Type: ");          printTokenTypeName(node->Kind.TerminalType);            // Ensure node->value is not NULL before printing            // Print the terminal value based on its type          if (node->Kind.TerminalType == IDENTIFIER\_TOKEN) {              printf("| Identifier: %s", (char\*)node->value);          }          else if (node->Kind.TerminalType == CHAR\_LITERAL\_TOKEN) {              printf("| Char: '%c'",\*((char\*)(node->value)));          }          else if (node->Kind.TerminalType == DIGIT\_TOKEN) {             printf("| Integer: %d", \*((int\*)(node->value)));          }          printf("\n"); // Newline for clarity      }      else { // Non-Terminal node          printf("[NON-TERMINAL] Type: %d", node->Kind.nonTerminalType);          printf("\n");          // Debug print for number of children          printf("Child Count: %d\n", node->numberOfChildrens);          // Recursively print all children          for (int i = 0; i < node->numberOfChildrens; i++) {              if (node->childrens[i] == NULL) {                  printf("Warning: NULL child at index %d\n", i);              } else {                  printAST(node->childrens[i], indent - 5);              }          }      }  }  /\*\*   \* Converts an input filename to a NASM output filename by replacing or adding the .nasm extension   \*   \* @param input\_filename The source filename   \* @return Newly allocated string with the output filename, or NULL on error   \*         The caller is responsible for freeing this memory   \*/  char\* filename\_convert\_to\_extension(const char \*input\_filename, const char \*new\_extension) {      // Validate parameters      if (input\_filename == NULL || new\_extension == NULL) {          return NULL;      }      // Find the last dot in the filename      const char \*dot\_position = strrchr(input\_filename, '.');      // Calculate the base name length      size\_t base\_len;      if (dot\_position != NULL) {          base\_len = dot\_position - input\_filename;      } else {          base\_len = strlen(input\_filename);      }      // Ensure the new extension starts with a dot      const char \*extension\_with\_dot = new\_extension[0] == '.' ? new\_extension : NULL;      char \*temp\_extension = NULL;      if (!extension\_with\_dot) {          size\_t ext\_len = strlen(new\_extension);          temp\_extension = (char\*)malloc(ext\_len + 2); // +1 for dot, +1 for null          if (temp\_extension == NULL) {              return NULL;          }          temp\_extension[0] = '.';          strcpy(temp\_extension + 1, new\_extension);          extension\_with\_dot = temp\_extension;      }      // Allocate memory for new filename      size\_t new\_len = base\_len + strlen(extension\_with\_dot) + 1;      char \*output\_filename = (char\*)malloc(new\_len);      if (output\_filename == NULL) {          free(temp\_extension);          return NULL;      }      // Copy base and append new extension      strncpy(output\_filename, input\_filename, base\_len);      output\_filename[base\_len] = '\0';      strcat(output\_filename, extension\_with\_dot);      // Clean up temporary string if used      if (temp\_extension) {          free(temp\_extension);      }      return output\_filename;  }  int Compiler\_Check\_Vaild\_FileName(char\* fname){      // checking to see if the file ends up with .txt      return 1;  }  int main(int argc, char\* argv[]){      // checking goten atguments      if(argc != 2){          printf("EXPECTING EXACTLY ONE ARGUMENT");          return 1;      }      // checking if got a valid filename      if(!Compiler\_Check\_Vaild\_FileName(argv[1])){          // here, got an invalid file name          printf("INVALID FILE NAME\n");          return 1;      }        // creating new File object pointer      FILE \*file = fopen(argv[1], "r");      if(file == NULL){          printf("SOURCE CODE PATH DOESN'T EXISTS");          return 1;      }      // ================================================      // first stage, converting the text into tokens      // the lexer holds the file name for the error handling, seting it      SetsourceCodeFileName(argv[1]);      // get the tokens array      Token \*tokenarray = getTokens(file);      // if we were at panic mode in the lexer      if(tokenarray == NULL){          // exiting program          exit(EXIT\_SUCCESS);      }      // ===================================================      // second stage, the parser      Parser\* parser = parser\_init();      // intiating the parser structure      init\_action\_table(parser);      init\_goto\_table(parser);      init\_Production\_Rules(parser);        // creating the Abstact Syntax Tree and checking syntax of the code      TreeNode \* root = create\_abstact\_syntax\_tree(parser, tokenarray);        // freeing the parser and the token array      VirtualFree(parser, 0, MEM\_RELEASE);      free(tokenarray);      // checking if we were at panic mode at the parser      if(root == NULL){          // exiting program          exit(EXIT\_SUCCESS);      }      // ===================================================      // third stage      // creting the scope tree and checking semantic errors      Scope\* main\_scope;      process\_program(root, &main\_scope);        // checking if we were at panic mode at the semantic phase      if(main\_scope == NULL){          // freeing the ast tree          freeAst(root);          // exiting program          exit(EXIT\_SUCCESS);      }        // ============================================      // forth stage      // back-end, generating the code in assembly      // creating the destination code      char\* dest\_file\_name\_nasm;      char\* dest\_file\_name\_obj;      char\* dest\_file\_name\_exe;        // creating the new files names      dest\_file\_name\_nasm = filename\_convert\_to\_extension(argv[1], "nasm");      dest\_file\_name\_obj = filename\_convert\_to\_extension(argv[1], "obj");      dest\_file\_name\_exe = filename\_convert\_to\_extension(argv[1], "exe");      // creating new file(or override if exists)      FILE \*dest\_file = fopen(dest\_file\_name\_nasm, "w");        if(dest\_file == NULL){          printf("COMPILER ERROR");          exit(EXIT\_FAILURE);      }      // generating the nasm code into the destination file      CodeGenerator\_Generate(root, main\_scope, dest\_file);      // closing the file      fclose(dest\_file);        // compiler finished, freeing left over memory      freeAst(root);      freeScopeTree(main\_scope);      // final stage, creating the obj file and linking it with kernel32 lib      char buffer[1000];      // create the obj command      sprintf(buffer, "%s -f win64 \"%s\" -o \"%s\"", NASM\_FILE\_PATH,  dest\_file\_name\_nasm, dest\_file\_name\_obj);      // executing the command      int nasm\_status = system(buffer);      if (nasm\_status != 0) {          printf("Error compiling NASM file\n");          return 1;  // Exit on error      }        // creating the linking command      sprintf(buffer, "%s %s /LIBPATH:\"%s /subsystem:console /machine:x64 /ENTRY:main /out: %s",LINKER\_FILE\_PSTH, dest\_file\_name\_obj, KERNEL32\_FILE\_PATH, dest\_file\_name\_exe);      // executing the command      int gcc\_status = system(buffer);      if (gcc\_status != 0) {         printf("Error linking with microsoft linker\n");         return 1;  // Exit on error        }      // finished!      printf("Executable created successfully!\n");      return 0;  } |