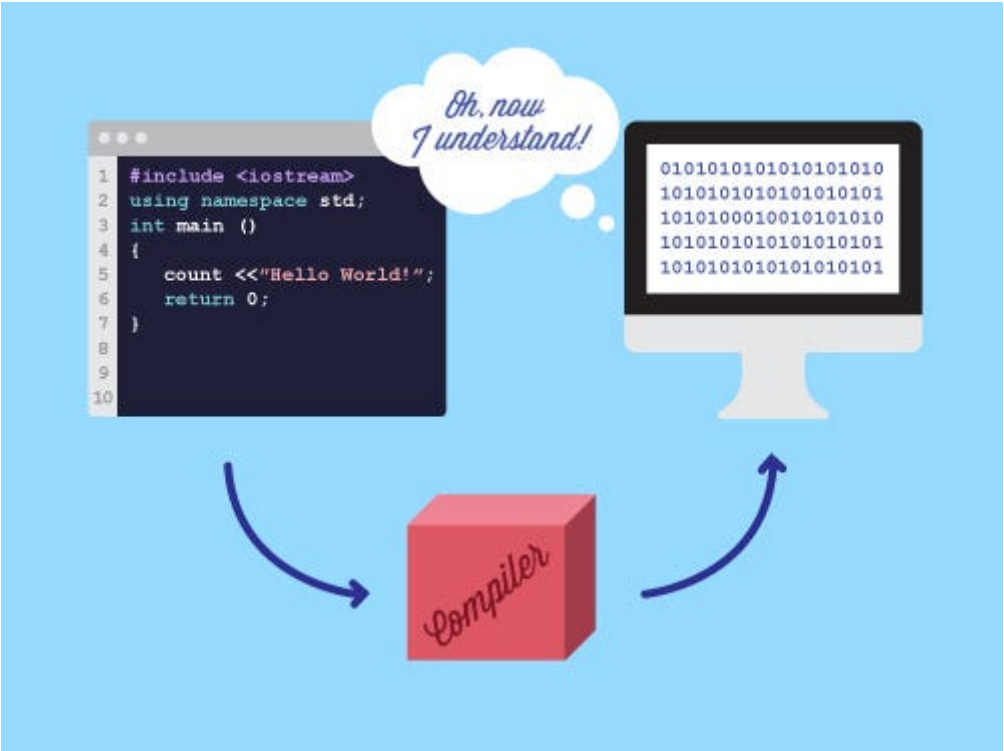
עבודת גמר

לקבלת תואר טכנאי תוכנה



שם הסטודנט: אוריאל שבת

ת.ז הסטודנט: 328106844

שם הפרויקט: קומפיילר Cqual

שמות המנחים: אורי וולטמן ומיכאל צ'רנובילסקי

תוכן

מבוא

השראה:

אני מגיל צעיר מאוד מתעניין בטכנולוגיה וכששמעתי שאפשר לבחור כפרויקט לפתח קומפיילר ידעתי שאני רוצה לעשות את זה. בזמן שסיפרו אילו נושאים לפרויקטים אפשר לעשות אמרו שלבנות קומפיילר זה נושא קשה שמעט עושים גרם לי עוד יותר לרצות לבחור לבנות קומפיילר מכיוון שזה מאתגר וקשה ורציתי אתגר שיעזור לי לפתור בעיות מורכבות ויפתח את היכולת שלי לחפש מידע באינטרנט ואת היכולת שלי לתכנת.

מטרה

ארצה להוציא מהפרויקט הזה את המיטב. הן מבחינת לקיחת אתגר ופיתוח עצמי, והן מבחינת רכישת ידע בתחום שלא
התעסקתי בו בעבר, קומפיילרים.
ארצה להבין לעומק כיצד קומפיילר עובד ואת התאוריה עליה מתבסס.
ארצה לבחון ולפתח את הידע והכישורים שלי בפרויקט בסדר גודל כזה. פיתוח של אלגוריתמים חכמים ויעילים אשר יפתרו
את הבעיות האלגוריתמיות השונות העולות בפרויקט זה, תוך לימוד עצמי של ידע חדש וצבירת ניסיון בנושאים שלא
התעמקתי בהם בעבר, כמו מכונות מצבים, עיצוב שפה, מבנה נתונים ועוד.

תיאור הפרויקט

הפרויקט שלי הינו פיתוח קומפיילר, אשר מתרגם מסמך טקסט המכיל קוד בשפה שאני עיצבתי, Cqual, לקוד אסמבלי x86.
שפת התכנות היא הגדרה של חוקים תחביריים וסמנטיים, שנועדו להגדיר תהליכי חישוב שיבוצעו על ידי המחשב. הגדרת השפה היא חלק בלתי נפרד מבניית הקומפיילר.
שפה Cqual תכיל :

- משתנים וטיפוסים: השפה תומכת במשתנים מגוונים וטיפוסים מורכבים. התמיכה במשתנים מסוגים שונים כמו מספרים שלמים, עשרוניים, מחרוזות ותווים מוסיפה רמות נוספות של גמישות למתכנתים.

- ביצוע מותנה: תנאים כמו if ו- else ושילוב עם לולאות. לדוגמה: אחרי אם הלולאה לא מתקיימת אז יתבצע קוד לפי התנאי.
- ביצוע חוזר לולאות: בשפה קיימים לולאות -for ו- while.
- תת משימות, פונקציות/פעולות כמו פונקציות מתקדמות, שמורכבות מתת-משימות יכולות להפוך את הקוד לקריא יותר, לניהול טוב יותר של המשימות, ולהפוך את הקוד לניתן לתחזוקה.

ספר הפרויקט

ספר הפרויקט הזה הוא תיעוד של כל התהליך שלי שהוא יצירת קומפיילר והוא מסביר במפורט את כל נושא וחלק שלקחו חלק מהיצירה של הקומפיילר.
הספר מכיל בתוכו את הגישות השונות לפתרון הבעיות האלגוריתמיות שבאות בכל שלב בקומפיילר, את קו המחשבה שלי לאורך הפיתוח, מידע על התאוריות של פיתוח קומפיילר ומקורות מידע שהשתמשתי בהן במהלך הפיתוח.

ספר השפה

הקדמה

קצת על השפה Cqual

בזמן שחשבתי על הבסיס של הפרויקט שלי גיליתי שהשם של השפה C# זה בעצם המשך של השם של C++ וב#C יש 4 פלוסים מחוברים מה שיוצר האשטאג, בגלל זה עלה לי הרעיון ליצור שפה מלפני C שהיא מורכבת מ4 מינוסים מה שיוצר את הסימן שווה ובשחיברתי את C עם = יצא C-equal אז קראתי לשפה Cqual.

מה יהיה בספר השפה

בספר השפה תתואר שפת התכנות Cqual. יתוארו אבני השפה, תכולת השפה, ודקדוק השפה.

אבני השפה

משתנים – Variables

משתנה מייצג מקום בזיכרון בו אפשר לשמור ערכים.
מקום זה בזיכרון מיוצג על ידי שם המשתנה (name-Variable), שנקרא גם מזהה (Identifier).

שמות משתנים

1. שם משתנה הוא רצף של אותיות בשפה האנגלית.
2. אין להשתמש במילים שמורות כמזהים.
3. קיימת הבחנה בין אותיות גדולות וקטנות (sensitive Case).

טיפוסי משתנים

לכל משתנה בשפה Cqual יש גם טיפוס)type-Data(אשר מציין את סוג הערכים שהוא יכול להכיל.
ישנם שני סוגים של טיפוסי משתנים:
• int – משתנה מטיפוס מספר שלם.
ס מכיל ערכים מסוג מספרים שלמים. 1, -15, 79 0 וכו'.
• char – משתנה מטיפוס תו.
ס מכיל ערכים מסוג תו. a, g, 0, 7 f וכו'.
הגדרת משתנים
הגדרה כללית של משתנה:

;<Data-type> <Identifier>;
דוגמאות:

;int x
;char c

ביטויים - Expressions & Statements

Expression

יחידה תחבירית בשפת תכנות שניתן להעריכה על מנת לקבוע את ערכה. שילוב של אחד או יותר קבועים, משתנים,
פונקציות, אופרטורים, ו – Expression נוספים, שהשפה מפרשת, ומחשבת
כדי לייצר ("להחזיר") ערך. תהליך זה, עבור ביטויים מתמטיים, נקרא הערכה (Evaluation).
בפשטות, הערך המתקבל הוא בדרך כלל אחד מהסוגים הפרימיטיביים השונים, כמו ערך מספרי, ערך בוליאני וכו'.
דוגמאות ל – Expressions:

- 15 + 3
- 4
- (x-6)/y
- (x + 15) - 3 * (y – 4)

Statement

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

דוגמאות ל – Statements:

- תנאים - if
- לולאות – while,for
- הצהרה על משתנה – int x;
- השמת ערך למשתנה – int x = 4;

אופרטורים – Operators

חשבוניים:

- חיבור - +
- חיסור - –
- כפל - *
- חילוק - /

לוגיים:

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

תכולת השפה

בחלק זה תתואר תכולת השפה ואיך כל חלק בשפה נכתב בצורה נכונה מבחינה דקדוקית.

כל פקודה בשפה Cqual תסתיים עם נקודה פסיק ; למעט תנאים, לולאות בסוף בלוקים ובקריאה לפונקציה. בסוף כל בלוק צריך להוסיף את המילה end.

השמה

כפי שציינתי לעיל משתנה הוא מקום בזיכרון בו אפשר לשמור ערך. השמה מאפשרת לנו לשמור את הערך הרצוי במקום זה בזיכרון.
סימול של השמה מבוצע באמצעות הסימן שווה - =
• הגדרה כללית להשמה:
<Identifier>=<Expression>;
על מנת שההשמה תהיה חוקית, טיפוס המשתנה אליו עושים את ההשמה, כלומר ה -<type-Data> של ה - < Identifier> צריך לתאום לטיפוס הערך המושם, כלומר ה - <type-Data> של ה - <Expression>.

• דוגמאות:
int num = 17;
char ch="h";

תנאים ולולאות

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

תנאים – Conditions

לתנאי יכולים להיות שני חלקים:
• if

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

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

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

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

• דוגמא להגדרת תנאי בעזרת שימוש ב – if:

```
if <condition> then <statement>;end
```

לולאות – Loops

לולאה דומה מאוד במבנה שלה לתנאי, if, אך ההבדל היחיד הוא שחלק הקוד שבתוך הלולאה מתבצע כל עוד התנאי תקף

)כל עוד הביטוי נותן תוצאת אמת(, ולאו דווקא 0 או 1 פעמים. כלומר לולאה יכולה להתבצע מספר רב של פעמים.

- while
- for

דוגמא כללית להגדרת לולאה בעזרת שימוש ב - while:

```
while (<condition> ){
<statements>end
}end
```

דוגמא כללית להגדרת לולאה בעזרת שימוש ב - for:

```
for(<assignment>;<condition>;<expression>){
<statements>end
}end
```

דקדוק השפה

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

מהי שפה?

שפה היא אוסף המשפטים שמצייתים לחוקים המוגדרים בתחביר של השפה. משפטים אלו מורכבים ממילים / אסימונים (Tokens)המוגדרים בשפה.

תחביר השפה

תחביר השפה Cqual, כמו רוב שפות התכנות, הוא תחביר חופשי הקשר (grammar free Context).

הדקדוק מורכב מ – Terminals ו – Terminals-Non. הסימנים (Terminals) הם המילים (Tokens) שנקלטו כקלט מקטע הקוד, בעוד שהמשתנים (Terminals-Non) הם רצפי סימנים ומשתנים.

תחביר השפה מוגדר באמצעות שילוב הסימנים והמשתנים, בכללים שנקראים כללי יצירה (rules Production). כללי היצירה בעצם מגדירים את המשתנים, באמצעות הסימנים המוגדרים בשפה ומשתנים אחרים.

Tokens

להלן האסימונים, ה – Tokens של השפה Cqual:

```
EPSILON, // ε
ID,
LPAREN, // (
RPAREN, // )
LBRACE, // {
RBRACE, // }
COMMA, // ,
IF,
INT,
Char,
WHILE,
FOR,
RETURN,
DOUBLE,
THEN,
PRINT,
INPUT,
NUMBER,
CHAR,
NEQ, // !=
GT, // >
LT, // <
GTE, // >=
LTE, // <=
EQ, // ==
ASSIGN, // =
PLUS, // +
MINUS, // -
STAR, // *
SLASH, // /
```

SEMICOLON, // ;
END_OF_INPUT // $

BNF

Form Naur-Backus היא צורת כתיבה פורמלית (Notation) עבור תיאור Grammars של שפות נטולות הקשר (Context languages free). צורת כתיבה זו משמשת לעיתים קרובות לתיאור שפות תכנות (שהן לרוב שפות נטולות הקשר).
BNF עוזר לכתוב בצורה חד-חד משמעית את כללי ה – Grammar של שפה מסוימת, באופן יחסית קל וקריא.

להלן ה – BNF של השפה Cqual:

S'-> statement_list
S'-> ε

statement_list -> statement statement_list
statement_list -> ε

function_declaration -> type_specifier ID (formal_parameters) block
formal_parameters -> type_specifier ID formal_parameters_tail
formal_parameters -> ε
formal_parameters_tail -> , type_specifier ID formal_parameters_tail
formal_parameters_tail -> ε
parameters -> expression parameters_tail
parameters_tail -> , expression parameters_tail
parameters_tail -> ε

block -> { statement_list }

statement -> function_declaration
statement -> declaration
statement -> assignment

```
statement -> if_statement
statement -> for_statement
statement -> while_statement
statement -> function_call
statement -> return_statement
statement -> expression_statement
statement -> Print
statement -> Input

declaration -> type_specifier ID ;
declaration -> type_specifier ID = expression ;

assignment -> ID = expression ;
function_call -> ID ( parameters )

if_statement -> IF condition THEN statement

for_statement -> FOR ( assignment ; condition ; expression ) block

while_statement -> WHILE ( condition ) block

return_statement -> RETURN expression ;

expression -> expression + term
expression -> expression - term
expression -> term

Print -> print ( type_specifier , expression ) ;

Input -> input ( type_specifier , ID ) ;

term -> term * factor
term -> term / factor
term -> factor
```

factor -> ID
factor -> (expression)
factor -> factor_base
factor_base -> number
factor_base -> string

type_specifier -> int
type_specifier -> String
type_specifier -> double

condition -> expression == expression
condition -> expression != expression
condition -> expression > expression
condition -> expression < expression
condition -> expression >= expression
condition -> expression <= expression

expression_statement -> expression ;

דוגמא לתוכנית בשפת Cqual:

```
int main(int q, int z end) {
    int a=6;
    int w=7;
    while (w > 5 ){
    w=w-1;
    if a < 4 then z=9;end
    }
    for( x=6;;x>5;x+2){
    char t="t";end
    }end

}
int sum(int x, int y end) {
    int s=6;
    int ssh=22;end
}end
```

רקע תיאורטי בתחום הפרויקט:

מהו קומפיילר:

קומפיילר הוא תוכנת מחשב המתרגמת משפת מחשב אחת לשפת מחשב אחרת. לרוב משפה עילית לשפת מכונה. קומפיילר נוצר בשל הקושי לכתוב ולהבין קוד בשפת מכונה לכן יצרו שפות תכנות יותר מובנות וברורות ובשביל שהן יעבדו היה צריך להאמיר אותן לשפת מכונה וכך נוצר הקומפיילר. קומפיילר הוא תוכנית מורכבת ומסובכת רוב הקומפיילרים הנמצאים כיום בשימוש מעבירים את הקלט דרך מספר שלבים: השלב הראשון הוא ניתוח - lexical analysis בשלב הזה הקומפיילר עובר על הקלט ומשיך אותו ליחידות זיהוי. השלב השני תצורה - syntax analysis לוקח את היחידות האלה ובודק אם הדקדוק של הקלט נכון לפי שפת התכנות. השלב השלישי הוא פיזור - semantic analysis בודק אם הקוד נכון מבחינה סמנטית. השלב הרביעי הוא יצירת קוד זמני - intermediate code generation השלב הזה יוצר קוד זמני שכבר יכול להפוך לשפת מכונה. השלב החמישי הוא יעילות - optimization כאן הקומפיילר הופך את הקוד ליעיל יותר ומהיר יותר. השלב השישי והאחרון הוא המרת קוד - code generation כאן הקומפיילר לוקח את הקוד היעיל ומייצר את הקוד בשפת מכונה שהמחשב יכול להריץ.

תהליכים עיקריים בפרויקט:

התהליכים העיקריים בפרויקט שלי יהיו יצירת ששת השלבים של הקומפיילר ולחבר אותם ביחד:

1. Lexer
2. Parser
3. Semantic analysis
4. Intermediate code generation
5. Error Handling
6. Code generation

מבנה וארכיטקטורת הפרויקט

ניתוח מילולי (Lexical analysis)

ניתוח מילולי הוא השלב הראשון בקומפיילר והוא רכיב שמתפקד לקריאת קוד מקור ולזיהוי טוקנים. ה-lexer מקבל מחרוזת המייצגת את הקוד בשפת תכנות כלשהי וממיר את המחרוזת לטוקנים. כל טוקן מייצג יחיד משמעותי בשפת התכנות, והוא מיוצג על ידי סוג (קטגוריה) וערך (תוכן). הסוג מציין את סוג המילה או הקטע בשפה (למשל: מזהה, מספר, אופרטור), והערך מציין את התוכן המדויק של הטוקן. ה-lexer מבצע את ההמרה ממחרוזת לטוקנים באמצעות אוטומט סופי (DFA Deterministic Finite Automaton).

ניתוח תחבירי (Parsing)

ניתוח תחבירי (Syntax Analysis) הוא השלב בתהליך ההמרה שבו נבדקת ונקבעת המבנה התחבירי של קוד המקור של שפת התכנות. במילים פשוטות, ניתוח תחבירי מתרגם את קוד המקור מסדר התווים שלו למבנה תחבירי המגיע לידי ביטוי באמצעות עץ תחביר (AST abstract syntax tree).

ניתוח סמנטי (Semantic analysis)

ניתוח סמנטי (Semantic Analysis) הוא השלב בתהליך ההמרה שמתרגם את הקוד מתוך המבנה התחבירי שלו למשמעות פנימית של התכנית. במילים פשוטות, ניתוח סמנטי נוגע להבנה של המשמעות האמיתית של הקוד, ולוודא שהתכנית תואמת למשמעות של השפה התכנותית.

תרגום לשפת ביניים(Generation Code Representation (Intermediate

תרגום לשפת ביניים (Intermediate Code Generation) הוא השלב בתהליך הקומפילציה בו נפיק קוד ביניים שמשמש כפלט לשלב הבא בקומפיילר. קוד ביניים זה מהווה גרסה פשוטה ותואמת יותר של הקוד המקורי, והוא מהווה גשר בין שפת התכנות המקורית לבין השפה התוכנית היעד.

אופטימיזציה (Optimization)

אופטימיזציה בקומפיילר היא התהליך שבו נעשה מאמץ לשפר את ביצועי הקוד המקורי, תוך שמירה על התוצאה הלוגית השקולה של התכנית. המטרה היא ליצור קוד יעיל יותר, המפרט את הפעולות בצורה יעילה יותר ובעל תוצאות דומות או זהות לתוצאות הקוד המקורי. אופטימיזציה בקומפיילר יכולה לכלול יתרונות בביצועים, אפשרויות ניתוח, ושיפורים כלליים לקוד. היא ניתנת לביצוע במגוון שלבים בתהליך הקומפילציה, או גם במהלך ההרצה של התוכנית. כאשר אופטימיזציה מתבצעת בשלב מאוחר יותר, נקראת אופטימיזציה בזמן ריצה (Run-time Optimization).

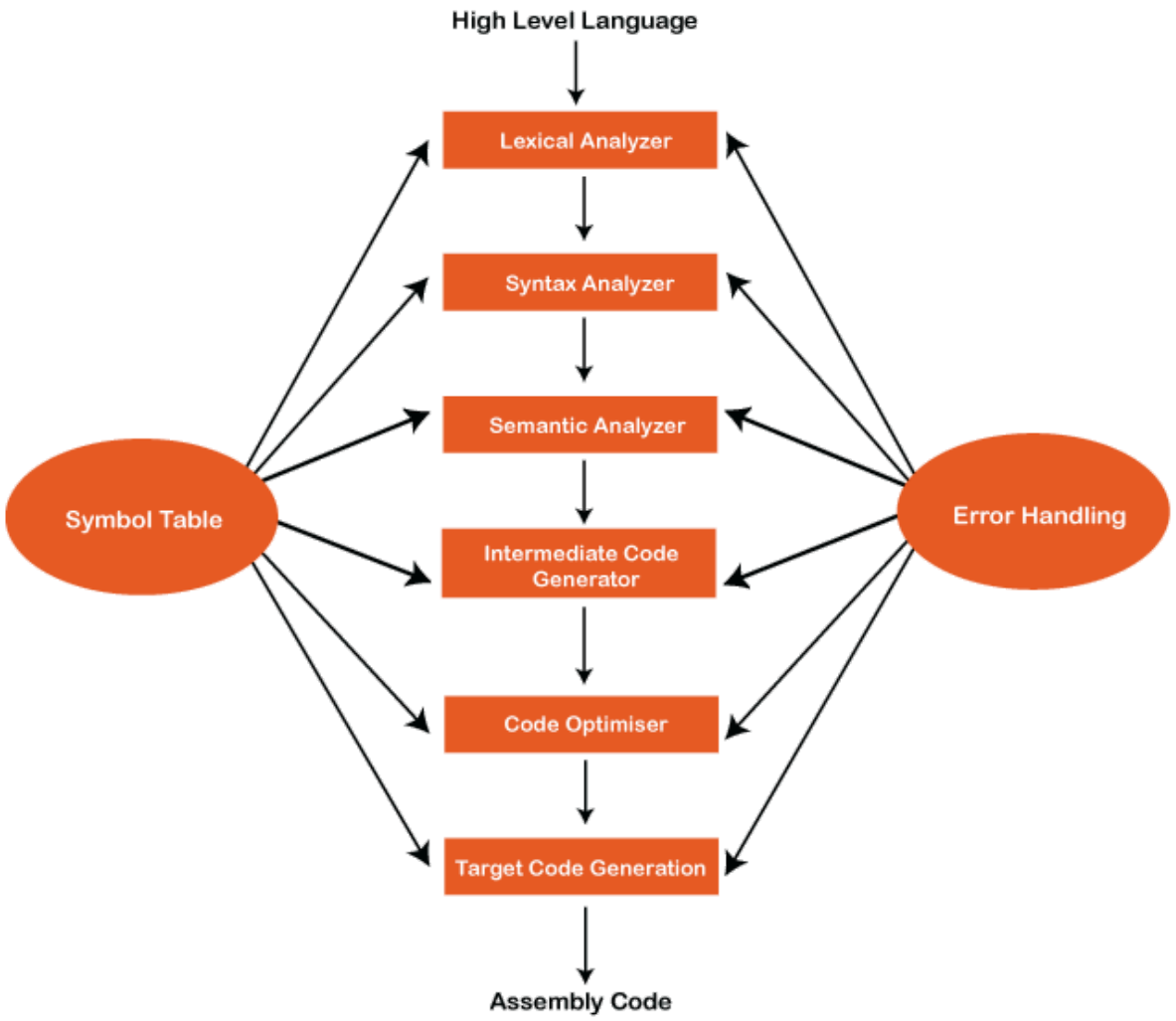
יצירת קוד היעד (Code Generation)

יצירת קוד היעד (Code Generation) בקומפיילר היא השלב בתהליך הקומפילציה שבו הקוד המקורי מתורגם לשפת התכנות אחרת, שנקראת שפת היעד. תהליך זה כולל את הפיכת המבנה התחבירי של הקוד למבנה סמנטי ביניים, ובסופו תרגום לקוד בשפת היעד.

התמודדות עם שגיאות (Handling Error)

תהליך התמודדות עם שגיאות בקומפיילר מהווה חלק חשוב בתהליך הקומפילציה שמטרתו לזהות ולדווח על שגיאות בקוד המקורי. שגיאות אלו יכולות להיות שפתיות, סמנטיות, או שגיאות בתהליכי הקומפילציה עצמן. התמודדות עם שגיאות היא שלב בלתי נפרד בתהליך הפיתוח שמבטיח שהקוד המקורי יהיה תקין ופועל כצפוי.

תרשים Design Level down-Top



הסבר מילולי של התרשים

1. ניתוח מילולי (Lexical analysis):

תפקיד- המזהה יחידים (טוקנים) בקוד המקור וממירם לסדרה כללית של סימנים.

קלט: קוד מקורי בשפה תכנות.

פלט: סדרת טוקנים.

2. ניתוח תחבירי (Parsing):

תפקיד: בודק את היחסים התחביריים בין הטוקנים, בונה מבנה תחבירי כגון עץ תחביר.

קלט: סדרת טוקנים.

פלט: מבנה תחביר (כמו עץ תחביר).

3. ניתוח סמנטי (Semantic analysis):

תפקיד: בודק את תקינות המשמעות של הקוד, כמו התאמת סוגים, בדיקת תקינות סמנטית.

קלט: מבנה תחביר.

פלט: קוד סמנטי (Intermediate Code).

4. תרגום לשפת ביניים (Generation Code Representation Intermediate):

תפקיד: השלב של תרגום לשפת ביניים בקומפיילר הוא ליצור קוד בשפת ביניים או מבנה נתון המשמש כגשר בין הקוד המקורי בשפת התכנות המקורית לבין הקוד המכונה הסופי או קוד בשפת תכנות יעד.

קלט: המבנה התחבירי או הקוד המקורי של השפה המקורית, שכבר עבר ניתוח תחבירי וסמנטי.

פלט: קוד בשפת ביניים.

5. אופטימיזציה (Optimization):

תפקיד: משפר את ביצועי הקוד, בדרך כלל על ידי הכללת שיפורים בקוד ביני.

קלט: קוד ביני.

פלט: קוד ביני משופר.

6. יצירת קוד היעד (Code Generation):

תפקיד: יוצר קוד ייעודי למכונה או לסביבה מסוימת מתוך קוד ביני.

קלט: קוד ביני משופר.

פלט: קוד מכונה או קוד בשפת תכנות יעד.

הגדרת הבעיה האלגוריתמית

בפרויקט קיימות כמה בעיות אלגוריתמיות שונות המחולקות לחלקים שונים בקומפיילר.

הבעיה האלגוריתמית הראשונה:

היא בחלק הראשון של הקומפיילר והיא לקלוט מקובץ תווים ולשייך אותם לסוגים שונים של אסימונים כמו משתנים, מילים שמורות, מספרים ועוד. כאן הקומפיילר צריך "להבין" כמה תווים מהקלט נחשבים לאסימונים אחת ולאן היחידה שייכת.

הבעיה האלגוריתמית השנייה:

היא בחלק השני של הקומפיילר והיא: אחרי השלב הראשון ששייך את הקלט ליחידות זיהוי צריך לעבור על היחידות ולראות אם הן מתאימות לדקדוק של השפה שאני הולך להמציא.

הבעיה האלגוריתמית השלישית:

היא בחלק השלישי של הקומפיילר היא: לבדוק האם הביטוי שהתקבל הגיוני לדוגמה: מחרוזת ועוד נכון או לא נכון זה ביטוי לא הגיוני. אז האלגוריתם צריך לבדוק את זה.

הבעיה האלגוריתמית הרביעית:

היא בחלק הרביעי של הקומפיילר היא: להמיר את הקוד התקין מהשפה המומצאת לשפה מכונה כך שהמכונה תוכל להריץ את הקוד.

הבעיה האלגוריתמית החמישית:

הקומפיילר חייב לטפל בשגיאות בקוד המקור ולדווח עליהן למתכנת בצורה מובנת.

הבעיה האלגוריתמית השישית:

היא בחלק השישי של הקומפיילר היא: לייצר את הקוד הגמור לשפת מכונה.

סקירת אלגוריתמים בתחום הבעיה

תהליך הניתוח התחבירי, ה – Parsing, הוא התהליך המשמעותי והמורכב ביותר מבחינה אלגוריתמית ורעיונית בתהליך הקומפילציה. כעת אציג שיטות שונות ואלגוריתמים שונים הנפוצים בשלב זה.

מונחים

כמה מונחים שאשתמש בהם בתיאור האלגוריתמים .

Derivation

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

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

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

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

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

Left-most Derivation

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

Right-most Derivation

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

דוגמה:

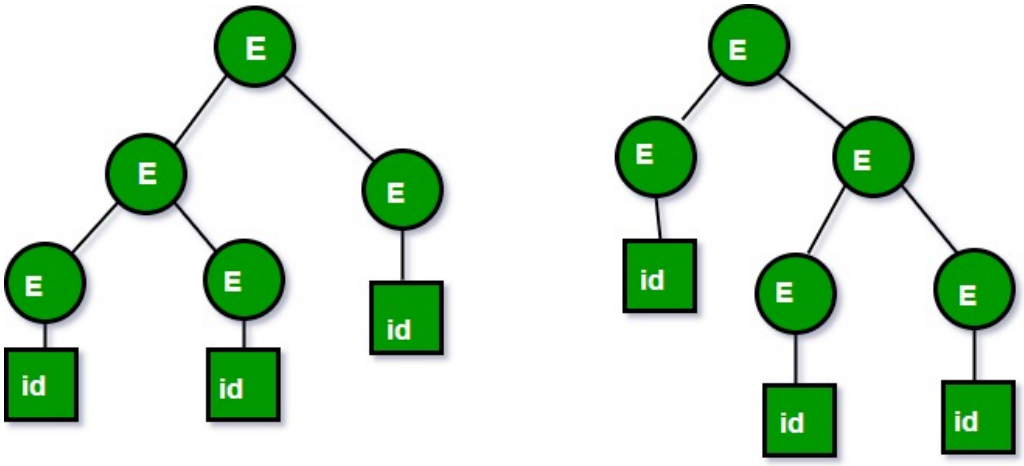
נתון ה-grammar:

E-> E + E

E -> E * E

E -> id

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



Left Factoring

אם יותר מ – rule Production אחד מתחיל באותה קידומת, אז ה – Parser לא יכול לבצע הכרעה באיזה מהחוקים הוא צריך לבחור בשביל לנתח את הקלט הנוכחי.

דוגמה

אם כלל ייצור מסוים נראה כך:

המנתח לא יודע להחליט אחרי איזה חוק לעקוב, כיוון ששני החוקים מתחילים באותו Terminal (או terminal-Non).

על מנת להסיר בעיה זאת משתמשים בטכניקה שנקראת factoring Left.

factoring Left ממירה את ה – Grammar כך שלא יהיו חוסר הוודאויות האלו. היא עובדת כך שעבור כל קידומת

שמשומשת יותר מפעם אחת יוצרים כלל חדש וההמשך של הכלל הישן משורשר לכלל החדש.

דוגמא

הכלל הקודם יוכל כעת להיראות כך:

S -> S S + | S S * | a

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

S -> S S S' | a

S' -> + | *

Parsing Algorithms

על מנת ליצור Tree Parse עליו יתבסס תהליך הקומפילציה, ישנם כמה אלגוריתמים הנקראים Algorithms Parsing.

אלגוריתמים אלה מתחלקים לשני סוגים עיקריים.

1. Top Down Parsing (TDP)
2. Bottom Up Parsing (BUP)

שפות תכנות הן בדרך כלל languages free-Context. נהוג לפרש CFL באמצעות מכונות מצבים, ובאופן יותר ספציפי

מכונות מצבים המשתמשות במחסנית (machines Pushdown). לכן האלגוריתמים שכעת אציג ישתמשו באוטומט מחסנית

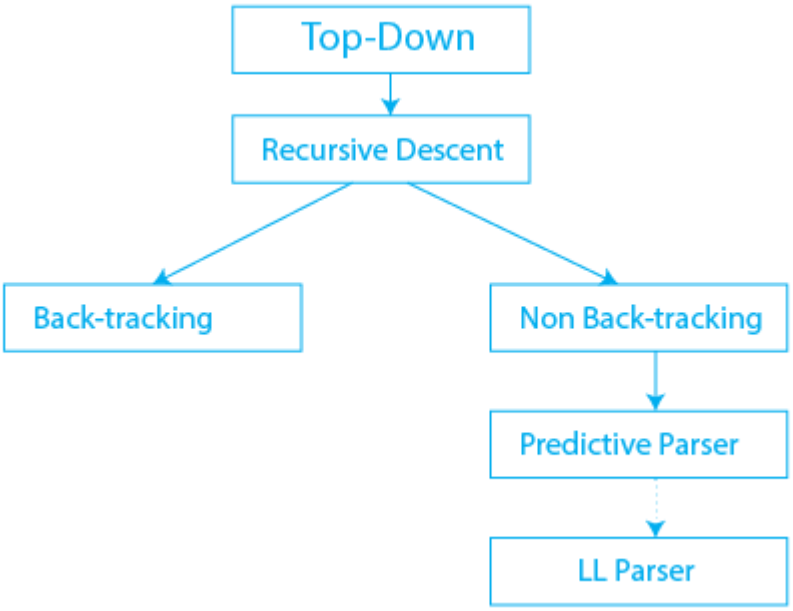
לרוב, על מנת לבצע את פעולת ה – Parsing.

Top Down Parsing (TDP)

טכניקה בה עוברים מהחלקים העליונים לחלקים התחתונים של העץ התחבירי, על ידי שימוש בכללי השכתוב של

Grammar השפה. עוברים מה – Grammar לקלט.

דיאגרמה המתארת מספר סוגים של Parsing Down Top:



Recursive Descent Parsing

ניתוח יורד רקורסיבי הוא טכניקת ניתוח תחבירי לבניית פרסר המפרק ומבין את מבנה הקוד בצורה רקורסיבית. בתהליך זה, כל כלל דקדוקי מיוצג בפונקציה נפרדת המנסה לזהות את המבנה המתאים במחרוזת הקלט. אם כלל מסוים מתאים לקלט, הוא חוזר עם אישור והמשתמש מתקדם לנתח את שאר הקלט; אחרת, הוא נכשל ומאפשר להמשיך לתהליך ניתוח אלטרנטיבי. ניתוח יורד רקורסיבי נפוץ במיוחד לשפות עם דקדוק פשוט יחסית כמו שפות תכנות מסוימות, ומאפשר פרסור אינטואיטיבי יחסית ליישום. הוא דורש לעיתים התאמות בדקדוק כדי להתמודד עם מצבים כמו שאיבת שמאל (Left Recursion) שיכולים לגרום לתקיעות בלולאה אינסופית, ולכן מצריך תכנון קפדני.

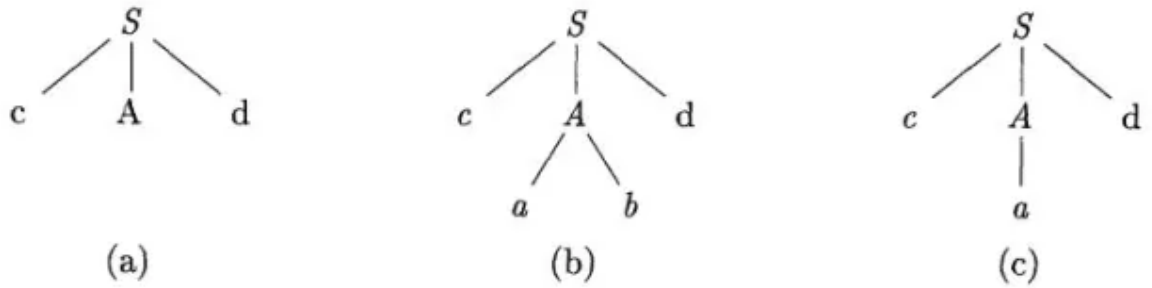
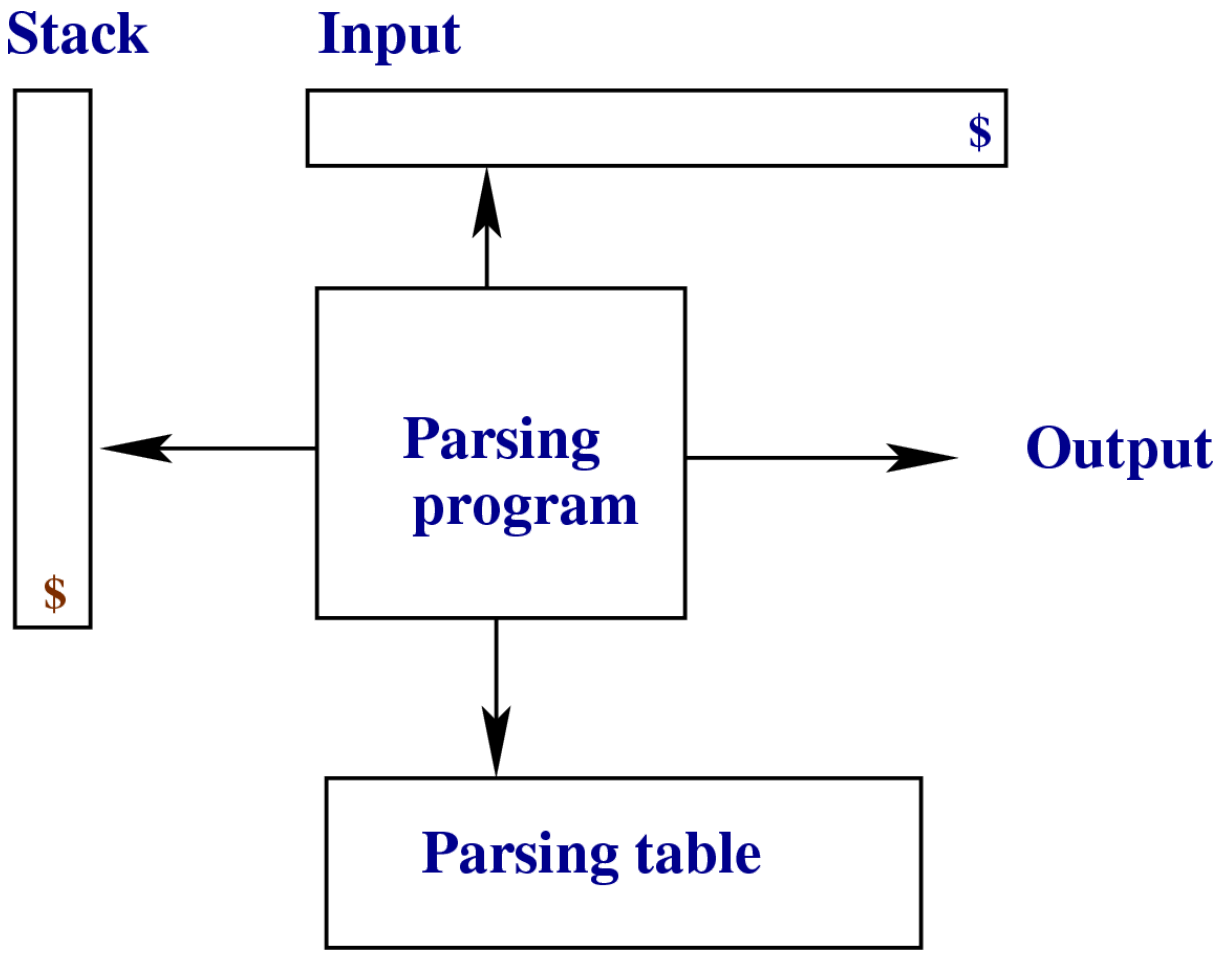


Figure 4.14: Steps in a top-down parse

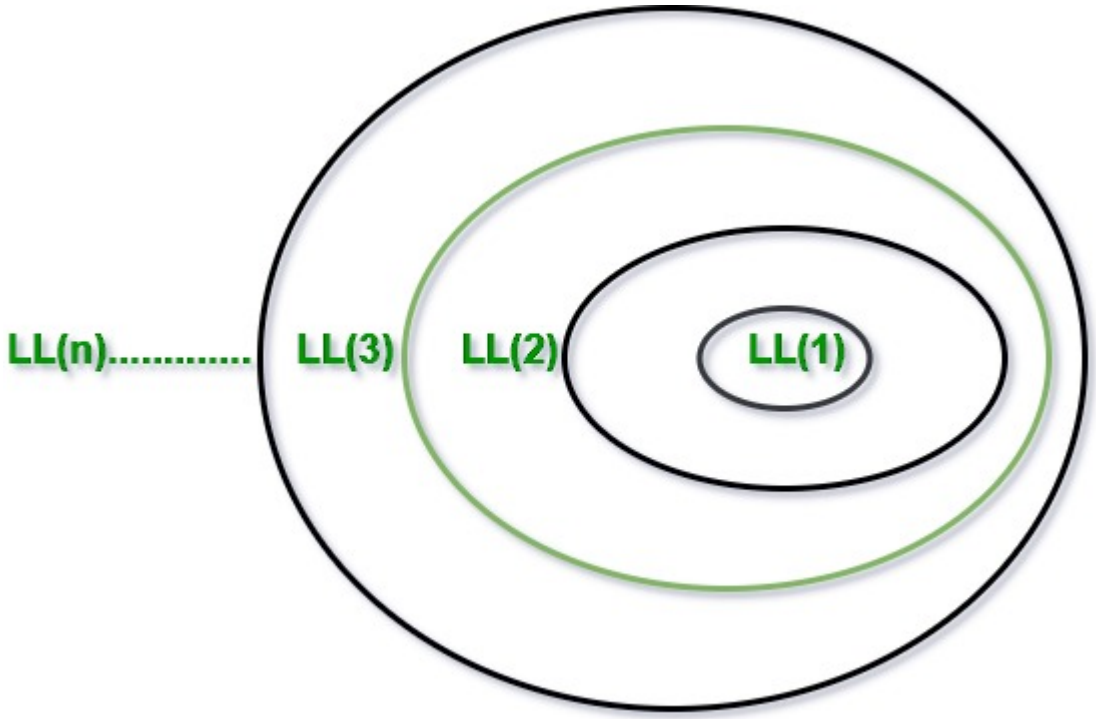
Predictive Parsing

הוא גישה לניתוח תחבירי המיועדת להתמודד עם דקדוקים ללא שמאל שמאל (Left Recursion).
הוא משתמש במבנה נתונים המכונה טבלת ניתוח, המסייעת בקבלת החלטות במהלך הניתוח.
הטבלה הזו מבוססת על כללי הדקדוק של השפה ומאפשרת לפרסר לקבל החלטות מבוססות על סמל הקלט הנוכחי. יתרון מרכזי של ניתוח זה הוא הביצועים המשופרים שלו בהשוואה לשיטות ניתוח רקורסיביות, בכך שהוא נמנע מקריאות פונקציה רקורסיביות ומיישם גישה ישירה יותר. עם זאת, כדי להשתמש ב-Predictive Parsing, יש להמיר את הדקדוק לפורמט נטול שאיבה שמאלית ולוודא שהוא מתאים לניתוח באמצעות תחזיות מקדימות של סמל אחד או יותר (LL(k)).



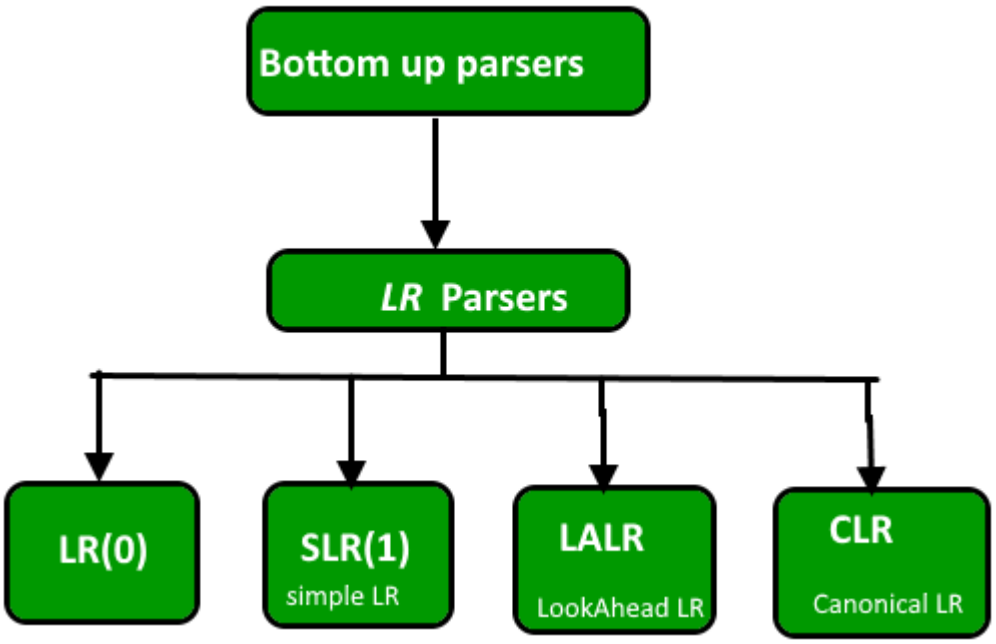
LL(k) parser

מנתח LL(k) הוא סוג של מנתח תחבירי המתאים לניתוח דקדוקים חסרי הקשר. המונח "LL" מתייחס לסריקה משמאל לימין (Left-to-right) והפקת נגזרות שמאליות (Leftmost derivation), ו-"k" מציין את מספר סימני הקלט הנצפים מראש (Lookahead). מנתח זה משתמש במבנה נתונים, כמו טבלאות ניתוח תחביריות, כדי לקבוע איזה כלל דקדוקי ליישם על סמך k הסימנים הבאים בתור בקלט. היתרון של מנתח LL(k) הוא שהוא מספק דרך שיטתית ויעילה לבנות עץ תחבירי מהקלט, במיוחד בשפות תכנות בעלות דקדוק פשוט יחסית או כזה שניתן לשינוי כדי להתאים לניתוח מקדים של k סימנים. עם זאת, במקרים של דקדוקים מורכבים יותר, ייתכן שיידרשו ערכי k גבוהים מאוד, מה שמגביל את היעילות והשימושיות של מנתח LL(k).



Bottom Up Parsing (BUP)

טכניקה בה עוברים מהחלקים התחתונים לחלקים העליונים של העץ התחבירי, על ידי שימוש בכללי השכתוב של Grammar השפה. עוברים מהקלט אל ה - Grammar.
דיאגרמה המתארת מספר סוגים של Parsing Up Bottom:



LR Parser

מנתח LR הוא סוג של מנתח תחבירי חזק לניתוח דקדוקים חסרי הקשר, שבו המונח "LR" מתייחס לסריקה משמאל לימין (Left-to-right) והפקת נגזרות ימניות (Rightmost derivation). מנתחי LR ידועים ביכולתם להתמודד עם מגוון רחב של דקדוקים חסרי הקשר וביעילותם בניתוח תחבירי של שפות תכנות מורכבות. הם משתמשים בטבלת ניתוח וגרף מצב כדי לקבוע את הצעדים שיש לנקוט במהלך הניתוח. הגרסאות הנפוצות ביותר כוללות את LR(0), SLR (Simple LR), LALR (Look-Ahead LR)), ו-LR מלא. כל אחת מהגרסאות הללו שונה במורכבות וביכולת שלה לנתח דקדוקים שונים, אך כולן מתמקדות ביכולת לזהות דקדוקים המאפשרים הפקת נגזרות ימניות בצורה יעילה.

Precedence Parser

מנתח תחבירי לפי קדימות, או Precedence Parser, הוא סוג של מנתח שמבוסס על עקרון הקדימות כדי לקבוע את סדר הערכת הביטויים. זהו כלי חשוב במיוחד עבור שפות תכנות שבהן יש חשיבות רבה לסדר הפעולות, כמו חישובים מתמטיים מורכבים או ביטויים לוגיים. בפרסר זה, לכל אופרטור מוגדרת רמת קדימות שונה, המשמשת לקביעת סדר ההערכה של הביטוי. כמו כן, ניתנת חשיבות גם לאסוציאטיביות, שמאפשרת לקבוע את כיוון ההערכה, מה שמקל על הבנת התוצאה. מנתח לפי קדימות משתמש במבני נתונים, כמו טבלאות קדימות, כדי לקבל החלטות לגבי הפקת עץ תחבירי שמתאים לביטויים המורכבים מהאופרטורים ומהאופרנדים בקלט.

CYK Parser

מנתח CYK (Cocke-Younger-Kasami) הוא אלגוריתם לניתוח תחבירי של שפות חסרות הקשר, שמתבסס על טכניקת דינמית רב-ממדית. האלגוריתם משתמש בטבלה דו-ממדית, בה כל תא מייצג קבוצה של משתנים המסוגלים לייצר תת-רצף מהקלט. מנתח CYK יעיל במיוחד כאשר הדקדוק מוצג בצורת נורמלית חומסקי (CNF), שבה כל כלל דקדוקי מורכב משני משתנים בלבד או מסימן טרמינלי אחד. האלגוריתם פועל בשלושה שלבים: הוספת טרמינלים, חיבור זוגות של טבלאות שנוצרו קודם, ולבסוף בדיקת יצירת המחרוזת המלאה מהסימן ההתחלתי של הדקדוק. זהו אחד הכלים החזקים לניתוח דקדוקים מורכבים ומאפשר הערכה מדויקת של הקלטים בזמן פולינומיאלי.

מבנה נתונים

מבנה נתונים ב-lexer:

במסגרת תהליך הניתוח המילולי (Lexical Analysis) בקומפיילר, נשתמש באוטומט סופי (Finite Automaton) הנקרא "אוטומט מצב" (State Machine) או "אוטומט סופי עם פעולות" (Finite Automaton with Actions). המטרה של האוטומט הסופי בניתוח מילולי היא לזהות ולסווג טוקנים בקוד המקור. אוטומט סופי בניתוח מילולי מכיל שלבים הנקראים "מצבים", והקשרים ביניהם מסומנים באמצעות מעברים עם חץ. כל מצב מייצג את המצב הנוכחי של האוטומט, והחץ מציין את המעבר ממצב נוכחי למצב נוסף בהתאם לקלט שהוא מקבל.
אני הולך לממש את האוטומט הסופי בעזרת מטריצת מצבים.

מטריצת מצבים: היא מבנה נתונים המתאר את המעבר בין המצבים (states) של אוטומט סופי. כל מקום במטריצה מייצג את המעבר בין זוג מצבים נתונים. המטריצה עשויה להכיל מידע על המצב הבא שבו ימצא האוטומט לאחר קליטת סימבול מסוים במצב נתון. לכל שורה במטריצה מתאים מצב יחיד, ולכל עמודה מתאימה סימבול או קבוצת סימבולים אפשריים. על פי התא שנמצא בשורה ובעמודה ניתן לראות לאיזה מצב האוטומט ימצא את עצמו אם יקליט סימבול מסוים.

Input/ state	STATE(0)	STATE(1)	STATE(2)	STATE(3)	STATE(4)	STATE(5)	STATE(6)
Default	0	1	2	3	0	0	0
0-9	2	2	3	3	0	0	0
.	0	0	2	0	0	0	0
"	1	1	1	1	1	1	1
Operator	4	0	0	0	0	0	0
Separator	5	0	0	0	0	0	0
;	6	0	0	0	0	0	0

בשלב הניתוח המילולי נשתמש באוטומט סופי מכיוון שמבנה הנתונים הזה מאפשר קביעה של המצב בו צריך לעבור בשביל להמיר את הטוקנים ביעילות של(O(n.

מבנה נתונים ב-parser

יש הרבה דרכים לממש ניתוח תחבירי, אני בחרתי לממש בקומפילר שלי Parser Reduce-Shift.
בניתוח תחבירי, הקומפיילר צריך לקרוא את הקוד המקור ולהבין את מבנה הדקדוק של השפה.
בשלב זה, הקומפיילר יוצר עץ תחביר, או טבלת תחביר (Parse Table) המסבירים את המבנה התחבירי של הקוד.

עץ תחביר (Parse Tree): הוא מבנה נתונים המייצג את התצורה הבנויה של ביטוי או פקודה בשפת תכנות. בהקשר של בניית קומפיילר, עץ התחביר משמש להצגת היררכיה התחבירית של קוד מקור.

טבלת התחביר: טבלת התחביר בניתוח תחבירי בקומפיילר היא טבלה שמאגדת את המידע הדרוש לביצוע שלב זה בתהליך הניתוח. הניתוח התחבירי (Syntax Analysis) הוא השלב השני בתהליך הקומפילציה שבו הקוד מקור נבדק לפי כללי התחביר של השפה ונבנית מבנה תחבירי עץ התחביר (Parse Tree) או עץ התחביר המופשט (Abstract Syntax Tree - AST).
טבלת התחביר נועדה לסייע לקומפיילר בבניית ובתחזוקת עץ התחביר במהלך הניתוח. היא תכניס רעיונות כיצד לממש את הכללים התחביריים של השפה. ישנם שני סוגים נפוצים של טבלות תחביר: טבלת תחביר שמירה (Parsing Table) וטבלת ייצוג (Representation Table). בפרויקט אני אממש את טבלת התחביר באמצעות מטריצה.

מחסנית: שימוש במחסנית (Stack) בשלב ניתוח תחבירי בקומפיילר מציין את השימוש במחסנית כחלק מהתהליך של בניית עץ התחביר. המחסנית משמשת לאחסון וניהול מידע במהלך הניתוח, והיא כלי חשוב במנגנונים של בניית התחביר של הקוד המקור.

השימוש במחסנית נכנס לתמוך בתהליך ניתוח התחביר באופן הבא:

1. ניהול הסימבולים והמצב התחבירי.
2. מניע עבור פעולות Shift.
3. פעולות Reduce ובניית העץ.
4. טיפול בשגיאות.
5. מתן התראות לקוד מקור.

Reduce-Shift Actions:

הפעולות של Reduce ו-Shift הן שני סוגים של פעולות במהלך הקריאה של הקוד המקור על ידי הניתוח התחבירי. הפעולה של Shift מציינת קריאה נוספת מקוד המקור והזזת הניתוח התחבירי למצב קודם. הפעולה של Reduce מציינת הפקדה של אחת מן החוקים של דקדוק השפה, כלומר, הקצרה של יחידה קוד בעץ התחביר.

Action Table: טבלת הפעולות (Action Table) היא טבלה המכילה את ההוראות הדקדוקיות המציינות אילו פעולות יש לבצע בכל פרסום בזמן קריאת הקוד המקור. כל תא בטבלה מקושר לסימבול בסטק של הפרסר ולסימבול הבא מקוד המקור (ה-Look-Ahead).
בטבלת הפעולות, כל שורה מיוצגת על ידי מצב בו יכול להיות הפרסר (סימולטנית) וכל עמודה מיוצגת על ידי סימבול מהקוד המקור (Look-Ahead). כל תא בטבלה מכיל את הפעולה שצריך לבצע במצב ועם הסימבול הנתון.
טבלת הפעולות נוצרת בתהליכי הניתוח התחבירי, ותלויה בדקדוק של השפה.

goto table:

טבלת Goto מתארת את המעברים בין מצבי הפרסר למצבים נוספים כאשר הפרסר יבצע מעבר ממצב שבו הוא מסתמך על סימבולים מסוימים למצב בו הוא מתמקד בסימבול אחר. המצבים בהם יכול להימצא הפרסר הם המצבים בהם הוא משתמש במספר סימבולים מקוד המקור כדי לקבוע את המצב הבא.

מימוש lalr parser:

מפרט דקדוק:

הגדר את הדקדוק עבור השפה שלך באמצעות סימון רשמי כמו BNF. דקדוק זה מתאר את כללי התחביר של השפה, כולל סדר האסימונים ומבנה מבני השפה.
ודא שהדקדוק הוא חד משמעי ומתאים לניתוח באמצעות טכניקות LALR. זה בדרך כלל כרוך בפתרון אי בהירות או קונפליקטים בדקדוק.

בניית LR(0) פריטים:

פריטי LR(0) מייצגים מצבים אפשריים של המנתח בזמן שהוא מנתח את הקלט. כל פריט LR(0) מתאים לכלל ייצור עם נקודה המציינת את מיקום הניתוח הנוכחי.
לדוגמה, בהינתן כלל ייצור כמו $A \to \alpha\beta$, הפריטים של LR(0) יהיו $A \to \cdot\alpha\beta$, $A \to \alpha\cdot\beta$ ו- $A \to \cdot\alpha\beta$.

בניית LR(1) פריטים:

הרחב פריטי LR(0) לפריטי LR(1) על ידי הוספת סמל מבט קדימה. סמל מבט זה מציין את האסימון הבא שצפוי המנתח לאחר ניתוח הצד הימני של הייצור.
פריטי LR(1) נראים כמו $A \to \alpha\cdot\beta, a$, כאשר 'a' הוא סמל המבט קדימה.

בנה את טבלת הניתוח של LALR(1):

השתמש בפרטי LR(1) כדי לבנות את טבלת הניתוח LALR(1), המייצגת מעברים בין מצבי מנתח בהתבסס על אסימוני קלט.
טבלת הניתוח מורכבת בדרך כלל מערכים עבור כל מצב מנתח וכל סמל מסוף או לא מסוף. ערכים מציינים אם להזיז, להקטין או לבצע פעולה אחרת בהתבסס על המצב הנוכחי וסמל הקלט.

יישום מנתח:

כתוב קוד ליישום המנתח באמצעות טבלת הניתוח שנבנתה בשלב הקודם.

המנתח שומר על ערימה כדי לעקוב אחר מצבים ולנתח צמתים של עצים בזמן שהוא מעבד אסימוני קלט.
הוא קורא אסימונים מה-lexer (נתח לקסיקלי) ומשתמש בטבלת הניתוח כדי לקבוע את הפעולה הבאה: הסט, צמצום או קבל/דחה.

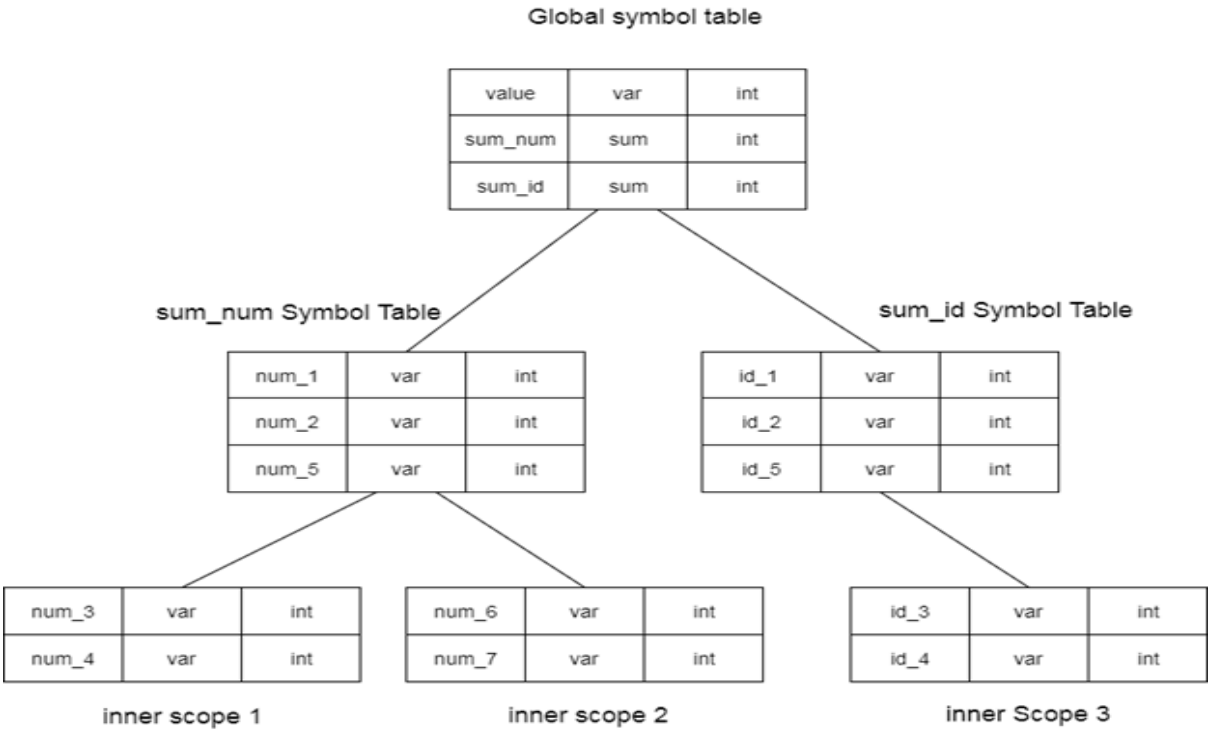
טיפול בשגיאות:

הטמע מנגנוני טיפול בשגיאות כדי לטפל בשגיאות תחביר או בקלט לא חוקי בחן.
אסטרטגיות לשחזור שגיאות עשויות לכלול שחזור מצב פאניקה (דילוג על אסימונים עד הגעה לנקודת סנכרון), הפקות שגיאות (שימוש בהפקות מיוחדות לטיפול בשגיאות נפוצות), או מתן הודעות שגיאה משמעותיות למשתמש.

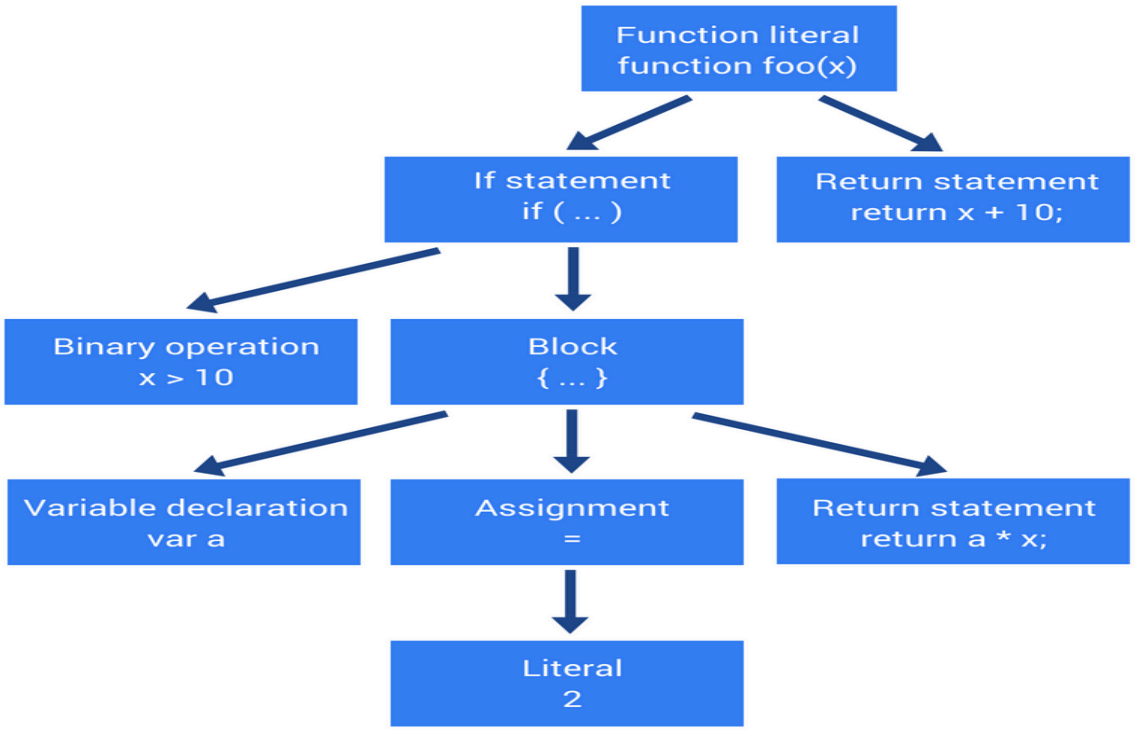
מבנה נתונים ב-Semantic analysis:

ניתוח סמנטי בקומפיילר כולל ניתוח של משמעות קוד המקור. מבני נתונים שונים משמשים בשלב זה כדי לייצג ולתפעל את המידע הסמנטי המופק מקוד המקור. הנה מבני נתונים שהשתמשתי בניתוח סמנטי:

טבלת סמלים (Symbol Table): טבלת סמלים היא מבנה נתונים המאחסן מידע על סמלים בתוכנה כגון משתנים, פונקציות, מחלקות וכו'. היא כוללת בדרך כלל תכונות כגון שם, סוג, היקף, מיקום זיכרון וכל מידע רלוונטי אחר. יש הרבה דרכים לממש את טבלת הסמלים אך אני בחרתי לממש את טבלת הסמלים בדרך הכי נפוצה והיא דרך טבלת גיבוב. טבלת גיבוב היא מבנה נתונים המאחסן מפתח וערך, המאפשר הכנסה, מחיקה ושליפה יעילה של ערכים על סמך המפתחות המשתייכים אליהם. הוא משתמש בפונקציית hashing כדי למפות מפתחות למדדים במערך, שבו מאוחסנים הערכים המתאימים. טכניקות טיפול בהתנגשות כגון שרשור או כתובות פתוחות משמשות כדי להתמודד עם מצבים שבהם שיש מספר מפתחות לאותו אינדקס. טבלאות Hash מציעות זמן גישה מהיר (O(1) בממוצע) והן נמצאות בשימוש נפוץ ביישומים שונים לאחסון ואחזור נתונים לפי מפתח.



עץ תחביר מופשט (AST) : AST נבנה במהלך הניתוח ומשמש לאורך כל הניתוח הסמנטי.
צמתים ב-AST מייצגים מבנים תחביריים של קוד המקור, ובמהלך ניתוח סמנטי, הם מסומנים במידע נוסף כגון מידע על סוג. עץ תחביר מופשט (AST) הוא מבנה נתונים היררכי המייצג את המבנה התחבירי של קוד המקור. כל צומת בעץ מתאים למבנה תחבירי, עם התכונות המאחסנות מידע על המבנה (כגון מזהים או מילוליים). לצמתים יש יחסי הורה-ילד, וניתן ליישם אלגוריתמי מעבר כדי לנתח או להפוך את הקוד ביעילות. ה-AST משמש כייצוג מובנה המאפשר משימות כמו ניתוח סמנטי ויצירת קוד בקומפיילרים.



מבנה נתונים ב-Code generation:

בשלב יצירת קוד בקומפיילר, התוכנית שנכתבה בשפת תכנות ברמה גבוהה מתורגמת לקוד מכונה ברמה נמוכה, שמתאים לביצוע על מעבד או פלטפורמה ספציפית. תהליך זה מחייב את הקומפיילר להבין את מבנה הקוד ולקבל החלטות מושכלות כדי לייצר קוד יעיל המנצל באופן מיטבי את הארכיטקטורה של היעד. השלבים כוללים תרגום הוראות לשפת ביניים, ניתוח יעילות הקוד, ולבסוף התאמתו לדרישות החומרה של המעבד הספציפי. ביצירת הקוד נלקחים בחשבון גורמים כמו אופטימיזציה של לולאות, ניהול זיכרון וחישובי כתובות, כדי להבטיח ביצועים מקסימליים. כך, שלב זה הוא מהותי להבטחת הפיכת התוכנית הנכתבת על ידי מפתחי התוכנה לקוד יעיל, בר ביצוע ומהיר. בשלב יצירת הקוד הקומפיילר משתמש טבלת הסמלים שנוצרה בשלב הקודם וביחד עם עץ תחביר מופשט.

26

Code Generation Example

Statements	*Code Generated*	*Register Descriptor*	*Address Descriptor*
		Registers empty	
t := a - b	MOV a,R0 SUB b,R0	R0 contains t	t in R0
u := a - c	MOV a,R1 SUB c,R1	R0 contains t R1 contains u	t in R0 u in R1
v := t + u	ADD R1,R0	R0 contains v R1 contains u	u in R1 v in R0
d := v + u *live*(d)=true all other dead	ADD R1,R0 MOV R0,d	R0 contains d	d in R0 d in R0 and memory

תיאור המערכת

מחלקת ה-Lexer

כותרת הפונקציה	הסבר
void allTheTokens(char str[], int* items, Token alltokens[], int size)	הפונקציה הראשית של הניתוח המילולי O(1)
State transition(State currentState, char input);	הפונקציה מחליטה לאיזה מצב שייחס התו O(1)
int isKeyword(char* str);	הפונקציה בודקת אם המחרוזת היא מילה שמורה O(1)
char* processStringToken(char str[], int* i, int* a, State* current);	הפונקציה מחברת את התווים במחרוזת O(n)
void processStringTokenResult(char* keyword, Token Tokens[], int* j, int* tokens, int* b, int line, int* a, int* i);	הפונקציה הופכת את המחרוזת לטוקן O(n)
void processNumberToken(char str[], int* i, int* a, State* current, Token Tokens[], int* j, int* tokens, int* b, int line);	הפונקציה הופכת את המספר לטוקן O(n)
void processOperatorToken(char str[], int* i, Token Tokens[], int* j, int* tokens, int* b, int line, State* current);	הפונקציה הופכת את אופרטור לטוקן O(n)
void processSeparatorToken(char str[], int* i, Token Tokens[], int* j, int* tokens, int* b, int line, State* current);	הפונקציה הופכת את הסוגריים לטוקן O(n)

void processSemicolonToken(char str[], int* i, Token Tokens[], int* j, int* tokens, int* b, int line, State* current);	הפונקציה הופכת את את הנקודה פסיק לטוקן O(n)
void printTokens(Token Tokens[], int tokens);	הפונקציה מדפיסה את הטוקנים O(n)

מחלקת ה-Parser

כותרת הפונקציה	הסבר
int pos_terminal(char* str)	הפונקציה בודקת אם האסימון הוא terminal O(1)
int pos_nonterminal(char* str)	הפונקציה בודקת אם האסימון הוא non terminal O(1)
int funShift(Token* tokens, int input_size, char* token)	הפונקציה מבצעת shift וממשיכה את הניתוח O(1)
int funReduce(Token* tokens, int input_size, char* token)	הפונקציה מבצעת reduce וממשיכה את הניתוח O(n)
int funAccept(Token* tokens, int input_size, char* token)	הפונקציה מסיימת את הניתוח O(1)
int funError(Token* tokens, int input_size, char* token)	הפונקציה מודיע על שגיאה O(1)
void lr_parse(Token* tokens , int input_size)	הפונקציה הראשית של ה-parser

void init()	הפונקציה מאתחלת את המטריצות O(1)

מחלקת ה-Semantic Analysis

כותרת הפונקציה	הסבר
int checkVariableType(SymbolTable* table, const char* varName, const char* expectedType, int scope)	הפונקציה בודקת אם המשתנה הוא הסוג שהוא אמור להיות
void semanticAnalysis(ASTNode* node, SymbolTable* table, int currentScope)	הפונקציה מבצעת את הניתוח
void perform(ASTNode* root)	הפונקציה הראשית של ה-Semantic Analysis
int checkassigntment(SymbolTable* table, const char* varName, int scope);	הפונקציה בודקת אם המשתנה שומר מידע מתאים לו

מחלקת ה-Code Generation

כותרת הפונקציה	הסבר
void code_generator_create();	הפונקציה יוצרת משתנה האחראי על יצירת הקוד
void code_generator_destroy();	הפונקציה הורסת את המשתנה
void code_generator_init();	הפונקציה מאתחלת את יצירת הקוד
int code_generator_register_alloc();	הפונקציה מחזירה את מספר האוגר הפנוי
void code_generator_register_free(int r);	הפונקציה משחררת את האוגר
char* code_generator_register_name(int r);	הפונקציה מחזירה את השם של האוגר
char* code_generator_label_create();	הפונקציה יוצרת תווית
char* code_generator_symbol_address(SymbolTableEntry* entry);	הפונקציה מחזירה את הכתובת של המשתנה
void code_generator_output(char* format, ...);	הפונקציה רושמת לקובץ מידע
void code_generator_output_data_segment(SymbolTable* table);	הפונקציה רושמת לקובץ מידע הקשור למשתנים
void code_generator_generate(SymbolTable* table,ASTNode* parse_tree);	הפונקציה מייצרת את הקוד של התוכנית
void code_generator_block(ASTNode* block);	הפונקציה מייצרת את הקוד של בלוק
void code_generator_stmt(ASTNode* stmt);	הפונקציה מייצרת את הקוד של statement

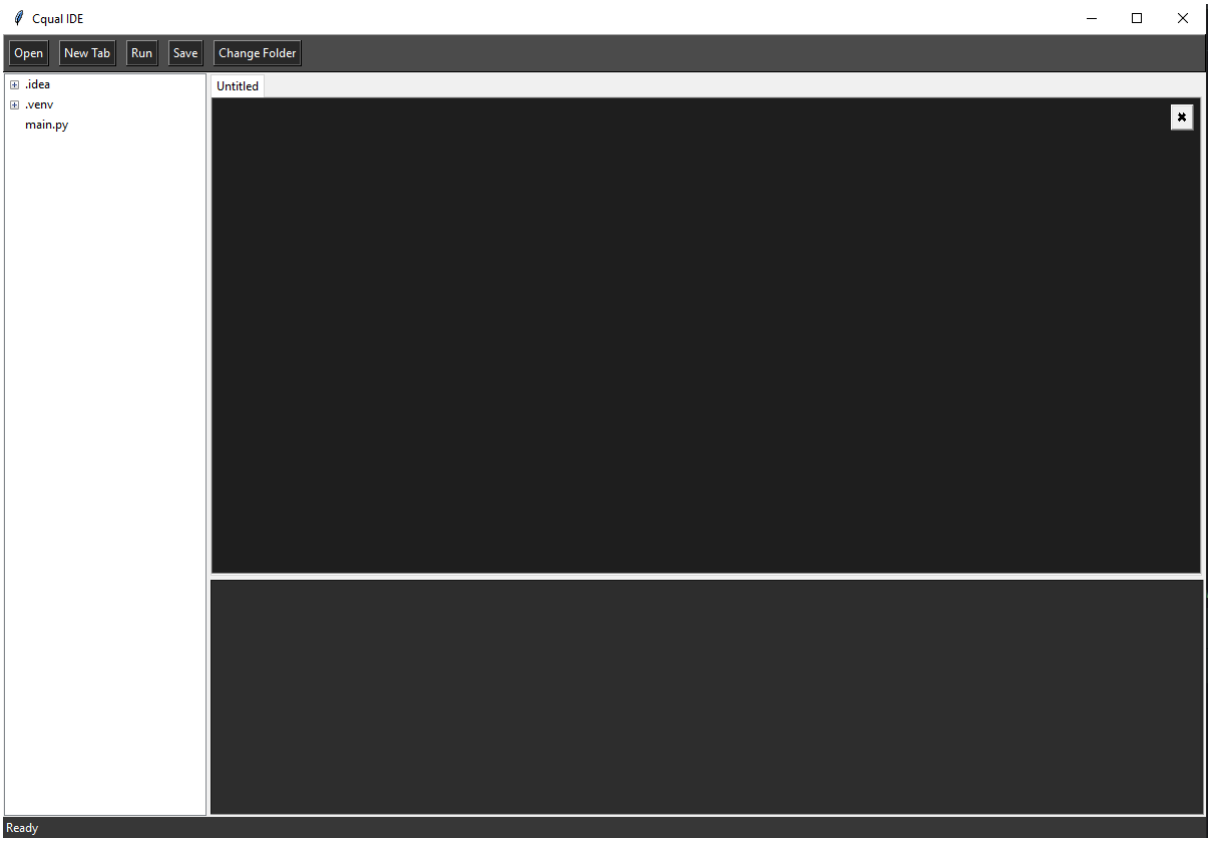
הפונקציה מייצרת את הקוד של declaration	void code_generator_decl(ASTNode* decl);
הפונקציה מייצרת את הקוד של assignment	void code_generator_assign(ASTNode* assign);
הפונקציה מייצרת את הקוד של while loop	void code_generator_while(ASTNode* _while);
הפונקציה מייצרת את הקוד של expression	void code_generator_expression(ASTNode* expr);
הפונקציה מייצרת את הקוד של binary expression	void code_generator_binary_expression(int left_register, int right_register, TokensType op);
הפונקציה מייצרת את הקוד של if statement	void code_generator_if_statement(ASTNode * _if);
הפונקציה מייצרת את הקוד של condition	void code_generator_condition(ASTNode* condition);
הפונקציה מייצרת את הקוד של function	void code_generator_function(ASTNode* func);
הפונקציה מייצרת את הקוד של קריאה לפונקציה	void code_generator_funCall(ASTNode* func);
הפונקציה מייצרת את הקוד של for loop	void code_generator_for(ASTNode* _for);
הפונקציה מייצרת את הקוד של print function	void code_generator_print(ASTNode* print);
הפונקציה מייצרת את הקוד של input function	void code_generator_input(ASTNode* input);

יצירת סביבת עבודה

אחרי שסיימתי את הקומפיילר הרגשתי שמשהו חסר אז החלטתי ליצור סביבת עבודה ששם יהיה אפשר לכתוב קוד בשפה החדשה. בגלל שיצירת סביבת עבודה היא לא חובה אז החלטתי לבחור לפתח משהו יחסית פשוט בפייתון שיעשה את העבודה.

כשחיפשתי איך ליצור ממשק גרפי בפייתון נתקלתי ב tkinter שהיא הרחבה פשוטה ומהירה שמקלה מאוד על המפתח לכן בחרתי אותה.

מכיוון שלא רציתי להתמקד בזה יותר מידי בסביבת העבודה הזאת יש רק את הדברים הכי בסיסיים שמצאתי לנכון להוסיף.



שפות תכנות:

שפת התכנות שאני משתמש בה בפרויקט היא c

סביבת עבודה:

סביבת העבודה שלי היא Microsoft Visual Studio Community 2022 (64-bit) - Current
Version 17.7.6
Command Prompt
(PyCharm 2024.1.1 (Community Edition

הפסאודו קוד של התוכנית הראשית

פונקציה ReadFile (מצביע לגודל size)

בקש מהמשתמש את שם הקובץ
פתח קובץ עם שם הקובץ במצב קריאה
אם הקובץ לא נפתח בהצלחה
    החזר NULL
עברו לסוף הקובץ כדי למצוא את גודל הקובץ
אחסן את גודל הקובץ משתנה length
חזור לתחילת הקובץ
הקצאת זיכרון למחרוזת בגודל אורך + 1
אתחול אינדקס i ל-0
אם לא בסוף הקובץ
    קרא תו C מהקובץ
    אחסן את C במחרוזת במיקום i
    הגדל את i
הגדר את המיקום האחרון של מחרוזת ל- NULL terminator
עדכן את הגודל ל i
סגור את הקובץ
החזר את המחרוזת

הגדר מערך T NULL
הגדר מקום כ-0 לאחסון המיקום הנוכחי

פונקציה ראשי

הקצה מקום בזיכרון של אסימונים בגודל 100
קריאה לפונקציה init()
אתחול i, j
קריאה ל-ReadFile והעברת הפניה ל-J כדי לקבל את תוכן הקובץ ב-Str
עבדו את Str כדי לסמן אותו ל-T, מעדכנים את A במספר האסימונים
עבור כל אסימון ב-T עד A
    הדפס את נתוני האסימון
קריאה ל lr_parser עם האסימונים T בגודל A
החזר 0

מדריך למשתמש

היי תודה שבחרת להשתמש בקומפיילר הזה, קומפיילר זה כלי רב עוצמה שהופך את קוד המקור שלך לתוכניות שהמחשב שלך יכול להבין ולהפעיל. אם אתה חדש בקומפיילרים, אל תדאג! מדריך זה ידריך אותך בתהליך צעד אחר צעד.

מה זה קומפיילר?

קומפיילר הוא כמו מתרגם למחשב שלך. הוא לוקח את הקוד שאתה כותב בשפת תכנות ברמה גבוהה, כגון Python או C++, ומתרגם אותו לצורה שחומרת המחשב שלך יכולה להבין ולבצע במקרה של הקומפיילר הזה הקוד שאתה יכול לרשום הוא בשפה בשם - Cqual הקומפיילר מתרגם לשפת אסמבלי. תהליך זה כולל מספר שלבים:

ניתוח לקסיקלי: השלב הראשון הוא ניתוח מילוני. כאן, המהדר מפרק את הקוד שלך ליחידות קטנות הנקראות אסימונים. חשבו על אסימונים כעל אבני הבניין של הקוד שלכם, כמו מילים במשפט.

ניתוח תחביר: לאחר מכן מגיע ניתוח תחביר. המהדר מסדר את האסימונים למבנה הנקרא עץ ניתוח, המייצג את המבנה הדקדוקי של הקוד שלך. עץ זה עוזר לקומפיילר להבין את הקשרים בין חלקים שונים של הקוד שלך.

ניתוח סמנטי: ברגע שהמבנה של הקוד שלך מובן, המהדר בודק אותו לאיתור שגיאות לוגיות ומוודא שהוא עומד בכללי שפת התכנות. שלב זה נקרא ניתוח סמנטי ועוזר לתפוס טעויות שאולי אינן ברורות מעצם הסתכלות בקוד.

יצירת קוד ביניים: לאחר אימות נכונות הקוד, הקומפיילר עשוי ליצור ייצוג ביניים של הקוד. קוד ביניים זה הוא לרוב פשוט ואחיד יותר מקוד המקור המקורי, מה שמקל על ביצוע אופטימיזציה ותרגום לקוד מכונה.

אופטימיזציה: בשלב זה, המהדר מנסה לשפר את היעילות של הקוד שלך על ידי אופטימיזציה שלו. זה יכול לכלול דברים כמו סידור מחדש של הוראות או ביטול קוד מיותר כדי לגרום לתוכנית שלך לפעול מהר יותר ולהשתמש בפחות זיכרון.

יצירת קוד: לבסוף, הקומפיילר מתרגם את הקוד הממוטב לאסמבלי. קוד מכונה זה הוא מה שבוצע בפועל כאשר אתה מפעיל את התוכנית שלך.

כיצד להשתמש בקומפיילר הזה

השימוש בקומפיילר הוא קל! הינה מדריך פשוט שיעזור לך להתחיל:

כתוב את הקוד שלך: ראשית, כתוב את הקוד שלך בעורך טקסט או בסביבת פיתוח משולבת (IDE).

פתח Command Prompt: פתח את Command Prompt במחשב שלך. זה המקום שבו אתה תהיה אינטראקציה עם הקומפיילר.

הרכיב את הקוד שלך: השתמש בשורת הפקודה כדי לנווט לספרייה שבה הקוד שלך נשמר. לאחר מכן, הקלד את הפקודה כדי להרכיב את הקוד שלך באמצעות הקומפיילר. לדוגמה:

```
compiler.c your-program.txt
```

הפעל את התוכנית שלך: אם תהליך ההידור הצליח, הקומפיילר יפיק קובץ אסמבלי שתוכל להפעיל במחשב שלך.

הצג את הפלט: אם התוכנית שלך מפיקה פלט כלשהו, כגון טקסט או מספרים, היא תוצג לאחר הפעלתה.

וזהו! עכשיו שאתה יודע איך להשתמש בקומפיילר נשאר לך רק להמשיך לכתוב קוד!.

רפלקציה

אני בחרתי בנושא בניית קומפיילר מכיוון שבניית שפה מאפס ולגרום לה לעבוד נשמע לי פרויקט מעניין שרציתי לעשות כבר הרבה זמן ועוד כששמעתי שבניית קומפיילר זה פרויקט קשה ואתגרי זה היה לי ברור שזה הפרויקט שאני הולך לעשות. בשבילי העבודה על הפרויקט הייתה באמת מאתגרת מכיוון שאין המון מידע מדויק באינטרנט היו רגעים בהם אני הייתי תקוע והייתי צריך לחשוב על דרכים יצירתיות לפתור את אותן בעיות אבל בסך הכל העבודה על הפרויקט הייתה טובה ולמדתי הרבה ממנה ולמצוא מידע חשוב באינטרנט ואיך אפשר לממש את המידע שמצאתי. בעזרת העבודה על הפרויקט הרחבתי את הידע שלי על איך שפת תכנות עובדת מבפנים ואת השלבים ביצירת קוד מכונה משפת תכנות. בעזרת הפרויקט אני מרגיש שרכשתי כלים שיעזרו לי בהמשך כמו איך למצוא מידע ממוקד מתוך כמות גדולה של מידע, השתמשתי בידע התיאורטי שלי של מבני נתונים והשתמשתי במבני נתונים שקודם היו לי פחות ברורים ועכשיו אני מכיר אותם טוב. היו זמנים בפרויקט שממש נתקעתי ואני לא יכול להתקדם כי אין לי את הידע הנדרש מה שקרה לי בעיקר בשלב השני של הקומפיילר שהוא ה-parser אך בסוף קראתי מאמרים ומאתרים באינטרנט עד שהצלחתי למצוא את הפתרון ולהתקדם. אם הייתי מתחיל היום מחדש את הפרויקט אני חושב שהייתי משתמש בשפת תכנות שונה מ C מכיוון שב- C אין תכנות מונחה עצמים היה לי יותר מסורבל לכתוב את הקוד ואם הייתי בוחר שפת תכנות כמו -java אז גם לא הייתי צריך ליצור מאפס את כל מבני הנתונים וגם לא הייתי צריך לנהל את הזיכרון מה שהיה מקל עליי משמעותית וחוסך הרבה זמן. אם הייתי יכול לשנות את תהליך העבודה שלי הייתי משנה את אופן הלמידה שלי, אופן הלמידה שלי הוא כזה לקרוא ולחקור את השלב הנוכחי שאני נמצא בו כרגע ואחרי שאני מסיים אותו לקרוא על השלב הבא מה שגרם לתחושה של תקיעות מכיוון שלא היה לי את הידע להמשיך ברצף את השלבים מה מאט את הקצב שלי.

ביבליוגרפיה

geeksforgeeks
https://www.geeksforgeeks.org/lalr-parser-with-examples/?ref=lbp
https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

codingninjas.com
https://www.codingninjas.com/studio/library/parsing-in-compiler-design
https://www.codingninjas.com/studio/library/lexer-and-lexer-generators

guru99.com
https://www.guru99.com/compiler-design-lexical-analysis.html

cratecode.com
https://cratecode.com/info/lexical-analysis

cboard.cprogramming.com
https://cboard.cprogramming.com/c-programming/114351-implementation-slr-parser.htmla

javatpoint.com
https://www.javatpoint.com/lalr-1-parsing
https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm

serokell.io
https://serokell.io/blog/how-to-implement-lr1-parser

hal.science
https://hal.science/hal-01633123/document
chat GPT
Compilers: Principles, Techniques, and Tools by Aho, Lam, Sethi, and Ullman

נספח- קוד הפרויקט

main.h

```
#include "Lexer.h"
#include "Parser.h"
#pragma once
extern Token t[100];
extern int place;
```

main.c

```
#define  _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Lexer.h"
#include "Parser.h"
char* read_file(int* size)
{
    FILE* file;
    char filename[256];
    scanf("%s", filename);

    file = fopen(filename, "r");
    if (file == NULL) return NULL;
    fseek(file, 0, SEEK_END);
    int length = ftell(file);
    fseek(file, 0, SEEK_SET);
    char* string = malloc(sizeof(char) * (length + 1));
    char c;
    int i = 0;
    while ((c = fgetc(file)) != EOF)
    {
        string[i] = c;
        i++;
    }
    string[i] = '\0';
    *size = i;
    fclose(file);
    return string;
}
Token* t = NULL;

int place = 0;
int main(){
    t = malloc(sizeof(Token) * 100);
    init();
    int i,j=0;
    int a;

    char* str = read_file(&j);
    allTheTokens(str,&a,t,j);
    for (i = 0; i < a; i++)
    {
        printf("%s ", t[i].data);
```

```
    }
    lr_parse(t, a);
            return 0;
}
```

Lexer.h

```
#pragma once


typedef enum
{
    EPSILON, // ε
    ID,
    LPAREN, // (
    RPAREN, // )
    LBRACE, // {
    RBRACE, // }
    COMMA, // ,
    IF,
    INT,
    String,
    WHILE,
    FOR,
    RETURN,
    DOUBLE,
    THEN,
    ELSE,
    NUMBER,
    STRING,
    NEQ, // !=
    GT, // >
    LT, // <
    GTE, // >=
    LTE, // <=
    EQ, // ==
    ASSIGN, // =
    PLUS, // +
    MINUS, // -
    STAR, // *
    SLASH, // /
    SEMICOLON, // ;
    END_OF_INPUT // $
}TokensType;

typedef enum {
    STATE_INITIAL,
    STATE_IN_STRING,
    STATE_IN_KEYWORD,
    STATE_IN_IDENTIFIER,
    STATE_IN_NUMBER,
    STATE_IN_DOUBLE,
    STATE_IN_OPERATOR,
    STATE_IN_SEPARATOR,
    STATE_IN_SEMICOLON
```

```
} State;


typedef struct {
    TokensType type;
    char* data;
    int linenum;
    int wordnum;
}Token;
void allTheTokens(char str[], int* items, Token alltokens[], int size);
State transition(State currentState, char input);
int isKeyword(char* str);

// Constants
#define NUM_KEYWORDS 8



// Token processing functions
char* processStringToken(char str[], int* i, int* a, State* current);
void processStringTokenResult(char* keyword, Token Tokens[], int* j, int* tokens, int* b, int line, int* a, int*
i);
void processNumberToken(char str[], int* i, int* a, State* current, Token Tokens[], int* j, int* tokens, int* b, int
line);
void processOperatorToken(char str[], int* i, Token Tokens[], int* j, int* tokens, int* b, int line, State* current);
void processSeparatorToken(char str[], int* i, Token Tokens[], int* j, int* tokens, int* b, int line, State*
current);
void processSemicolonToken(char str[], int* i, Token Tokens[], int* j, int* tokens, int* b, int line, State*
current);

// Print functions
void printTokens(Token Tokens[], int tokens);

int posOp(char* str);

// Copy tokens from source to destination
void copyTokens(Token destination[], Token source[], int tokens);
```

Lexer.c

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "Lexer.h"


char* keywords[] = { "if", "int","string", "while", "for", "return", "double","then", "else"};

void allTheTokens(char str[], int* items, Token alltokens[], int size) {
    int tokens = 0;
    int i, j = 0, a = 0, b = 0;
    int line = 0;
    State current = STATE_INITIAL;

    Token Tokens[100];
    char* keyword;
```

```
    for (i = 0; i < size; i++) {
        if (str[i] == '\n') {
            line++;
            b = 0;
            i++;
        }

        current = transition(current, str[i]);

        switch (current) {
        case STATE_IN_STRING:
            keyword = processStringToken(str, &i, &a, &current);
            processStringTokenResult(keyword, Tokens, &j, &tokens, &b, line,&a,&i);
            break;
        case STATE_IN_NUMBER:
            processNumberToken(str, &i, &a, &current, Tokens, &j, &tokens, &b, line);
            break;
        case STATE_IN_OPERATOR:
            processOperatorToken(str, &i, Tokens, &j, &tokens, &b, line, &current);
            break;
        case STATE_IN_SEPARATOR:
            processSeparatorToken(str, &i, Tokens, &j, &tokens, &b, line, &current);
            break;
        case STATE_IN_SEMICOLON:
            processSemicolonToken(str, &i, Tokens, &j, &tokens, &b, line, &current);
            break;
        default:
            break;
        }

        current = STATE_INITIAL;
    }



    printTokens(Tokens, tokens);

    *items = tokens;
    copyTokens(alltokens, Tokens, tokens);
}

/// <summary>
///
/// </summary>
/// <param name="str"></param>
/// <param name="i"></param>
/// <param name="a"></param>
/// <param name="current"></param>
/// <returns></returns>
char* processStringToken(char str[], int* i, int* a, State* current) {
    char* keyword = NULL;

    while (*current == STATE_IN_STRING) {
        keyword = (char*)realloc(keyword, sizeof(char) * ((*a)+1));
        keyword[*a+1] = '\0';
        keyword[*a] = str[*i];
        (*a)++;
        (*i)++;
        *current = transition(*current, str[*i]);
    }
```

```
    return keyword;
}
State transition(State currentState, char input) {
    int index,  startCondition,  endCondition;
    State nextState;
    static const State transitions[][128] = {
        [STATE_INITIAL] = { STATE_IN_STRING, STATE_IN_NUMBER, STATE_IN_OPERATOR,
STATE_INITIAL, STATE_IN_SEMICOLON,STATE_IN_SEPARATOR },
        [STATE_IN_STRING] = { STATE_IN_STRING, STATE_INITIAL },
        [STATE_IN_NUMBER] = { STATE_IN_NUMBER, STATE_IN_DOUBLE, STATE_INITIAL },
        [STATE_IN_DOUBLE] = { STATE_IN_DOUBLE, STATE_INITIAL },
        [STATE_IN_OPERATOR] = { STATE_INITIAL },
        [STATE_IN_SEPARATOR] = { STATE_INITIAL },
        [STATE_IN_SEMICOLON] = { STATE_INITIAL },
    };

    static const int charConditions[][3] = {
        { 'A', 'Z', STATE_IN_STRING },
        { 'a', 'z', STATE_IN_STRING },
        { '0', '9', STATE_IN_NUMBER },
        { '.', '.', STATE_IN_DOUBLE },
        { ';', ';', STATE_IN_SEMICOLON },
        { '=', '=', STATE_IN_OPERATOR },
        { '<', '<', STATE_IN_OPERATOR },
        { '>', '>', STATE_IN_OPERATOR },
        { '!', '!', STATE_IN_OPERATOR },
        { '+', '+', STATE_IN_OPERATOR },
        { '-', '-', STATE_IN_OPERATOR },
        { '*', '*', STATE_IN_OPERATOR },
        { '/', '/', STATE_IN_OPERATOR },
        { '{', '{', STATE_IN_SEPARATOR },
        { '}', '}', STATE_IN_SEPARATOR },
        { '(', '(', STATE_IN_SEPARATOR },
        { ')', ')', STATE_IN_SEPARATOR },
        { '\', '\', STATE_IN_STRING },
        { '"', '"', STATE_IN_STRING },
        { ',', ',', STATE_IN_SEPARATOR },
    };

     index = (unsigned char)input;

    for (int j = 0; j < sizeof(charConditions) / sizeof(charConditions[0]); ++j) {
        startCondition = charConditions[j][0];
        endCondition = charConditions[j][1];
        nextState = (State)charConditions[j][2];

        if (index >= startCondition && index <= endCondition) {
            return nextState;
        }
    }

    return transitions[currentState][index];
}
void processStringTokenResult(char* keyword, Token Tokens[], int* j, int* tokens, int* b, int line,int* a,int *i) {
    Tokens[*j].data = malloc(sizeof(char) * (*a));
    strcpy(Tokens[*j].data, keyword);

    if (strcmp(Tokens[*j].data,"end")==0) {
        Tokens[*j].wordnum = ++(*b);
        Tokens[*j].linenum = line;
```

```
        Tokens[*j].type = EPSILON;
        (*j)++;
        (*tokens)++;
    }

    else if (isKeyword(keyword)!=-1) {
        Tokens[*j].wordnum = ++(*b);
        Tokens[*j].linenum = line;
        Tokens[*j].type = IF +isKeyword(keyword);
        (*j)++;
        (*tokens)++;
    }
    else if (keyword[0] == '"') {
        Tokens[*j].wordnum = ++(*b);
        Tokens[*j].linenum = line;
        Tokens[*j].type = STRING;
        (*j)++;
        (*tokens)++;
    }
    else {
        Tokens[*j].wordnum = ++(*b);
        Tokens[*j].linenum = line;
        Tokens[*j].type = ID;
        (*j)++;
        (*tokens)++;
    }

    if (*a >= 1)
        (*i)--;
    *a = 0;
}

void processNumberToken(char str[], int* i, int* a, State* current, Token Tokens[], int* j, int* tokens, int* b, int
line) {
    int check = 0;
    int flag = 0;
    Tokens[*j].data = NULL;
    while (*current == STATE_IN_NUMBER) {
        Tokens[*j].data = (char*)realloc(Tokens[*j].data, sizeof(char) * (check + 1) + 1);
        Tokens[*j].data[check + 1] = '\0';
        Tokens[*j].data[check] = str[*i];
        (*i)++;
        check++;
        *current = transition(*current, str[*i]);
        flag++;
    }

    if (str[*i] == '.') {
        Tokens[*j].data[check++] = str[*i];
        *current = transition(*current, str[++(*i)]);
        flag = 0;
        while (*current == STATE_IN_DOUBLE) {
            Tokens[*j].data = (char*)realloc(Tokens[*j].data, sizeof(char) * (check + 1) + 1);
            Tokens[*j].data[check] = str[*i];
            Tokens[*j].data[check + 1] = '\0';
            (*i)++;
            check++;
            flag++;
            *current = transition(*current, str[*i + 1]);
        }
    }
```

```
    Tokens[*j].type = NUMBER;
    if (flag >= 1)
        (*i)--;

    Tokens[*j].wordnum = ++(*b);
    Tokens[*j].linenum = line;

    (*j)++;
    (*tokens)++;
    *current = transition(*current, str[*i]);
}

void processOperatorToken(char str[], int* i, Token Tokens[], int* j, int* tokens, int* b, int line, State* current)
{
    Tokens[*j].data = NULL;
    Tokens[*j].data = (char*)realloc(Tokens[*j].data, sizeof(char) * 2 );
    Tokens[*j].data[0] = str[*i];
    Tokens[*j].data[1] = '\0';
    Tokens[*j].wordnum = ++(*b);
    Tokens[*j].linenum = line;
    Tokens[*j].type = NEQ + posOp(Tokens[*j].data);
    (*j)++;
    (*tokens)++;
    *current = transition(*current, str[*i]);
}

void processSeparatorToken(char str[], int* i, Token Tokens[], int* j, int* tokens, int* b, int line, State*
current) {
    Tokens[*j].data = NULL;
    Tokens[*j].data = (char*)realloc(Tokens[*j].data, sizeof(char) * 2);
    Tokens[*j].wordnum = ++(*b);
    Tokens[*j].data[0] = str[*i];
    Tokens[*j].data[1] = '\0';
    Tokens[*j].linenum = line;
    Tokens[*j].type = LPAREN+posSep(Tokens[*j].data);

    (*j)++;
    (*tokens)++;
    *current = transition(*current, str[*i]);
}

void processSemicolonToken(char str[], int* i, Token Tokens[], int* j, int* tokens, int* b, int line, State*
current) {
    Tokens[*j].data = NULL;
    Tokens[*j].data = (char*)realloc(Tokens[*j].data, sizeof(char) * 2);
    Tokens[*j].wordnum = ++(*b);
    Tokens[*j].data[0] = str[*i];
    Tokens[*j].data[1] = '\0';
    Tokens[*j].linenum = line;
    Tokens[*j].type = SEMICOLON;
    (*j)++;
    (*tokens)++;
    *current = transition(*current, str[*i]);
}

void printTokens(Token Tokens[], int tokens) {
    int i;
    for (i = 0; i < tokens; i++) {
        printf("Token %d: %s , State: %d\n", i + 1, Tokens[i].data, Tokens[i].type);
    }
```

```
}

int isKeyword(char* str) {
    int i;
    for ( i = 0; i < sizeof(keywords)/sizeof(keywords[0]); i++) {
        if (strcmp(str, keywords[i]) == 0) {
            return i;
        }
    }
    return -1;
}
int posSep(char* str) {
    static char* sep[] = { "(",")","{","}",","};
    int i;
    for (i = 0; i < sizeof(sep) / sizeof(sep[0]); i++) {
        if (strcmp(str, sep[i]) == 0) {
            return i;
        }
    }
    return -1;
}

int posOp(char* str) {
    int i;
    static char* ops[] = { "!=",
     ">",
     "<",
     ">=",
     "<=",
     "==",
     "=","+","-","*","/"};
    for (i = 0; i < sizeof(ops)/sizeof(ops[0]); i++) {
        if (strcmp(str, ops[i]) == 0) {
            return i;
        }
    }
    return -1;
}

void copyTokens(Token destination[], Token source[], int tokens) {
    int i;
    for ( i = 0; i < tokens; i++) {
        destination[i] = source[i];
    }
}
```

Parser.h

```
#pragma once
#include "Lexer.h"
#include "stack.h"
#include "SymbolTable.h"


typedef struct {
    char type[4]; // 'r' for strings starting with 'r', 's' for strings starting with 's', 'i' for integers
```

```
} MatrixCell;
typedef struct {
    int row;
    int col;
    char data[5];
}insert;
typedef struct {
    const char* non_terminal;
    int reduction_number;
} ReductionRule;

stack s;
SymbolTable st;
int input_pos;
int current_state;
char action[10]; // Increase size if necessary for your actions
char* scope;
char** scopes;
int numscopes;


int pos_terminal(char* str);
int pos_nonterminal(char* str);
int  funShift(Token* tokens, int input_size, char* token);
int funReduce(Token* tokens, int input_size, char* token);
int funAccept(Token* tokens, int input_size, char* token);
int funError(Token* tokens, int input_size, char* token);
void lr_parse(Token* tokens , int input_size);
void init();
```

Parser.c

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "main.h"

#define ROWS 105
#define COLS 30

// Structure to hold the matrix data

MatrixCell matrix[ROWS][COLS];
int goto_table[ROWS][23];

ReductionRule reduction_rules[] = {
    {"S'", 1},
 {"S'", 0},
 {"statement_list", 2},
 {"statement_list", 1},
 {"function_declaration", 6},
 {"formal_parameters", 3},
 {"formal_parameters_tail", 4},
 {"formal_parameters_tail", 1},
 {"parameters", 2},
 {"parameters_tail", 0},
```

```
 {"parameters_tail", 3},
 {"block", 3},
 {"statement", 1},
 {"statement", 1},
 {"statement", 1},
 {"statement", 1},
 {"statement", 1},
 {"statement", 1},
 {"statement", 1},
 {"statement", 1},
 {"statement", 1},
 {"declaration", 3},
 {"declaration", 5},
 {"assignment", 4},
 {"function_call", 4},
 {"if_statement", 4},
 {"for_statement", 9},
 {"while_statement", 5},
 {"return_statement", 3},
 {"expression", 3},
 {"expression", 3},
 {"expression", 1},
 {"term", 3},
 {"term", 3},
 {"term", 1},
 {"factor", 1},
 {"factor", 3},
 {"factor", 1},
 {"factor_base", 1},
 {"factor_base", 1},
 {"type_specifier", 1},
 {"type_specifier", 1},
 {"type_specifier", 1},
 {"condition", 3},
 {"condition", 3},
 {"condition", 3},
 {"condition", 3},
 {"condition", 3},
 {"condition", 3},
 {"expression_statement", 2} };
int pos_terminal(char* str) {
   static char* terminals_ordered[] = {
      "ε",        // Empty string (often used to represent an empty production)
   "ID",       // Identifier
   "(",        // Open parenthesis
   ")",        // Close parenthesis
   ",",        // Comma
   "{",        // Open curly brace
   "}",        // Close curly brace
   ";",        // Semicolon
   "=",        // Assignment operator
   "IF",       // 'if' keyword
   "THEN",     // 'then' keyword, though not typically used in C-like languages
   "FOR",      // 'for' keyword
   "WHILE",    // 'while' keyword
   "RETURN",   // 'return' keyword
   "+",        // Addition operator
   "-",        // Subtraction operator
   "*",        // Multiplication operator
   "/",        // Division operator
   "number",   // Numeric literal
```

```
    "string",   // String literal
    "INT",      // 'int' keyword for integer type specifier
    "String",   // 'String' type specifier, though in C typically 'char*' is used
    "double",   // 'double' keyword for double-precision floating-point type specifier
    "==",       // Equality comparison operator
    "!=",       // Inequality comparison operator
    ">",        // Greater than operator
    "<",        // Less than operator
    ">=",       // Greater than or equal to operator
    "<=",       // Less than or equal to operator
    "$"         // End of input symbol
    };
    for (int i = 0; i < sizeof(terminals_ordered) / sizeof(terminals_ordered[0]); i++)
    {
        if (strcmp(str, terminals_ordered[i]) == 0)
            return i;
    }
    return -1;
}
int pos_nonterminal(char* str) {
    static const char* non_terminals[] = {
    "S'",
        "statement_list",
        "function_declaration",
        "formal_parameters",
        "formal_parameters_tail",
        "parameters",
        "parameters_tail",
        "block",
        "statement",
        "declaration",
        "assignment",
        "function_call",
        "if_statement",
        "for_statement",
        "while_statement",
        "return_statement",
        "expression",
        "term",
        "factor",
        "factor_base",
        "type_specifier",
        "condition",
        "expression_statement"
    };
    for (int i = 0; i < sizeof(non_terminals) / sizeof(non_terminals[0]); i++)
    {
        if (strcmp(str, non_terminals[i]) == 0)
            return i;
    }
    return -1;
}
char* symbols[] = {  "ε",
    "ID",
     "(",
     ")",
     "{",
     "}",
     ",",
    "IF",
    "INT",
```

```
    "String",
    "WHILE",
    "FOR",
    "RETURN",
    "DOUBLE",
    "THEN",
    "ELSE",
    "number",
    "string",
     "!=",
     ">",
    "<",
    ">=",
    "<=",
    "==",
    "=",
    "+",
     "-",
     "*",
    "/",
    ",",
    "$" };

/// <summary>
///
/// </summary>
/// <param name="tokens"></param>
/// <param name="input_size"></param>
/// <param name="token"></param>
/// <returns></returns>
int  funShift(Token* tokens, int input_size, char* token) { // Shift
    int nextState = atoi(action + 1);
    if (strcmp(token, "WHILE") == 0 || strcmp(token, "FOR") == 0)
        scope = _strdup(token);
    if (strcmp(token, "{") == 0) {
        numscopes++;

    }
    if (strcmp(token, "(") == 0) {
        int temp = stack_pop(&s);
        scope = _strdup(((ASTNode*)stack_top(&s))->data);
        stack_push(&s, temp);
        numscopes++;

    }
    ASTNode* newNode;
    if (input_pos < input_size)
        newNode = createASTNode("Terminal", token, tokens[input_pos].data, numscopes - 1);
    else
        newNode = createASTNode("Terminal", token, token, numscopes - 1);
    stack_push(&s, newNode);
    //printAST(newNode, 0);

    stack_push(&s, nextState); // Push next state
    input_pos++;
    current_state = nextState;
    return 1;
}

int funReduce(Token* tokens, int input_size, char* token) {  // Reduce
    int ruleIndex = atoi(action + 1);
```

```
    int popCount = reduction_rules[ruleIndex].reduction_number; // Assuming each entry in the stack is a
state-symbol pair
    if (strcmp(token, "}") == 0 && numscopes > 1) {
        numscopes--;
    }

    ASTNode* reducedNode = createASTNode(reduction_rules[ruleIndex].non_terminal, token, NULL,
numscopes-1);
    // Pop states and symbols from the stack as per the reduction rule
    for (int i = 0; i < popCount; i++) {
        stack_pop(&s);
        ASTNode* Node = (ASTNode*)stack_pop(&s);
        addChild(reducedNode, Node);
    }
    int topState;
    // Peek the new top state after popping

    topState = stack_top(&s);
    //printf("%s", stack_top(&s));


    // Use the goto_table to find the next state based on the non-terminal and top state
    int nextState = goto_table[topState][pos_nonterminal(reducedNode->type)];

    stack_push(&s, reducedNode); // Push the non-terminal resulted from reduction

    //printAST(reducedNode, 0);
    stack_push(&s, nextState); // Push the state obtained from goto_table
    //printf("%s", stack_top(&s));
    current_state = nextState; // Update current state
    return 1;
}

int funAccept(Token* tokens, int input_size, char* token) {
    printf("\nParsing successful\n");
    stack_pop(&s);
    printAST(stack_top(&s), 0);
    buildSymbolTableFromAST(stack_top(&s), &st);
    printSymbolTable(&st);
    //perform(stack_top(&s));
    return 0;
}

int funError(Token* tokens, int input_size, char* token) {
    fprintf(stderr, "\nSyntax error\n");
    return 0;
}

void lr_parse(Token* tokens, int input_size) {
    numscopes = 1;


    scopes = malloc(sizeof(char*));
    symbolTableInit(&st);
    stack_init(&s);


    typedef int (*pFun)(Token* , int , char* );

    int flagRun = 1;
    while (flagRun) {
```

```
        char* token = (input_pos < input_size) ? symbols[tokens[input_pos].type] : "$";

        strcpy(action, matrix[current_state][pos_terminal(token)].type);
        char possibales[] = "sraE";
        pFun arFun[] = { funShift, funReduce, funAccept, funError };
        char* pos = strchr(possibales, action[0]);
        flagRun =  arFun[pos - possibales](tokens, input_size, token);
    }
}
void init() {
    // Initialize the matrix with default values


    insert insert2[][1] = { {0,0,"s3"},
            {0,1,"s14"},
            {0,2,"s25"},
            {0,9,"s15"},
            {0,11,"s16"},
            {0,12,"s17"},
            {0,13,"s18"},
            {0,18,"s27"},
            {0,19,"s28"},
            {0,20,"s20"},
            {0,21,"s21"},
            {0,22,"s22"},
            {1,29,"acc"},
            {2,0,"s3"},
            {2,1,"s14"},
            {2,2,"s25"},
            {2,9,"s15"},
            {2,11,"s16"},
            {2,12,"s17"},
            {2,13,"s18"},
            {2,18,"s27"},
            {2,19,"s28"},
            {2,20,"s20"},
            {2,21,"s21"},
            {2,22,"s22"},
            {13,1,"s30"},
            {14,2,"s32"},
            {14,8,"s31"},
            {15,1,"s35"},
            {15,2,"s25"},
            {15,18,"s27"},
            {15,19,"s28"},
            {16,2,"s36"},
            {17,2,"s37"},
            {18,1,"s35"},
            {18,2,"s25"},
            {18,18,"s27"},
            {18,19,"s28"},
            {19,7,"s39"},
            {19,14,"s40"},
            {19,15,"s41"},
            {23,16,"s42"},
            {23,17,"s43"},
            {25,1,"s35"},
            {25,2,"s25"},
            {25,18,"s27"},
            {25,19,"s28"},
            {30,2,"s45"},
```

{30,7,"s46"},
{30,8,"s47"},
{31,1,"s35"},
{31,2,"s25"},
{31,18,"s27"},
{31,19,"s28"},
{32,1,"s35"},
{32,2,"s25"},
{32,18,"s27"},
{32,19,"s28"},
{33,10,"s51"},
{34,14,"s40"},
{34,15,"s41"},
{34,23,"s52"},
{34,24,"s53"},
{34,25,"s54"},
{34,26,"s55"},
{34,27,"s56"},
{34,28,"s57"},
{36,1,"s59"},
{37,1,"s35"},
{37,2,"s25"},
{37,18,"s27"},
{37,19,"s28"},
{38,7,"s61"},
{38,14,"s40"},
{38,15,"s41"},
{40,1,"s35"},
{40,2,"s25"},
{40,18,"s27"},
{40,19,"s28"},
{41,1,"s35"},
{41,2,"s25"},
{41,18,"s27"},
{41,19,"s28"},
{42,1,"s35"},
{42,2,"s25"},
{42,18,"s27"},
{42,19,"s28"},
{43,1,"s35"},
{43,2,"s25"},
{43,18,"s27"},
{43,19,"s28"},
{44,3,"s66"},
{44,14,"s40"},
{44,15,"s41"},
{45,20,"s20"},
{45,21,"s21"},
{45,22,"s22"},
{47,1,"s35"},
{47,2,"s25"},
{47,18,"s27"},
{47,19,"s28"},
{48,7,"s70"},
{48,14,"s40"},
{48,15,"s41"},
{49,3,"s71"},
{50,0,"s74"},
{50,4,"s73"},
{50,14,"s40"},
{50,15,"s41"},

{51,1,"s14"},
{51,2,"s25"},
{51,9,"s15"},
{51,11,"s16"},
{51,12,"s17"},
{51,13,"s18"},
{51,18,"s27"},
{51,19,"s28"},
{51,20,"s20"},
{51,21,"s21"},
{51,22,"s22"},
{52,1,"s35"},
{52,2,"s25"},
{52,18,"s27"},
{52,19,"s28"},
{53,1,"s35"},
{53,2,"s25"},
{53,18,"s27"},
{53,19,"s28"},
{54,1,"s35"},
{54,2,"s25"},
{54,18,"s27"},
{54,19,"s28"},
{55,1,"s35"},
{55,2,"s25"},
{55,18,"s27"},
{55,19,"s28"},
{56,1,"s35"},
{56,2,"s25"},
{56,18,"s27"},
{56,19,"s28"},
{57,1,"s35"},
{57,2,"s25"},
{57,18,"s27"},
{57,19,"s28"},
{58,7,"s82"},
{59,8,"s31"},
{60,3,"s83"},
{62,16,"s42"},
{62,17,"s43"},
{63,16,"s42"},
{63,17,"s43"},
{67,3,"s84"},
{68,1,"s85"},
{69,7,"s86"},
{69,14,"s40"},
{69,15,"s41"},
{73,1,"s35"},
{73,2,"s25"},
{73,18,"s27"},
{73,19,"s28"},
{76,14,"s40"},
{76,15,"s41"},
{77,14,"s40"},
{77,15,"s41"},
{78,14,"s40"},
{78,15,"s41"},
{79,14,"s40"},
{79,15,"s41"},
{80,14,"s40"},
{80,15,"s41"},

```
{81,14,"s40"},
{81,15,"s41"},
{82,1,"s35"},
{82,2,"s25"},
{82,18,"s27"},
{82,19,"s28"},
{83,5,"s90"},
{84,5,"s90"},
{85,0,"s94"},
{85,4,"s93"},
{87,0,"s74"},
{87,4,"s73"},
{87,14,"s40"},
{87,15,"s41"},
{88,7,"s96"},
{90,0,"s3"},
{90,1,"s14"},
{90,2,"s25"},
{90,9,"s15"},
{90,11,"s16"},
{90,12,"s17"},
{90,13,"s18"},
{90,18,"s27"},
{90,19,"s28"},
{90,20,"s20"},
{90,21,"s21"},
{90,22,"s22"},
{93,20,"s20"},
{93,21,"s21"},
{93,22,"s22"},
{96,1,"s35"},
{96,2,"s25"},
{96,18,"s27"},
{96,19,"s28"},
{97,6,"s100"},
{98,1,"s101"},
{99,3,"s102"},
{99,14,"s40"},
{99,15,"s41"},
{101,0,"s94"},
{101,4,"s93"},
{102,5,"s90"} };
```

```
insert insert3[][1] = {
 {3,6,"r3"},
{3,29,"r3"},
{4,0,"r12"},
{4,1,"r12"},
{4,2,"r12"},
{4,9,"r12"},
{4,11,"r12"},
{4,12,"r12"},
{4,13,"r12"},
{4,18,"r12"},
{4,19,"r12"},
{4,20,"r12"},
{4,21,"r12"},
{4,22,"r12"},
{5,0,"r13"},
{5,1,"r13"},
{5,2,"r13"},
```

```
{5,9,"r13"},
{5,11,"r13"},
{5,12,"r13"},
{5,13,"r13"},
{5,18,"r13"},
{5,19,"r13"},
{5,20,"r13"},
{5,21,"r13"},
{5,22,"r13"},
{6,0,"r14"},
{6,1,"r14"},
{6,2,"r14"},
{6,9,"r14"},
{6,11,"r14"},
{6,12,"r14"},
{6,13,"r14"},
{6,18,"r14"},
{6,19,"r14"},
{6,20,"r14"},
{6,21,"r14"},
{6,22,"r14"},
{7,0,"r15"},
{7,1,"r15"},
{7,2,"r15"},
{7,9,"r15"},
{7,11,"r15"},
{7,12,"r15"},
{7,13,"r15"},
{7,18,"r15"},
{7,19,"r15"},
{7,20,"r15"},
{7,21,"r15"},
{7,22,"r15"},
{8,0,"r16"},
{8,1,"r16"},
{8,2,"r16"},
{8,9,"r16"},
{8,11,"r16"},
{8,12,"r16"},
{8,13,"r16"},
{8,18,"r16"},
{8,19,"r16"},
{8,20,"r16"},
{8,21,"r16"},
{8,22,"r16"},
{9,0,"r17"},
{9,1,"r17"},
{9,2,"r17"},
{9,9,"r17"},
{9,11,"r17"},
{9,12,"r17"},
{9,13,"r17"},
{9,18,"r17"},
{9,19,"r17"},
{9,20,"r17"},
{9,21,"r17"},
{9,22,"r17"},
{10,0,"r18"},
{10,1,"r18"},
{10,2,"r18"},
{10,9,"r18"},
```

```
{10,11,"r18"},
{10,12,"r18"},
{10,13,"r18"},
{10,18,"r18"},
{10,19,"r18"},
{10,20,"r18"},
{10,21,"r18"},
{10,22,"r18"},
{11,0,"r19"},
{11,1,"r19"},
{11,2,"r19"},
{11,9,"r19"},
{11,11,"r19"},
{11,12,"r19"},
{11,13,"r19"},
{11,18,"r19"},
{11,19,"r19"},
{11,20,"r19"},
{11,21,"r19"},
{11,22,"r19"},
{12,0,"r20"},
{12,1,"r20"},
{12,2,"r20"},
{12,9,"r20"},
{12,11,"r20"},
{12,12,"r20"},
{12,13,"r20"},
{12,18,"r20"},
{12,19,"r20"},
{12,20,"r20"},
{12,21,"r20"},
{12,22,"r20"},
{14,7,"r35"},
{14,14,"r35"},
{14,15,"r35"},
{14,16,"r35"},
{14,17,"r35"},
{20,1,"r40"},
{21,1,"r41"},
{22,1,"r42"},
{23,0,"r31"},
{23,3,"r31"},
{23,4,"r31"},
{23,7,"r31"},
{23,10,"r31"},
{23,14,"r31"},
{23,15,"r31"},
{23,23,"r31"},
{23,24,"r31"},
{23,25,"r31"},
{23,26,"r31"},
{23,27,"r31"},
{23,28,"r31"},
{24,0,"r34"},
{24,3,"r34"},
{24,4,"r34"},
{24,7,"r34"},
{24,10,"r34"},
{24,14,"r34"},
{24,15,"r34"},
{24,16,"r34"},
```

{24,17,"r34"},
{24,23,"r34"},
{24,24,"r34"},
{24,25,"r34"},
{24,26,"r34"},
{24,27,"r34"},
{24,28,"r34"},
{26,0,"r37"},
{26,3,"r37"},
{26,4,"r37"},
{26,7,"r37"},
{26,10,"r37"},
{26,14,"r37"},
{26,15,"r37"},
{26,16,"r37"},
{26,17,"r37"},
{26,23,"r37"},
{26,24,"r37"},
{26,25,"r37"},
{26,26,"r37"},
{26,27,"r37"},
{26,28,"r37"},
{27,0,"r38"},
{27,3,"r38"},
{27,4,"r38"},
{27,7,"r38"},
{27,10,"r38"},
{27,14,"r38"},
{27,15,"r38"},
{27,16,"r38"},
{27,17,"r38"},
{27,23,"r38"},
{27,24,"r38"},
{27,25,"r38"},
{27,26,"r38"},
{27,27,"r38"},
{27,28,"r38"},
{28,0,"r39"},
{28,3,"r39"},
{28,4,"r39"},
{28,7,"r39"},
{28,10,"r39"},
{28,14,"r39"},
{28,15,"r39"},
{28,16,"r39"},
{28,17,"r39"},
{28,23,"r39"},
{28,24,"r39"},
{28,25,"r39"},
{28,26,"r39"},
{28,27,"r39"},
{28,28,"r39"},
{29,6,"r2"},
{29,29,"r2"},
{35,0,"r35"},
{35,3,"r35"},
{35,4,"r35"},
{35,7,"r35"},
{35,10,"r35"},
{35,14,"r35"},
{35,15,"r35"},

```
{35,16,"r35"},
{35,17,"r35"},
{35,23,"r35"},
{35,24,"r35"},
{35,25,"r35"},
{35,26,"r35"},
{35,27,"r35"},
{35,28,"r35"},
{39,0,"r49"},
{39,1,"r49"},
{39,2,"r49"},
{39,9,"r49"},
{39,11,"r49"},
{39,12,"r49"},
{39,13,"r49"},
{39,18,"r49"},
{39,19,"r49"},
{39,20,"r49"},
{39,21,"r49"},
{39,22,"r49"},
{46,0,"r21"},
{46,1,"r21"},
{46,2,"r21"},
{46,9,"r21"},
{46,11,"r21"},
{46,12,"r21"},
{46,13,"r21"},
{46,18,"r21"},
{46,19,"r21"},
{46,20,"r21"},
{46,21,"r21"},
{46,22,"r21"},
{61,0,"r28"},
{61,1,"r28"},
{61,2,"r28"},
{61,9,"r28"},
{61,11,"r28"},
{61,12,"r28"},
{61,13,"r28"},
{61,18,"r28"},
{61,19,"r28"},
{61,20,"r28"},
{61,21,"r28"},
{61,22,"r28"},
{62,0,"r29"},
{62,3,"r29"},
{62,4,"r29"},
{62,7,"r29"},
{62,10,"r29"},
{62,14,"r29"},
{62,15,"r29"},
{62,23,"r29"},
{62,24,"r29"},
{62,25,"r29"},
{62,26,"r29"},
{62,27,"r29"},
{62,28,"r29"},
{63,0,"r30"},
{63,3,"r30"},
{63,4,"r30"},
{63,7,"r30"},
```

```
{63,10,"r30"},
{63,14,"r30"},
{63,15,"r30"},
{63,23,"r30"},
{63,24,"r30"},
{63,25,"r30"},
{63,26,"r30"},
{63,27,"r30"},
{63,28,"r30"},
{64,0,"r32"},
{64,3,"r32"},
{64,4,"r32"},
{64,7,"r32"},
{64,10,"r32"},
{64,14,"r32"},
{64,15,"r32"},
{64,16,"r32"},
{64,17,"r32"},
{64,23,"r32"},
{64,24,"r32"},
{64,25,"r32"},
{64,26,"r32"},
{64,27,"r32"},
{64,28,"r32"},
{65,0,"r33"},
{65,3,"r33"},
{65,4,"r33"},
{65,7,"r33"},
{65,10,"r33"},
{65,14,"r33"},
{65,15,"r33"},
{65,16,"r33"},
{65,17,"r33"},
{65,23,"r33"},
{65,24,"r33"},
{65,25,"r33"},
{65,26,"r33"},
{65,27,"r33"},
{65,28,"r33"},
{66,0,"r36"},
{66,3,"r36"},
{66,4,"r36"},
{66,7,"r36"},
{66,10,"r36"},
{66,14,"r36"},
{66,15,"r36"},
{66,16,"r36"},
{66,17,"r36"},
{66,23,"r36"},
{66,24,"r36"},
{66,25,"r36"},
{66,26,"r36"},
{66,27,"r36"},
{66,28,"r36"},
{70,0,"r23"},
{70,1,"r23"},
{70,2,"r23"},
{70,7,"r23"},
{70,9,"r23"},
{70,11,"r23"},
{70,12,"r23"},
```

```
{70,13,"r23"},
{70,18,"r23"},
{70,19,"r23"},
{70,20,"r23"},
{70,21,"r23"},
{70,22,"r23"},
{71,0,"r24"},
{71,1,"r24"},
{71,2,"r24"},
{71,9,"r24"},
{71,11,"r24"},
{71,12,"r24"},
{71,13,"r24"},
{71,18,"r24"},
{71,19,"r24"},
{71,20,"r24"},
{71,21,"r24"},
{71,22,"r24"},
{72,3,"r8"},
{74,3,"r10"},
{75,0,"r25"},
{75,1,"r25"},
{75,2,"r25"},
{75,9,"r25"},
{75,11,"r25"},
{75,12,"r25"},
{75,13,"r25"},
{75,18,"r25"},
{75,19,"r25"},
{75,20,"r25"},
{75,21,"r25"},
{75,22,"r25"},
{76,3,"r43"},
{76,7,"r43"},
{76,10,"r43"},
{77,3,"r44"},
{77,7,"r44"},
{77,10,"r44"},
{78,3,"r45"},
{78,7,"r45"},
{78,10,"r45"},
{79,3,"r46"},
{79,7,"r46"},
{79,10,"r46"},
{80,3,"r47"},
{80,7,"r47"},
{80,10,"r47"},
{81,3,"r48"},
{81,7,"r48"},
{81,10,"r48"},
{86,0,"r22"},
{86,1,"r22"},
{86,2,"r22"},
{86,9,"r22"},
{86,11,"r22"},
{86,12,"r22"},
{86,13,"r22"},
{86,18,"r22"},
{86,19,"r22"},
{86,20,"r22"},
{86,21,"r22"},
```

```
            {86,22,"r22"},
            {89,0,"r27"},
            {89,1,"r27"},
            {89,2,"r27"},
            {89,9,"r27"},
            {89,11,"r27"},
            {89,12,"r27"},
            {89,13,"r27"},
            {89,18,"r27"},
            {89,19,"r27"},
            {89,20,"r27"},
            {89,21,"r27"},
            {89,22,"r27"},
            {91,0,"r4"},
            {91,1,"r4"},
            {91,2,"r4"},
            {91,9,"r4"},
            {91,11,"r4"},
            {91,12,"r4"},
            {91,13,"r4"},
            {91,18,"r4"},
            {91,19,"r4"},
            {91,20,"r4"},
            {91,21,"r4"},
            {91,22,"r4"},
            {92,3,"r5"},
            {94,3,"r7"},
            {95,3,"r9"},
            {100,0,"r11"},
            {100,1,"r11"},
            {100,2,"r11"},
            {100,9,"r11"},
            {100,11,"r11"},
            {100,12,"r11"},
            {100,13,"r11"},
            {100,18,"r11"},
            {100,19,"r11"},
            {100,20,"r11"},
            {100,21,"r11"},
            {100,22,"r11"},
            {103,3,"r6"},
            {104,0,"r26"},
            {104,1,"r26"},
            {104,2,"r26"},
            {104,9,"r26"},
            {104,11,"r26"},
            {104,12,"r26"},
            {104,13,"r26"},
            {104,18,"r26"},
            {104,19,"r26"},
            {104,20,"r26"},
            {104,21,"r26"},
            {104,22,"r26"} };
goto_table[0][1] = 1;
goto_table[0][2] = 4;
goto_table[0][8] = 2;
goto_table[0][9] = 5;
goto_table[0][10] = 6;
goto_table[0][11] = 10;
goto_table[0][12] = 7;
goto_table[0][13] = 8;
```

```
goto_table[0][14] = 9;
goto_table[0][15] = 11;
goto_table[0][16] = 19;
goto_table[0][17] = 23;
goto_table[0][18] = 24;
goto_table[0][19] = 26;
goto_table[0][20] = 13;
goto_table[0][22] = 12;
goto_table[2][1] = 29;
goto_table[2][2] = 4;
goto_table[2][8] = 2;
goto_table[2][9] = 5;
goto_table[2][10] = 6;
goto_table[2][11] = 10;
goto_table[2][12] = 7;
goto_table[2][13] = 8;
goto_table[2][14] = 9;
goto_table[2][15] = 11;
goto_table[2][16] = 19;
goto_table[2][17] = 23;
goto_table[2][18] = 24;
goto_table[2][19] = 26;
goto_table[2][20] = 13;
goto_table[2][22] = 12;
goto_table[15][16] = 34;
goto_table[15][17] = 23;
goto_table[15][18] = 24;
goto_table[15][19] = 26;
goto_table[15][21] = 33;
goto_table[18][16] = 38;
goto_table[18][17] = 23;
goto_table[18][18] = 24;
goto_table[18][19] = 26;
goto_table[25][16] = 44;
goto_table[25][17] = 23;
goto_table[25][18] = 24;
goto_table[25][19] = 26;
goto_table[31][16] = 48;
goto_table[31][17] = 23;
goto_table[31][18] = 24;
goto_table[31][19] = 26;
goto_table[32][5] = 49;
goto_table[32][16] = 50;
goto_table[32][17] = 23;
goto_table[32][18] = 24;
goto_table[32][19] = 26;
goto_table[36][10] = 58;
goto_table[37][16] = 34;
goto_table[37][17] = 23;
goto_table[37][18] = 24;
goto_table[37][19] = 26;
goto_table[37][21] = 60;
goto_table[40][17] = 62;
goto_table[40][18] = 24;
goto_table[40][19] = 26;
goto_table[41][17] = 63;
goto_table[41][18] = 24;
goto_table[41][19] = 26;
goto_table[42][18] = 64;
goto_table[42][19] = 26;
goto_table[43][18] = 65;
```

```
goto_table[43][19] = 26;
goto_table[45][3] = 67;
goto_table[45][20] = 68;
goto_table[47][16] = 69;
goto_table[47][17] = 23;
goto_table[47][18] = 24;
goto_table[47][19] = 26;
goto_table[50][6] = 72;
goto_table[51][2] = 4;
goto_table[51][8] = 75;
goto_table[51][9] = 5;
goto_table[51][10] = 6;
goto_table[51][11] = 10;
goto_table[51][12] = 7;
goto_table[51][13] = 8;
goto_table[51][14] = 9;
goto_table[51][15] = 11;
goto_table[51][16] = 19;
goto_table[51][17] = 23;
goto_table[51][18] = 24;
goto_table[51][19] = 26;
goto_table[51][20] = 13;
goto_table[51][22] = 12;
goto_table[52][16] = 76;
goto_table[52][17] = 23;
goto_table[52][18] = 24;
goto_table[52][19] = 26;
goto_table[53][16] = 77;
goto_table[53][17] = 23;
goto_table[53][18] = 24;
goto_table[53][19] = 26;
goto_table[54][16] = 78;
goto_table[54][17] = 23;
goto_table[54][18] = 24;
goto_table[54][19] = 26;
goto_table[55][16] = 79;
goto_table[55][17] = 23;
goto_table[55][18] = 24;
goto_table[55][19] = 26;
goto_table[56][16] = 80;
goto_table[56][17] = 23;
goto_table[56][18] = 24;
goto_table[56][19] = 26;
goto_table[57][16] = 81;
goto_table[57][17] = 23;
goto_table[57][18] = 24;
goto_table[57][19] = 26;
goto_table[73][16] = 87;
goto_table[73][17] = 23;
goto_table[73][18] = 24;
goto_table[73][19] = 26;
goto_table[82][16] = 34;
goto_table[82][17] = 23;
goto_table[82][18] = 24;
goto_table[82][19] = 26;
goto_table[82][21] = 88;
goto_table[83][7] = 89;
goto_table[84][7] = 91;
goto_table[85][4] = 92;
goto_table[87][6] = 95;
goto_table[90][1] = 97;
```

```
goto_table[90][2] = 4;
goto_table[90][8] = 2;
goto_table[90][9] = 5;
goto_table[90][10] = 6;
goto_table[90][11] = 10;
goto_table[90][12] = 7;
goto_table[90][13] = 8;
goto_table[90][14] = 9;
goto_table[90][15] = 11;
goto_table[90][16] = 19;
goto_table[90][17] = 23;
goto_table[90][18] = 24;
goto_table[90][19] = 26;
goto_table[90][20] = 13;
goto_table[90][22] = 12;
goto_table[93][20] = 98;
goto_table[96][16] = 99;
goto_table[96][17] = 23;
goto_table[96][18] = 24;
goto_table[96][19] = 26;
goto_table[101][4] = 103;
goto_table[102][7] = 104;
    for (int i = 0; i < ROWS; i++) {
        for (int j = 0; j < COLS; j++) {
            strcpy(matrix[i][j].type, "E"); // Empty cell
        }
    }
    for (int i = 0; i < sizeof(insert2) / sizeof(insert2[0]); i++) {


        strcpy(matrix[insert2[i]->row][insert2[i]->col].type, insert2[i]->data);

    }
    for (int i = 0; i < sizeof(insert3) / sizeof(insert3[0]); i++) {


        strcpy(matrix[insert3[i]->row][insert3[i]->col].type, insert3[i]->data);

    }
}
```

Stack.h

```
#define STACK_MAX_SIZE 100
#include "Tree.h"
enum { FALSE, TRUE };
typedef unsigned char Boolean;

typedef struct StackNode {
    void* data;
    struct StackNode* next;
} StackNode;
```

```
typedef struct Stack {
    StackNode* top;
} stack;
//======= PROTOTYPE =====================
void stack_init(stack* s);
Boolean  stack_empty(stack*);
Boolean  stack_full(stack*);
void* stack_pop(stack* s);
void stack_push(stack* s, void* info);
void* stack_top(stack* s);
void stackFree(stack* stack);
```

Stack.c

```
// my_stack.c
#include <stdio.h>
#include <stdlib.h>
#include "stack.h"


/********************** Function bodies **************/

// Initialize the stack
void stack_init(stack* stack) {
    stack->top = NULL;
}
Boolean stack_empty(stack* s) {
    return s->top == -1 ? 1 : 0;
}
Boolean  stack_full(stack* s) {
    return s->top == STACK_MAX_SIZE ? 1 : 0;
}

// Push an ASTNode onto the stack
void stack_push(stack* stack, void* data) {
    StackNode* newNode = (StackNode*)malloc(sizeof(StackNode));
    if (newNode == NULL) {
        fprintf(stderr, "Failed to allocate memory for stack node.\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->next = stack->top;
    stack->top = newNode;
}

// Pop an ASTNode from the stack
void* stack_pop(stack* stack) {
    if (stack->top == NULL) {
        return NULL;
    }
    StackNode* topNode = stack->top;
    void* data = topNode->data;
    stack->top = topNode->next;
    free(topNode);
    return data;
}

// Peek at the top ASTNode of the stack
void* stack_top(stack* stack) {
```

```
    if (stack->top == NULL) {
        return NULL;
    }
    return stack->top->data;
}
void stackFree(stack* stack) {
    while (stack->top != NULL) {
        StackNode* temp = stack->top;
        stack->top = stack->top->next;
        free(temp);
    }
}
```

Tree.h

```
#pragma once
typedef struct ASTNode {
    char* type; // Type of node (e.g., "Expression", "Statement")
    char* data;
    char* value; // Literal value, for nodes representing literals or identifiers
    int scope;
    struct ASTNode** children; // Array of pointers to child nodes
    int childrenCount; // Number of children
} ASTNode;

ASTNode* createASTNode(const char* type, const char* value,char* data,int scope);
void addChild(ASTNode* parent, ASTNode* child);
void printAST(ASTNode* node, int level);
void freeAST(ASTNode* node);
```

Tree.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Tree.h"



// Create a new Node
ASTNode* createASTNode(const char* type, const char* value,char* data,int scope) {
    ASTNode* node = (ASTNode*)malloc(sizeof(ASTNode));
    node->type = _strdup(type); // Copy the type
    node->value = value ? _strdup(value) : NULL; // Copy the value, if provided
    node->data = _strdup(data);
    node->children = NULL; // Initially, no children
    node->scope= scope;
    node->childrenCount = 0; // Initially, no children
    return node;
}
void addChild(ASTNode* parent, ASTNode* child) {
    if (!parent || !child) return; // Safety check

    // Resize the children array to accommodate the new child
    parent->childrenCount++;
    parent->children = (ASTNode**)realloc(parent->children, sizeof(ASTNode*) * parent->childrenCount);
```

```
    parent->children[parent->childrenCount - 1] = child; // Add the new child
}
void printAST(ASTNode* node, int level) {
    if (!node) return; // Base case

    // Indentation for levels
    for (int i = 0; i < level; i++) {
        printf("  ");
    }

    // Print the current node
    if(node->type)
     printf(" type: %s", node->type);
    if (node->value) {
        printf(" (%s)", node->value);
        printf(" scope: %d ", node->scope);
    }
    printf("\n");

    // Recursively print children
    for (int i = 0; i < node->childrenCount; i++) {
        printAST(node->children[i], level + 1);
    }
}
void freeAST(ASTNode* node) {
    if (!node) return; // Base case

    // Free children recursively
    for (int i = 0; i < node->childrenCount; i++) {
        freeAST(node->children[i]);
    }

    // Free the current node
    free(node->type);
    if (node->value) {
        free(node->value);
    }
    if (node->children) {
        free(node->children);
    }
    free(node);
}
```

SymbolTable.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "Tree.h"
typedef enum {
    SYMBOL_TYPE_VARIABLE,    // For variable declarations
    SYMBOL_TYPE_FUNCTION,    // For function declarations
    SYMBOL_TYPE_TYPE_SPECIFIER, // For type specifiers like int, String, double
    SYMBOL_TYPE_PARAMETER,   // For formal function parameters
    SYMBOL_TYPE_EXPRESSION,  // For generic expressions
    SYMBOL_TYPE_TERM,        // For terms within expressions
    SYMBOL_TYPE_FACTOR,      // For factors within terms
    SYMBOL_TYPE_CONDITION,   // For conditions in if, while, and for statements
    SYMBOL_TYPE_BLOCK,       // For blocks of statements
```

```
    SYMBOL_TYPE_STATEMENT,   // For generic statements (if applicable)
    SYMBOL_TYPE_IF_STATEMENT,// For if statements
    SYMBOL_TYPE_FOR_STATEMENT, // For for loops
    SYMBOL_TYPE_WHILE_STATEMENT, // For while loops
    SYMBOL_TYPE_FUNCTION_CALL, // For function call expressions
    SYMBOL_TYPE_RETURN_STATEMENT, // For return statements
    SYMBOL_TYPE_ASSIGNMENT, // For assignment statements
    SYMBOL_TYPE_DECLARATION // For declarations (variables and possibly functions)
} SymbolType;

typedef struct SymbolTableEntry {
    char* name;
    SymbolType type;
    char* dataType;
    char* scopeIdentifier; // String to represent the scope
    char* data;
    struct SymbolTableEntry* next;
} SymbolTableEntry;

#define SYMBOL_TABLE_SIZE 100

typedef struct SymbolTable {
    SymbolTableEntry* entries[SYMBOL_TABLE_SIZE];
} SymbolTable;

unsigned int hash(const char* key);
void symbolTableInit(SymbolTable* table);
SymbolTableEntry* symbolTableInsert(SymbolTable* table, const char* name, SymbolType type, const
char* dataType,char* data, const int scopeIdentifier);
SymbolTableEntry* symbolTableLookup(SymbolTable* table, const char* name, const int scopeIdentifier);
void symbolTableSetParams(SymbolTable* table,ASTNode* node, const char* name, const int
scopeIdentifier);
void symbolTableFree(SymbolTable* table);
void traverseASTAndBuildSymbolTable(ASTNode* node, SymbolTable* table);
void buildSymbolTableFromAST(ASTNode* root, SymbolTable* table);
void printSymbolTable(SymbolTable* table);
int checkentry(SymbolTable* table, char* name, int scope);
void findsymbols(SymbolTable* table, ASTNode* node);
SymbolTableEntry* findparams(SymbolTable* table, ASTNode* node);
```

SymbolTable.c

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "SymbolTable.h"



unsigned int hash(const char* key) {
    unsigned int hash = 0;
    for (; *key; key++) {
        hash = hash * 33 + *key;
    }
    return hash % SYMBOL_TABLE_SIZE;
}

void symbolTableInit(SymbolTable* table) {
```

```
    memset(table->entries, 0, sizeof(SymbolTableEntry*) * SYMBOL_TABLE_SIZE);
}

SymbolTableEntry* symbolTableInsert(SymbolTable* table, const char* name, SymbolType type, const
char* dataType,char* data, const int scopeIdentifier) {
    unsigned int index = hash(name);
    SymbolTableEntry* entry = (SymbolTableEntry*)malloc(sizeof(SymbolTableEntry));
    if (!entry) {
        fprintf(stderr, "Memory allocation failed for symbol table entry.\n");
        exit(1);
    }
    entry->name = _strdup(name);
    entry->type = type;
    entry->dataType = _strdup(dataType);
    entry->data = _strdup(data);
    entry->scopeIdentifier = scopeIdentifier;
    entry->next = table->entries[index];
    table->entries[index] = entry;
    return entry;
}

SymbolTableEntry* symbolTableLookup(SymbolTable* table, const char* name, const int scopeIdentifier) {
    unsigned int index = hash(name);
    for (SymbolTableEntry* entry = table->entries[index]; entry != NULL; entry = entry->next) {
        if (strcmp(entry->name, name) == 0 && entry->scopeIdentifier<=scopeIdentifier ) {
            return entry; // Found
        }
    }
    return NULL; // Not found
}

void symbolTableFree(SymbolTable* table) {
    for (int i = 0; i < SYMBOL_TABLE_SIZE; i++) {
        SymbolTableEntry* entry = table->entries[i];
        while (entry) {
            SymbolTableEntry* temp = entry;
            entry = entry->next;
            free(temp->name);
            free(temp->dataType);
            free(temp->data);
            free(temp);
        }
    }
}
void traverseASTAndBuildSymbolTable(ASTNode* node, SymbolTable* table) {
    if (!node) return; // Base case: node is NULL
    char* name = NULL;
    SymbolTableEntry* params=NULL;


    if (strcmp(node->type, "function_declaration") == 0) {
        // Iterate over the children to find the variable name and data type
        for (int i = 0; i < node->childrenCount ; i++) {
            ASTNode* child = node->children[i];



            findsymbols(table, child);
            if (strcmp(child->type, "formal_parameters") == 0) {

                findsymbols(table, child);
```

```
            for (int i = 0; i < child->childrenCount; ++i) {
                child = child->children[i];
                i = 0;
                findsymbols(table, child);
            }
        }
    }

}


if (strcmp(node->type, "declaration") == 0) {
    // Assuming the variable name and type are determined by traversing the children
    const char* varName = NULL;
    const char* dataType = NULL;
    const char* data = NULL;
    // Iterate over the children to find the variable name and data type
    for (int i = 0; i < node->childrenCount; i++) {
        ASTNode* child = node->children[i];

        if (strcmp(child->type, "expression") == 0) {

            for (int i = 0; i < child->childrenCount; ++i) {
                child = child->children[i];
                i = -1;
                if (data!=NULL) {
                    strcpy(data,"error");
                }
                // Look for the assignment operator "=" to find the start of the right side
                else if (strcmp(child->value, "number") == 0|| strcmp(child->value, "string") == 0||
strcmp(child->value, "ID") == 0) {
                    if (data != NULL) {
                        data = realloc(data, sizeof(char) * (strlen(data) + strlen(child->data)));
                        strcat(data, child->data);
                    }
                    else
                        data = child->data; // Return the node representing the right side
                    printf("%d %s ",child->scope, data);
                }

            }
        }
        if ( strcmp(child->type, "Terminal") == 0) {
            // Next sibling node should be the variable name

            varName = child->data;

        }
        if (strcmp(child->type, "type_specifier") == 0) {

            dataType = child->children[0]->value;
        }
        if (varName && dataType && checkentry(table,varName,child->scope)==-1) {
            symbolTableInsert(table, varName, SYMBOL_TYPE_VARIABLE, dataType,data, node->scope);
        }

    }


}
```

```
    // Continue traversing the tree
    for (int i = 0; i < node->childrenCount; i++) {
        traverseASTAndBuildSymbolTable(node->children[i], table, node->scope);
    }
}

void buildSymbolTableFromAST(ASTNode* root, SymbolTable* table) {
    traverseASTAndBuildSymbolTable(root, table); // Start the traversal from the root
}
void printSymbolTable(SymbolTable* table) {
    printf("Symbol Table:\n");
    for (int i = 0; i < SYMBOL_TABLE_SIZE; i++) {
        SymbolTableEntry* entry = table->entries[i];
        int j = 0;
        while (entry != NULL) {
            printf("Index: %d, Name: %s, Type: %s, DataType: %s, Data: %s, Scope: %d \n",
                i,
                entry->name,
                entry->type == SYMBOL_TYPE_VARIABLE ? "Variable" : "Function",
                entry->dataType,
                entry->data,
                entry->scopeIdentifier);
            entry = entry->next;

        }
    }
}
int checkentry(SymbolTable* table, char* name,int scope) {
    if (symbolTableLookup(table, name, scope) != NULL)
        return 1;
    return -1;
}

void findsymbols(SymbolTable* table, ASTNode* node) {
    if (node == NULL)
        return;
    const char* varName = NULL;
    const char* dataType = NULL;
    const char* data = NULL;
    if(node->childrenCount==0)
    if (strcmp(node->value, "ID") == 0) {
        // Next sibling node should be the variable name
        varName = node->data;
        dataType = node->type;

        if (varName && dataType&&checkentry(table,varName,node->scope)==-1) {
            symbolTableInsert(table, varName, SYMBOL_TYPE_FUNCTION, dataType,varName,
node->scope);
        }
    }
    // Iterate over the children to find the variable name and data type
    for (int i = 0; i < node->childrenCount; i++) {
        ASTNode* child = node->children[i];
        //findsymbols(table,child)
        if (strcmp(child->type, "factor_base") == 0) {
            data = child->children[0]->data;
        }
        if (strcmp(child->type, "Terminal") == 0&& strcmp(child->value, "ID") == 0) {
            // Next siblchilding node should be the variable name
            varName = child->data;
```

```
        }
        if (strcmp(child->type, "type_specifier") == 0) {

            dataType = child->children[0]->value;
        }
        if (varName && dataType && checkentry(table, varName, node->scope) == -1) {
            symbolTableInsert(table, varName, SYMBOL_TYPE_VARIABLE, dataType,data, node->scope);
        }
    }
}
```

SemanticAnalysis.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "SymbolTable.h"

int checkVariableType(SymbolTable* table, const char* varName, const char* expectedType,  int scope);
void semanticAnalysis(ASTNode* node, SymbolTable* table,  int currentScope);
void perform(ASTNode* root);
```

SemanticAnalysis.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
//#include "SymbolTable.h"
#include "SemanticAnalysis.h"
#include "Tree.h"



// Function to check the type of a variable
int checkVariableType(SymbolTable* table, const char* varName, const char* expectedType, int scope) {
    SymbolTableEntry* entry = symbolTableLookup(table, varName, scope);
    if (entry != NULL) {
        if(strcmp(entry->dataType, expectedType) == 0)
            return 1;
    }
    return 0;
}
int checkassigntment(SymbolTable* table, const char* varName, int scope) {
    SymbolTableEntry* entry = symbolTableLookup(table, varName, scope);
    if (strcmp(entry->dataType, "INT") == 0) {
        if(entry->data!=NULL&& (entry->data[0] - '0') >= 0 && (entry->data[0] - '0') <= 9)
        return 1;
        if (entry->data == NULL)
            return 1;
    }
    if (strcmp(entry->dataType, "String") == 0 && entry->data[0] =="") {
        return 1;
    }
    return 0;
```

```
}


void semanticAnalysis(ASTNode* node, SymbolTable* table,  int currentScope) {
   if (!node) return;

   if (strcmp(node->value, "ID")==0&&node->data) {
      SymbolTableEntry* entry = symbolTableLookup(table, node->data, currentScope);
      if (!entry) {
         printf("Semantic Error0: Variable %s is not declared in the current scope %d.\n", node->data,
currentScope);
         return;
      }
      if (entry->type == SYMBOL_TYPE_VARIABLE) {
         // Additional checks for the variable can go here (type checks, etc.)
         if (checkVariableType(table, node->data,entry->dataType, node->scope) == 0) {
            printf("Semantic Error1: Variable %s is not the correct type in the current scope %d.\n",
node->data, currentScope);
         }
         else if (checkassigntment(table, node->data,node->scope) == 0) {
            printf("Semantic Error2: Variable %s is not the correct type in the current scope %d.\n",
node->data, currentScope);
         }
      }
      if (entry->type == SYMBOL_TYPE_FUNCTION) {

      }
   }
   // Check for other node types (function calls, operations, etc.) here...
   if (strcmp(node->type, "function")==0) {
      SymbolTableEntry* entry = symbolTableLookup(table, node->data, 1);
      if (!entry) {
         printf("Semantic Error: function %s is not declared.\n", node->data);
      }

   }
   // Recursively analyze children nodes
   for (int i = 0; i < node->childrenCount; i++) {
      semanticAnalysis(node->children[i], table, node->children[i]->scope);
   }
}
// Main function to kick off the analysis
void perform(ASTNode* root) {
   SymbolTable table;
   symbolTableInit(&table);
   buildSymbolTableFromAST(root, &table);
   semanticAnalysis(root, &table, 0); // Start analysis from the root with global scope
   symbolTableFree(&table);
}
```

CodeGeneration.h

```
#pragma once

#include <stdio.h>

#include <stdbool.h>

#include "SemanticAnalysis.h"

#include "Lexer.h"

#include "Tree.h"

#define NUM_OF_REGISTERS 8

#define LABEL_FORMAT "$$L%d"

/* ---------- Structs ---------- */

// Struct of a register in the code generator's register array

typedef struct Register

{

	char name[2]; // The name of the register for target code output

	char high[2];

	char low[2];

	bool inuse; // Whether the register is currently in use or not

} Register;

// Struct of the code generator

typedef struct Code_Generator

{

	Register registers[NUM_OF_REGISTERS]; // Array of registers to be used in the code generation
process

	FILE* dest_file; // A pointer to the output file for the generated code
```

```
} Code_Generator;

Code_Generator* generator;

SymbolTableEntry** vars;
int SizeVars;

// Create a new code generator and points the compiler's code generator to it
void code_generator_create();
// Frees the memory allocated for the code generator of the compiler
void code_generator_destroy();
// Initializes the compiler's code generator
void code_generator_init();
// Searches for a free register in the code generator registers array.
// If found, marks it as inuse and returns the index of that register.
// If not found, terminates the compiler.
int code_generator_register_alloc();
// Marks the register in index r in the code generator registers array as unused
void code_generator_register_free(int r);
// Returns the register name of register r in the code generator registers array
char* code_generator_register_name(int r);
// Allocates a new lable name and returns a pointer to it
char* code_generator_label_create();
// Performs the right address computation for the given symbol according to its
// place in the program (global / local),
// and returns a string that represents that address.
// If the entry is NULL, returns NULL.
char* code_generator_symbol_address(SymbolTableEntry* entry);
```

```
// Outputs the given formated string to the target file
void code_generator_output(char* format, ...);
// Outputs the data segment of the program to the target file
void code_generator_output_data_segment(SymbolTable* table);
// Generates the assembly code for the given parse tree
void code_generator_generate(SymbolTable* table,ASTNode* parse_tree);
// Generates a block (BLOCK)
void code_generator_block(ASTNode* block);
// Generates a statement (STMT)
void code_generator_stmt(ASTNode* stmt);
// Generates a variable declaration statement (DECL)
void code_generator_decl(ASTNode* decl);
// Generates an assignment statement (ASSIGN)
void code_generator_assign(ASTNode* assign);

// Generates a while statement (WHILE)
void code_generator_while(ASTNode* _while);
// Generates an expression
void code_generator_expression(ASTNode* expr);
// Generates a binary expression of the form: Expr -> Expr binary_operator Expr
void code_generator_binary_expression(int left_register, int right_register, TokensType op);

void code_generator_if_statement(ASTNode* _if);

void code_generator_condition(ASTNode* condition);

void code_generator_function(ASTNode* func);
```

```
void code_generator_funCall(ASTNode* func);

void code_generator_for(ASTNode* _for);

void code_generator_print(ASTNode* print);

void code_generator_input(ASTNode* input);
```

CodeGeneration.c

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdarg.h>
#include <stdbool.h>
#include "SymbolTable.h"
#include "Lexer.h"
#include "Tree.h"
#include "CodeGeneration.h"

// Create a new code generator and points the compiler's code generator to it
void code_generator_create() {
	generator = (Code_Generator*)calloc(1, sizeof(Code_Generator));
	if (generator == NULL) {
	printf("%s", "Error! cant create a output file");
```

```
        exit(1);
        }

}
// Frees the memory allocated for the code generator of the compiler
void code_generator_destroy() {
        if (generator != NULL)
        {
        // Close the destination file
        fclose(generator->dest_file);
        free(generator);
        generator = NULL;
        }
}
// Initializes the compiler's code generator
void code_generator_init() {
        generator->dest_file = fopen("output.txt", "w");  // Open for writing
        if (generator->dest_file == NULL) {
        fprintf(stderr, "Failed to open file output for writing.\n" );
        free(generator);
        exit(EXIT_FAILURE);
        }

        strcpy(generator->registers[0].name, "AX");
        strcpy(generator->registers[0].low, "AL");
        strcpy(generator->registers[0].high, "AH");
        generator->registers[0].inuse = false;
        strcpy(generator->registers[1].name, "BX");
```

```
        strcpy(generator->registers[1].low, "BL");
        strcpy(generator->registers[1].high, "BH");
        generator->registers[1].inuse = false;
        strcpy(generator->registers[2].name, "CX");
        strcpy(generator->registers[2].low, "CL");
        strcpy(generator->registers[2].high, "CH");
        generator->registers[2].inuse = false;
        strcpy(generator->registers[3].name, "DX");
        strcpy(generator->registers[3].low, "DL");
        strcpy(generator->registers[3].high, "DH");
        generator->registers[3].inuse = false;

}
// Searches for a free register in the code generator registers array.
// If found, marks it as inuse and returns the index of that register.
// If not found, terminates the compiler.
int code_generator_register_alloc() {
        for (int i = 0; i < NUM_OF_REGISTERS; i++)
        {
        if (generator->registers[i].inuse != true) {
                generator->registers[i].inuse = true;
                return i;
        }
        }
        // If couldn't find a free register
        printf("Couldn't find a free register");
        //compiler_destroy();
        exit(1);
```

```
}
// Marks the register in index r in the code generator registers array as unused
void code_generator_register_free(int r) {
	generator->registers[r].inuse = false;
}
// Returns the register name of register r in the code generator registers array
char* code_generator_register_name(int r) {
	return generator->registers[r].name;
}
// Allocates a new lable name and returns a pointer to it
char* code_generator_label_create() {
	// A static integer to create unique lables
	static int label_num = 0;
	// Allocate memory for the label
	char* label = (char*)calloc(8, sizeof(char));
	if (label == NULL) exit(1);
	// Create label
	sprintf(label, LABEL_FORMAT, label_num);
	// Increment static int for next label
	label_num++;
	// Return the new label
	return label;
}
// Performs the right address computation for the given symbol according to its
// place in the program (global / local),
// and returns a string that represents that address.
// If the entry is NULL, returns NULL.
char* code_generator_symbol_address(SymbolTableEntry* entry) {
```

```
}
// Outputs the given formated string to the target file
void code_generator_output(char* format, ...) {
        // Check if the generator or its dest_file is NULL before proceeding
        if (generator == NULL || generator->dest_file == NULL) {
        fprintf(stderr, "Error: Code generator or destination file is not initialized.\n");
        exit(EXIT_FAILURE);
        }
        va_list args;                                   // Declare a va_list type variable
        va_start(args, format);                                 // Initialize the va_list with the ...
        vfprintf(generator->dest_file, format, args); // Forward ... to vfprintf
        va_end(args);                                   // Clean up the va_list
}
// Outputs the data segment of the program to the target file
void code_generator_output_data_segment(SymbolTable* table) {
        int size = 0;
        vars = symbolTableAll(table,&size);
        SizeVars = size;
        code_generator_output(".STACK 100h\n");
        code_generator_output(".DATA\n");
        code_generator_output("\t ; -------------------------\n");
        code_generator_output("\t ; Your variables here\n");
        code_generator_output("\t ; -------------------------\n");



        for (int i = 0; i < size; i++)
        {
```

```
        for (int j = i+1; j < size; j++)
        {
                if (strcmp(vars[i]->name, vars[j]->name) == 0) {
                int temp = strlen(vars[j]->name);
                vars[j]->name = realloc(vars[j]->name, sizeof(char) * temp + 2);
                vars[j]->name[temp] = '2';
                vars[j]->name[temp+1] = '\0';
                }
        }
        }
        for (int i = 0; i < size; i++)
        {

        /*      vars[i]->name = realloc(vars[i]->name, sizeof(char) *strlen(vars[i]->name)+3);
        sprintf(vars[i]->name+strlen(vars[i]->name)-1, "%d", hash(vars[i]->name));*/
        code_generator_output("\t%s dw ?\n",vars[i]->name);

        }
}

// Generates the assembly code for the given parse tree
void code_generator_generate(SymbolTable* table,ASTNode* parse_tree) {
        int size = 0;
        ASTNode** funcs = find_all_node_of_type(parse_tree, "function_declaration", &size);
        code_generator_create();
        code_generator_init();
        // - Generate the data segment of the program
        code_generator_output_data_segment(table);
```

```
    // - Generate the code segment of the program
    code_generator_output(".CODE\n");
    code_generator_output("start:\n");
    code_generator_output("\tmov ax, DATA\n");
    code_generator_output("\tmov ds, ax\n");
    code_generator_output("\t; -------------------------\n");
    code_generator_output("\t; Your code here\n");
    code_generator_output("\t; -------------------------\n");
    code_generator_output("\tcall main\n");




    code_generator_output("\n\texit:\n");
    code_generator_output("\tmov ax, 4c00h\n");
    code_generator_output("\tint 21h\n");



    for (int i = 0; i < size; i++)
    {
    code_generator_function(funcs[i]);
    }
    code_generator_output("END start\n");



}
```

```
// Generates a block (BLOCK)
void code_generator_block(ASTNode* block) {
	int size = 0;
	ASTNode** all = NULL;
	// BLOCK -> done
	if (block == NULL)
	return;
	if (block->childrenCount == 1)
	// Go to the parent scope when exiting a block
	return;
	// BLOCK -> STMT BLOCK
	else
	{
	for (int i = block->children[1]->childrenCount-1; i >=0; i--)
	{
		all=find_all_node_of_type(block->children[1]->children[i], "statement", &size);
		for (int j = size-1; j >= 0; j--)
		{
		// Generate the statement
		code_generator_stmt(all[j]);
		}

		size = 0;

	}

	// Generate the block
	code_generator_block(find_first_node_of_type(block->children[0], "block"));
```

```
        }
}
// Generates a statement (STMT)
void code_generator_stmt(ASTNode* stmt) {
        if (stmt == NULL)
        return;
        typedef void (*pFun)(ASTNode* );
        pFun arFun[] = { code_generator_decl
,code_generator_assign,code_generator_if_statement,code_generator_for ,code_generator_while
,code_generator_expression ,code_generator_funCall,code_generator_function ,code_generator_print
,code_generator_input };
        static const char* non_terminals[] = {
        "declaration",
        "assignment",
        "if_statement",
        "for_statement",
        "while_statement",
        "expression",
        "function_call",
        "function_declaration",
        "Print",
        "Input"
        };
        for (int i = 0; i < sizeof(non_terminals) / sizeof(non_terminals[0]); i++)
        {
        if (strcmp(stmt->children[0]->type, non_terminals[i]) == 0)
                arFun[i](stmt->children[0]);
        }

}
```

```
void code_generator_for(ASTNode* _for) {
	char* for_label = code_generator_label_create();
	char* done_label = code_generator_label_create();
	code_generator_output("%s:\n", for_label);

	code_generator_assign(_for->children[6]);

	code_generator_condition(_for->children[4]);

	code_generator_output("%s\n", done_label);
	// Generate code for the loop body
	code_generator_block(_for->children[0]);
	// Jump back to the start of the loop
	code_generator_expression(_for->children[2]);
	code_generator_output("\tmov [%s] , ax\n", _for->children[6]->children[3]->data);
	code_generator_output("\tjmp %s\n", for_label);

	// Label for the end of the loop
	code_generator_output("%s:\n", done_label);
}

void code_generator_function(ASTNode* func) {
	code_generator_output("proc %s\n", func->children[func->childrenCount-2]->data);
	code_generator_output("\tmov BP,SP\n");
	code_generator_block(func->children[0]);
	code_generator_output("ret\n");
	code_generator_output("endp %s\n", func->children[func->childrenCount - 2]->data);
```

```
}

void code_generator_funCall(ASTNode* func) {
	code_generator_output("\tcall %s\n", func->children[func->childrenCount-1]->data);
}

// Generates a variable declaration statement (DECL)
void code_generator_decl(ASTNode* decl) {
	if (decl == NULL) {
	fprintf(stderr, "Error: Declaration node is null or lacks variable name.\n");
	exit(EXIT_FAILURE);
	}

	// Handle different data types
	const char* resDirective;
	ASTNode* type = find_first_node_of_type(decl, "type_specifier")->children[0];
	ASTNode* var = decl->children[3];
	if (strcmp(type->data, "int") == 0) {
	resDirective = "resd";  // Reserve space for one double word (integer)
	}
	else if (strcmp(type->data, "double") == 0) {
	resDirective = "resq";  // Reserve space for one quad word (double)
	}
	else if (strcmp(type->data, "char") == 0) {
	resDirective = "resb";  // Reserve space for one byte (char)
	}
	else {
```

```
        fprintf(stderr, "Error: Unsupported type '%s' for variable '%s'.\n", decl->type, decl->data);
        exit(EXIT_FAILURE);
        }




        // Check if initialization is required
        if (decl->childrenCount > 0 && decl->children[0] != NULL && decl->children[0]->value != NULL) {
        code_generator_expression(decl->children[1]);  // Generate the initialization value
        code_generator_output("\tmov [%s], ax\n", var->data);  // Store the result in the variable
        }
}



void code_generator_print(ASTNode* print) {
        if (strcmp(print->children[print->childrenCount - 3]->children[0]->value, "INT") == 0) {
        code_generator_expression(print->children[2]);
        code_generator_output("\tmov dx , ax\n");
        code_generator_output("\tadd dx , 48\n");
        code_generator_output("\tmov ah , 2\n");
        code_generator_output("\tint 21h\n");
        }
        else {
        code_generator_expression(print->children[2]);
        code_generator_output("\tmov dx , ax\n");
        code_generator_output("\tmov ah , 2\n");
        code_generator_output("\tint 21h\n");
        }
```

```
        code_generator_output("\tmov dl , 10\n");
        code_generator_output("\tmov ah , 2\n");
        code_generator_output("\tint 21h\n");
}

void code_generator_input(ASTNode* input) {
        if (strcmp(input->children[input->childrenCount - 3]->children[0]->value, "INT") == 0) {
        code_generator_output("\tmov ah , 7\n");
        code_generator_output("\tint 21h\n");
        }
        else {
        code_generator_output("\tmov ah , 7\n");
        code_generator_output("\tint 21h\n");
        }
}


// Generates an assignment statement (ASSIGN)
void code_generator_assign(ASTNode* assign) {
        code_generator_expression(assign->children[1]);
        SymbolTableEntry* entry=malloc(sizeof(SymbolTableEntry));
        // Assign the expression value to the variable
        for (int i = 0; i < SizeVars; i++)
        {
        if (strstr(vars[i]->name, assign->children[assign->childrenCount - 1]->data) !=NULL &&
vars[i]->scopeIdentifier <= assign->scope) {
                entry=vars[i]; // Found
        }
```

```
        }



        code_generator_output("\tmov [%s], ax\n", entry->name);
        // Free result register because we don't need it anymore
        code_generator_register_free(0);
}

void code_generator_if_statement(ASTNode* _if) {
        char* if_label = code_generator_label_create();



        // Generate the condition
        code_generator_condition(_if->children[2]);
        // Check if condition
        code_generator_output(" %s\n", if_label);

        code_generator_stmt(_if->children[0]);

        code_generator_output("%s:\n", if_label);

        // Free expression register because we don't need it anymore
        code_generator_register_free(0);

}

// Generates a while statement (WHILE)
void code_generator_while(ASTNode* _while) {
```

```
        char* while_label = code_generator_label_create();
        char* done_label = code_generator_label_create();
        code_generator_output("%s:\n",while_label);

        // Generate the condition
        code_generator_condition(_while->children[2]);
        // Check while condition

        code_generator_output("%s\n", done_label);
        // Generate code for the loop body
        code_generator_block(_while->children[0]);
        // Jump back to the start of the loop
        code_generator_output("\tjmp %s\n", while_label);

        // Label for the end of the loop
        code_generator_output("%s:\n", done_label);
}

void code_generator_term(ASTNode* node) {
        if (node == NULL) {
        return;
        }

        // Traverse child nodes: left term (if exists), operator, right term
        if (node->childrenCount == 3) {
        // Generate code for the first term
        code_generator_expression(node->children[0]);
```

```
// Store the result of the first term to avoid being overwritten
code_generator_output("\tpush ax\n");

// Generate code for the second term
code_generator_expression(node->children[2]);

// Retrieve the first term result into another register
code_generator_output("\tmov bx, ax\n");
code_generator_output("\tpop ax\n");

// Now, perform the operation given by the middle child (operator)
if (strcmp(node->children[1]->value, "*") == 0) {
        code_generator_output("\timul ax, bx\n");  // mul
}
else if (strcmp(node->children[1]->value, "+") == 0) {
        code_generator_output("\tadd ax, bx\n");  // add
}
else if (strcmp(node->children[1]->value, "-") == 0) {
        code_generator_output("\tsub bx, ax\n");  // sub
        code_generator_output("\tmov ax, bx\n");

}
else if (strcmp(node->children[1]->value, "/") == 0) {
        code_generator_output("\tmov dx, ax\n");
        code_generator_output("\tmov ax, bx\n");
        code_generator_output("\tmov bx, dx\n");
        code_generator_output("\txor dx dx\n");
        code_generator_output("\tdiv bx\n"); // div
```

```
        }
        }
        else {
        // It's just one term without operation, process it directly
        code_generator_expression(node->children[0]);
        }
}

void code_generator_expression(ASTNode* node) {
        if (node == NULL) {
        fprintf(stderr, "Error: Null expression node.\n");
        exit(EXIT_FAILURE);
        }

        // Assuming the node's type dictates the kind of expression or operand
        if (strcmp(node->type, "expression") == 0 || strcmp(node->type, "term") == 0) {
        // This node type is a container and should delegate to its child or children
        code_generator_term(node);
        }
        else if (strcmp(node->type, "factor") == 0) {
        // This can delegate further down or handle simple factors
        code_generator_expression(node->children[0]);  // Assuming one child
        }
        else if (strcmp(node->type, "factor_base") == 0) {
        // Delegate to terminal
        code_generator_expression(node->children[0]);
        }
        else if (strcmp(node->type, "Terminal") == 0) {
```

```
        // Terminal node: directly load the value into eax
        if (node->data) {
                if (strcmp(node->value, "ID") == 0) {
                code_generator_output("\tmov ax, [%s]\n", node->data);
                }
                else code_generator_output("\tmov ax, %s\n", node->data);
        }
        }
}
void code_generator_condition(ASTNode* condition) {
        // Generate code for the left and right expressions of the condition
        code_generator_expression(condition->children[0]); // Evaluate the left side
        code_generator_output("\tpush ax\n");                // Save left result on stack
        code_generator_expression(condition->children[2]); // Evaluate the right side
        code_generator_output("\tpop bx\n");                 // Pop left result into ebx

        // Compare results
        code_generator_output("\tcmp ax, bx\n");        // Compare eax and ebx

        // Set appropriate flags based on comparison, actual jumps will depend on later usage
        if (strcmp(condition->children[1]->data, "==") == 0) {
        code_generator_output("\tjne somewhere  ; Jump if equal\n");
        }
        else if (strcmp(condition->children[1]->data, "!=") == 0) {
        code_generator_output("\tje ");
        }
        else if (strcmp(condition->children[1]->data, ">") == 0) {
        code_generator_output("\tjle ");
```

```
        }
        else if (strcmp(condition->children[1]->data, "<") == 0) {
        code_generator_output("\tjge ");
        }
        else if (strcmp(condition->children[1]->data, ">=") == 0) {
        code_generator_output("\tjl ");
        }
        else if (strcmp(condition->children[1]->data, "<=") == 0) {
        code_generator_output("\tjg ");
        }
        else {
        fprintf(stderr, "Unsupported condition operator %s.\n", condition->children[1]->data);
        }
}
// Generates a binary expression of the form: Expr -> Expr binary_operator Expr
void code_generator_binary_expression(int left_register, int right_register, TokensType op) {
        typedef void (*pFun)(int , int );
        pFun arFun[] = { code_generator_divide ,code_generator_mul ,code_generator_add
,code_generator_sub };
        TokensType list[] = { SLASH,STAR,PLUS,MINUS};
        for (int i = 0; i < sizeof(list) / sizeof(list[0]); i++)
        {
        if (op==list[i])
                arFun[i](left_register,right_register);
        }
}
```

main.py

```
import os
from tkinter import *
from tkinter import ttk, filedialog, messagebox
import subprocess

class ModernIDE:
    def __init__(self, root):
        self.root = root
        self.root.title('Cqual IDE')
        self.root.geometry("1200x800")
        self.file_path = ''
        self.current_folder = os.getcwd()
        self.setup_ui()

    def setup_ui(self):
        self.setup_toolbar()
        self.setup_main_layout()
        self.setup_status_bar()

    def setup_toolbar(self):
        toolbar = Frame(self.root, bd=1, relief=RAISED, bg='#4b4b4b')
        Button(toolbar, text="Open", command=self.open_file, bg='#2a2a2a',
fg='white').pack(side=LEFT, padx=5, pady=5)
        Button(toolbar, text="New Tab", command=self.add_tab, bg='#2a2a2a',
fg='white').pack(side=LEFT, padx=5, pady=5)
        Button(toolbar, text="Run", command=self.run_code, bg='#2a2a2a',
fg='white').pack(side=LEFT, padx=5, pady=5)
        Button(toolbar, text="Save", command=self.save_file, bg='#2a2a2a',
fg='white').pack(side=LEFT, padx=5, pady=5)
        Button(toolbar, text="Change Folder", command=self.change_folder,
bg='#2a2a2a', fg='white').pack(side=LEFT, padx=5, pady=5)
        toolbar.pack(side=TOP, fill=X)

    def setup_main_layout(self):
        main_pane = PanedWindow(self.root, orient=HORIZONTAL)
        main_pane.pack(fill=BOTH, expand=True)

        file_manager_frame = Frame(main_pane, bg='#3a3a3a')
        file_manager_frame.pack(fill=Y, expand=False)
```

```
        self.file_tree = ttk.Treeview(file_manager_frame, show="tree")
        self.file_tree.pack(fill=BOTH, expand=True)
        self.populate_tree(self.file_tree, '', self.current_folder)
        self.file_tree.bind("<Double-1>", self.open_file_from_tree)
        main_pane.add(file_manager_frame, minsize=200)

        right_pane = PanedWindow(main_pane, orient=VERTICAL)
        main_pane.add(right_pane)

        self.notebook = ttk.Notebook(right_pane)
        right_pane.add(self.notebook, height=500)

        self.add_tab("Untitled")
        self.code_output = Text(height=10, bg="#2d2d2d", fg="white",
insertbackground="white", font=("Courier New", 12))
        right_pane.add(self.code_output)

    def setup_status_bar(self):
        self.status_bar = Label(self.root, text="Ready", anchor=W,
bg='#3a3a3a', fg='white')
        self.status_bar.pack(side=BOTTOM, fill=X)

    def add_tab(self, tab_title="Untitled"):
        editor = Text(self.notebook, bg="#1e1e1e", fg="white",
insertbackground="white", font=("Courier New", 12))
        index = self.notebook.index('end')
        self.notebook.add(editor, text=tab_title)
        self.notebook.tab(index, text=tab_title)
        self.notebook.select(index)

        # Add close button on the right of each tab
        def on_close_tab():
            self.notebook.forget(index)
        Button(editor, text="✖", command=on_close_tab).pack(side=TOP,
anchor=SE, padx=5, pady=5)
        return editor

    def set_file_path(self, path):
        self.file_path = path
        self.update_status_bar(path)
```

```
    def update_status_bar(self, message):
        self.status_bar.config(text=message)

    def open_file(self):
        path = filedialog.askopenfilename(initialdir=self.current_folder,
filetypes=[('C Files', '*.c')])
        if path:
            with open(path, 'r') as file:
                code = file.read()
                editor = self.add_tab(os.path.basename(path))
                editor.delete('1.0', END)
                editor.insert('1.0', code)
                self.set_file_path(path)

    def save_file(self):
        editor = self.notebook.nametowidget(self.notebook.select())
        path = self.file_path or filedialog.asksaveasfilename(filetypes=[('C
Files', '*.c')])
        if path:
            with open(path, 'w') as file:
                code = editor.get('1.0', END)
                file.write(code)
                self.set_file_path(path)
                self.notebook.tab(self.notebook.index(editor),
text=os.path.basename(path))

    def run_code(self):
        if self.file_path == '':
            messagebox.showinfo("Save Your Code", "Please save your code
before running.")
            return

        output_file = self.file_path.replace(".c", "")
        compile_command = f'gcc "{self.file_path}" -o "{output_file}"'
        process = subprocess.Popen(compile_command, stdout=subprocess.PIPE,
stderr=subprocess.PIPE, shell=True)
        output, error = process.communicate()

        self.code_output.delete('1.0', END)

        if error:
```

```
            self.code_output.insert('1.0', error.decode('utf-8'))
        else:
            execute_command = f'{output_file}'
            process = subprocess.Popen(execute_command,
stdout=subprocess.PIPE, stderr=subprocess.PIPE, shell=True)
            output, error = process.communicate()
            self.code_output.insert('1.0', output.decode('utf-8'))
            self.code_output.insert('1.0', error.decode('utf-8'))

    def populate_tree(self, tree, parent, path):
        tree.delete(*tree.get_children(parent))
        for p in sorted(os.listdir(path)):
            full_path = os.path.join(path, p)
            if os.path.isdir(full_path):
                node = tree.insert(parent, 'end', text=p, open=False)
                self.populate_tree(tree, node, full_path)
            else:
                tree.insert(parent, 'end', text=p)

    def open_file_from_tree(self, event):
        item = self.file_tree.selection()[0]
        parent_path = self.file_tree.parent(item)
        file_name = self.file_tree.item(item, "text")
        while parent_path:
            file_name = os.path.join(self.file_tree.item(parent_path, "text"),
file_name)
            parent_path = self.file_tree.parent(parent_path)

        if os.path.isfile(file_name):
            with open(file_name, 'r') as file:
                code = file.read()
                editor = self.add_tab(os.path.basename(file_name))
                editor.delete('1.0', END)
                editor.insert('1.0', code)
                self.set_file_path(file_name)

    def change_folder(self):
        folder = filedialog.askdirectory(initialdir=self.current_folder)
        if folder:
            self.current_folder = folder
            self.populate_tree(self.file_tree, '', folder)
```

```
if __name__ == "__main__":
    root = Tk()
    ide = ModernIDE(root)
    root.mainloop()
```