עבודת גמר

לקבלת תואר

טכנאי תוכנה

הנושא: בניית קומפיילר

המגיש : גיא שון עדן

המנחה : מיכאל צ'רנובילסקי

מאי 2019 **תשע"ט**

תוכן עניינים

מבוא

מטרה

לבחון את הידע והכישורים התכנותיים שלי בפרוייקט בסדר גודל כזה.
לפתח אלגוריתמים חכמים אשר יפתרו בעיות שונות תוך כדי שימוש בכלי תכנות מונחה עצמים וחשיבה שברשותי. בנוסף רכישת נסיון בקרב מכונות מצבים, דבר שלא נתקלתי בו בעבר.

תאור המערכת

המערכת הינה מהדר – "קומפיילר" אשר מתרגם מסמך טקסטואלי המכיל קוד בשפה שאני הגדרתי, אימג'ינרי, לקוד אסמבלי 32 ביט. שפת תכנות היא הגדרה של אוסף חוקים תחביריים וסמנטיים, שנועדו להגדיר תהליכי חישוב המבוצעים במחשב. הגדרת שפת התכנות היא חלק בלתי נפרד מבניית המהדר – המהדר עושה שימוש בהגדרת השפה כדי לנתח את קטע הקוד שנקלט וכדי לייצר את תוכנית היעד.

כאשר מגדירים שפת תכנות, מתייחסים כאמור לשלושה מישורים: האחד מילונאי, השני תחבירי והשלישי לשוני. המישור המילונאי מגדיר אילו מילים שייכות לשפה, ואילו לא [לדוגמה: המילה if היא מילה מקובלת בשפה, בעוד שהמילה Hello@#, איננה]. המישור התחבירי מגדיר איזה רצפי מילים של השפה הם חוקיים, ואיזה הם לא [לדוגמה: הרצף int x = 3; הוא רצף חוקי בשפת Java, בעוד שהרצף if x is 5 then, איננו חוקי בשפת Java]. המישור הלשוני מתייחס למשמעות רצפי המילים, והוא מגדיר חוקים כלליים שחייבים להתקיים בכל רצף מילים בשפה [לדוגמה: חובת ההצהרה - לפני השימוש במשתנה, קיימת חובה להצהיר עליו].

תחביר השפה הינו תחביר-חופשי-הקשר [Context-Free Grammar] ומקובל להגדיר אותו לפי BNF – Backus Naur Form. הדקדוק מורכב מ-Terminals ומ-Non-Terminals. הסימנים [Terminals] הם המילים שנקלטו כקלט מקטע הקוד, בעוד שהמשתנים [Non Terminals] הם רצפי סימנים ומשתנים. תחביר השפה מוגדרת באמצעות שילוב הסימנים והמשתנים, בכללים שנקראים כללי יצירה [Productions]. כללי היצירה בעצם מגדירים את המשתנים, באמצעות הסימנים.

מנתח מילונאי, המחלק את הטקסט במסמך לרצף של טוקנים (טוקן הוא רצף של אותיות או סימנים המוגדר בשפה). חלק זה נעשה באמצעות מכונת מצבים בת 3 שכבות.

מנתח תחבירי, הבונה מרצף הטוקנים עץ. חלק זה בנוי בעזרת מכונת מצבים. מפה המקבלת את המצב הנוכחי, ואת הטוקן הנוכחי, ולפיו יודעת מהו המצב הבא.

מנתח לשוני, המעבד את העץ שנבנה על ידי המנתח התחבירי ובונה עץ פשוט יותר, הבנוי על ירושה, ומשמיט את צמתי הביניים הלא נחוצים.

מחולל קוד ביניים, העובר על העץ הלשוני והופך אותו לרשימה (וקטור) של "פקודות", פקודה בנוייה מ4 חלקים, אופרנד א', אופרנד ב', פעולה, ותוצאה. בחלק זה כבר ניתן לראות ממשית כיצד המהדר מפרק את הקוד ומתרגם אותו לרצף של פקודות פשוטות.

מחולל קוד סופי, אשר יעבור על רשימת ה"פקודות" ויתרגם אותם לאסמבלי 32 ביט, יחולל את סגמנט הנתונים, וישמור על סדר האוגרים (אוגר תפוס / פנוי).

שפת התכנות ופירוט סביבת העבודה והכלים

פיתוח המערכת נעשה בשפת התכנות C++.

סביבת העבודה היא - Microsoft .NET Framework ,

תוך כדי שימוש ב- Microsoft Visual Studio 2017.

פיתוח המערכת נעשה תוך כדי שימוש בכלי Windows Command Prompt Application שסביבת העבודה מציעה למשתמש.

מפרטי תוכנה

ניסוח וניתוח הבעיה האלגוריתמית

במהלך בניית הפרוייקט נתקלתי במספר בעיות אלגוריתמיות אשר היו מהותיות לקיום המערכת. כעת אמנה אותם, אנתח אותם ובפרק הבא אציע פיתרון לבעיות.

בעיה אלגוריתמית מספר 1, **ניתוח מילוני**

כיצד ניתן לפרש בצורה חד משמעית את רצף התווים כטוקן מסויים? למשל, כאשר נראה את התו '=', נוכל להניח שמדובר באתחול\השמה של ערך לתוך משתנה, אך מה אם מייד אחריו יופיע עוד פעם '='? נצטרך להתייחס ל2 התווים כטוקן אחד המייצג השוואה.

בעיה אלגוריתמית מספר 2, **ניתוח תחבירי**

המנתח המילונאי מזרים טוקנים, מילים תקינות הכלולות בשפה אל המנתח התחבירי. מנתח זה צריך למצוא הגיון בסדר הטוקנים ולבנות ממנו עץ אשר ייצג תוכנה הגיונית. מאחר שמדובר בעץ ולכל צומת ישנן תכונות שונות ו"ילדים" שונים זה מזה, צריך למצוא דרך לשמור על התכונות הייחודיות של כל אחד מהם, ועדיין לשמור על היכולת להתייחס אליהם כמכלול ולעבור עליהם בצורה איטרטיבית בסופו של דבר. על מנת לבנות עץ מדוייק יש להגדיר "נוסחאות" מדוייקות שייצגו את השפה, וההגיון שבה. מקובל לייצג את נוסחאות אלו בצורת backus naur. עוד אתגר מעבר להגדרת השפה, היא "מצבים מקבילים". למשל ב#C, מה קורה כאשר המצב הנכחי הוא statement, והטוקן הנכחי הוא identifier? איך נדע האם לצפות להשמה (assignment) כמו "X = 3;", או לקריאה לפעולה שהיא תכונה של העצם כמו "X.foobar()"? אם המצב הנכחי הוא הכרזה והטוקן הנכחי הוא identifier, איך נדע האם לצפות לsemicolon או לפסיק? איך נוכל לדעת תמיד למה לצפות באופן מדוייק?

בעיית הניתוח הלשוני היא ללא ספק הבעיה הכי גדולה בפרוייקט ומהווה כ - 60% מכל הפרוייקט, זהו החלק הכי חשוב במהדר ובלעדיו לוגיקת השפה לא תתקיים.

בעיה אלגוריתמית מספר 3, **פישוט העץ התחבירי**

עד כה ראינו רצף לוגי מאוד הגיוני, המנתח המילוני בודק טוקנים ורואה שכולם בשפה. המנתח התחבירי מרכיב מהטוקנים משפטים ובודק שאלו משפטים תקינים בשפה. ועכשיו, המנתח הלשוני צריך לקבל את התכנה בצורת העץ התחבירי ולבדוק אם היא הגיונית. פה ייבדק הרצף הלוגי של התכנה, האם יש שימוש במשתנה שלא הוכרז? האם יש חוסר תאימות בין סוגי משתנים? בכדי לפענח את העץ התחבירי נצטרך למצוא שיטה יעילה לעבור עליו, ולפשט אותו. כיצד נזהה את הטיפוס של צומת בעץ? כיצד נסרוק את העץ בצורה שתאפשר לנו לקבל את הערכים הנורשים מאחיו השמאליים והוריו, ולאחר מכן את הערכים הנוצרים מילדיו?

העץ התחבירי נבדל מעץ הניתוח בכך שהוא מכיל אך ורק את מה שנדרש על ידי המתרגם לתרגום הקוד. העץ התחבירי פשוט יותר גם מבנית וגם רעיונית. הוא ממוקד וללא צמתים מקשרים, ומטרתו היחידה הוא לייצג את התוכנית במבנה שיאפשר בקלות יחסית לתרגמו לייצוג ביניים.

חשוב להדגיש: הבדיקה הסמנטית היא בדיקת הקלט האחרונה בתהליך ההידור, ולכן עץ תחבירי שנפלט ממנה, מייצג תוכנית תקינה.

פיתוח הפתרון ויישומו

<u>פיתוח פתרון לבעיה אלגוריתמית מספר 1, **ניתוח מילונאי**</u>

נשתמש במכונת מצבים. גם כאשר נהיה במצב מקבל, נמשיך לקרוא עד אשר נגיע למבוי סתום (ללא מצבים או ללא קלט). בכדי לממש את מכונת המצבים נשתמש במפה.

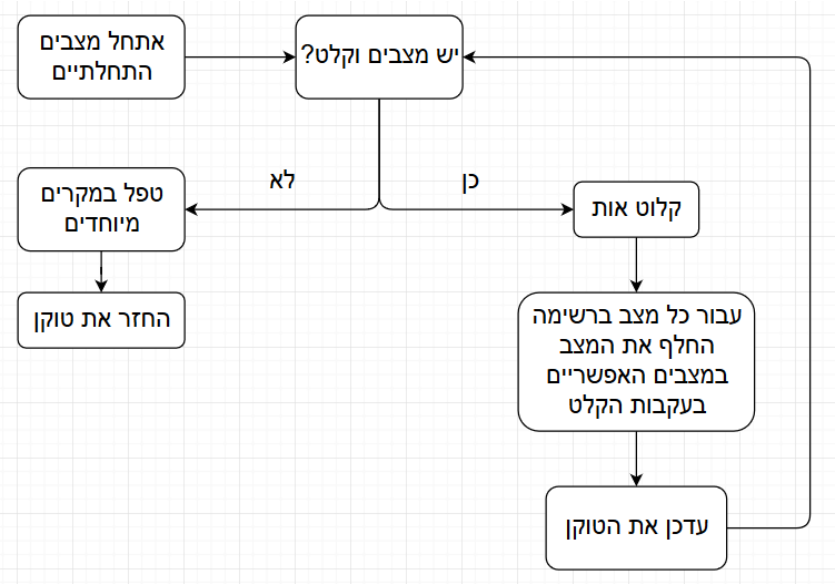
מפה היא סוג של פונקציה – בעבור קלט נתון [בתנאי שהוגדר במפה קודם לכן], היא מחזירה ערך אחד מסוים. המפה שהגדרתי משתמשת בשני מפתחות, ועבור שילוב מתאים של השניים מתקבל ערך יחיד. המפה בעצם מגדירה את המעברים של האוטומט – עבור מפתח שמייצג מצב ומפתח שמייצג אות קלט, יתקבל המצב הבא שאליו אמור לעבור האוטומט.

חשוב לציין : האוטומט הוא דטרמיניסטי באופן הזה שעבור קלט נתון יש מצב מקבל אחד בלבד שאליו יכול הקלט להגיע. עם זאת, מאחר והאוטומט בנוי משילוב של מספר תת-אוטומטים, קיימים מספר מצבים התחלתיים, תכונה שבדרך כלל מיוחסת לאוטומטים סופיים לא דטרמיניסטיים.

תאור אלגוריתם הפעולה המחזירה את הטוקן הבא

1. **אתחל** את האוטומט הסופי
2. **כל עוד** יש מצבים וגם יש קלט:
   א. **קלוט** את האות הבאה.
   ב. **עבור** כל מצב ברשימה:
      (1) **החלף** ברשימת המצבים בין המצב ובין כל המצבים האפשריים בעקבות הקלט
   ג. **עדכן** את <u>token</u>
3. **טפל** במקרים מיוחדים
4. **החזר** את <u>token</u>

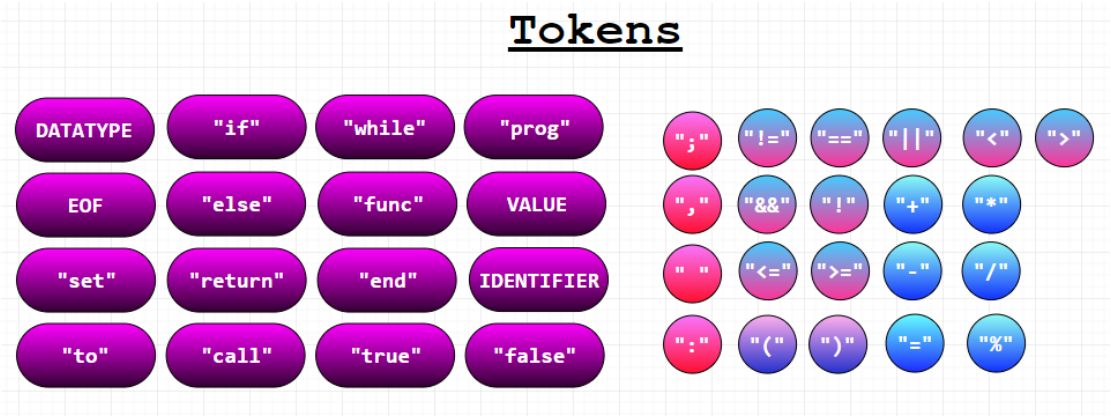
<u>תרשים האלגוריתם:</u>



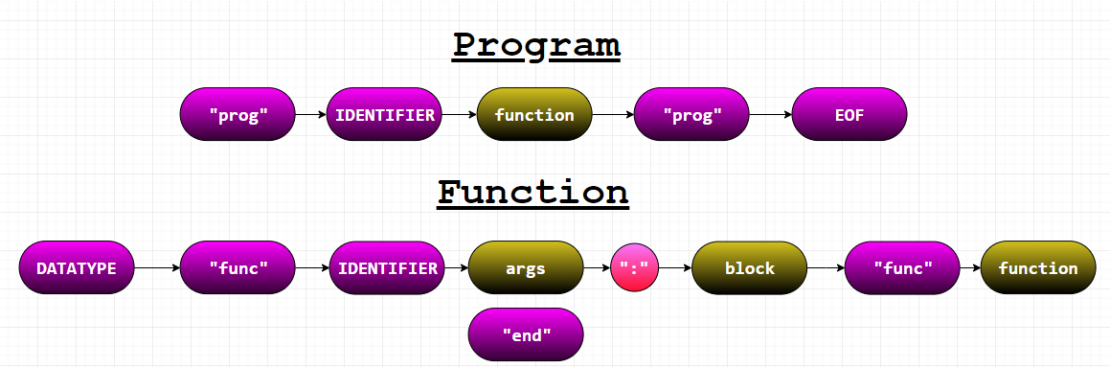
פיתוח פתרון לבעיה אלגוריתמית מספר 2, **ניתוח תחבירי**

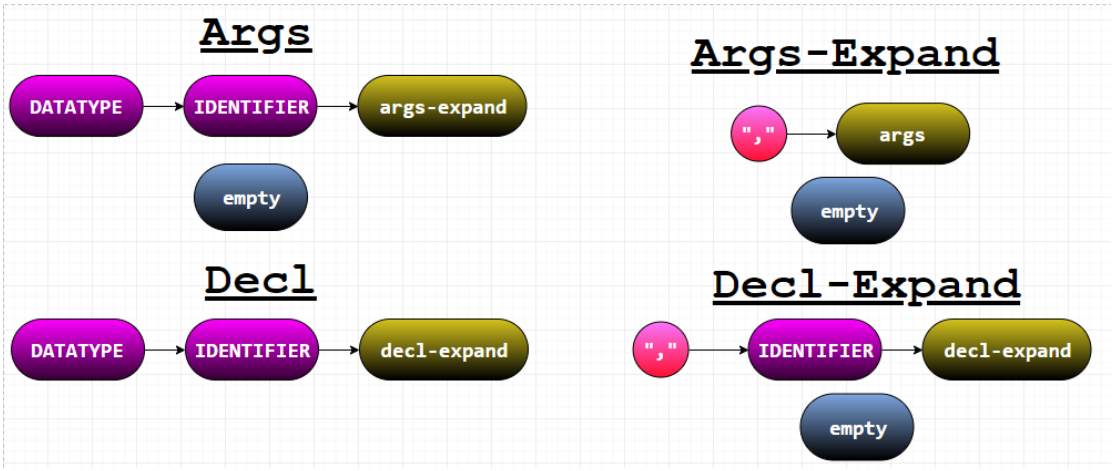
קודם כל, ננסח לשפה נוסחאות מדוייקות, במקום להשתמש בנוסחאות backus naur, בחרתי להשתמש בתרשימים ויזואליים, אשר מדוייקים באותה מידה.

אבני הבנייה של השפה (הטוקנים):

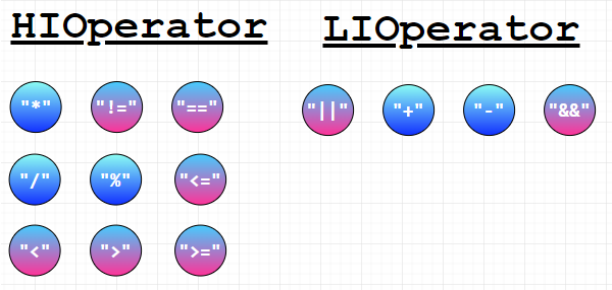


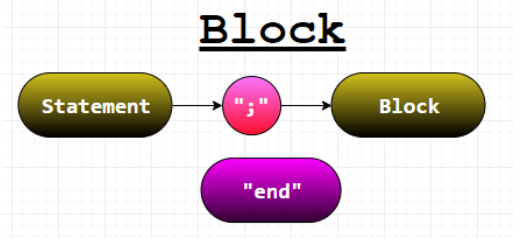
ה"נוסחאות":

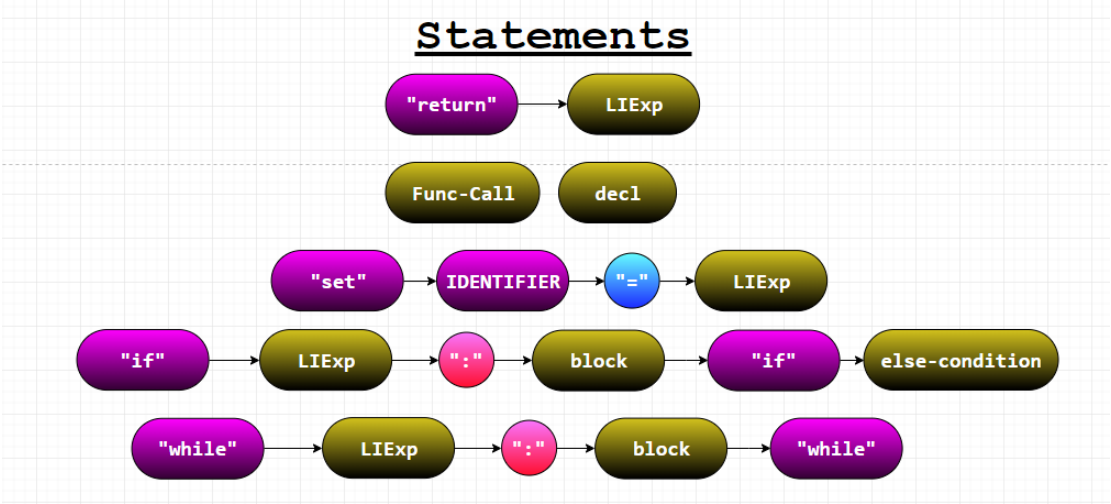


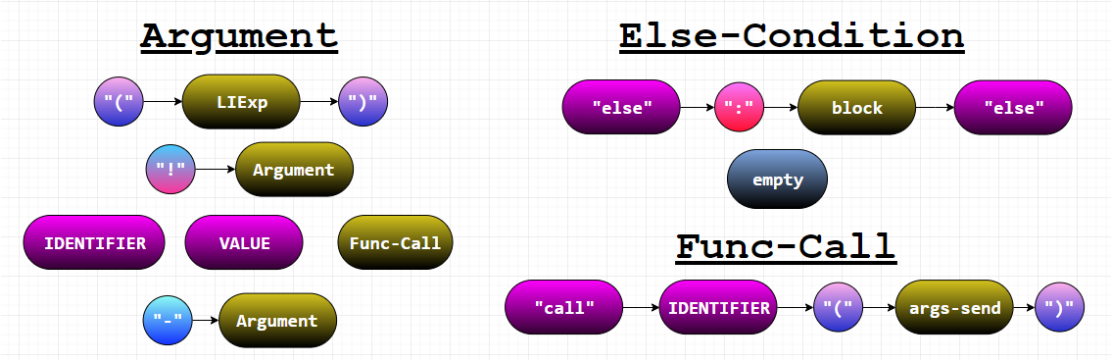


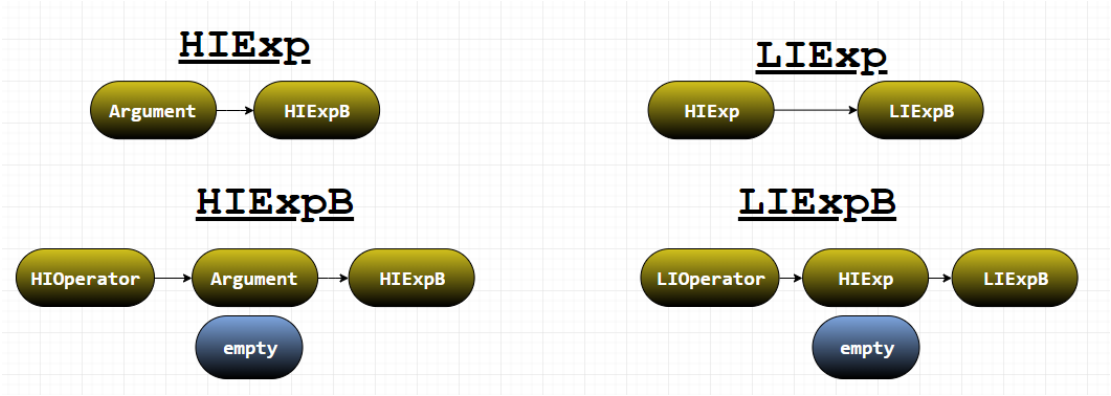

Args-Send
LIExp
args-send-expand
empty
Args-Send-Expand
","
args-send
empty


HIOperator
"*"
"!="
"=="
"/"
"%"
"<="
"<"
">"
">="
LIOperator
"||"
"+"
"-"
"&&"


Block
Statement
";"
Block
"end"


Statements
"return"
LIExp
Func-Call
decl
"set"
IDENTIFIER
"="
LIExp
"if"
LIExp
":"
block
"if"
else-condition
"while"
LIExp
":"
block
"while"


Argument
"("
LIExp
")"
"!"
Argument
IDENTIFIER
VALUE
Func-Call
"-"
Argument
Else-Condition
"else"
":"
block
"else"
empty
Func-Call
"call"
IDENTIFIER
"("
args-send
")"



כפי שניתן לראות על פי התרשימים, השפה תוכננה כך שתמיד עבור טוקן נוכחי ומצב נוכחי תהיה רק אפשרות אחת כתוצאה. משמע, אין 2 נוסחאות הנמצאות באותו ההקשר ומתחילות באותו הטוקן. כל סוג Statement מתחיל בטוקן שונה.

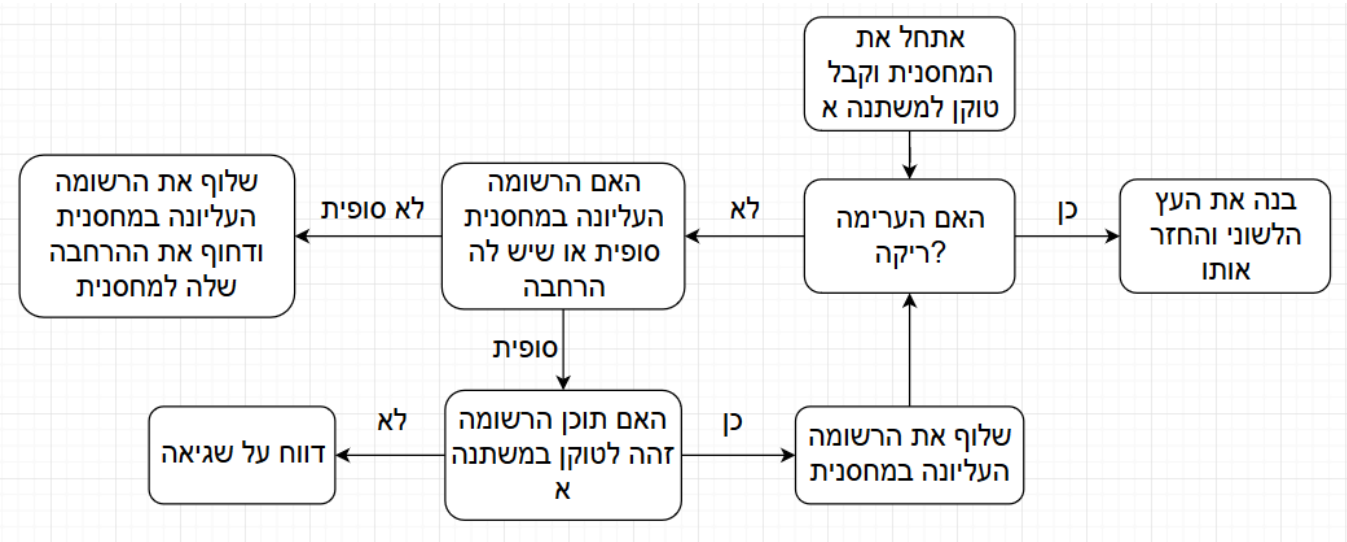
באשר לבעיית בניית עץ אשר ייצג את כל תכונותיהן של הנוסחאות השונות, השתמשתי בירושה. ייצרתי מבנה נתונים בו יש מחלקת בסיס אשר מכילה מספר שלם המייצג את סוג הטיפוס, ווקטור של בנים של צומת זה בעץ. לאחר מכן ייצרתי עבור כל נוסחה בעץ מחלקה היורשת ממחלקת הבסיס, בה תכונות שונות בהתאם לסוג הnode.

תאור אלגוריתם מימוש מנתח תחבירי מעלה מטה באופן איטרטיבי

אלגוריתם זה לא מקבל קלט, אלא משתמש במחלקת הLexical Analyzer המוגדרת כתכונה שלו בכדי לקבל טוקנים כקלט. פעולה זו מחזירה את העץ הלשוני (שהוא בעצם עץ התחביר המפושט).

1. **אתחל** את המחסנית
2. a = getToken()
3. **כל עוד** המחסנית אינה ריקה:
   a. x = stack.top()
   b. **אם** x הוא סימן [terminal]:
      i. **אם** x שווה ל-a:
         1. שלוף מהמחסנית
         2. a = getToken()
      ii. **אחרת**:
         1. **דווח** על שגיאה
   c. **אחרת:**
      i. שלוף מהמחסנית
      ii. דחוף למחסנית את ההרחבה של x, בסדר הפוך ובכל דחיפה הכנס את האיבר לעץ התחביר, כבנו של x
4. **בנה** את <u>העץ הלשוני</u>
5. **החזר** את <u>העץ הלשוני</u>

<u>תרשים לאלגוריתם:</u>



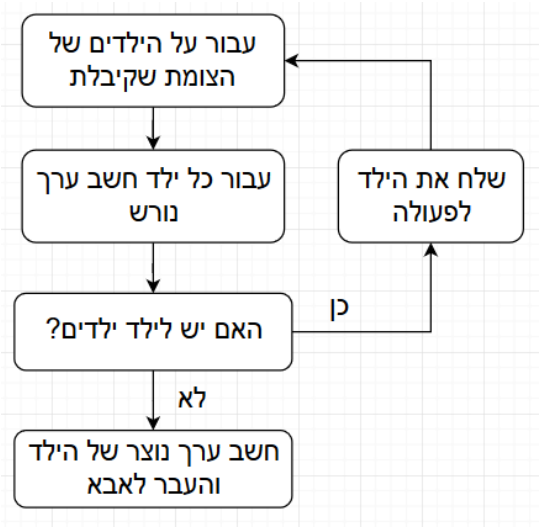
פיתוח פתרון לבעיה אלגוריתמית מספר 3, **פישוט העץ התחבירי**

באשר לבעיית הזיהוי, המספר המזהה שמייצג את סוג הטיפוס משמש אותנו כאשר אנו צריכים לעבור (באירטציה, לולאה) על ווקטור הילדים של חולייה בעץ. כאשר אנו עוברים על ווקטור של חוליות שונות שונות נתייחס לכולן כחוליית הבסיס, ממנה הן יורשות. אך בתוך הלולאה נוכל לדעת איך להתייחס לכל אחת מהן בעזרת המספר המזהה, ובעזרתו נוכל לדעת איך לעשות dynamic_cast, פעולה ב++C אשר מאפשרת לך להמיר מחלקת בסיס למחלקה היורשת ממנה.

דוגמה פשוטה יותר לאסטרטגיה זו היא חיות. מחלקת חיה מכילה תכונות אשר משותפות לכל החיות. ממחלקה זו יורשות מחלקת כלב ומחלקת חתול. לא נוכל ליצור רשימה התכיל כלבים וחתולים, אך נוכל ליצור רשימה המכילה חיות. לאחר מכן, אם נדע שטיפוס מסוג חיה הוא כלב **בוודאות**, נוכל לעשות לטיפוס החיה dynamic_cast לכלב, ולהתייחס אליו ככלב, משמע לגשת לתכונות שבלעדיות לו.

פישוט העץ יעשה בצורה הבאה, נסרוק את העץ בסריקה תוכית. עבור כל ילד של הצומת שקיבלה הפעולה, נחשב את הערך שאנו מורישים לו, ואז נקרא לפעולה עליו. כשסיימנו לעבור על כל הילדים (בחזרה מהרקורסיה), נחשב את הערך שהצומת שקיבלה הפעולה יצר. ערך זה יועבר להורה של הצומת.

תאור אלגוריתם ניתוח העץ התחבירי



קוד הפתרון [פסאודו קוד]:

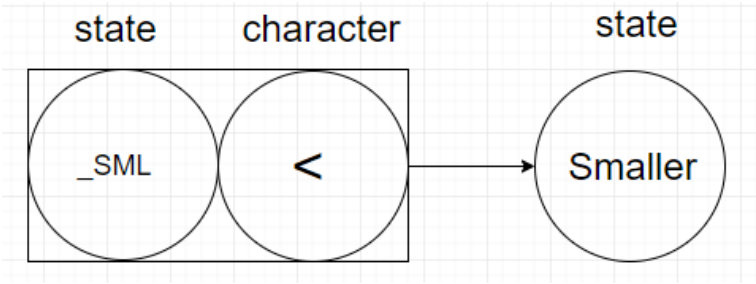
1. **עבור** כל בן של הצומת:
   a. חשב ערך נורש של הבן
   b. חפש לעומק בבן
2. חשב ערך נוצר של הצומת והעבר להורה

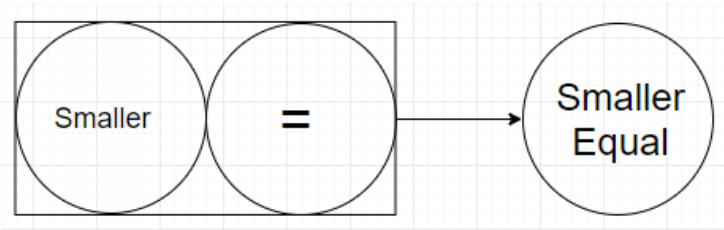
מבנה נתונים

מפה

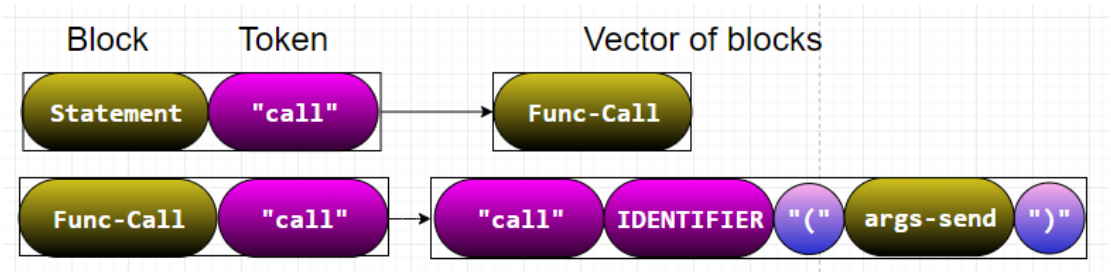
במהלך הפרוייקט עשיתי שימוש במפה על מנת לממש 2 מכונות מצבים, אחת למנתח המילונאי, ואחת למנתח התחבירי. מפה היא מבנה נתונים הפועל כפונקציה, בעבור קלט מסויים, תתקבל תוצאה אחת בלבד.

בכדי להשתמש במפה כמכונת מצבים למנתח המילונאי, הגדרתי את הקלט כמבנה הנתונים Pair, המכיל זוג אובייקטים. במקרה הזה האובייקטים הם מחרוזת המייצגת מצב נכחי, והתו המתקבל. הפלט הוגדר כמחרוזת המייצגת את המצב אליו הנך עובר.





בכדי להשתמש במפה כטבלת תרגום למנתח התחבירי, הגדרתי את הקלט כמבנה הנתונים Pair, המכיל זוג אובייקטים. במקרה הזה האובייקטים הם מבנה נתונים מסוג ParserStackEntry המייצג בלוק בנייה שעלינו לפרק, וטוקן נוכחי. הפלט הינו ווקטור המכיל בלוקי בנייה. הפלט בעבור בלוק הבנייה תלוי בטוקן, בעבור טוקנים שונים ימצא תרגום שונה.



<u>ווקטור</u>

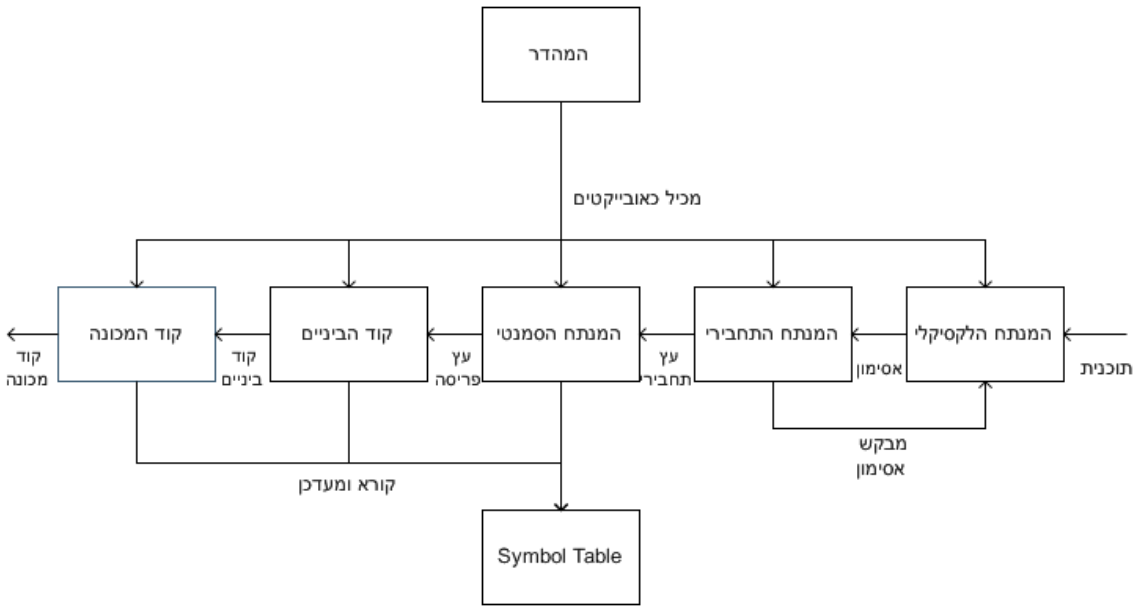
בניגוד לשפות תכנות אחרות בהן הווקטור משמש כמבנה נתונים המאחסן זוג של מספרי float, ב++C ווקטור הוא כמו שילוב של מערך, רשימה מקושרת, תור ומחסנית. כמו map, ו-pair הוא חלק מספריית STL – Standard Tenplate Library. המבנה ווקטור בעל תכונת האינדקס של מערך, בעל תכונות הראש, זנב, וגמישות (הקצאה דינאמית) של רשימה מקושרת, בעל תכונת השליפה והדחיפה של מחסנית ותור. הווקטור הוא המבנה השמיש ביותר ב++C (לדעתי האישית) והרביתי להשתמש בו, במידת הצורך.

תכנון

חלוקה למודולים

מודול	Header files	Source files
מנתח מילונאי	LexicalAnalyzerBase.h LexicalAnalyzerStateMachine.h LexicalAnalyzer.h	LexicalAnalyzerStateMachine.cpp LexicalAnalyzer.cpp
מנתח תחבירי	ParserBase.h Parser.h	Parser.cpp
מנתח לשוני	ParseTree.h	ParseTree.cpp
מחולל קוד ביניים	SyntaxTree.h MiddleCodeGenerator.h	SyntaxTree.cpp MiddleCodeGenerator.cpp
מחולל תכנית יעד	CodeGenerator.h	CodeGenerator.h
Symbol Table	Types.h SymbolTable.h	Types.cpp SymbolTable.cpp

<u>הקשר בין המודולים:</u>



רשימת הפעולות

Lexical Analyzer State Machine			
שם הפעולה	קלט	פלט	תפקיד
init_states			מאתחלת את וקטור המצבים לכל המצבים ההתחלתיים של האוטומט
state_machine_move	תו		מזיז את כל המצבים בוקטור המצבים למצב הבא בעקבות התו
no_states		בוליאני	מחזירה האם לא קיימים מצבים עכשוויים באוטומט
get_lexeme		Lexeme	מחזירה את מילת הקלט, המצב וסוג המילה בו נמצא האוטומט
get_next_state	תו ומצב	מצב	מחזירה את המצב שיתקבל ממעבר מהמצב הנתון בעקבות התו הנתון
init_middle			מאתחלת את מפת המעברים של האוטומט
init_finals			מאתחלת את מפת המצבים הסופים של האוטומט
init_keywords			מאתחלת את רשימת המילים השמורות
check_keyword			משנה את סוג האסימון במידה והוא מילה שמורה שלא זוהתה

Lexical Analyzer			
שם הפעולה	קלט	פלט	תפקיד
get_token		Token	מחזירה את האסימון הבא לפי הקלט
getLineNumber		מספר שלם	מחזירה את מספר השורה בה נמצאת קריאת הקלט

Parser			
שם הפעולה	קלט	פלט	תפקיד
parse		SyntaxTree*	מנתחת את הקלט ויוצרת ממנו SyntaxTree
init_table			מאתחלת את הטבלה שמכילה את כללי היצירה
init_stack			מאתחלת ומכינה את המחסנית לניתוח
reverse	וקטור של ParserStackEntry	וקטור של ParserStackEntry	מחזירה את הוקטור שמקבלת בסדר הפוך

Parse Tree			
שם הפעולה	קלט	פלט	תפקיד
insert	ParserTreeNodeBase*, ParserStackEntry*	ParserTreeNodeBase*	מכניסה את ParserStsckEntry לעץ, כך שהוא בן ל- ParserTreeNodeBase ומחיזרה את הצומת החדש שנוצר.
dfEvaluation	ParserTreeNodeBase*		סורקת את העץ בסריקה בסדר תחילי.
createNode	ParserStackEntry*	ParserTreeNodeBase*	יוצרת ומחזירה צומת חדש לעץ, שתוכנו הוא PSEntry.

ParserTreeNodeBase			
שם הפעולה	קלט	פלט	תפקיד
compInherited			מחשב את הערכים הנורשים של הצומת [מאחיו השמאליים ואבותיו]
compSynthesized			מחשב את הערכים הסינטזיים של הצומת [מבניו ואחיו הימניים]

SyntaxTreeNodeBase			
שם הפעולה	קלט	פלט	תפקיד
generate	MiddleCodeGenerator*	וקטור של *Command	מחזיר את ייצוג הביניים של צומת העץ
setAsNext	SyntaxTreeNodeBase*		מטפלת בקשר בין הצומת לזו שאחריה

MiddleCodeGenerator			
שם הפעולה	קלט	פלט	תפקיד
generate	SyntaxTree*	וקטור של *Command	מייצר ומחזיר את ייצוג הביניים של העץ שהתקבל
getTemp	מספר שלם	TempEntry*	מחזיר רשומה פנויה של משתנה זמני
backpatch			מסדרת את התגיות והקפיצות בקוד הביניים

Symbol Table			
שם הפעולה	קלט	פלט	תפקיד
addMethod		MethodEntry*	יוצרת, שומרת ומחזירה רשומת-פעולה חדשה ריקה
getMethod	מחרוזת	MethodEntry*	מחזירה רשומת-פעולה קיימת ששמה הוא המחרוזת. אם לא קיימת תחזיר NULL
getMethod	מספר שלם	MethodEntry*	מחזירה רשומת-פעולה קיימת שה-id שלה הוא המספר אם לא קיימת תחזיר NULL
addIOFunc	מחרוזת	MethodEntry*	יוצרת, שומרת ומחזירה רשומת-פעולה חדשה שסוגה הוא IOFunc
getIOFunc	מחרוזת	MethodEntry*	מחזירה רשומת-פעולה קיימת ששמה הוא המחרוזת. אם לא קיימת תחזיר NULL

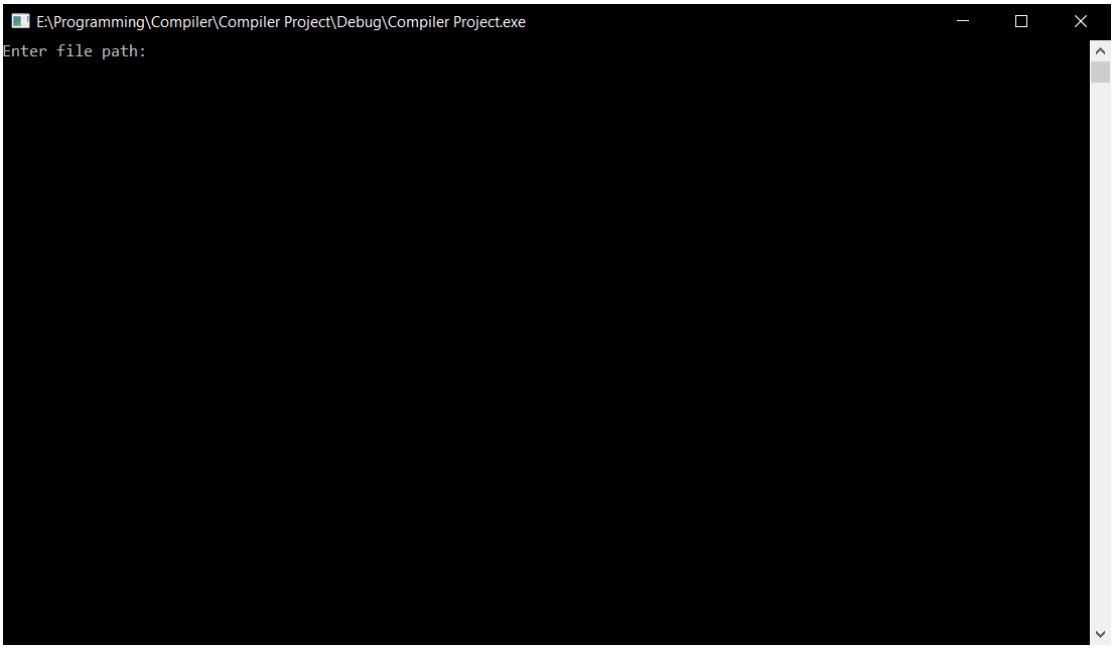
CodeGen			
שם הפעולה	**קלט**	**פלט**	**תפקיד**
generate	קוד הביניים ושם הקובץ		מתרגמת את קוד הביניים לשפת היעד ואת קוד היעד כותבת בקובץ
add_data_code			מכניסה אל קוד היעד את מקטע הנתונים [Data Segment]
generate_code_seg	קוד ביניים		מתרגמת את קוד הביניים ומכניסה אותו אל קוד היעד את בקטע הקוד [Code Segment]
store_all_regs			שומר את כל האוגרים בזיכרון ומשחרר אותם
store_reg	Register*		שומר את תוכן האוגר בזכרון ומשחרר אותו
store_var	BaseEntry*		שומר את המשתנה בזכרון
clear_reg	Register*		משחרר את האוגר
get_reg	Argument	Register*	מחזירה אוגר שיחזיק את הערך של Argument
get_spec_reg	מספר שלם ו- *BaseEntry	Register*	מחזירה אוגר ספציפי שמספרו הוא המספר, שיחזיק את הערך של *BaseEntry
assign	Register*, BaseEntry*		מגדירה שערך האוגר הוא ערך המשתנה, ללא שינוי פיזי בקוד היעד

MethodEntry			
שם הפעולה	קלט	פלט	תפקיד
add_var	מחרוזת ומספר שלם	VarEntry*	יוצרת, שומרת ומחזירה רשומת-משתנה חדשה ששמה הוא המחרוזת, והטיפוס שלה הוא המספר
get_var	מחרוזת	VarEntry*	מחזירה רשומת-משתנה קיימת ששמה הוא המחרוזת. אם לא קיימת תחזיר NULL
add_temp	מספר שלם	TempEntry*	יוצרת, שומרת ומחזירה רשומת-משתנה-זמני חדשה שהטיפוס שלה הוא המספר
get_temp	מספר שלם	TempEntry*	מחזירה רשומת-משתנה קיימת שטיפוסה הוא המספר. אם לא קיימת תחזיר NULL

מדריך למשתמש

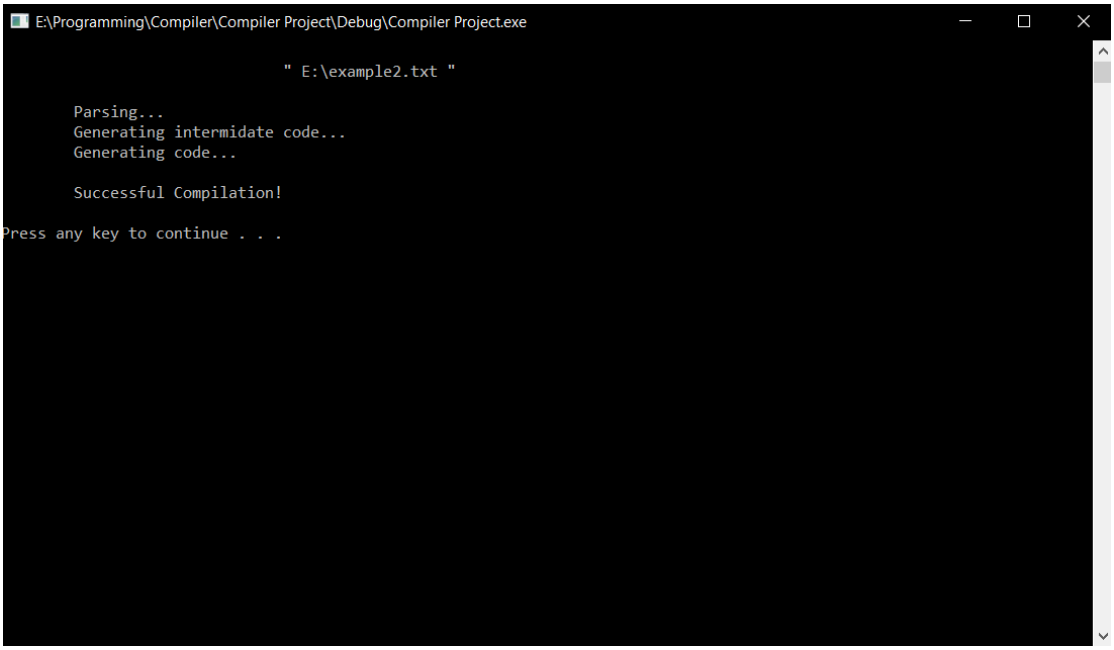
כעת אתן הסבר קצר עבור המשתמש כיצד לתפעל את המערכת כראוי.

עם פתיחת התכנה, מוצגת ההודעה הבאה:



הודעה זו חוזרת על עצמה פעמיים, בפעם הראשונה יש להכניס את מיקום הקובץ אותו תרצה לתרגם לאסמבלי, בפעם השניה את מיקום הקובץ המתורגם שיפלט.

לאחר מכן, יודפסו הודאות שגיאה\אזהרה. אם הודפסה הודעת שגיאה הקומפיילר עצר, התכנה לא תקינה ויש לתקן אותה ולנסות שוב. אם הודפסו הודעות אזהרה בלבד הכל בסדר, וניתן להריץ את התכנה. שים לב שיתכנו זגיאות זמן ריצה עקב התעלמות מאזהרות אלו.
אם אין שגיאות הודעה זו תודפס בסוף:



לאחר קומפילציה מוצלחת, יש להריץ את קובץ האסמבלי דרך masm32 ולקבל קובץ obj. לאחר מכן יש להריץ את קובץ הobj דרך link32. לאחר הלינקר יתקבל קובץ exe.

<u>השפה:</u>

שפת התכנות היא פרוצדוראלית ו-Case Sensitive.

טיפוסי המשתנים בשפה:

1. מספר שלם – בשפה: int
2. מחרוזת – בשפה: str
3. ערך בוליאני – בשפה: bool

שם משתנה בשפה מכיל אותיות ומספרים בלבד. כל פקודה תסתיים בנקודה-פסיק (;) וסיום כל בלוק יסתיים ב-end ומיד לאחר מכן במילה השמורה שמגדירה את התחלת הבלוק [לדוגמה, בלוק של פונקציה יסתיים ב-end function].

התוכנית תתחיל במילה Program ומיד אחריה שם התוכנית.

כל פונקציה תוצהר על ידי שימוש בטיפוס המוחזר, המילה השמורה function, שם הפונקציה והארגומנטים שלה, עם הפרדת פסיקים ביניהם, ובסוף נקודותיים. הקריאה לפונקציה תתבצע באמצעות המילה השמורה call, מיד אחריה שם הפונקציה והפרמטרים המתאימים בסוגרים, מופרדים בפסיקים.

האופרטורים הקיימים בשפה:

1. אופרטורים אריתמטיים: חיבור (+), חיסור (-), כפל (*), חילוק (/) ומודולו (%).
2. אופרטורים לוגיים: וגם (&&), או (||) ולא (!).
3. אופרטורים השוואתיים: גדול (<), גדול-שווה (=<), קטן (>), קטן-שווה (=>), שווה (==) ולא-שווה (=!).

הפקודות בשפה:

1. הצבה:	set <var> = <exp>
2. קריאה:	call <func> (<args>)
3. הצהרה על משתנה:	<type> <name>
4. הצהרה על פונקציה:	<type> function <name><args>:
5. תנאי:	if <exp> :
6. אחרת:	else :
7. לולאה:	while <exp> :
8. החזרה:	return <exp>

הערכת הפתרון

תכנון העבודה והביצוע הסופי היו שונים מכמה בחינות. במהלך ביצוע העבודה נאלצתי לשנות את הגדרות של השפה, כיוון שהן לא אפשרו ניתוח תחבירי בעזרת מנתח מעלה-מטה.

הוספת אופטימיזציה היא חשובה על מנת להפוך את המהדר למכליל יותר. האופטימיזציה היא תהליך חשוב גם מכיוון שהיא משפרת את איכות קוד היעד ובכך משפר את זמן הריצה של התוכנית.

לדעתי, קיים מקום לשיפור שפת התכנות. הוספת מערכים היא מעשה חשוב. מערך הוא אחד המבנים הבסיסיים והחזקים ביותר, שעל כל מתכנת מתחיל ללמוד להשתמש בו בתבונה. הוספת מערכים חשובה לא רק למען הגשמת המטרה שלשמה נבנה המהדר, אלא גם על מנת שהשפה תהיה turing complete, משמע ניתן להשתמש בה לכל מטרה ולתכנת איתה כל תכנה.

תוספת נוספת לשפת התכנות היא מבנים. מבנים הם תוספת משמעותית לשפה, כיוון שהם פותחים דלת לקראת תכנות מונחה עצמים.

ביבליוגרפיה

תודות:
תודה רבה למיכאל צ'רנובילסקי שעורר בי השראה ודחף אותי לעשות פרוייקט שמאתגר אותי.

אתרי אינטרנט:

- https://stackoverflow.com/
- https://en.cppreference.com/w/
- https://www.wikipedia.org/
- https://stackoverflow.com/questions/3060946/implementing-a-state-machine-in-c-how

נספח – חלקי קוד חשובים:

כל הפרוייקט מורכב מ4617 שורות קוד. מסיבה זו נכללו רק חלקי הקוד הפעילים, ללא classים ומימושים של פעולות נורשות.

<u>אלגוריתם מכונת המצבים של המנתח המילונאי:</u>

```
Token LexicalAnalyzer::get_token()
{
	Lexeme lexeme;
	char c;
	this->number_of_characters_read = 0;

	//initialize default list of states.
	this->state_machine->init_states();

	this->current_lexeme.lexeme = "";
	this->current_lexeme.state  = "";
	this->current_lexeme.type   = Tokens::Not_Final;

	//Peek into the next character and get the possible states.
	c = input.peek();
	this->state_machine->state_machine_move(c);

	//This while loop runs as long as we got input and as long as we have possible states.
	//It will onlt stop after we ran out of states, hopfully by then we will have a final lexeme.
	//If the lexeme is not final when we exit the loop, we report to
	//the user that he has a compilation error, and that the token was unidentified.
	while (!this->state_machine->no_states() && input.good())
	{
		//if we heve possible states from the peek, it means the next character is a part of the
		//lexeme so we read it as well.
		input.get();
		this->number_of_characters_read++;
		//We call get_lexeme(), this function will return the lexeme, it's type, and it's state.
		//If our lexeme is not final it will return an empty lexeme, with the type and state
		//"Not_Final"
		lexeme = this->state_machine->get_lexeme();
		//If it is final, we copy it's properties to our current_lexeme.
		if (lexeme.type != Tokens::Not_Final)
		{
			this->current_lexeme.lexeme = lexeme.lexeme;
			this->current_lexeme.state = lexeme.state;
			this->current_lexeme.type = lexeme.type;
		}
		//We have to peek and do one more check, because even if we have a final lexeme,
		//it still can be changed by the character after it.
		//For example '<' is a finished lexeme,
		//but it can be expanded by a '='.
		c = input.peek();
		this->state_machine->state_machine_move(c);
	}
	//checks if the current lexeme is a keyword.
	this->state_machine->check_keyword(&this->current_lexeme);
```

```
	if (this->current_lexeme.type == Tokens::Identifier)
	{
		//To avoid the use of an asm x86 keyword as an id
		this->current_lexeme.lexeme += "_";
	}

	else if (this->current_lexeme.type == Tokens::End_Of_Line)
	{
		this->line_number++;
		return get_token();
	}

	else if (this->current_lexeme.type == Tokens::White_Space)
	{
		return get_token();
	}

	Token token;
	token.lexeme = this->current_lexeme.lexeme;

	if (lexeme.type == Tokens::Not_Final)
	{
		Error_Handler::report(this->line_number, "Token \'" + lexeme.lexeme + "\' was not identified");
	}

	if (this->current_lexeme.type == Tokens::Not_Final) // handling special cases
	{
		if (input.eof()) // reached end of file
		{
			if (this->number_of_characters_read == 0)
			{
				token.type = Tokens::End_Of_File;
			}
			else
			{
				Error_Handler::report(this->line_number, "Token was probably not finished");
			}
		}
		else
		{
			Error_Handler::report(this->line_number, "Token was not identified");
		}
	}

	else
		token.type = this->current_lexeme.type;

	return token;

}
```

קובץ הheader הבסיסי מנתח המילונאי:

```
#pragma once
#include <string>
#include <queue>

typedef std::queue<struct Token> TokenQueue;

namespace Tokens
{
	enum TokenTypes
	{
		Start_Of_File, Identifier, If, Else, While, Return, Value,
		End_Of_File, End_Of_Line, Comma, Colon,
		Not_Equals, Equal_Sign, Equals, Bigger, Smaller, Bigger_Equal,
		Smaller_Equal, Plus, Minus, Multiply, Divide, Mod,
		DataType, Prog, End, Set, Func,
		Round_Brackets_Open, Round_Brackets_Close,
		And, Or, Not, Not_Final, White_Space, Semi_Colon, Call
	};
}

struct Lexeme
{
	std::string lexeme;
	std::string state;
	int type;
};

struct Token
{
	int type;
	std::string lexeme;
};
```

קובץ הheader הבסיסי של המנתח התחבירי (parser):

```
#pragma once
#include "SymbolTable.h"
#include "LexicalAnalyzerBase.h"
namespace Entries
{
	enum NonTerminals
	{
		program_ = 50, block_, statement_, decl_, decl_expand_, assignment_,
		func_call_, condition_, else_condition_, loop_,
		function_, args_, args_expand_, return_, args_send_, args_send_expand
	};

	enum Type
	{
		Terminal = 100, NonTerminal, Epsilon
	};

	enum Expressions
	{
		LowImportanceExp = 150, LowImportanceExpB, HighImportanceExp, HighImportanceExpB,
		Argument, HighImportanceOperators, LowImportanceOperators
	};
}

class ParserTreeNodeBase;

struct ParserStackEntery // Parser Stack Entry struct
{
	Entries::Type type;
	int value;
	ParserTreeNodeBase* ParserTreeNode;
	int line; // For error reporting

	bool operator<(const ParserStackEntery &p) const
	{
		return (value < p.value);
	}
};
```

<u>האלגוריתם העיקרי של המנתח התחבירי (parser):</u> אלגוריתם זה מוסבר בפרטי פרטים בעמוד 11.

```
SyntaxTree * Parser::parse()
{
	init_stack();
	ParserStackEntery cur;
	Token tok = lexer->get_token();

	while (!parser_stack.empty()) {
		cur = parser_stack.top();
		if (cur.type == Entries::Terminal) {
			if (cur.value == tok.type) {
				dynamic_cast<ParserTreeNodeTerminal*>(cur.ParserTreeNode)->type = tok.type;
				dynamic_cast<ParserTreeNodeTerminal*>(cur.ParserTreeNode)->value = tok.lexeme;
				parser_stack.pop();
				tok = lexer->get_token();
			}
			else {
				Error_Handler::report(cur.line, "Unexpceted token: \"" + tok.lexeme + "\"");
			}
		}
		else
		{	parser_stack.pop();
			ParseTable::iterator tmp_it = parse_table.find(std::make_pair(cur, tok.type));
			ParseTable::iterator ep_it = parse_table.find(std::make_pair(cur, Entries::Epsilon));
			if (tmp_it != parse_table.end()) {
				std::vector<ParserStackEntery> vec = reverse(tmp_it->second);
				for (ParserStackEntery i : vec){
					i.line = lexer->getLineNumber();
					i.ParserTreeNode = parse_tree->insert(cur.ParserTreeNode, &i);
					parser_stack.push(i);
				}
			}
			else if (ep_it != parse_table.end())
			{
				std::vector<ParserStackEntery> vec = reverse(ep_it->second);
				for (ParserStackEntery i : vec){
					i.line = lexer->getLineNumber();
					i.ParserTreeNode = parse_tree->insert(cur.ParserTreeNode, &i);
					parser_stack.push(i);
				}
			}
			else{
				Error_Handler::report(lexer->getLineNumber(), "Unexpceted token \"" + tok.lexeme +
"\"");
			}
		}
	}
	if (tok.type != Tokens::End_Of_File){
		Error_Handler::report(lexer->getLineNumber(), "End of file had already been reached");
	}
	parse_tree->dfEvaluation(parse_tree->root);
	syntax_tree->root = dynamic_cast<ParserTreeNodeProgram*>(parse_tree->root)->node_ptr;

	return syntax_tree;
}
```

כללי היצירה של השפה (534 שורות):

```
void Parser::init_table()
{
	parse_table.clear();
	int x;
	ParserStackEntery entry;
	std::vector<ParserStackEntery> list;

	// {x} <--- means x can be repeated, or can be ignored.
	/* Program */
	/* Program -> fileStart function {function} fileEnd */
	entry = { Entries::NonTerminal, Entries::program_ };
	x = Tokens::Prog;
	list.push_back({ Entries::Terminal    , Tokens :: Prog });
	list.push_back({ Entries::Terminal    , Tokens :: Identifier });
	list.push_back({ Entries::NonTerminal , Entries:: function_ });
	list.push_back({ Entries::Terminal    , Tokens::Prog });
	list.push_back({ Entries::Terminal    , Tokens :: End_Of_File });
	parse_table.insert(  std::make_pair(  std::make_pair(entry, x)  , list)   );
	list.clear();

	/* Function */
	/* Function -> id arguments colon curly_bracket_open block  */
	entry = { Entries::NonTerminal, Entries::function_ };
	x = Tokens::DataType;
	list.push_back({ Entries::Terminal    , Tokens:: DataType });
	list.push_back({ Entries::Terminal    , Tokens:: Func });
	list.push_back({ Entries::Terminal    , Tokens:: Identifier });
	list.push_back({ Entries::NonTerminal , Entries::args_ });
	list.push_back({ Entries::Terminal    , Tokens:: Colon });
	list.push_back({ Entries::NonTerminal , Entries::block_ });
	list.push_back({ Entries::Terminal    , Tokens::Func });
	list.push_back({ Entries::NonTerminal , Entries::function_ });
	parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
	list.clear();

	/* Function End */
	entry = { Entries::NonTerminal, Entries::function_ };
	x = Tokens::End;
	list.push_back({ Entries::Terminal   , Tokens:: End });
	parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
	list.clear();

	/* Args -> type id argsexpen */
	entry = { Entries::NonTerminal, Entries::args_ };
	x = Tokens::DataType;
	list.push_back({ Entries::Terminal   , Tokens::DataType });
	list.push_back({ Entries::Terminal   , Tokens::Identifier });
	list.push_back({ Entries::NonTerminal   , Entries::args_expand_ });
	parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
	list.clear();
```

```
/* Args -> epsilon */
entry = { Entries::NonTerminal, Entries::args_ };
x = Entries::Epsilon;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Args Expand ->  ',' args*/
entry = { Entries::NonTerminal, Entries::args_expand_ };
x = Tokens::Comma;
list.push_back({ Entries::Terminal   , Tokens::Comma });
list.push_back({ Entries::NonTerminal   , Entries::args_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Args Expand -> epsilon */
entry = { Entries::NonTerminal, Entries::args_expand_ };
x = Entries::Epsilon;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Return -> return liexp */
entry = { Entries::NonTerminal, Entries::return_ };
x = Tokens::Return;
list.push_back({ Entries::Terminal   , Tokens::Return });
list.push_back({ Entries::NonTerminal   , Entries::LowImportanceExp });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement -> Return */
entry = { Entries::NonTerminal, Entries::statement_ };
x = Tokens::Return;
list.push_back({ Entries::NonTerminal   , Entries::return_});
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement: Function Call */
/* Func -> call funcName () */
entry = { Entries::NonTerminal, Entries::func_call_ };
x = Tokens::Call;
list.push_back({ Entries::Terminal    , Tokens::Call });
list.push_back({ Entries::Terminal    , Tokens::Identifier });
list.push_back({ Entries::Terminal    , Tokens::Round_Brackets_Open });
list.push_back({ Entries::NonTerminal , Entries::args_send_ });
list.push_back({ Entries::Terminal    , Tokens::Round_Brackets_Close });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
/* ArgSend -> liexp argsendexpen */
entry = { Entries::NonTerminal, Entries::args_send_ };
x = Tokens::Identifier;
list.push_back({ Entries::NonTerminal , Entries::LowImportanceExp });
list.push_back({ Entries::NonTerminal , Entries::args_send_expand });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Value;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Not;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Round_Brackets_Open;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Call;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Minus;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

list.clear();


/* ArgSend -> epsilon */
entry = { Entries::NonTerminal, Entries::args_send_ };
x = Entries::Epsilon;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* ArgSendExpen -> , liexp argsendexpen */
entry = { Entries::NonTerminal, Entries::args_send_expand };
x = Tokens::Comma;
list.push_back({ Entries::Terminal    , Tokens ::Comma });
list.push_back({ Entries::NonTerminal , Entries::args_send_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* ArgSendExpen -> epsilon */
entry = { Entries::NonTerminal, Entries::args_send_expand };
x = Entries::Epsilon;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
/* Block */
/* Block -> statement block */
entry = { Entries::NonTerminal, Entries::block_ };
x = Entries::Epsilon;
list.push_back({ Entries::NonTerminal, Entries::statement_ });
list.push_back({ Entries::Terminal, Tokens::Semi_Colon });
list.push_back({ Entries::NonTerminal, Entries::block_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Block -> end */
entry = { Entries::NonTerminal, Entries::block_ };
x = Tokens::End;
list.push_back({ Entries::Terminal, Tokens::End });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();


//Statements
/* Statement -> Declaration */
entry = { Entries::NonTerminal, Entries::statement_ };
x = Tokens::DataType;
list.push_back({ Entries::NonTerminal, Entries::decl_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement -> Assignment */
entry = { Entries::NonTerminal, Entries::statement_ };
x = Tokens::Set;
list.push_back({ Entries::NonTerminal, Entries::assignment_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/*Statement -> FuncCall */
entry = { Entries::NonTerminal, Entries::statement_ };
x = Tokens::Call;
list.push_back({ Entries::NonTerminal, Entries::func_call_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement -> Condition */
entry = { Entries::NonTerminal, Entries::statement_ };
x = Tokens::If;
list.push_back({ Entries::NonTerminal, Entries::condition_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
/* Statement -> Loop */
entry = { Entries::NonTerminal, Entries::statement_ };
x = Tokens::While;
list.push_back({ Entries::NonTerminal, Entries::loop_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement: Declaration */
/* Dec -> type id expention */
entry = { Entries::NonTerminal, Entries::decl_ };
x = Tokens::DataType;
list.push_back({ Entries::Terminal, Tokens::DataType });
list.push_back({ Entries::Terminal, Tokens::Identifier });
list.push_back({ Entries::NonTerminal, Entries::decl_expand_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement: Declaration: Expention */
/* DecExp -> , id decExpen */
entry = { Entries::NonTerminal, Entries::decl_expand_ };
x = Tokens::Comma;
list.push_back({ Entries::Terminal, Tokens::Comma });
list.push_back({ Entries::Terminal, Tokens::Identifier });
list.push_back({ Entries::NonTerminal, Entries::decl_expand_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement: Declaration: Expention */
/* DecExp -> epsilon */
entry = { Entries::NonTerminal, Entries::decl_expand_ };
x = Entries::Epsilon;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement: Assignment */
/* Assignment -> set id "=" (expression) */
entry = { Entries::NonTerminal, Entries::assignment_ };
x = Tokens::Set;
list.push_back({ Entries::Terminal, Tokens::Set });
list.push_back({ Entries::Terminal, Tokens::Identifier });
list.push_back({ Entries::Terminal, Tokens::Equal_Sign });
list.push_back({ Entries::NonTerminal, Entries::LowImportanceExp });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
/* Statement: Condition */
/* Cond -> if exp then block else */
entry = { Entries::NonTerminal, Entries::condition_ };
x = Tokens::If;
list.push_back({ Entries::Terminal, Tokens::If });
list.push_back({ Entries::NonTerminal, Entries::LowImportanceExp });
list.push_back({ Entries::Terminal, Tokens::Colon });
list.push_back({ Entries::NonTerminal, Entries::block_ });
list.push_back({ Entries::Terminal, Tokens::If });
list.push_back({ Entries::NonTerminal, Entries::else_condition_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement: Else Condition */
/* else then [block] */
entry = { Entries::NonTerminal, Entries::else_condition_ };
x = Tokens::Else;
list.push_back({ Entries::Terminal, Tokens::Else });
list.push_back({ Entries::Terminal, Tokens::Colon });
list.push_back({ Entries::NonTerminal, Entries::block_ });
list.push_back({ Entries::Terminal, Tokens::Else });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement: Else Condition */
entry = { Entries::NonTerminal, Entries::else_condition_ };
x = Entries::Epsilon;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Statement: Loop */
/* Loop -> while (boolExp) do block */
entry = { Entries::NonTerminal, Entries::loop_ };
x = Tokens::While;
list.push_back({ Entries::Terminal, Tokens::While });
list.push_back({ Entries::NonTerminal, Entries::LowImportanceExp });
list.push_back({ Entries::Terminal, Tokens::Colon });
list.push_back({ Entries::NonTerminal, Entries::block_ });
list.push_back({ Entries::Terminal, Tokens::While });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Expressions */
/* High Importance Operators -> Multiply */
entry = { Entries::NonTerminal, Entries::HighImportanceOperators };
x = Tokens::Multiply;
list.push_back({ Entries::Terminal, Tokens::Multiply });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
/* High Importance Operators -> Division */
entry = { Entries::NonTerminal, Entries::HighImportanceOperators };
x = Tokens::Divide;
list.push_back({ Entries::Terminal, Tokens::Divide });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* High Importance Operators -> Modulu */
entry = { Entries::NonTerminal, Entries::HighImportanceOperators };
x = Tokens::Mod;
list.push_back({ Entries::Terminal, Tokens::Mod });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* High Importance Operators -> Equal */
entry = { Entries::NonTerminal, Entries::HighImportanceOperators };
x = Tokens::Equals;
list.push_back({ Entries::Terminal, Tokens::Equals });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* High Importance Operators -> NotEqual */
entry = { Entries::NonTerminal, Entries::HighImportanceOperators };
x = Tokens::Not_Equals;
list.push_back({ Entries::Terminal, Tokens::Not_Equals });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* High Importance Operators -> Bigger */
entry = { Entries::NonTerminal, Entries::HighImportanceOperators };
x = Tokens::Bigger;
list.push_back({ Entries::Terminal, Tokens::Bigger });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* High Importance Operators -> Bigger Equals */
entry = { Entries::NonTerminal, Entries::HighImportanceOperators };
x = Tokens::Bigger_Equal;
list.push_back({ Entries::Terminal, Tokens::Bigger_Equal });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* High Importance Operators -> Smaller */
entry = { Entries::NonTerminal, Entries::HighImportanceOperators };
x = Tokens::Smaller;
list.push_back({ Entries::Terminal, Tokens::Smaller });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
/* High Importance Operators -> Smaller Equals */
entry = { Entries::NonTerminal, Entries::HighImportanceOperators };
x = Tokens::Smaller_Equal;
list.push_back({ Entries::Terminal, Tokens::Smaller_Equal });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Low Importance Operators -> Plus */
entry = { Entries::NonTerminal, Entries::LowImportanceOperators };
x = Tokens::Plus;
list.push_back({ Entries::Terminal, Tokens::Plus });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Low Importance Operators -> Minus */
entry = { Entries::NonTerminal, Entries::LowImportanceOperators };
x = Tokens::Minus;
list.push_back({ Entries::Terminal, Tokens::Minus });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Low Importance Operators -> Or */
entry = { Entries::NonTerminal, Entries::LowImportanceOperators };
x = Tokens::Or;
list.push_back({ Entries::Terminal, Tokens::Or });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/* Low Importance Operators -> And */
entry = { Entries::NonTerminal, Entries::LowImportanceOperators };
x = Tokens::And;
list.push_back({ Entries::Terminal, Tokens::And });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();


/* ARGUMENTS */
/*    Arg -> (Exp)    */
entry = { Entries::NonTerminal, Entries::Argument };
x = Tokens::Round_Brackets_Open;
list.push_back({ Entries::Terminal, Tokens::Round_Brackets_Open });
list.push_back({ Entries::NonTerminal, Entries::LowImportanceExp });
list.push_back({ Entries::Terminal, Tokens::Round_Brackets_Close });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
/*    Arg -> Not Exp    */
entry = { Entries::NonTerminal, Entries::Argument };
x = Tokens::Not;
list.push_back({ Entries::Terminal, Tokens::Not });
list.push_back({ Entries::NonTerminal, Entries::Argument });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/*    Arg -> Value    */
entry = { Entries::NonTerminal, Entries::Argument };
x = Tokens::Value;
list.push_back({ Entries::Terminal, Tokens::Value });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/*    Arg -> Identifier    */
entry = { Entries::NonTerminal, Entries::Argument };
x = Tokens::Identifier;
list.push_back({ Entries::Terminal, Tokens::Identifier });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/*    Arg -> Func call    */
entry = { Entries::NonTerminal, Entries::Argument };
x = Tokens::Call;
list.push_back({ Entries::NonTerminal, Entries::func_call_ });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();

/*    Arg -> '-' arg   */
entry = { Entries::NonTerminal, Entries::Argument };
x = Tokens::Minus;
list.push_back({ Entries::Terminal, Tokens::Minus });
list.push_back({ Entries::NonTerminal, Entries::Argument });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
/* HighImportanceExp -> arg hiexpb */
entry = { Entries::NonTerminal, Entries::HighImportanceExp };
list.push_back({ Entries::NonTerminal, Entries::Argument });
list.push_back({ Entries::NonTerminal, Entries::HighImportanceExpB });

x = Tokens::Identifier;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Value;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Not;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Round_Brackets_Open;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Call;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Minus;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
/* HIExpB -> highImportnceOperator arg hiexpb */
entry = { Entries::NonTerminal, Entries::HighImportanceExpB };
x = Tokens::Multiply;
list.push_back({ Entries::NonTerminal, Entries::HighImportanceOperators });
list.push_back({ Entries::NonTerminal, Entries::Argument });
list.push_back({ Entries::NonTerminal, Entries::HighImportanceExpB });
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Divide;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Mod;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Equals;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Not_Equals;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Bigger;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Bigger_Equal;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Smaller;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

x = Tokens::Smaller_Equal;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

list.clear();

/* HIExpB -> epsilon */
entry = { Entries::NonTerminal, Entries::HighImportanceExpB };
x = Entries::Epsilon;
parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
list.clear();
```

```
        /* LIExp -> HIExp LIExpB */
        entry = { Entries::NonTerminal, Entries::LowImportanceExp };
        x = Tokens::Identifier;
        list.push_back({ Entries::NonTerminal, Entries::HighImportanceExp });
        list.push_back({ Entries::NonTerminal, Entries::LowImportanceExpB });

        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

        x = Tokens::Value;
        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

        x = Tokens::Not;
        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

        x = Tokens::Round_Brackets_Open;
        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

        x = Tokens::Call;
        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

        x = Tokens::Minus;
        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

        list.clear();

        /* LIExpB -> liop hiexp liexpb */
        entry = { Entries::NonTerminal, Entries::LowImportanceExpB };
        x = Tokens::Plus;
        list.push_back({ Entries::NonTerminal, Entries::LowImportanceOperators });
        list.push_back({ Entries::NonTerminal, Entries::HighImportanceExp });
        list.push_back({ Entries::NonTerminal, Entries::LowImportanceExpB });

        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

        x = Tokens::Minus;
        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

        x = Tokens::And;
        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));

        x = Tokens::Or;
        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
        list.clear();

        /* LIExpB -> epsilon */
        entry = { Entries::NonTerminal, Entries::LowImportanceExpB };
        x = Entries::Epsilon;
        parse_table.insert(std::make_pair(std::make_pair(entry, x), list));
        list.clear();


}
```

אלגוריתם הכנסה לעץ הניתוח התחבירי:

```
ParserTreeNodeBase* ParseTree::insert(ParserTreeNodeBase* parent, ParserStackEntery* data)
{
	//Creates a new node from the data and fills with relevant information
	ParserTreeNodeBase* n = createNode(data);
	n->line = data->line;
	n->parent = parent;
	n->sym = parent->sym;
	if (data->value != Entries::function_)
		n->functionId = parent->functionId;
	else
		n->functionId = n->sym->addMethod()->id;

	//Inserts the new node into the parent's children into the begining,
	//moving the other children down the vector
	parent->children.insert(parent->children.begin(), n);

	return n;
}
```

אלגוריתם סריקה לעומק – ניתוח עץ התחביר:

```
void ParseTree::dfEvaluation(ParserTreeNodeBase * n)
{
	for (ParserTreeNodeBase* m : n->children)
	{
		m->compInherited();
		dfEvaluation(m);
	}
	n->compSynthesized();
}
```

דוגמא לפונקצית compInherited – עבור חוליית ArgSend, המחשבת את הערכים שירש מחוליות מעליו בעץ:

```
void ParserTreeNodeArgSend::compInherited()
{
	if (parent->node_type == ParserTreeEnum::func_call_)
		exp_list = dynamic_cast<ParserTreeNodeFuncCall*>(parent)->exp_list;

	else if (parent->node_type == ParserTreeEnum::arg_send_expand)
		exp_list = dynamic_cast<ParserTreeNodeArgSend*>(parent->parent)->exp_list;
}
```

דוגמא לפונקצית compSynthesized – עבור חוליית Function:

```
void ParserTreeNodeFunction::compSynthesized()
{
	if (children.size() != 1)
	{
		//The fifth child of a Function block is a block node.
		block_ptr = dynamic_cast<ParserTreeNodeBlock*>(children[5])->node_ptr;
		if (!(dynamic_cast<ParserTreeNodeBlock*>(children[5])->hasRet)){
		Error_Handler::report(line, "Function \'" + (method_ent->name) + "\' has no return statement");
		}
		node_ptr = new SyntaxTreeNodeFunction(method_ent, block_ptr, line);
		func_list->push_back(node_ptr);
	}
}
```

מסמך הheader של מחולל קוד הביניים:

```
#ifndef __MCG_H
#define __MCG_H
#include "SyntaxTree.h"
#include "SymbolTable.h"
#include <fstream>
//These classes are declared here for pre-processor reasons (so we can use them without creating an include
//conflict) more info here: https://stackoverflow.com/questions/8526819/c-header-files-including-each-other-
mutually
class SyntaxTree;
class Command;
class SyntaxTreeNodeBase;

namespace Commands{
	enum CmdsEnum{
		Or, And, Not, Equal, NotEqual, Bigger, Smaller, BigEqu,
		SmaEqu, Plus, Minus, Mul, Div, Mod, UnrayMinus, Assign, Copy, IfNot, If,
		Goto, Proc, Return, EndProc, Call, IO, NoOp
	};
}
struct Argument{
	BaseEntry* base_ent;
	MethodEntry* method_ent;
	std::string val;
	Command* cmd;
};
class Command{
public:
	Command(Argument a, Argument b, Commands::CmdsEnum oper, BaseEntry* res) :
		arg1(a), arg2(b), op(oper), result(res), labeled(false) {};
	~Command();
	void output(std::ofstream& f);
public:
	static Argument NULLARG;
	Commands::CmdsEnum op;
	Argument arg1;
	Argument arg2;
	BaseEntry* result;
	int interCodeLine;
	bool labeled; // is a label  needed?
	int label; // label number for code generation
};

class MiddleCodeGenerator
{
public:
	MiddleCodeGenerator(SymbolTable* s) : symbol_table(s) {};
	~MiddleCodeGenerator();
public:
	TempEntry* getTemp(Types::TypesEnum type);
	std::vector<Command*> generate(std::string out_file, SyntaxTree* syntax_t);
private:
	void backpatch();
public:
	SymbolTable* symbol_table;
	std::vector<std::pair<Command*, SyntaxTreeNodeBase*>> bplist; // backpatch list
	std::vector<Command*> code; // final inter code
};

#endif
```

אלגוריתם ההמרה (תרגום) לקוד הביניים מהעץ הלשוני:

```
std::vector<Command*> MiddleCodeGenerator::generate(std::string of, SyntaxTree* syn)
{
	//generate is a recursive function which every syntax tree node has. It is used to generate commands
	//out of the contents of a node. When we encounter a jump to a node we havn't generated yet, we
	//push a pair of the jump command and the tree node to a backpatch list. The backpatch function
	//connects the jumps to the generated command, we call it after we generated the whole program.
	code = syn->root->generate(this);

	int i = 0;
	for (Command* q : code)
	{
		q->interCodeLine = i++;
	}

	backpatch();

	// print to file
	std::ofstream file;
	file.open(of);

	for (Command* q : code)
	{
		q->output(file);
	}

	file.close();
```