

ספר פרוייקט

מגישה: אביה גריידי

מספר זהות: 325449957

סמינר בית יעקב גברא

מנחה: המורה רבקי נילסון

תאריך הגשה: 22/06/23 | ג' תמוז תשפ"ג

תוכן עניינים

4	הצעת פרוייקט
6	מבוא / תקציר
6	הרקע לפרוייקט:
7	תהליך המחקר:
8	סקירת ספרות:
8	מטרות ויעדים:
9	מטרת העל:
9	מטרות נוספות:
9	יעדים:
9	אתגרים
10	מדדי הצלחה
10	תיאור מצב קיים
10	רקע תיאורטי
18	ניתוח חלופות מערכת
28	תיאור החלופה הנבחרת
31	אפיון המערכת
31	ניתוח דרישות המערכת
31	מידול המערכת
31	אפיון פונקציונאלי
32	ביצועים עיקריים
33	אילוצים
33	תיאור הארכיטקטורה
33	ארכיטקטורת רשת
33	תיאור פרוטוקולי התקשורת
33	שרת-לקוח
33	תיאור הצפנות
33	תרשים USE CASES / UML של המערכת המוצעת
33	רשימת Use Cases
33	Use Cases עיקריים

34	USE CASE DIAGRAM
34	מבני נתונים בפרוייקט
39	תרשים מחלקות
39	תיאור המחלקות המוצעות
39	תיאור התוכנה
39	אלגוריתמים מרכזיים
50	תיאור מסד הנתונים
50	תרשים מסכים
50	מדריך למשתמש
53	בדיקות והערכה
53	ניתוח יעילות
53	אבטחת מידע
53	פיתוחים עתידיים
53	מסקנות
54	ביבליוגרפיה
54	שלמי תודה

הצעת פרוייקט

סמל מוסד: 560342

שם מכללה: סמינר גברא

שם הסטודנט: אביה גריידי

ת.ז. הסטודנט: 325449957

שם הפרויקט: קומפיילר

תיאור הפרויקט:

הגדרת שפה עילית חדשה בסינטקס מוכר למתכנתים. הקוד יהודר לקובץ בשפת C#.

הגדרת הבעיה האלגוריתמית:

נבנה קומפיילר שתפקידו להדר את קוד המשתמש ולבדוק את תקינותו, אם הוא אכן תקין מבחינה דקדוקית, תחבירית וסמנטית – הקוד יהודר לשפת C# ויוצג קוד בשפת C# למשתמש.

רקע תיאורטי בתחום הפרויקט:

מחשבים אינם בעלי בינה ואינם מבצעים דבר פרט למספר מוגבל של פקודות חישוב אלקטרוניות. כל מחשב בנוי על שבב אלקטרוני אחד או יותר הקרוי מעבד המסוגל לקבל בצורה בינארית נתוני קלט, לבצע עליהם פעולה מסוימת ולחשב את הפלט. הכנסת הנתונים והפקודות למעבד מתבצעת בצורה של הכנסת קוד (מספר) של פקודה, יחד עם הנתונים שיעובדו. רצף הפקודות שניתן למעבד נקרא שפת מכונה. אותה שפת מכונה היא התוכנה המורצת במחשב.

בשל הקושי לתכנת את המחשב בשפת מכונה, פותחו עם השנים שפות תכנות מופשטות יותר, הנקראות בשם הקיבוצי שפות עיליות. הדקדוק המגדיר שפה עילית הוא מורכב יותר מדקדוק שפת המכונה, אך עם זאת ברור יותר לבני אדם ומאפשר התייחסות מופשטת יותר להיבטים שונים של החומרה והתוכנה, תחזוקתיות קלה יותר, ופיתוח מהיר יותר.

תפקידו של המהדר, אם כך, הוא להמיר את הדקדוק העילי לדקדוק שפת המכונה. באופן כללי ניתן לומר שככל ששפת התכנות מופשטת יותר, כך על המהדר להיות מותאם ומורכב יותר.

כך נפתח בפנינו עולם שלם של שפות תכנות, וכל אחד הרוצה לבנות שפה תכנותית משלו – יכול לעשות זאת ולבנות לכל שפה קומפיילר ייחודי משלה.

שפת C# הינה שפה נפוצה כיום בתכנות, היא שפה מונחית עצמים ומתממשת עם web. סינטקס השפה הזו מבוסס על ספריות רבות, ולצורך קידוד נדרשת למידה והכרה של הספריות האלו.

לכן החלטתי ליצור שפה חדשה בסינטקס שאינו מבוסס על ספריות קוד שיהודר לקובץ בשפת #C עבור הטמעת הקוד בסביבות עבודה של #C.

תהליכים עיקריים בפרויקט:

1. הידור קוד המשתמש - ההידור מחולק ל:
א. אנליזה: בו הקוד מחולק לחלקים המרכיבים אותו ונוצר ייצוג ביניים של הקוד, האנליזה גם היא מחולקת ל3 שלבים:
1. ניתוח לקסיקלי-דקדוקי - חלוקת קוד המשתמש לקטגוריות שמורות בשפה.
2. ניתוח סינטקטי - תחבירי - יצירת עץ גזירה לכל ביטוי מקוד המשתמש.
3. ניתוח סמנטי - משמעות הקוד, בדיקת תקינות הקוד מבחינת משתנים והפניות נכונות.
ב. סינתזה: יצירת התרגום לשפת היעד #C מתוך תרגום הביניים.
2. אם התגלו טעויות - החזרת רשימת הטעויות, אם לא - תרגום קוד המתכנת לשפת #C.
3. יצירת קובץ קוד בסיומת .cs.

תיאור הטכנולוגיה :

שפת תכנות בצד השרת: C.

הקוד יהודר לשפת #C.

מסד נתונים:

פרוטוקולי תקשורת:

לוחות זמנים:

- חקר המצב הקיים - ספטמבר
- הגדרת הדרישות - ספטמבר
- אפיון המערכת - אוקטובר
- אפיון בסיס הנתונים - נובמבר
- עיצוב המערכת - דצמבר
- בניית התוכנה - ינואר, פברואר
- בדיקות - מרץ
- הכנת תיק פרויקט - אפריל
- הטמעת המערכת - מאי
- הגשת פרויקט סופי - מאי

חתימת הסטודנט: **אזיה**

חתימת רכז המגמה: ר.נילסון

אישור משרד החינוך:

מבוא / תקציר

הרקע לפרוייקט:

כשהתחלתי לחשוב על נושא לפרוייקט גמר ידעתי שאני מחפשת משהו מאתגר. פרוייקט שיוציא ויבטא את הידע והכלים שרכשי ואת היכולות שלי. בנוסף רציתי לממש אותו באופן שיקנה לי ניסיון ופרקטיקה בבניית פרוייקט בעצמי, משלב חקר הנושא, הלמידה העצמית, התכנון ועד לשלב היישום עד הפרטים הקטנים. שמעתי על המושג "קומפילר" ושאפשר לבנות כזה לבד והחלטתי שאני הולכת לבנות כזה בעצמי.

למעשה, חשבתי על שפה חדשה מסויימת שארצה לפתח – שפה שדומה מאד לפייתון מבחינת תחביר וסמנטיקה. פייתון היא שפה פופולרית שנמצאת בשימוש נרחב בתעשייה ובאקדמיה, והתחביר שלה ידוע כקל לקריאה ולכתיבה. עם זאת ישנו היבט אחד בתחביר של python שיכול להיות מאתגר ומעצבן עבור מתכנתים מתחילים ומנוסים כאחד והוא השימוש ברווח לבן או טאב להזחה. הזחה מבוססת רווחים פירושו שהמבנה של תוכנית python נקבע לפי כמות הרווחים בתחילת כל שורה. זה יכול להיות מבלבל ומקור לשגיאות, במיוחד כאשר עובדים עם קוד שהועתק והודבק ממקורות שונים או כאשר משתפים פעולה עם מתכנתים אחרים שיש להם העדפות הזחה שונות. כדי לטפל בבעיה זו, החלטתי ליצור שפה דמוית Python המשתמשת ב- { } במקום רווח לבן עבור הזחה. תחביר זה דומה לזה המשמש בשפות תכנות פופולריות אחרות כמו C, Java, ו JavaScript – ויש לו יתרון שהוא מוכר ואינטואיטיבי יותר למתכנתים רבים. על ידי שימוש ב- { } להזחה, קיוויתי להפוך את השפה לנגישה יותר וקלה יותר לשימוש עבור מתכנתים בכל רמות המיומנות.

לאחר ששקלתי מספר אפשרויות לפלט של המהדר, החלטתי לתרגם את השפה דמוית פייתון ל-C#. היו כמה סיבות מדוע בחרתי ב-C# כשפת היעד של המהדר.

ראשית, C# היא שפה מודרנית מונחה עצמים שנמצאת בשימוש נרחב בתעשייה לבניית יישומים חזקים וניתנים להרחבה. יש לה סט עשיר של תכונות שהופכים אותה לכלי רב עוצמה לפיתוח תוכנה.

שנית, ל-C# יש מערכת חזקה המספקת בטיחות בזמן קומפילציה ועוזרת לתפוס שגיאות בשלב מוקדם בתהליך הפיתוח. זה חשוב במיוחד עבור פרויקטים בקנה מידה גדול שבהם שמירה על איכות הקוד ומזעור באגים היא חיונית.

שלישית, C# היא שפה מודפסת סטטית אשר מבצעת קומפילציה לקוד מכונה יעיל, מה שהופך אותה למתאימה ליישומים קריטיים לביצועים. יש לה גם סביבת זמן ריצה מתוכננת היטב המספקת המפשטות את ניהול הזיכרון וטיפול בשגיאות.

לבסוף, ל-C# יש קהילה גדולה ופעילה של מפתחים התורמים לפרויקטים בקוד פתוח, חולקים ידע ושיטות עבודה מומלצות ומספקים תמיכה והכוונה. זה מקל על מציאת משאבים וכלים לבנייה ותחזוקה של תוכנה ב-C#.

החלטתי ליצור קובץ ב-C# ולא להריץ את התוכנית. למה?

יצירת קובץ C# כפלט של המהדר מציעה מספר יתרונות על פני הפעלה ישירה של התוכנית:

ראשית, יצירת קובץ כפלט של המהדר מאפשרת למשתמשים להפיץ ולשתף את התוכניות שלהם ביתר קלות. C# היא שפה בשימוש נרחב שנתמכת בפלטפורמות וארכיטקטורות רבות ושונות, מה שאומר שניתן להרכיב ולהפעיל תוכניות במגוון רחב של מערכות. על ידי יצירת קובץ C# כפלט של המהדר, משתמשים יכולים להפיץ את התוכניות שלהם כקובצי הפעלה עצמאיים שניתן להפעיל בכל מערכת שתומכת ב-C#.

שנית, על ידי יצירת קובץ C# המשתמש יכול למנף את הכלים והספריות הקיימים במערכת האקולוגית של C#. לדוגמה, הוא יכול להשתמש ב- Visual Studio או ב- JetBrains Rider כדי לערוך, לנפות באגים וכד' בקוד שנוצר, מה שמספק חווית פיתוח מוכרת ועוצמתית.

שלישית, על ידי יצירת קובץ C#, אני יכולה לספק פורמט פלט ברור ושקוף שניתן להבין ולשנות בקלות על ידי המשתמש. זה מאפשר למשתמש לבדוק את הקוד שנוצר, לבצע שינויים במידת הצורך ולשלב אותו בפרויקטים הקיימים שלו ב-C#.

לאחר בניית השלב הראשון בבניית הקומפיילר שמתי לב שמה שעשיתי עד כה הוא גנרי. זאת אומרת שגם אם אשנה את רשימת המילים התקינות בשפה שלי – התוכנית תעבוד!

לכן החלטתי להמשיך ולבצע את הפרויקט אבל ברמה יותר גבוהה – בצורה גנרית (– דבר שהקשה לא מעט על השלבים הבאים...), והרעיון של השפה הנ"ל יהווה דוגמא לשפה שמותאמת אישית.

בעולם של היום, קיימות אינספור שפות תכנות המשמשות לפיתוח יישומי תוכנה למטרות שונות. עם זאת, יצירת שפת תכנות חדשה מאפס יכולה להיות משימה מורכבת ומאתגרת, במיוחד עבור לא מומחים בתחום מדעי המחשב. כאן נכנס הפרויקט שלי לתמונה. הפרויקט שלי נועד לספק מענה לאנשים שרוצים להמציא שפת תכנות משלהם מבלי לעבור את תהליך המורכב של עיצוב והטמעה של מהדר. אני בונה מהדר גנרי המאפשר למשתמשים להזין את מפרטי השפה שלהם, כולל סוגי האסימונים ודקדוק חסר הקשר (CFG) – מושגים שיפורטו בהמשך – ומייצר מהדר שיכול לתרגם קוד שנכתב בשפת המשתמש לשפת C#. פרויקט זה נועד להקל על אנשים להביע את רעיונותיהם בשפת תכנות ייחודית, מבלי ללמוד את המורכבות של עיצוב מהדר. על ידי מתן ממשק ידידותי למשתמש להגדרת מפרטי שפות ויצירת מהדרים, אני מקווה להעצים אנשים ליצור שפות תכנות משלהם ולהביא את הרעיונות שלהם למציאות. בספר פרויקט זה אתאר את תהליך בניית המהדר הגנרי, לרבות תכנון ויישום מחולל המהדר, וכן את האתגרים שעמדנו בפני והפתרונות שפיתחתי.

הפרויקט יורכב ממספר רכיבים, כולל כלי הגדרת שפה המאפשר למשתמשים לציין את התחביר והסמנטיקה של השפה החדשה שלהם, מנתח הממיר את קוד המקור לעץ תחביר מופשט, ומחולל קוד שמתרגם את עץ התחביר המופשט לתוך קוד C#.

תהליך המחקר:

על מנת לבצע את המשימה הנדרשת – בניית קומפיילר – חקרתי רבות אודות הרקע התיאורטי של הנושא ועל דרכי היישום בפועל – ימים ובעיקר לילות...

המהדר מורכב מכמה שלבים, שכל אחד מהם מבצע משימה מסוימת בתהליך התרגום. שלבים אלה כוללים ניתוח לקסיקלי (המכונה גם סריקה), ניתוח תחביר (ידוע גם כניתוח

סינטקטי), ניתוח סמנטי, יצירת קוד ואופטימיזציה (- בפרוייקט שלי לא ערכתי אופטימיזציה - מעבר לקוד ביניים ואז לשפת מכונה - אלא יצרתי קובץ בסיומת cs עם קוד המשתמש המתורגם לשפת C#).

מהדר גנרי הוא סוג של מהדר שנועד להיות גמיש וניתן להתאמה למגוון רחב של שפות תכנות. במקום להיות מותאם לשפה ספציפית, מהדר גנרי מספק מסגרת להגדרת התחביר והסמנטיקה של שפה, ויצירת מהדר שיכול לתרגם תוכניות הכתובות בשפה זו לשפה שונה.

למעשה עבדתי שלב שלב. בעבור כל שלב למדתי את החלק התיאורטי ועברתי לביצוע.

תוך כדי חקירת הנושא נחשפתי למושגים שונים.

הפרוייקט שלי כולל מושגים כמו תורת השפה הפורמלית, ביטויים רגולריים, תורת האוטומטים, CFG - דקדוק חסר הקשר, ואלגוריתמי ניתוח.

תורת השפה הפורמלית מספקת את הבסיס לתיאור התחביר והסמנטיקה של שפות תכנות תוך שימוש בדקדוקים. תורת האוטומטים מספקת מסגרת להבנת ההתנהגות של מהדרים ומערכות תוכנה אחרות המבצעות מניפולציות בשפות פורמליות. אלגוריתמי ניתוח משמשים לניתוח מבנה התוכניות ויצירת עץ תחביר מופשט המייצג את מבנה התוכנית.

סקירת ספרות:

קורס דיגיטלי

גד דהן, על שלב הניתוח הלקסיקלי ועל שלב הניתוח הסינטקטי.

אתרים נוספים בהם נעזרתי בכתיבת הפרוייקט:

הסבר על הניתוח הסמנטי ועל שלב יצירת קוד היעד:

- https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.htm
- <https://pgrandinetti.github.io/compiler/page/implementation-semantic-analysis/>
- <https://www.geeksforgeeks.org/compiler-construction-tools/>

התמקצעות בשפת C:

- <https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/strcmp-wcsncmp-mbsncmp?view=msvc-170>

טיפול בשגיאות:

- <https://stackoverflow.com/questions/72654401/a-breakpoint-instruction-debugbreak-statement-or-a-similar-call-was-execu>

מטרות ויעדים:

המטרה שלי הייתה לרכוש ידע בתחומים נוספים ולהתנסות בבניית פרוייקט מאתגר בעצמי.

מטרת העל:

לפתח מהדר גנרי המאפשר למשתמשים להגדיר שפות תכנות מותאמות אישית משלהם ולהרכיב את הקוד שלהם לשפת C#. המטרה היא לספק כלי גמיש וידידותי למשתמש המפשט את תהליך יצירת השפה ומאפשר למפתחים להתמקד בעיצוב והטמעה של תכונות השפה שלהם.

לפתח את המהדר כך שיכול לתרגם קוד שנכתב בשפה אחת לשפה שונה המבוססת על מפרטים מוגדרים על ידי המשתמש. זה מאפשר למשתמשים לכתוב קוד בשפה המועדפת עליהם ולתרגם אותו לשפה חדשה מבלי ללמוד תחביר או דקדוק חדשים. המטרה היא להקל על מפתחים לעבוד עם שפות תכנות מרובות ולהקל על פיתוח תוכנות שניתן להשתמש בהן בסביבות שונות.

מטרות נוספות:

- תכנון המערכת תוך שימת דגש על כתיבה נכונה, מאורגנת ומקצועית של הקוד.
- לבדוק את הפרויקט ביסודיות כדי לוודא שהוא פועל בצורה נכונה ומהימנה, ושהוא מייצר קוד C# נכון ויעיל עבור מגוון רחב של תוכניות קלט.

יעדים:

היעדים שלי:

- קליטת האסימונים – המילים התקינות בשפה מהמשתמש.
- קליטת ה-CFG – דקדוק חסר הקשר – הדקדוק התחבירי של השפה מהמשתמש.
- קליטת האילוצים הסמנטיים מהמשתמש.
- בניית מנתח לקסיקלי.
- בניית מנתח סינטקטי.
- בניית מנתח סמנטי.
- קליטת הקוד מהמשתמש.
- ניתוח לקסיקלי של הקוד – הפרדה ללקסמות וזיהוי האסימונים.
- ניתוח סינטקטי של הקוד – בדיקת תקינות התחביר של הקוד ויצירת עץ ניתוח.
- ניתוח סמנטי של הקוד – בדיקת תקינות הסמנטיקה של הקוד.
- יצירת קובץ שמכיל את תרגום הקוד לשפת C#.

אתגרים

במהלך החשיבה על האלגוריתם וכן במהלך כתיבת האלגוריתם ניצבו בפניי אתגרים רבים.

אחד האתגרים המרכזיים שעמד בפני היה מימוש השלבים בצורה גנרית, להבטיח שהמהדר יהיה גמיש מספיק כדי להכיל מגוון רחב של שפות מוגדרות על ידי המשתמש, תוך שמירה על רמת דיוק ויעילות גבוהה. זה דרש לשקול היטב את העיצוב של המהדר, ולבדוק אותו בקפדנות כדי לזהות ולטפל בכל בעיה פוטנציאלית.

האתגרים היוו בעבורי מורכבות של ממש, אך בעזרת ד' צלחתי אותם והם עזרו לי לא מעט ברכישת מיומנויות חדשות, כמו:

- חקירת נושא הפרוייקט.
- לימוד עצמי של חומר חדש.
- התמצאות בסוגי שפות שונות.
- שימת לב למקרי קצה.
- פתרון באגים.
- כושר ריכוז בפרטים קריטיים רבים על מנת לצרפם לאלגוריתם אחד.
- עמידה בזמני הדד-ליין להגשות.

מדדי הצלחה

בניית קומפיילר על כל שלביו בהתאם לקלט המשתמש:

- בניית מנתח לקסיקלי המפריד את הקוד לאסימונים שהוגדרו על ידי המשתמש.
- בניית מנתח סינטקטי הבודק את תקינות המבנה התחבירי של הקוד בהתאם להגדרת המשתמש.
- בניית מנתח סינטקטי הבודק את תקינות הסמנטיקה של הקוד בהתאם להגדרת המשתמש.
- יצירת קובץ המכיל תרגום הקוד של המשתמש מהשפה החדשה לשפת C# כך שהתוכנית זהה ב - 100%.

תיאור מצב קיים

נכון לעכשיו, תעשיות רבות מסתמכות על שפות תכנות שהן למטרות כלליות ואינן מותאמות לצרכיהן הספציפיים.

רקע תיאורטי

מהדר הוא תוכנה המתרגמת קוד מקור שנכתב בשפת תכנות אחת לתוכנית מקבילה בשפה אחרת. המטרה העיקרית של מהדר היא ליצור קוד מכונה או שפת יעד אחרת שניתן להפעיל במחשב.

למעשה, יש סוגים שונים של שפות. יש שפות פרוצדורליות ושפות סקריפט.

סוג שפת התכנות המחייב את כל הקוד להיות מוגדר בתוך פונקציות, כמו C ו C# - מכונה בדרך כלל כשפת תכנות פרוצדורלית. תכנות פרוצדורלי היא פרדיגמת תכנות המתמקדת בהגדרת פרוצדורות (כלומר, פונקציות) המבצעות משימות ספציפיות ושינוי נתונים באמצעות שימוש בהצהרות. בשפות תכנות פרוצדורליות, זרימת התוכנית העיקרית היא בדרך כלל סדרה של קריאות פונקציות המבצעות מניפולציות בנתונים ומבצעות פעולות. לעומת זאת Python ו JavaScript - הן שתיהן דוגמאות לשפות סקריפט התומכות במספר פרדיגמות תכנות, כולל תכנות פרוצדורלי, מונחה עצמים ופונקציונלי. שפות אלו מאפשרות גמישות רבה יותר מבחינת היכן ניתן להגדיר קוד, כמו מתן אפשרות לשימוש במשתנים גלובליים והגדרת קוד מחוץ לפונקציות.

גם מבחינת תחביר יש הבדלים:

תחביר C#: היא שפה בהקלדה סטטית, כלומר יש להצהיר על משתנים עם סוג נתונים ספציפי בזמן ההידור. Python משתמש בצורה דינמית, מה שאומר שלא צריך להצהיר על משתנים עם סוג נתונים ספציפי והם יכולים לשנות סוג בזמן ריצה.

בפרויקט שלי הנחתי למשתמש לבחור את סגנון השפה שלו. ההבדלים ניכרו בעיקר בשלב הניתוח הסמנטי.

להלן השלבים הכרוכים בבניית מהדר:

1. ניתוח לקסיקלי: זהו השלב הראשון של תהליך ההידור. בשלב זה, קוד הקלט מפורק לרצף של אסימונים – מילים תקינות בשפה שניתן לעבד על ידי המהדר. זה כרוך בזיהוי מילות מפתח, מזהים, אופרטורים, ואלמנטים אחרים של שפת התכנות.
2. ניתוח סינטקטי: זהו השלב השני של תהליך הקומפילציה. בשלב זה, האסימונים שנוצרו בשלב הניתוח המילוני מנותחים כדי לקבוע אם הם יוצרים תחביר תקף בהתאם לכללי הדקדוק של שפת התכנות. זה כולל בניית עץ ניתוח או עץ תחביר מופשט.
3. ניתוח סמנטי: זהו השלב השלישי בתהליך הקומפילציה. בשלב זה, המהדר בודק האם התחביר של התוכנית נכון מבחינה סמנטית. זה כולל בדיקת סוגים – טיפוסים, בדיקת היקף ובדיקות אחרות כדי להבטיח שהתוכנית מעוצבת היטב ומשמעותית.
4. יצירת קוד: זהו השלב הרביעי בתהליך הקומפילציה. בשלב זה, המהדר מייצר את קוד היעד. זה כרוך בתרגום הקוד משפת המשתמש לשפת C#.

ניתוח לקסיקלי | Lexical Analyzer

הגדרות קשורות:

Lexeme – לקסמה: סדרת אותיות או תווים המופרדת משאר התוכנית באופן מוסכם. לדוגמא על ידי רווח או נקודה.

Token – אסימון: רצף תווים – לקסמה המוכרת בשפה.

סוגי אסימונים:

ישנם מספר סוגים של אסימונים הנפוצים בבניית מהדר ובשלבי הניתוח של תהליך הקומפילציה. להלן כמה מהסוגים הנפוצים ביותר:

1. מילות מפתח: אלו מילים שמורות בעלות משמעות ספציפית בשפת התכנות, כגון: `if`, `else`, `for`, `while` וכד'.

2. מזהים: אלו הם שמות שניתנו למשתנים, פונקציות או ישויות תוכנה אחרות, כגון: `x`, `myFunction`, `myClass` וכד'.

3. אופרטורים: אלו הם סמלים או מילים המבצעים פעולה ספציפית בשפת התכנות, כגון `+`, `-`, `*`, `=`, או `&&`.

ישנם מספר סוגים של אופרטורים:

- אופרטורים אריתמטיים: אופרטורים אלו משמשים לביצוע פעולות מתמטיות על ערכים מספריים. לדוגמא: חיבור (+), חיסור (-), כפל (*), חילוק (/) ומודולוס (%).
 - אופרטורים להשוואה: אופרטורים אלו משמשים להשוואת ערכים ולהחזרת ערך בוליאני true או false על סמך ההשוואה. לדוגמא: גדול מ- (>), קטן מ- (<), שווה ל- (=), לא שווה ל- (!=), גדול או שווה ל- (>=), וקטן או שווה ל- (<=).
 - אופרטורים לוגיים: אופרטורים אלו משמשים לשילוב ערכים בוליאניים ולהחזרת תוצאה בוליאנית. לדוגמא: לוגי AND (&), לוגי OR (||), ולוגי לא (!).
 - אופרטורים להקצאה: אופרטורים אלו משמשים להקצאת ערך למשתנה. לדוגמא: שווה (=), פלוס-שווה (+), מינוס-שווה (-), כפול-שווה (*), חילוק-שווה (/), ו-מודולוס-שווה (%=).
 - אופרטורים סיביות: אופרטורים אלו משמשים לביצוע פעולות סיביות על ערכים בינאריים. לדוגמא: XOR (^), OR (|), AND (&), בכיוון סיביות (^), העברה שמאלה (<<) והזזה ימינה (>>).
 - אופרטורים מותנים: אופרטורים אלו משמשים ליצירת ביטויים מותנים המחזירים ערך המבוסס על תנאי בוליאני. האופרטור המותנה הנפוץ ביותר הוא האופרטור הטרינרי (:?), המשמש להקצאת ערך למשתנה המבוסס על תנאי.
4. מפרידים, סימני פיסוק: אלו הם סמלים או תווים המפרידים בין חלקים שונים של התוכנית, כגון פסיקים, נקודות פסיק או סוגריים.
5. ליטרלים: אלו הם ערכים קבועים המשמשים בתוכנית, כגון מספרים, מחרוזות או ערכים בוליאניים.
6. הערות ותווים מיוחדים: מבחינה טכנית אלה אינם אסימונים, ההערות משמשות כדי לספק הסברים או הערות הניתנות לקריאה על ידי אדם בתוך הקוד. תווים מיוחדים אלו לדוגמא טאבים, רווחים, שורה חדשה וכד'. המהדר מתעלם מהם.
- שלב זה אחראי על קריאת הקלט, הפרדתו ליחידות לקסיקליות (לקסמות), זיהוי כל לקסמה מאיזה סוג היא ויצירת אסימון בהתאם המכיל שם ותכונות, דיווח על שגיאות – לקסמות לא חוקיות בשפה.
- לדוגמא, בעבור הקוד:

```
if(x==5)
```

ניצור מערך שיכיל:

if	(x	==	5)
----	---	---	----	---	---

כאשר כל תא במערך הוא מסוג אסימון ומכיל את התכונות המיוחדות לו.

על מנת להכריז על האסימונים התקינים בשפה נשתמש בביטויים רגולריים וניצור אוטומט דטרמיניסטי סופי (DFA) המבטא את הביטויים הרגולריים שישמש לסריקת הקלט וזיהוי האסימונים.

ביטויים רגולריים ואוטומטים משמשים לתיאור תבניות ומבנים במחרוזות ובנתונים אחרים. ביטוי רגולרי הוא רצף של תווים המגדיר דפוס חיפוש. הוא משמש להתאמה ולטיפול בטקסט בהתבסס על קבוצה של כללים ותבניות. ניתן להשתמש בהם גם כדי להגדיר תבניות מורכבות יותר באמצעות מטא-תווים, כגון * (אפס או יותר מופעים של התו הקודם), + (מופע אחד או יותר של התו הקודם), ו-? (אפס או מופע אחד של התו הקודם).

אוטומט, המכונה גם מכונת מצבים סופיים, הוא מודל מתמטי המתאר מערכת שיכולה להיות באחד ממספר סופי של מצבים בכל זמן נתון. הוא משמש לתיאור ההתנהגות של מערכות שניתן לעצב כרצף של אירועים או תשומות – במקרה שלי – רצף התווים המזהים לקסמה. האוטומט מורכב מקבוצה של מצבים, קבוצה של תשומות ומערכת של מעברים המתארים כיצד המערכת עוברת ממצב אחד לאחר על סמך הקלט.

אני השתמשתי בביטויים רגולריים כדי להגדיר את התבניות המוכרות על ידי האוטומט ואת המעברים בין מצבים באוטומט.




איך זה עובד?

נגדיר כל אסימון בשפה בביטוי רגולרי.

לדוגמא:

האסימונים if, else, elif.

נסמן:

מצב  | מצב סופי  | מעבר (בעבור תו מסוים)  ϵ משמש לייצוג המחרוזת הריקה, שהיא מחרוזת באורך אפס.

בעבור כל ביטוי מתחילים ממצב ההתחלה.

המטרה: להפוך את הביטויים הרגולריים לאוטומט מצבים סופי (דטרמיניסטי).

נהפוך תחילה את הביטויים הרגולריים לאוטומט לא סופי (אי-דטרמיניסטי).

מה ההבדל בין אוטומט דטרמיניסטי לאי-דטרמיניסטי?

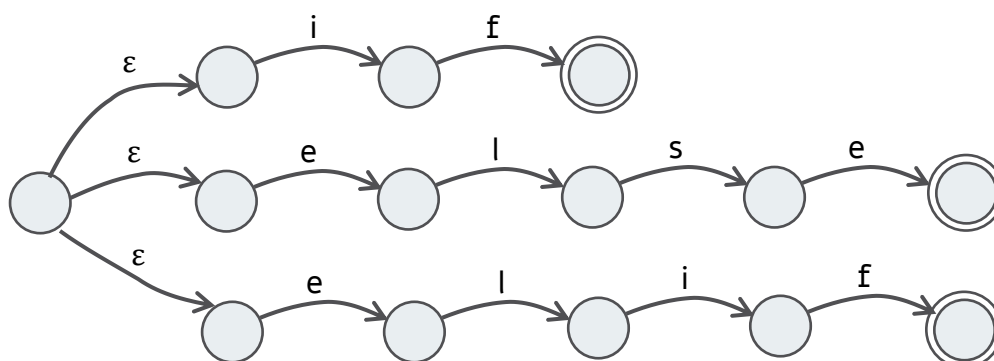
אם אני נמצאת בנקודה מסוימת:

באוטומט דטרמיניסטי יש לי רק דרך אחת לעבור – אפשרות אחת. זה מתבצע פעולה אחר פעולה עד הסוף.

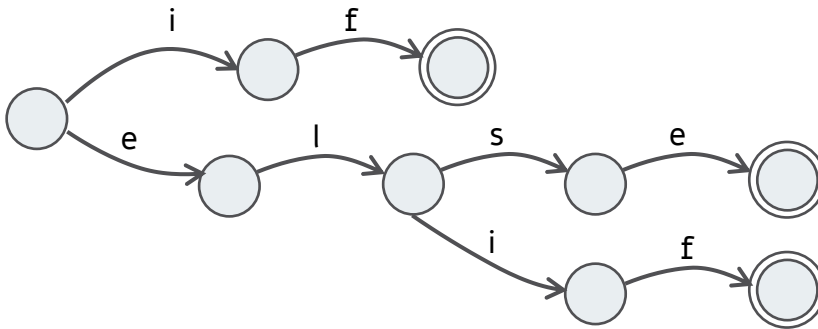
לעומת זאת באוטומט לא דטרמיניסטי בחלק מהנקודות אני יכולה לבצע 2 פעולות או יותר, יש כמה דרכים שאפשר לעבור אליהן.

בדוגמא:

מנקודת ההתחלה, בעבור התו e יש לי שני דרכים אפשר לעבור.



נהפוך את האוטומט הלא דטרמיניסטי לדטרמיניסטי.
למה? כי אני רוצה לבנות מערכת שפועלת באופן אוטומטי ולא נתקעת.



בעת נריץ את האוטומט הדטרמיניסטי על הקלט לזיהוי הביטוי המתאים.

לסיכום, בשלב זה קוד הקלט מפורק לרצף של אסימונים שניתן לעבד על ידי המהדר. לאחר מכן, האסימונים מועברים לשלב הבא של תהליך ההידור, שהוא שלב ניתוח התחביר.

ניתוח סינטקטי | Syntactic Analyzer

זהו השלב המורכב ביותר. (במיוחד בעקבות כך שהחלטתי להפוך את הפרוייקט לגנרי!) אחרי שזיהינו את המילים בקוד השלב הבא הוא להבין את מבנה המשפט. לאיזה סוג של משפט הוא מתאים והאם המשפט הזה הוא תקני בשפה. לכל שפת תכנות יש דקדוק שמאפיין את החוקיות של המשפטים שבה. לדוגמא, אחרי `if` חייב להיות (). לפי הדקדוק ניתן לזהות האם משפט מסוים שנכתב בשפה הוא משפט תקין או לא.

המטרה: להבין את המבנה של התוכנית.

איך נבצע את זה?

נחלק את התוכנית שלנו למבנה היררכי. לדוגמא: תכנית C בנויה מפונקציות (גם `main` היא פונקציה), כל פונקציה בנויה מהצהרות ופקודות, כל פקודה בנויה מביטויים וכו' צורה פשוטה אך מדויקת לניסוח מבנה של תכנית (בשפת תוכנית ספציפית) היא על ידי: **דקדוק חסר הקשר**.

CFG | Context Free Grammar

דקדוק חסר הקשר נותן לי בעצם את המבנה, איזה סוגים של משפטים יכולים להיות בתוך המכלול של השפה שלי.

תפקידו של ה- `parser` – המנתח, לקרוא את רצף האסימונים, לוודא שהם מקיימים את הדקדוק או להתריע על שגיאות ולבנות את עץ הגזירה של התכנית.

לכאורה אפשר לנסות לזהות את התחביר של השפה דרך ביטוי רגולרי אבל לאוטומט סופי יש מספר סופי של מצבים ולכן הוא לא יכול לספור כמות לא מוגבלת של סימנים. לדוגמא, הוא לא יוכל לספור את מספר מופעי הסוגריים שבביטוי חשבוני על מנת לוודא שהביטוי תקין. כדי לתאר את התחביר של השפה נצטרך מבנה פורמלי חזק יותר והוא הדקדוק חסר הקשר – CFG.

המבנה:

$$G = (V, T, P, S)$$

V – non-terminals – משתנים שניתן להחליף בביטויים אחרים

T – terminals – קבוצת המילים בשפה שלי

P – חוקי גזירה – חוקים שמתארים את הקשרים בין הנון-טרמינלים השונים

S – משתנה תחילי – מצב התחלה

מהי גזירה?

גזירה היא סדרה של החלפות של אותיות לא טרמינליות תוך שימוש בחוקי הגזירה. זו בעצם סדרה של צעדים על מחרוזת המורכבת מטרמינלים ונון-טרמינלים בה בכל צעד מוחלף סמל X המופיע במחרוזת בתת-מחרוזת a תוך שימוש בכלל גזירה: $X \rightarrow a$.

לדוגמא:

$$S \rightarrow E$$

$$E \rightarrow E + T$$

$$| T$$

$$T \rightarrow F * T$$

$$| F$$

$$F \rightarrow \text{identifier}$$

זהו דקדוק המגדיר ביטוי חשבוני פשוט.

כדי להראות שמשפט שייך לשפה צריך להראות סדרת גזירות, החל ממצב S, שמסיקה את המשפט.

לדוגמא, הביטוי $\text{identifier} * \text{identifier} + \text{identifier}$.

$$S \rightarrow E \rightarrow E + T \rightarrow T + T \rightarrow F * T + T \rightarrow \text{identifier} * T + T \rightarrow \text{identifier} * F + T \rightarrow$$

$$\text{identifier} * \text{identifier} + T \rightarrow \text{identifier} * \text{identifier} + F \rightarrow$$

$$\text{identifier} * \text{identifier} + \text{identifier}$$

עבור כל משפט ספציפי ו-CFG ספציפי ניתן לבצע את הגזירה של המשפט.

יש גזירה ימנית וגזירה שמאלית.

Top-Down

בגזירה שמאלית יוצרים את עץ הגזירה מלמעלה למטה – מהשורש לעלים.

בגזירה זו, LL קוראים את הקלט משמאל לימין וגם גוזרים משמאל לימין.

בגזירה זו לוקחים את הכללים ומתוכם מנסים ליצור את המשפט. עוברים על כל האפשרויות עד שמגיעים למצב שאין אפשרות להרכיב את המשפט. אם מצאתי שהצלחתי – המשפט תקין.

אנחנו מתחילים ממצב ההתחלה ובכל שלב עלינו לבחור כלל גזירה מתוך מגוון כללי הגזירה של השפה הנתונה על ידי מעבר על הקלט. על פי הקלט בוחרים את כלל הגזירה. בכל שלב נבחר כלל גזירה ונבדוק מול הקלט אם יש התאמה ממשיכים לתו הבא בקלט ובמידה ואין התאמה נצטרך לחזור שלב אחורה כדי לקבוע כלל גזירה אחר. דרך זו מתאימה כאשר אפשר להכריע בין האפשרויות לגזירה באופן וודאי. בדקדוק כללי תיתכן יותר מאפשרות אחת לגזור את המשתנה השמאלי ביותר לפי האיבר הבא בקלט – אין לי תחזית ברורה, ולכן נצטרך לחזור שלב אחורה ולנסות כלל גזירה אחר. או להשתמש ב – lookahead קטן כדי להחליט מהו כלל הגזירה המתאים הבא.

Bottom-Up

בגזירה ימנית יוצרים את עץ הגזירה מלמטה למעלה – מהעלים לשורש.

בגזירה זו, LR קוראים את הקלט משמאל לימין וגוזרים מימין.

בגזירה זו לוקחים את המשפט, מתחילים לקרוא עד שרואים שאפשר להתאים כלל גזירה. ברגע שרואים שאי-אפשר לגזור – נתקעתי באמצע המשפט – המשפט לא תקין. אם הגעתי לסוף המשפט – המשפט תקין.

הניתוח מלמטה כלפי מעלה מאפשר לדחות את בחירת הכלל להפעלה עד שיהיה מספיק מידע מהקלט כדי להחליט. ברגע שהמנתח מוצא קבוצת מילים שמייצגת צד ימין של חוק מסוים הוא מחליף אותה בצד שמאל שלו וכל בונה צומת נוסף בעץ.

גזירה זו נעזרת ב**טבלת ניתוח action & GO-TO**, ובמהלך הניתוח במחסנית.

טבלה זו מיוצגת כאוטומט בה מסומל בעבור כל אסימון מהקלט וכלל גזירה אפשרי לאיזה מצב ניתן לעבור. אם עדיין לא ניתן להגיד שסיימתי כלל גזירה מסוים מבצעים פעולת **shift** – הוספה של האסימון למחסנית ומעבר לתו הבא. אם זיהיתי סיום כלל גזירה מבצעים פעולת **reduce** – החלפה של כל האיברים המתאימים לכלל הגזירה בכלל הגזירה. כאשר במחסנית נשאר כלל הגזירה ההתחלתי – הקוד תקין. אם נתקעתי באיזשהו שלב – הקוד לא תקין.

על מנת לבנות את טבלת הניתוח המנתח נעזר בקבוצת ה- **first**, קבוצת ה- **follow**, וקבוצת הסגור של כל נון-טרמינל.

קבוצת ה- first היא אוסף של כל האיברים שיכולים להיות הראשונים לנון-טרמינל מסוים.

קבוצת ה- follow היא אוסף של כל האיברים שיכולים להיות אחרי נון-טרמינל מסוים.

קבוצת הסגור היא אוסף כללי הגזירה בהם יכול להתחיל נון-טרמינל מסוים.

לסיכום, בשלב זה המנתח עובר על זרם האסימונים של הקוד ובודק את תקינות התחביר של הקוד על פי כללי הגזירה – CFG שהגדיר המשתמש. במהלך הניתוח המנתח בונה את עץ הגזירה AST שמתאר את המבנה ההיררכי של הקוד. את עץ הגזירה מעבירים לשלב הבא בתהליך ההידור שהוא שלב ניתוח הסמנטיקה.

ניתוח סמנטי | Semantic Analyzer

אחרי שניתחנו את המשפט מבחינה תחבירית, בעזרת עץ הניתוח, המהדר יכול לבצע ניתוח סמנטי כדי לבדוק אם קוד המקור הגיוני ועוקב אחר הסמנטיקה של שפת התכנות. לא כל משפט נכון תחבירית הוא גם נכון סמנטית. לדוגמא:

`!f(x==5)`

המשפט נכון תחבירית אבל לא בטוח שהוא נכון סמנטית. יש כמה דברים שצריך לבדוק. האם המשתנה x מוגדר? האם הוא מאותחל? האם הוא מטיפוס `int` או `float`? בדיקות כמו אלו הן חלק מתפקידי המנתח הסמנטי.

שלב זה אחראי על הבדיקות שלהלן:

- בדיקה שהמשתנה מוגדר.
 - כשמשתמשים במשתנה - בדיקה שהוא מאותחל - יש בו ערך.
 - בשפות פרוצדורליות - כשיש השמה - בדיקה שהטיפוסים שווים או תקינים.
 - בביטויים אריתמטיים / ביטויי תנאי וכד' - בדיקה על כל חלק בביטוי כך שהביטוי בכללותו מאותו טיפוס או תקין בהתאם להגדרות.
 - בדיקה על משתנה האם הוא מוכר בתוך הבלוק הנוכחי. פונקציה או לולאה.
 - העמסת פונקציות - בדיקה שהפונקציה קיבלה את מספר הפרמטרים הנדרש.
- בשלב זה על המנתח להתייחס בהתאם להמרות. טיפוס הגדרה וכו'

על מנת לבצע את הבדיקות הנ"ל המנתח נעזר ב**טבלת הסמלים**.

טבלת הסמלים – Symbol Table היא מבנה נתונים שמייצג טבלה כאשר לכל `identifier` שומרים את התכונות שלו כמו המיקום שלו בקוד, טיפוס, `scope` – תחום ההגדרה שלו וכד'.

הטבלה נבנית במהלך הניתוח. למשל: המנתח הלקסיקלי מזהה את המשתנים אבל לא מזהה מה הטיפוס. את הטיפוס מגלה המנתח הסמנטי...

מכיוון שיש סגנונות שפה שונות, על המשתמש – יוצר השפה להגדיר האם חובה להצהיר על משתנים או לא. כלומר, האם השפה היא סטטית או דינאמית וכן את הטיפוסים התקינים לכל חלק בביטויים לכל אופרטור והסריקה תתבצע בהתאם.

לסיכום, בשלב זה מתבצעת סריקה על עץ הניתוח כאשר תוך כדי מבצעים את הבדיקות הנ"ל כך מוודאים את תקינות הסמנטיקה של הקוד. במידה ואין שגיאות סמנטיות – הקוד תקין לחלוטין ואפשר לעבור לשלב הבא – שלב תרגום הקוד.

יצירת קוד היעד | Generating the code

זהו השלב האחרון, ייצור קוד של שפת המטרה. בשלב זה מפיק הקומפיילר קוד מטרה מקוד המקור.

איך זה עובד?

מגדירים מחרוזת לתוכה משרשרים על פי התחביר של שפת C# את הקוד המתורגם בהתאם לקוד המשתמש.

אני מגדירה את התחביר של שפת C#.

והרי למעשה מה שמעניין אותי – מה שאני צריכה להעתיק מהקוד של המשתמש זה בעיקר את המשתנים, השמות של הפונקציות וליטרלים שהמשתמש הגדיר כמו מחרוזות להדפסה בעת הוראת פלט וכד'. ואת זה אני אעתיק ברגע המתאים כשאני אהיה בסריקה בתוך נונ-טרמינל מסוים אשר מוגדר שבתוכו תוכל להיות התערבות מהמשתמש כמו בדוגמא הנ"ל.

מבצעים סריקה על עץ הניתוח ובעבור כל נונ-טרמינל של קוד המשתמש משרשרים למחרוזת הקוד את החלק המתאים מהנון-טרמינל של C#.

לסיכום,

על המשתמש להחליט על סוגי האסימונים, על הדקדוק התחבירי של השפה החדשה, על הסגנון ועל האילוצים הסמנטיים שלה. התוכנה תבנה את הקומפיילר ומעתה המשתמש יוכל לכתוב קטעי קוד בשפה החדשה שלו שיתורגמו לשפת C#.

ניתוח חלופות מערכתי

ניתוח לקסיקלי

בניית המנתח

בניית האוטומט לזיהוי האסימונים.

חלופה שיצרתי:

בתחילה יצרתי קובץ ובו הגדרתי בעצמי את המעברים מכל מצב בעבור כל תו קלט אפשרי בהתאם לאסימונים שהגדרתי לשפה שלי.

בעבור הדוגמא של האסימונים if , elif , else הקובץ נראה כך:

		?
1	i	o
3	e	o

		?
2	f	1
		!
		2
		?
4	l	3
		?
5	i	4
7	s	4
		?
6	f	5
		!
		6
		?
8	e	7
		!
		8

בך שהעמודה הראשונה מסמלת מאיזה מצב, העמודה השניה מסמלת בעבור איזה תו והעמודה השלישית מסמלת לאיזה מצב לעבור.

מכיוון שיש מצבים סופיים סימנתי בעזרת ? ! כאשר: ? – לא מצב סופי ו ! – מסמן מצב סופי.

בהמשך הורדתי את הסימונים הנ"ל ולא סימנתי מראש מיהו מצב סופי ומי לא ויצרתי קובץ נוסף ובו הייתה רשימה של כל המצבים הסופיים. בעבור כל מצב סופי היה מצוין איזה מספר מצב הוא ומה הערך – איזה אסימון הוא מייצג ומהו סוג האסימון.

2	1	if
6	1	elif
8	1	else

1	i	0
3	e	0
2	f	1
4	l	3
5	i	4
7	s	4
6	f	5
8	e	7

אלו היו הפונקציות:

```
void fillDFA(DFA* dfa)
{
    char line[255] = "line"; // שורה מהקובץ
    char delimiter[] = ","; // סימן הפרדה
    char* context = NULL; // תוכן מופרד מהשורה
    char* slot = NULL; // משבצת

    int i = 0;

    int stateId = 0;

    char* t[4] = { '\0', '\0', '\0' };

    FILE* f1;

    errno_t error = fopen_s(&f1, "D:\\אביה\\תוכנה\\הנדסת פרויקט\\Transitions.csv", "r");
    if (error != 0)
    {
        printf("\nCould not open the file Transitions :(");
        return;
    }

    printf("\nThe file Transitions was opened successfully !!! :)");

    addState(dfa, stateId);

    while (fgets(&line, 255, f1)) // הלולאה עוברת שורה שורה
    {
        slot = strtok_s(line, delimiter, &context); // משבצת

        while (atoi(slot) != stateId)
        {
            stateId++;
            addState(dfa, stateId);
        }

        i = 0;
        while (slot != NULL) // בכל שורה יש 3 משבצות
        {
            t[i++] = slot;
            // Get next slot:
            slot = strtok_s(NULL, delimiter, &context);
        }

        addTransition(dfa, atoi(t[0]), t[1][0], atoi(t[2]));
    }

    addState(dfa, 110);
}
```

```
void fillTokens(DFA* dfa)
{
    char line[255] = "line"; // שורה מהקובץ
    char delimiter[] = ","; // סימן הפרדה
    char* context = NULL; // תוכן מופרד מהשורה
    char* slot = NULL; // משבצת

    int tokenId = 1;
    char* value;
    int tokenIdType;

    int stateId;

    char* name = '\0';

    FILE* f1;

    errno_t error = fopen_s(&f1, "D:\\אביה\\הנדסת תוכנה\\פרויקט\\Tokens.csv", "r");
    if (error != 0)
    {
        printf("\nCould not open the file Tokens :(");
        return;
    }

    printf("\nThe file Tokens was opened successfully !!! :)");

    while (fgets(&line, 255, f1)) // הלולאה עוברת שורה שורה
    {
        slot = strtok_s(line, delimiter, &context); // משבצת
        value = slot;
        slot = strtok_s(NULL, delimiter, &context);
        tokenIdType = atoi(slot);
        slot = strtok_s(NULL, delimiter, &context);
        stateId = atoi(slot);

        dfa->states[stateId].idInArray = tokenId;
        dfa->states[stateId].isFinal = true;
        strcpy_s(dfa->states[stateId].name, (rsize_t)_countof(dfa->states[stateId].name), value);

        if (sizeTokens == 0)
        {
            tokens = (Token*)malloc(sizeof(Token));
            createToken(&tokens[sizeTokens++], 0, 0, "identifier");
            tokens = (Token*)realloc(tokens, sizeof(Token) * (sizeTokens + 1));
        }
        else
        {
            tokens = (Token*)realloc(tokens, sizeof(Token) * (sizeTokens + 1));
        }
        if (tokens)
        {
            createToken(&tokens[sizeTokens++], tokenId, tokenIdType, value);
            tokenId++;
        }
    }
}
```

אלגוריתם הניתוח

האלגוריתם מורכב משני שלבים: הפרדת הקוד ללקסמות ובעבור כל לקסמה זיהוי האסימון המתאים – אם בכלל.

אפרט על צורות ניתוח שונות שנוכחתי לראות באלגוריתמים רבים.

חלופה 1:

יש תווים מיוחדים שנקראים 'מפרידים' והם: רווח, '\n' – שורה חדשה, '\t' – טאב, '\f' – דף חדש.

כשאני מגיעה לאחד התווים האלה אני יודעת בוודאות שסיימתי לקסמה.

באלגוריתם כזה פשוט להפריד בין לקסמות ובעת נשאר רק לזהות את האסימון המתאים. לשם כך נעשה שימוש בתנאים כאשר האסימונים ידועים מראש הגדיר תנאי switch ובו הגדרה של כל אפשרויות האסימונים במשפטי case. צורה זו לא מתאימה כלל היות ועל המתכנת להכיר מראש מהם האסימונים ובהתאם לבנות את הפונקציה.

חלופה 2:

הפרדת הקוד ללקסמות בשיטה הנ"ל.

זיהוי האסימונים בקוד בצורה יותר חכמה: בניית אוטומט דטרמיניסטי סופי – DFA כאשר כל לקסמה שולחים לפונקציה שעוברת על האוטומט ובודקת האם מתאימה לאסימון תקין בשפה.

חלופה 3:

בתחילה עשיתי כך:

מעבר על הקוד עד שאני מזהה סיום לקסמה ובשביל כל לקסמה על ה-DFA בשביל לזהות את האסימון אליו מתאימה אם בכלל. כמו חלופה 2.

אבל בהרבה מקרים הלקסמה לא מסתיימת באחד המפרידים הנ"ל אלא הקוד כתוב מכמה לקסמות שכתובות רצוף ואני צריכה להבחין במקרים האלו לשים לב שנגמרה לקסמה ולנהוג בהתאם.

דוגמאות:

• `if(x==5)`

הלקסמות הן: `if, (, x, ==, 5,`

• `x++`

הלקסמות הן: `x, ++`

• ++X

הלקסמות הן: ++, X

מה עושים? 😞

שמתי לב:

בעת המעבר על לקסמה יכולים לקרות המקרים הבאים:

- התחלתי עם תו והגעתי למספר.
- התחלתי עם תו והגעתי לאופרטור.
- התחלתי עם מספר והגעתי לתו.
- התחלתי עם מספר והגעתי לאופרטור.
- התחלתי עם אופרטור והגעתי לתו.
- התחלתי עם אופרטור והגעתי לתו.

אז הפונקציה עובדת כך:

הגדרתי דגלים שיסמנו לי את סוג הערך שנקרא מהקוד. תו, מספר, הערה, מחרוזת...

אם התחלתי לקסמה עם הערה או מחרוזת אני ממשיך עד שאני אגיע לתו שמסמל את סוף ההערה או המחרוזת. לא משנה אילו תווים יש באמצע.

אם התחלתי עם תו או מספר אני ממשיכה עד שאני מגיעה לאופרטור.

אם התחלתי עם אופרטור אני ממשיכה עד שאני מגיעה לתו או מספר.

כשזיהיתי סיום לקסמה – שלחתי לפונקציה לזיהוי האסימון.

אלו הפונקציות:

פונקציית הניתוח:

(הפרדה ללקסמות)

```

void lex(DFA* dfa, char* code)
{
    char* s = code;
    char lexema[255] = "lexema";
    int number = 0;
    int alpha = 0;
    int operator = 0;
    int punctuation = 0;
    int comment = 0;
    int string = 0;
    int longComment = 0;
    int identifier = 1;
    int create = 0;
    int line = 0; // מספר שורות
    int index = 0;
    int firstIndex = 0;
    int lastIndex = 0;
    int length = 0;
    errno_t err;
    while (*s != '\0') // הסוף עד הקוד
    {
        length++;
        if (longComment == 1) // נשלח לזיהוי סוקן
        {
            if (*s == '#' && *(s + 1) == '#')
            {
                create = 1;
                index++;
                s++;
                length++;
            }
            // הערה קצרה או מתרוצת
        }
        else if (comment == 1)
        {
            if (*s == '\n')
            {
                create = 1;
            }
        }
        else if (string == 1)
        {
            if (*s == '"')
            {
                create = 1;
            }
        }
        else if (*s == '#')
        {
            if (alpha == 1 || number == 1 || operator == 1 || punctuation == 1)
            {
                create = 1;
                s--;
                index--;
                length--;
            }
            if (s[1] == '#')
            {
                longComment = 1;
            }
            else
            {
                comment = 1;
            }
        }
    }
}

```



```
else if (*s == '')
{
    string = 1;
}
else if (isdigit(*s))
{
    if (operator == 1)
    {
        create = 1;
        s--;
        index--;
        length--;
    }
    else
    {
        number = 1;
    }
}
else if (isalpha(*s) || *s == '_')
{
    if (operator == 1)
    {
        create = 1;
        s--;
        index--;
        length--;
    }
    else
    {
        alpha = 1;
    }
}
//הורגים פסדלגים עליהם
else if (' ' == *s || '\t' == *s || '\n' == *s || '/' == *s || '\r' == *s || '+' == *s || '\f' == *s)
{
    if (number == 1 || alpha == 1 || operator == 1)
    {
        create = 1;
    }
    else
    {
        firstIndex++;
    }
    if (*s == '\n')
    {
        line++;
    }
}
//סימני פיסוק
else if (*s == ';' || *s == '(' || *s == ')' || *s == '{' || *s == '}' || *s == '[' || *s == ']' || *s == ':' || *s == '?' || *s == ',' || *s == '.')
{
    if ((*s == '.' && number == 1))
    {
        number = 1;
    }
    else if (number == 1 || alpha == 1 || operator == 1)
    {
        create = 1;
        s--;
        index--;
        length--;
    }
    else
    {
        punctuation = 1;
        create = 1;
    }
}
```

```
//אופרטור
else
{
    if (number == 1 || alpha == 1 || punctuation == 1)
    {
        create = 1;
        s--;
        index--;
        length--;
    }
    else
    {
        operator = 1;
    }
}

if (create == 1)
{
    err = strncpy_s(lexema, _countof(lexema), code + firstIndex, length);
    lexema[length] = '\0';
    checkToken(dfa, lexema);
    //איפוס הדגלים
    number = 0;
    alpha = 0;
    operator = 0;
    comment = 0;
    string = 0;
    longComment = 0;
    punctuation = 0;
    create = 0;
    firstIndex = index+1;
    length = 0;
}
s++;
index++;
}

if (length != 0)
{
    err = strncpy_s(lexema, _countof(lexema), code + firstIndex, length);
    lexema[length] = '\0';
    checkToken(dfa, lexema);
}

for (int i = 0; i < sizeCodeTokens; i++)
{
    printf("\n %s -> %s\n", codeTokens[i].value, tokenTypes[codeTokens[i].idTokenType].name);
}
```

הפונקציה לזיהוי האסימון:

```
void checkToken(DFA* dfa, char* lexema)
{
    char* l = lexema;
    int idToken = -1;
    int idTokenType;
    char name[80];
    char value[80];
    char stringBuffer[80];
    int identifier = 1;
    int flag = 0;
    dfa->currentState = dfa->startState;
    if (*l != '_' && !isalpha(*l))
    {
        identifier = 0;
    }
    while (*l)
    {
        if (*l != '_' && !isalpha(*l) && !isdigit(*l))
        {
            identifier = 0;
        }
        if ((dfa->states[dfa->currentState]).transitions[hash(*l)].value == *l)
        {
            dfa->currentState = (dfa->states[dfa->currentState]).transitions[hash(*l)].idState;
            l++;
        }
        else
        {
            flag = 1;
            break;
        }
    }
    if (sizeCodeTokens == 0)
    {
        codeTokens = malloc(sizeof(Token));
    }
    else
    {
        codeTokens = (Token*)realloc(codeTokens, sizeof(Token) * (sizeCodeTokens + 1));
    }
    if (flag == 0 && dfa->states[dfa->currentState].isFinal)
    {
        idToken = dfa->states[dfa->currentState].idInArray;
        idTokenType = tokens[idToken].idTokenType;
        strcpy_s(value, (rsize_t)_countof(stringBuffer), tokens[idToken].value);
        createToken(&codeTokens[sizeCodeTokens++], idToken, idTokenType, value);
    }
    else if (identifier == 1)
    {
        for (int i = 0; i < sizeSymbolTable; i++)
        {
            if (strcmp(symbolTable[i].name, lexema))
            {
                idToken = symbolTable[i].id;
                break;
            }
        }
        if (idToken == -1)
        {
            idToken = sizeSymbolTable;
        }
        if (sizeSymbolTable == 0)
        {
            symbolTable = malloc(sizeof(Symbol));
        }
        else
        {
            symbolTable = (Symbol*)realloc(symbolTable, sizeof(Symbol) * (sizeSymbolTable + 1));
        }
        createSymbol(&symbolTable[sizeSymbolTable++], idToken, lexema);
        createToken(&codeTokens[sizeCodeTokens++], idToken, 2, lexema);
    }
    else
    {
        //לסמן כשגיאה
        createToken(&codeTokens[sizeCodeTokens++], -1, -1, "exception :(*)");
    }
}
```

ניתוח סינטקטי:

ישנם מספר סוגים של אלגוריתמי ניתוח המשמשים בניתוח תחביר.

ניתוח Top-Down

- ניתוח LL : זהו אלגוריתם ניתוח מלמעלה למטה שמתחיל בשורש של עץ הניתוח ובונה את העץ על ידי הרחבת לא-טרמינלים ברציפות. ניתוח LL ידוע בפשטות ובקלות היישום שלו.
- ניתוח $LL(k)$: זוהי גרסה של ניתוח LL המשתמשת ב- k סמלי מבט קדימה כדי לנתח את הדקדוק.

ניתוח Bottom-Up

- ניתוח LR : זהו אלגוריתם ניתוח מלמטה למעלה שמתחיל בעלים של עץ הניתוח ובונה את העץ על ידי הקטנת טרמינלים ברציפות. ניתוח LR חזק יותר מניתוח LL והוא יכול להתמודד עם מחלקה גדולה יותר של דקדוקים.
- ניתוח $SLR(1)$: זוהי גרסה של ניתוח LR המשתמשת במבט קדימה כדי לנתח את הדקדוק.
- ניתוח LALR : זוהי גרסה של ניתוח LR המשתמשת בקבוצה מצומצמת של סמלי מבט קדימה כדי להפחית את מספר המצבים במנתח LR.
- ניתוח CLR : זוהי גרסה נוספת של ניתוח LR המשתמשת בפריטי $LR(1)$ כדי לבנות את טבלת הניתוח.

ניתוח סמנטי

לניתוח סמנטי לא קיימים אלגוריתמים מובנים, אלא רעיונות ומבני נתונים המפשטים את הניתוח, ומהם חשבתי בניתי את האלגוריתם בעצמי.

יצירת קובץ הקוד

יצירת הקוד מותאמת ספציפית לכל מהדר ולכן אין לה אלגוריתמים מובנים בדרך-כלל לתרגום הקוד יוצרים מילון באופן בו מתאימים לכל מילה בשפה – כל אסימון – את המילה או המשפט המקביל לה בשפה החדשה.

תיאור החלופה הנבחרת

ניתוח לקסיקלי

בניית המנתח

לאחר שהגדרתי קבוצת אסימונים מסוימת לשפה שלי וראיתי שהאוטומט נבנה בשלמות, רציתי להוסיף מילים נוספות לשפה. על מנת להוסיף עוד מילים לשפה צריך לשבת ולעשות עוד עבודה שחורה ולהוסיף ידנית לקובץ בעבור כל אסימון את המעברים האפשריים שלו. כמוכן – זו עבודה מעצבנת. ואז חשבתי – למה שלא אבנה פונקציה שתבנה ותמלא את האוטומט לבד? הרי יש לכך כללים ברורים! וממחשבה – למעשה!....

בקובץ כתבתי את רשימת האסימונים בשפה שלי – בכל שורה אסימון כאשר אסימון מצוין סוג האסימון ובמידה ויש כמה אסימונים בעלי משמעות זהה לדוגמא: `&&`, `and` – הם יכתבו באותה שורה.

הפונקציה שבונה את המנתח עוברת שורה-שורה כאשר האיבר הראשון בשורה מציין את סוג האסימון, והאיברים הבאים הם רשימה של אסימונים בעלי משמעות זהה.

אלגוריתם הניתוח

פונקציית הניתוח שעשיתי בתחילה לא הייתה מספיק טובה למקרי קצה שונים. כמו לדוגמא, הביטוי `==++` לא יופרד אלא ישלח בשלמותו לזיהוי האסימון ותתקבל שגיאה לקסיקלית אבל למעשה אלו שני אסימונים `++` ו-`==`. ועוד, בפונקציה כזו על המתכנת לדעת מראש מהי ההגדרה של הערה, `##` או `//` וכד'.

לכן בניתי פונקציה חדשה ויעילה שעוברת במקביל גם על הקלט וגם על האוטומט DFA. האוטומט עובר על מחרוזת קלט ועבור כל תו וכל מצב הוא מחליט לאיזה מצב לעבור. אם האוטומט נמצא במצב סופי אנו אומרים שהאוטומט קיבל את מחרוזת הקלט ואם בסיום הקלט האוטומט נמצא במצב לא סופי אנו אומרים שהאוטומט דחה את מחרוזת הקלט. אם ממצב מסוים בעבור תו מסוים אין קשת יוצאת אנו אומרים שהאוטומט נתקע ולכן דחה את הקלט.

מה נעשה אם אותה מחרוזת מתאימה לשני אוטומטים? (דו משמעות)

לדוגמא, המחרוזת `==` יכולה להתפרש בשני דרכים. המנתח יכול לזהות בה שני מופעים של הסימן `=` או מופע אחד של האופרטור `==`.

הפתרון: חוק הרצף הארוך. המנתח תמיד ינסה למצוא את ההתאמה הגדולה ביותר.

ולכן- ברגע שהגעתי למצב סופי – יש לעשות סימן אבל להמשיך – אולי יש מילה יותר ארוכה שמתאימה אם כן – המילה הארוכה מסומנת כמצב סופי. אני ממשיכה עד שאני מגיעה למצב שנתקעתי ואין לאן להמשיך אז אני מסמנת את המצב האחרון שנתקלתי בו כאסימון.

ניתוח סינטקטי

בניית המנתח

בחרתי את ניתוח SLR(1). למה?

גזירה ימנית מתמודדת בצורה יותר טובה עם דקדוקים שונים מאשר גזירה שמאלית. מתוך אוסף המנתחים מנתח SLR(1) גם יעיל וגם יחסית קל יותר למימוש.

SLR(1) הוא אלגוריתם ניתוח פשוט ויעיל שיכול להתמודד עם מגוון רחב של דקדוקים נטולי הקשר. הוא קל ליישום וניתן להשתמש בו כדי ליצור טבלת מנתח שניתן להשתמש בה כדי לנתח קוד קלט במהירות וביעילות. ניתוח SLR(1) הוא אלגוריתם ניתוח מלמטה למעלה, מה

שאומר שהוא יכול להתמודד עם דקדוקים רקורסיביים שמאלה ויכול ליצור עץ ניתוח מלמטה למעלה. הוא יכול להיות שימושי במיוחד לטיפול בדקדוקים מורכבים ויצירת עצי ניתוח מדויקים.

האלגוריתם קורא מקובץ את שמות הנון-טרמינלים שהגדיר המשתמש אותם מחזיק באוטומט DFA (כמו את האסימונים) על מנת לזהות בעת מעבר על הגדרת הדקדוק באיזה נוו-טרמינל מדובר ולאחר מכן את הגדרת ה-CFG של השפה החדשה של המשתמש ותוך כדי ממלא את מבנה הנתונים שמחזיק אותו.

האלגוריתם עובר על מבנה הדקדוק. על מנת לבנות את טבלת הניתוח האלגוריתם נעזר בקבוצות ה-first, follow, וקבוצת הסגור של כל נוו-טרמינל.

לאחר בניית הקבוצות הנ"ל האלגוריתם עובר שוב על מבנה הדקדוק ויוצר את טבלת הניתוח בהתאם.

האלגוריתם לניתוח

אלגוריתם הניתוח עובר על זרם אסימונים של קוד המשתמש ובעבור כל אסימון מבצע את הפעולה הנדרשת shift, reduce, GOTO, או shift-reduce. כאשר מתבצעת הפעולה shift, דוחפים את האסימון למחסנית. כאשר מתבצע reduce מחליפים במחסנית את איברי כלל הגזירה שזוהה בכלל הגזירה ומבצעים GOTO – עדכון מספר המצב בו אני נמצאת כרגע. בנוסף יוצרים איבר בעץ הניתוח כאשר ילדיו הם איברי כלל הגזירה שהוצאו מהמחסנית. כך ממשיכים לנתח עד הסוף. או שנתקעתי – הקוד לא תקין. או שהגעתי לכלל הגזירה ההתחלתי – הקוד תקין מבחינה סינטקטית ואפשר לעבור לשלב הבא.

ניתוח סמנטי

בניית המנתח

על המשתמש להגדיר (בקובץ):

- את השמות-סוגי הטיפוסים התקינים בשפה.
- את סגנון השפה – האם חובה להגדיר משתנים או לא....
- כמו כן להגדיר טבלאות שמציינות לכל אופרטור האם הביטוי תקין או לא ואיזה טיפוס יוחזר מהביטוי. בטבלאות אלו נעזרתי בסריקת העץ.

האלגוריתם לניתוח

בשלב זה מתבצעת סריקה על העץ. במהלך הסריקה על העץ כאשר נתקלים ב-identifier מתעדכנת טבלת הסמלים שבה נעזרים בשביל הבדיקות הסמנטיות. טבלת הסמלים כוללת את ההגדרות הקשורות למשתנים ולפונקציות.

לכל טרמינל ונוו-טרמינל יש תכונה type על ידי תכונה זו אפשר לבדוק האם משמעות הקוד תקינה או לא. האלגוריתם סורק רקורסיבית את העץ. בעבור כל איבר בעץ מחשבים את הסוג שלו על פי הערך המוחזר מהתוצאה מהטבלה בהתאם לבנים שלו. אם התוצאה הסופית תקינה – הקוד תקין מבחינה סמנטית.

יצירת קובץ הקוד

על מנת לתרגם את הקוד לשפת C# הגדרתי את הדקדוק התחבירי של C#. הגדרתי מחרוזת לתוכה משרשרים את הקוד המתורגם. האלגוריתם ליצירת הקוד מבצע סריקה על עץ הניתוח ובעבור כל איבר בעץ משרשר למחרוזת את החלק המקביל המתאים מתוך ההגדרה של C# ובעת הצורך – שמות משתנים, טיפוסים וכד' מקוד המשתמש.

אפיון המערכת

ניתוח דרישות המערכת

- תוצאה אופטימלית ביותר.
- כתיבת הקוד בסיבוכיות היעילה ביותר.
- כתיבה בסטנדרטים מקצועיים.
- קוד ברור ומובן לצופה.
- תגובה מהירה בכל האפשר למשתמש.

מידול המערכת

- קליטת האסימונים – המילים התקינות בשפה מהמשתמש.
- קליטת ה-CFG – דקדוק חסר הקשר – הדקדוק התחבירי של השפה מהמשתמש.
- קליטת האילוצים הסמנטיים מהמשתמש.
- בניית מנתח לקסיקלי.
- בניית מנתח סינטקטי.
- בניית מנתח סמנטי.
- קליטת הקוד מהמשתמש.
- ניתוח לקסיקלי של הקוד – הפרדה ללקסמות וזיהוי האסימונים.
- ניתוח סינטקטי של הקוד – בדיקת תקינות התחביר של הקוד ויצירת עץ ניתוח.
- ניתוח סמנטי של הקוד – בדיקת תקינות הסמנטיקה של הקוד.
- יצירת קובץ שמכיל את תרגום הקוד לשפת היעד.

אפיון פונקציונאלי

- `int hash(char value);`
פונקציית hash שמקבלת תו מסוים ומחזירה את המיקום שלו במערך המעברים.
- `void fillRegularExpressions(DFA* dfa);`
פונקציה שקוראת מקובץ את הביטויים הרגולרים וממלאת אותם לאוטומט.
- `void fillDFA(DFA* dfa , char* fileName);`
פונקציה שקוראת מקובץ את האסימונים ובונה את האוטומט.
- `DFA* dfa(char* fileName);`

פונקציה שקוראת מקובץ את המילים השמורות בשפה ובונה להם אוטומט DFA בהתאם. (הפונקציה מזמנת את 2 הפונקציות הנ"ל)

- `void fillCFG(DFA* tokensDFA, DFA* CFG_DFA);`

פונקציה שקוראת מקובץ את הדקדוק חסר ובונה את המבנה שמחזיק אותו.

- `void findFirstAndClosedGroup(int id, int idNonTerminal, int idLast, int index);`

פונקציה רקורסיבית שמוצאת לנון-טרמינל מסוים את קבוצת ה-`first`. תוך כדי הפונקציה מוצאת גם את קבוצת הסגור.

- `void findFollow();`

פונקציה שיוצרת את קבוצת ה-`follow`.

- `void buildTable();`

פונקציה שיוצרת את טבלת ה-`GOTO & action` שמשמשת לניתוח הסינטקטי של הקוד.

- `void fillTables(DFA* tokensDFA);`

פונקציה שקוראת מקובץ את האילוצים הסמנטיים וממלאת את טבלאות האופרטורים.

- `void lex(DFA* dfa, char* s);`

פונקציה הניתוח הלקסיקלי.

- `void parser();`

פונקציה הניתוח הסינטקטי.

- `int typesCheck(AST* ast);`

פונקציה המנתח הסמנטי שבודקת את תקינות הטיפוסים.

- `void functionsCheck();`

פונקציית המנתח הסמנטי שבודקת את תקינות הפונקציות.

- `void GeneratingTheCode();`

פונקציה שיוצרת את הקובץ עם קוד המשתמש מתורגם ל-`C#`.

ביצועים עיקריים

משתמש המעוניין ליצור שפה חדשה משלו נכנס לתוכנה.

עליו להגדיר:

- מהם האסימונים – המילים התקינות בשפה.
- את התחביר הסינטקטי של השפה – `CFG`.
- את האילוצים הסמנטיים של השפה החדשה.

לאחר הגדרות אלו נבנה הקומפיילר על כל שלביו.

בעת, המשתמש יוכל להכניס קטע קוד בשפה החדשה שלו והקומפיילר יצור קובץ שבו קוד המשתמש מתורגם לשפת `C#`.

אילוצים

- שפת המשתמש יכולה לכלול: הוראות קלט, פלט, הצהרת משתנים, השמה, תנאים, לולאות ופונקציות. ללא מצביעים, ספריות מבני נתונים מורכבים וכו'.
- במערך האסימונים: $0 = \text{identifier}$ $1 = \langle \text{null} \rangle$ $\langle \$ \rangle =$ סיום הקלט, אחרון.
- אסימוני המשתמש מוגדרים מהספרה 2.
- אסימונים לא מתחילים באות גדולה ונונ-טרמינלים חייבים להתחיל באות גדולה.
- טיפוסים: אני הוספתי ש: $ok = 0$ $err = 1$ $op = 2$, אופרטור.
- טיפוס המשתמש מתחילים מהספרה 3.

תיאור הארכיטקטורה

המערכת כוללת ארבעה שלבים.

מנתח לקסיקלי -> מנתח סינטקטי -> מנתח סמנטי -> יצירת קובץ הקוד.

ארכיטקטורת רשת

לא רלוונטי.

תיאור פרוטוקולי התקשורת

לא רלוונטי.

שרת-לקוח

צד שרת נכתב בשפת C.

תיאור הצפנות

לא רלוונטי.

תרשים Use Cases / UML של המערכת המוצעת

רשימת Use Cases

- הגדרת השפה החדשה:
- הגדרת אסימונים.
- הגדרת CFG.
- הגדרת אילוצים סמנטיים.
- קליטת קוד המשתמש.

Use Cases עיקריים

UC₁

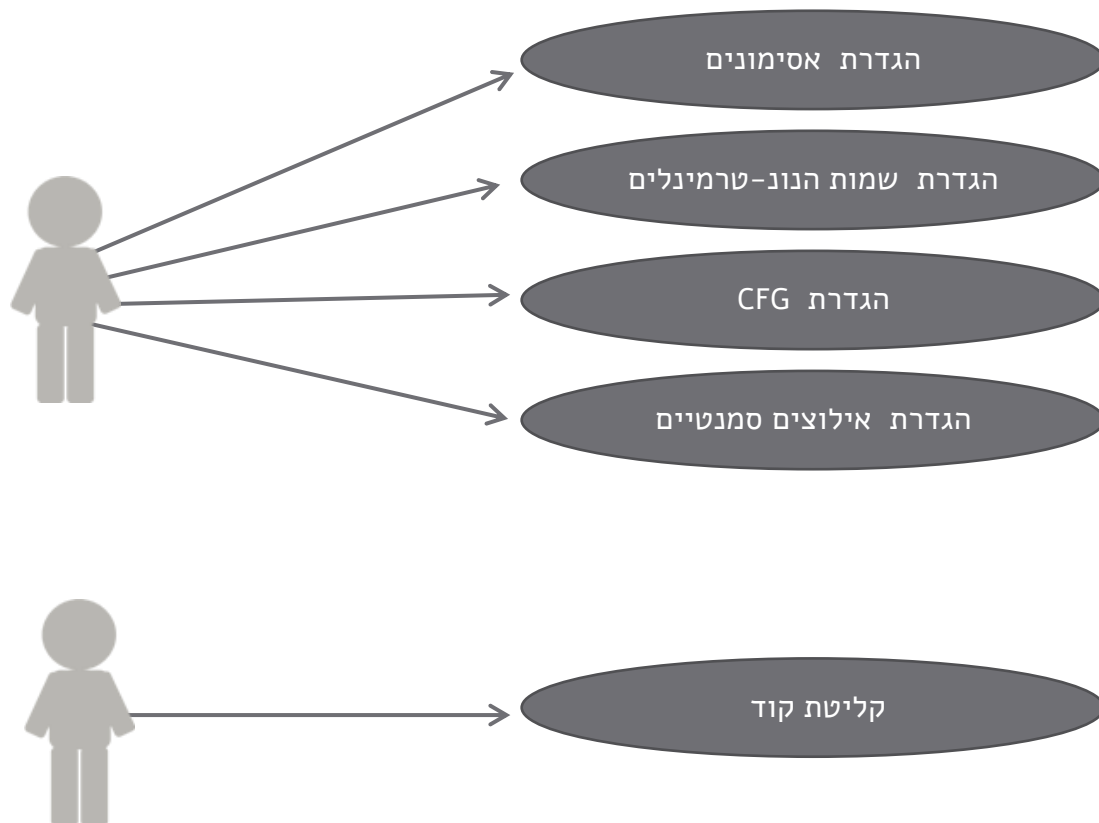
- UC₁: Identifier
- Name: הגדרת השפה.

- Description: יוצר השפה מגדיר את האסימונים התקינים, את הדקדוק התחבירי - CFG ואת האילוצים הסמנטיים של השפה החדשה.
- Actors: מתבנת.
- Frequency: בכל עת.
- Condition-Post: אישור שהשפה תקינה והוגדרה בהצלחה.

UC2

- UC2 :Identifier
- Name: כתיבת קוד.
- Description: המתבנת כותב את הקוד בשפת התכנות החדשה. הקוד יתורגם לשפת C#.
- Actors: מתבנת.
- Frequency: בכל עת.
- Condition-Post: המערכת תיצור קובץ בסיומת cs אשר יוצג למשתמש.

Use Case Diagram



מבני נתונים בפרויקט

ניתוח לקסיקלי

הגדרת האוטומט DFA:

האוטומט מיוצג על ידי גרף.

```
typedef struct {
    char value;
    int idState;
}Transition;
```

מעבר. בעבור תו מסויים – לאיזה מצב מוביל.

```
typedef struct {
    int id;
    bool isFinal;
    char name[255];
    int idInArray;
    int countTransitions;
    Transition transitions[95];
}State;
```

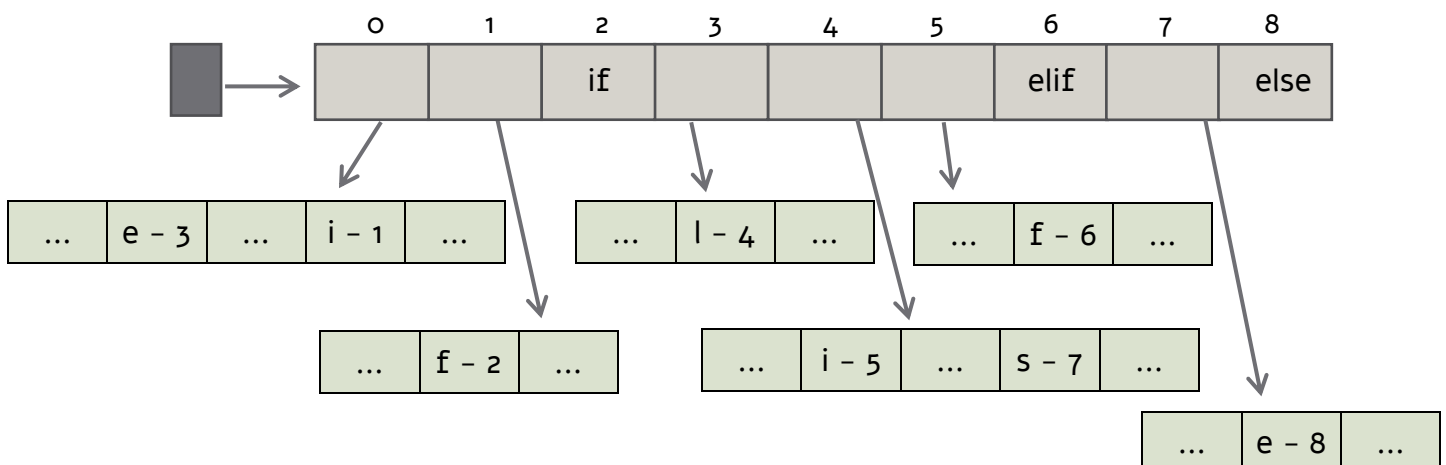
מצב. כל מצב מכיל: האם סופי או לא, אם כן – מה הערך שלו ומה המיקום שלו במערך. לכל מצב יש מערך שמיוצג על ידי טבלת hash של כל המעברים היוצאים ממנו.

```
typedef struct {
    int currentState;
    int countStates;
    State* states;
}DFA ;
```

המבנה שמחזיק את **האוטומט** כולו. מכיל מערך דינאמי של כל המצבים האפשריים.

דוגמא לאסימונים if, elif, else:

האוטומט DFA מערך המצבים לכל מצב-מערך המעברים



הגדרות הקשורות לאסימונים:

```
typedef struct {
    int id;
```

```
char name[255];
```

```
}TokenType;
```

סוג אסימון.

```
typedef struct {  
    int id;  
    char name[255];  
}Type;
```

טיפוס.

```
typedef struct{  
  
    int id;  
    int idTokenType;  
    int idType;  
    char value[255];  
  
}Token;
```

אסימון. כל אסימון מכיל מה הסוג שלו ואיזה טיפוס הוא.

```
typedef struct {  
    int id;  
    char name[255];  
    bool isDefined;  
    bool isInitialis;  
    char scope[255];  
    int* idTypes;  
    int count;  
}Symbol;
```

סמל מטבלת הסמלים. כל מזהה מכיל האם הוא מוגדר, האם הוא מאותחל, מה תחום ההגדרה שלו ומערך דינאמי של הטיפוסים אליהם משתנה במהלך התוכנית. (במידה ויכול להשתנות).

המבנים:

```
TokenType* tokenTypes;
```

מערך דינאמי של סוגי האסימונים.

```
Type* types;
```

מערך דינאמי של סוגי הטיפוסים.

```
Token* tokens;
```

מערך דינאמי של האסימונים.

```
Token* codeTokens;
```

מערך דינאמי של האסימונים של קוד המשתמש.

```
Symbol* symbolTable;
```

מערך דינאמי שמייצג את טבלת הסמלים.

ניתוח סינטקטי:**הגדרות הקשורות לדקדוק חסר הקשר:**

```
typedef struct {
```

```
    int id;  
    char value[255];  
    bool isTerminal;  
    int idTerminalOrNonTerminal;  
    int countValues;
```

```
}CFG_Value;
```

ערך במנה ה-CFG. כל איבר מכיל שם, האם הוא טרמינל או לא, אינדקס במערך שלו, ולאיבר הראשון נשמר כמה ערכים סך הכל יש לכלל הגזירה הזו.

```
typedef struct {
```

```
    int id;  
    char name[255];  
    int countOptions;  
    CFG_Value** options;
```

```
}NonTerminal;
```

נון-טרמינל. כל אחד מכיל את השם, והאופציות – כללי הגזירה אליהן יכול להתחלף.

```
typedef struct {  
    char** rules;  
    int size;  
}ClosedGroupTable
```

קבוצת סגור. קבוצת כללי הגזירה שנכללים בנון-טרמינל – לכל נון-טרמינל.

```
typedef struct {  
    int index;  
    int idNonTerminal;  
}ClosedGroup;
```

איבר בטבלת עזר לבניית טבלת הניתוח. הטבלה מכילה לנון-טרמינל מסוים לאיזה כלל גזירה להתייחס בקבוצת הסגור בעת הניתוח.

```
typedef struct {  
    char actionOrGOTO[255];  
}Table;
```

איבר בטבלת הניתוח. כל איבר מכיל מחרוזת המכילה את הפעולה הנדרשת שיש לבצע.

המבנים:

```
NonTerminal* nonTerminals;
```

מערך דינאמי של כל הנון-טרמינלים בשפה.

```
int** first;
```

מטריצה המייצגת את קבוצות ה-first.

```
int** follow;
```

מטריצה המייצגת את קבוצות ה-follow.

```
ClosedGroupTable* closedGroupTable;
```

טבלה של קבוצת הסגור לכל נון-טרמינל.

```
ClosedGroup* closedGroups;
```

מערך דינאמי שמייצג את טבלת העזר לבניית טבלת הניתוח.

```
Table** actionAndGOTO_Table;
```

טבלת הניתוח.

הגדרת מחסנית גנרית:

```
typedef struct {  
    void** data;  
    int size;  
    int top;  
}Stack;
```

מבנה שמייצג מחסנית גנרית.

למעשה יצרתי 2 מחסניות:

```
Stack parsingStack;
```

מחסנית עזר למהלך ניתוח הקוד בשלב הניתוח הסינטקטי.

כל איבר במחסנית מיוצג כך:

```
typedef struct {  
    bool isCode;  
    bool isTerminal;  
    int index;  
}Parse;
```

כל איבר מכיל האם הוא חלק מהקוד או שהוא מספר המציין את המצב בו אנו נמצאים במהלך הניתוח. אם האיבר הוא חלק מהקוד מצוין האם הוא טרמינל או לא. ובנוסף שומרים את האינדקס במערך הקוד של המשתמש.

```
Stack AST_Stack;
```

מחסנית עזר לבניית עץ הגזירה במהלך הניתוח.

כל איבר במחסנית זו מיוצג כך:

איבר בעץ הניתוח AST:

```
typedef struct{  
    int id;  
    int numChildrens;  
    bool isTerminal;  
    int idType;  
    struct AST** childrens;  
}AST;
```

למעשה זהו איבר בעץ הגזירה. כל איבר בעץ הגזירה מכיל את מספר הילדים שלו, האם הוא טרמינל או לא, מה הטיפוס שלו ומערך דינאמי של הילדים שלו.

המבנה:

```
AST* codeAST;
```

עץ הניתוח של קוד המשתמש.

ניתוח סמנטי

המבנה – מערך טבלאות האופרטורים לציון הטיפוס בהתאם לביטוי:

```
int*** operatorsTable
```

מערך דינאמי בו כל איבר הוא מטריצה שמסמנת בעבור כל ביטוי בין 2 טיפוסים את הטיפוס המתקבל. כל איבר במערך מייצג אופרטור.

תרשים מחלקות

לא רלוונטי.

תיאור המחלקות המוצעות

לא רלוונטי.

תיאור התוכנה

סביבת העבודה: visual studio 2022

שפת תכנות בצד השרת: C.

אלגוריתמים מרכזיים

הפונקציה הראשית main:

```
void main()
{
    DFA* tokensDFA;
    DFA* CFG_DFA;

    char code[1000] = "";

    printf("\n\nLEXICAL ANALYZER\n\n");
    מילוי סוגי האסימונים ובניית האוטומט.
    fillTokensTypes();
    tokensDFA = dfa("tokens");

    printf("\n\nSYNTACTIC ANALYZER\n\n");
    בניית אוטומט הנון-טרמינלים, הגדרת הדקדוק התחבירי ובניית טבלת הניתוח.
    CFG_DFA = dfa("NonTerminals");
    fillCFG(tokensDFA, CFG_DFA);
    createFirstAndFollow();
    buildTable();

    free(CFG_DFA);

    printf("\n\nSYNTACTIC ANALYZER\n\n");
    מילוי סוגי הטיפוסים וטבלאות האופרטורים על ביטויי הטיפוסים.
    fillTypes();
    fillTables(tokensDFA);
}
```

```
printf("\n\n---CHECK YUOR CODE---\n\n");

strcpy_s(code, (rsize_t)_countof(code), "a = 4.6 b =
true a = b + '3'");

printf("\n\nLEXICAL TEST\n\n");

lex(tokensDFA, code);

printf("\n\nSYNTACTIC TEST\n\n");

parser();

printf("\n\nSEMANTIC TEST\n\n");

semanticCheck();

printf("\n\nCODE GENERATION\n\n");

GeneratingTheCode();

}
```

ניתוח לקסיקלי:

בניית המנתח

אלגוריתם לבניית DFA:

```
void fillDFA(DFA* dfa , char* fileName)
{
```

```
    while (fgets(&line, 255, f1))
    {
```

הפונקציה קוראת מהקובץ שורה - שורה.

בעבור כל שורה עוברים משבצת-משבצת. כאשר כל המשבצות בשורה יובילו לאותו מצב סופי מכיוון שלכל המשבצות יש אותו משמעות של אסימון. שם האסימון יקבע על פי האסימון הראשון.

שומרים את השם ואת הסוג של האסימון הראשון.

```
    while (slot != NULL && strcmp(slot, "\n") )
    {
```

בעבור כל משבצת מתחילים ממצב 0.

```
        currentState = 0;
```

```
        while (*slot)
        {
```

בעבור כל תו שקוראים בודקים האם קיים בשבילו מעבר למצב הבא.

אם כן – עוברים למצב הבא – מעדכנים את המצב הנוכחי להיות המצב הבא.

אם לא – יוצרים מצב חדש ומעדכנים את המעבר בעבור התו הנוכחי מהמצב הנוכחי להיות המצב החדש שנוצר ומעדכנים את המצב הנוכחי להיות המצב החדש. כך עד לסיום האסימון.

}

כאשר סיימתי לקרוא אסימון:

אם אני האסימון הראשון בשורה:

- אני מוסיפה איבר חדש – אסימון חדש – למערך האסימונים בשפה – tokens. ששאר האסימונים בשורה מובילים לאותו מצב סופי.

- אני מוסיפה מצב חדש – מסמנת אותו להיות מצב סופי כאשר הערך שלו הוא שם האסימון. והתכונה של האינדקס במערך היא האינדקס של האסימון החדש במערך האסימונים. אני שומרת את מספר המצב ואילו יובילו האסימונים הבאים בשורה.

אם אני לא האסימון הראשון בשורה:

המעבר מהמצב האחרון שהגעתי אליו יוביל למצב הסופי שהאסימון הראשון הוביל אליו.

}

}

fillRegularExpressions(dfa);

בסיום מילוי האסימונים ממלאים גם את הביטויים הרגולריים עם אותו רעיון אלגוריתם כאשר מסומן לי בקובץ מתי השורה מובילה למצב חדש ומתי לא.

}

אלגוריתם הניתוח:

```
void lex(DFA* dfa, char* code)
{
```

פונקציית המנתח הלכסיקלי מקבלת את הקוד כמחרוזת. את המחרוזת צריך להפריד לאסימונים.

הגדרתי משתנים שיסמנו לי:

מיקום ואורך הלקסמה האחרונה שזיהיתי, מיקום ואורך המזהה האחרון שזיהיתי, דגל האם הלקסמה מתאימה להיות מזהה.

```
if (*s != '_' && !isalpha(*s))
{
    identifier = 0;
}
```

כאשר מתחילים בודקים האם התו הראשון עונה על ההגדרה של מזהה ומעדכנים את הדגל בהתאם. ומתחילים לעבור על האוטומט ממצב 0.

```
dfa->currentState = 0;
```

```
while (*s)
```

{

עוברים על כל מחרוזת הקוד תו-תו.

```
if (dfa->currentState == -1 && identifier == 0)
{
```

כאשר אני נתקעת – אני מגלה שבעבור התו שקראתי עכשיו הלקסמה לא אסימון שמור (הקשת היוצאת מהאוטומט מהמצב שהגעתי אליו היא -1) והיא ולא יכולה להיות מזהה, או אם הגעתי לתו שהוא וודאי מפריד בין לקסמות כמו רווח טאב וכד' אני בודקת מהי הלקסמה הארוכה ביותר שהגעתי אליה ואותה מסמנת כאסימון וממשיכה מהתו שאחרי אותה לקסמה.

```
addLexema(dfa, final , identifier ,
identifierValue);
```

כאשר מזהים לקסמה תקינה מוסיפים אותה כאסימון למערך אסימוני הקלט – codeTokens, ומעדכנים את המשתנים בהתאם.

}

```
if (dfa->currentState != -1)
{
```

```
dfa->currentState = (dfa->states[dfa->
>currentState]).transitions[hash(*s)].idState;
```

אם הלקסמה עדיין תקינה – היא יכולה להיות מזהה או אסימון – אני ממשיכה עד שאני נתקעת. איך ממשיכים? מקדמים את מחרוזת הקלט לתו הבא ואם הלקסמה תקינה לאסימון – מעדכנים את המצב הנוכחי להיות המצב שאליז מובילה הקשת בעבור התו הנוכחי מהקוד. במידה וזיהיתי שהמצב הבא הוא -1 זה אומר שהלקסמה לא תקינה להיות אסימון אני מסמנת את המצב הנוכחי כ-1 כדי לזכור זאת וממשיכה לעבור על הקלט כל עוד זה תקין להיות מזהה כשזה לא תקין להיות מזהה אני מסמנת זאת בדגל.

}

}

}

ניתוח סינטקטי:

בניית המנתח

הגדרת שמות ה-CFG – קריאה מקובץ ובניית אוטומט לשמירת שמות ה-CFG.

```
void fillCFG(DFA* tokensDFA , DFA* CFG_DFA)
{
```

הגדרת ה-CFG:

```
while (fgets(&line, 255, f1))
{
```

הפונקציה עוברת שורה-שורה.

```
if (*slot != '|')
{
```

בכל שורה המשבצת הראשונה מסמנת האם זו הגדרה של נון-טרמינל חדש או לא.

- כאשר זה נון-טרמינל חדש מוסיפים איבר למערך הנון-טרמינלים.
- כאשר הוא לא חדש - מוסיפים אופציה של כלל גזירה נוסף לנון-טרמינל שאני עומדת עליו עכשיו.

```
}
while ((slot = strtok_s(NULL, delimiter, &context))
&& *slot != '\n')
{
```

שאר המשבצות מהוות כלל גזירה.

בעבור כל משבצת:

- אם היא מתחילה באות גדולה = נון-טרמינל. אני שומרת את האינדקס שלו מהאוטומט של הנון-טרמינלים.
 - אם לא = טרמינל = אסימון. אני שומרת את האינדקס שלו מהאוטומט של האסימונים.
- בשביל כל משבצת מוסיפים איבר במערך לכלל הגזירה הנוכחי בנון-טרמינל הנוכחי.

```
    }
}
}

void findFirstAndCosedGroup(int id , int idNonTerminal ,int
idLast, int index)
{
```

מציאת קבוצת ה-first וקבוצת הסגור:

פונקציה רקורסיבית שעוברת לנון-טרמינל מסוים על כל האיברים הראשונים באופציות של כללי הגזירה שלו.

```
for (i = 0; i < nonTerminals[idNonTerminal].countOptions;i++)
{
    if (value.isTerminal == false)
    {
```

- אם זה נון-טרמינל - מוסיפים אותו לקבוצת ה-first, מוסיפים את כלל הגזירה הנוכחי לקבוצת הסגור של הנון-טרמינל המקורי שבשבילו זימנו את הפונקציה. ומבצעים קריאה נוספת לפונקציה ושולחים לה רקורסיבית גם את הנון-טרמינל הזה שהרי ה-first שלו הוא גם ה-first של המקורי.

```
    }
else
{
```

- אם זה טרמינל – מוסיפים אותו לקבוצת ה-first ומוסיפים את כלל הגזירה הנוכחי לקבוצת הסגור של הנון-טרמינל המקורי שבשבילו זימנו את הפונקציה.

```
if (!strcmp(value.value, "<null>"))
{
    findFirst(id, idLast, idNonTerminal,
    ++index);
}
```

- אם יש לכלל גזירה מסוים אפשרות של ϵ – נבצע קריאה רקורסיבית לפונקציה עם האינדקס הבא של כלל הגזירה הקודם שזימן את הפונקציה.

```
}
```

```
}
```

```
}
```

```
}
```

מציאת קבוצת ה-follow:

```
void findFollow()
{
    for (i = 0; i < sizeNonTerminals; i++)
    {
        for(j = 0; j < nonTerminals[i].countOptions; j++)
        {
            for (int t = 0; t < nonTerminals[i].options[j]-
            >countValues; t++)
            {
```

```
                if(value.isTerminal == false)
                {
```

- עוברים על כל מבנה הדקדוק. בכל מקום – כאשר בכלל גזירה מסוים נכלל נון-טרמינל קבוצת ה-follow שלו היא כל האיברים שיכולים להיות ראשונים באיבר שבא מיד אחריו.

- אם הוא עצמו האחרון באיברים של כלל הגזירה – נוסיף ל-follow שלו את ה-follow של הנון-טרמינל אליו שייך כלל הגזירה.

- אם האיבר הבא אחריו הוא טרמינל – מוסיפים את אותו טרמינל לקבוצה.

- אם האיבר הבא הוא נון-טרמינל – נעתיק לקבוצת ה-follow שלו את קבוצת ה-first של האיבר אחריו. ואם האיבר אחריו יכול להיות ϵ אז נעתיק גם את קבוצת ה-first של

האיבר שאחרי האיבר שאחריו וכן הלאה עד הסוף...

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

בניית טבלת הניתוח

```
void buildTable()
{
    for (i = 0; i < sizeNonTerminals; i++)
    {
        for (j = 0; j < nonTerminals[i].countOptions; j++)
```

```
{
  index = 0;
```

עוברים על כל מבנה הדקדוק. הטבלה למעשה עובדת כמו אוטומט שמסמן מכל נקודה בעבור כל קלט אפשרי לאיזה מצב לעבור. בעבור כל כלל גזירה מתחילים ממצב 0.

```
for (t = 0; t < nonTerminals[i].options[j]-
  >countValues;t++)
{
```

עוברים על כל האיברים של כל כלל גזירה.

```
if (cfg_value.isTerminal == false)
{
```

- אם האיבר הוא נון-טרמינל:

- אם המיקום בטבלה (בעבור המצב הנוכחי והאיבר הנוכחי) הוא ריק - ניצור מצב חדש (שורה חדשה בטבלה) ונמלא את המיקום ב- GOTO לאינדקס המצב החדש. נעדכן את המצב הנוכחי להיות המצב החדש.
- בנוסף לכך, אם אני האיבר הראשון בכלל הגזירה - ניצור שורה חדשה בטבלת העזר של קבוצות הסגור ונסמן לנו שצריך למלא לנ-טרמינל שאני עוברת עליו עכשיו את כללי הגזירה של קבוצת הסגור שלו מהמצב הנוכחי.
- אם המיקום בטבלה מלא - נעדכן את המצב הנוכחי להיות הערך שיש במשבצת בטבלה.
- אם אני האיבר האחרון בכלל הגזירה - (לאחר שעדכנתי את המצב הנוכחי) ממלאים reduce בכל מקום שיש follow. (וכן בתנאי שהמשבצת בטבלה ריקה).

```
else if (cfg_value.isTerminal == true)
{
```

- אם האיבר הוא טרמינל:

- אם המיקום בטבלה (בעבור המצב הנוכחי והאיבר הנוכחי) הוא ריק - ניצור מצב חדש (שורה חדשה בטבלה) ונמלא את המיקום ב- shift ואינדקס המצב החדש. נעדכן את המצב הנוכחי להיות המצב החדש.
 - אם המיקום בטבלה מלא - נעדכן את המצב הנוכחי להיות הערך שיש במשבצת בטבלה.
 - אם אני האיבר האחרון בכלל הגזירה - (לאחר שעדכנתי את המצב הנוכחי) ממלאים reduce בכל מקום שיש follow. (וכן בתנאי שהמשבצת בטבלה ריקה).
- כמו כן, אם זה כלל הגזירה ההתחלתי בעבור האיבר \$ = סיום הקלט - נמלא Accept שמסמן שסיימתי לעבור על הקוד והוא תקין.

```
}
```

```
for (i = 0; i < sizeClosedGroups; i++)
{
```

```
  for (j = 0; j <
    closedGroupTable[closedGroups[i].idNonTerminal].size;
    j++)
  {
```

לאחר מכן – עוברים על טבלת העזר של עדכון קבוצות הסגור. בשביל כל איבר בה – כל שורה – עוברים על כל כללי הגזירה שנמצאים בקבוצת הסגור של אותו נון-טרמינל (שמצוין בטבלה) – לכל כלל גזירה מתחילים מהמצב שמצוין בטבלה ועוברים כלולאה כל עוד המשבצת בטבלה של המצב הנוכחי ואיבר הנוכחי מכלל הגזירה מלאה.

אם המשבצת הנ"ל אכן מלאה מתקדמים לאיבר הבא בכלל הגזירה ומעדכנים את המצב הנוכחי להיות המצב שנמצא במשבצת.

במקביל עושים את אותו תהליך אבל ממצב 0.

ברגע שהמשבצת הנ"ל ריקה אני מעתיקה אליה את הערך שהגעתי אליו מהמעבר המקביל על הטבלה ממצב 0.

```
    }  
  }  
}
```

אלגוריתם הניתוח

```
void parser()  
{
```

המנתח הסינטקטי עובר על זרם האסימונים של קוד המשתמש בעצם על המערך codeTokens. הוא משתמש ב-2 מחסניות.

```
parsingStack = createStack();  
AST_stack = createStack();
```

דוחפים למחסנית הניתוח את המצב ההתחלתי בטבלת הניתוח – מצב 0.

```
while(true)  
{  
    if (!strcmp(actionName, "S"))//Shift
```

אם הפעולה הנדרשת היא shift:

- נוסיף את האסימון למחסנית הניתוח ולמחסנית עץ הגזירה.
- נוסיף את מספר הפעולה בטבלת הניתוח (מצב הפעולה) למחסנית הניתוח.
- נעבור לאיבר הבא בקלט.

```
}  
else if (!strcmp(actionName, "R"))//Reduce
```

אם הפעולה הנדרשת היא reduce:

- נבדוק כמה איברים מכיל כלל הגזירה.
- נשלוף מ-2 המחסניות את מספר האיברים בהתאם.
- ניצור איבר חדש לעץ – הנון-טרמינל שנגזר ונוסיף את האיברים הנ"ל להיות הילדים שלו. נדחוף אותו למחסנית עץ הגזירה.

- לאחר השליפה, נמצא את המצב ממנו ממשיכים הלאה ונבדוק בעבור הנון-טרמינל שגזרנו עכשיו לאיזה מצב צריך לעבור (GOTO). ונדחוף למחסנית הניתוח את הנון-טרמינל ואת המצב אליו עברנו.

```

    }

    else if(!strcmp(actionName, "Accept"))//Accept
    {
        אם הגענו לכאן = הקוד תקין מבחינה סינטקטית.
        success = true;
        break;
    }
    else
    {
        אם הגענו לכאן = הקוד לא תקין מבחינה סינטקטית.
        success = false;
        break;
    }
}

if (success == true)
{
    printf("\n\nThe code is correct!!!\n");
    codeAST = peek(&AST_Stack);
}
else
{
    printf("\n\nThe code is incorrect :(\n");
}
}

```

לדוגמא,

אם נתון הדקדוק:

$S \rightarrow E$

$E \rightarrow E + T$

$| T$

$T \rightarrow F * T$

$| F$

$F \rightarrow \text{identifier}$

טבלת הניתוח תהיה:

	identifier	+	*	\$	E	T	F
0	S8				GOTO1	GOTO4	GOTO7
1		S2		Accept			
2	S8					GOTO3	GOTO7
3		R 1 0	S5	R 1 0			

4		R 1 1	S5	R 1 1			
5	S8						GOTO6
6		R 2 0	R 2 0	R 2 0			
7		R 2 1	R 2 1	R 2 1			
8		R 3 0	R 3 0	R 3 0			

ובעבור הקוד: x^*y+z

מחסנית הניתוח תראה כך:

o |
o | x | 8 |
o | F | 7 |
o | T | 4 |
o | T | 4 | * | 5 |
o | T | 4 | * | 5 | y | 8 |
o | T | 4 | * | 5 | F | 6 |
o | T | 4 |
o | E | 1 |
o | E | 1 | + | 2 |
o | E | 1 | + | 2 | z | 8 |
o | E | 1 | + | 2 | F | 7 |
o | E | 1 | + | 2 | T | 3 |
o | E | 1 |

ניתוח סמנטי

בניית המנתח

בניית טבלאות האופרטורים

```
void fillTables(DFA* tokensDFA)
{
```

הפונקציה קוראת את הטבלאות מהקובץ.

```
while (fgets(&line, 255, f))
{
    while (slot != NULL && strcmp(slot, "\n"))
```



```
{
השורה הראשונה היא האופרטורים המתאימים לטבלה. נעבור על כולם ונסמן לכל
אופרטור את הכניסה שלו בטבלת האופרטורים.
```

```
}
השורות הבאות הן האילוצים הסמנטיים – הביטויים התקינים לאופרטורים שצוינו קודם.
for (int i = 0; i < sizeTypes; i++)
{
    for (int j = 0; j < sizeTypes; j++)
    {
        operatorsTable[sizeOperatorsTable][i][j] = atoi(slot);
    }
}
}
```

אלגוריתם הניתוח

פונקציה רקורסיבית שמבצעת סריקה על עץ הניתוח AST ומוודאת את תקינות הטיפוסים ואילוצים שונים בהתאם להגדרות המשתמש.

```
int typesCheck(AST* ast)
{
    if (ast->isTerminal == true)
    {
        return ast->idType;
    }
    AST** a = ast->childrens;
    L = typesCheck(a[i]);
    i++;
    for (; i < ast->numChildrens; i++)
    {
        if (a[i]->idType == 3)
        {
            אם זה אופרטור – איתו נבדוק את האחים.
            operatorId = tokens[a[i]->id].idType;
        }
        אם זה נון-טרמינל של השמה או הגדרה – נעדכן את טבלת הסמלים בטיפוס של המזהה.
        - וכן אם זה הגדרה של פונקציה – נחשב את מספר הארגומנטים שהיא מקבלת ואת
        הטיפוס שלהם. בנוסף, לכל המשתנים שיוגדרו בתוך הפונקציה נעדכן את התכונה
        .scope
        R = typesCheck(a[i]);
        L = operatorsTable[operatorId][L][R];
    }
    return L;
}
```

```
void functionsCheck()
{
הפונקציה הזו גם כן מבצעת סריקה על עץ הניתוח וכאשר היא נתקלת בנון-טרמינל שהוא
זימון של פונקציה היא בודקת האם מספר הפרמטרים שנשלחו והטיפוס שלהם זהה למספר
הפרמטרים והטיפוס שהפונקציה צריכה לקבל – הנתונים האלו מעודכנים בטבלת הסמלים.
```

- אם בתוך פונקציה יש משתנה נבדוק שהוא תקין מבחינת תחום ההגדרה שלו.

}

יצירת הקוד

```
void GeneratingTheCode()
{
```

הפונקציה סורקת רקורסיבית את עץ הניתוח - הפעם בשיטת שרש-שמאל-ימין.
כאשר נתקלים בנו-טרמינל שמוכר גם בדקדוק של C# (התחלה, קלט, פלט,) נשרשר
למחרוזת הקוד את התרגום המתאים.
במידה ויש אפשרות להתערבות מהמשתמש כמו הגדרת משתנה וכד' נחפש בתוך הנו-
טרמינל של עץ הניתוח את ההגדרה של המשתמש ונעתיק אותה.

}

תיאור מסד הנתונים

לא רלוונטי.

תרשים מסכים

לא רלוונטי.

מדריך למשתמש

ליוצר השפה:

הגדרת האסימונים:

כל שורה בקובץ מוגדרת כך:

מספר סוג האסימון האסימון.

לדוגמא:

1 +

1 *

הטור הראשון מציין את סוג האסימון והטור השני מייצג את האסימון בעצמו.

הגדרת הביטויים הרגולריים:

השורה הראשונה של ביטוי רגולרי בקובץ מוגדרת כך:

שם הביטוי האם מוביל למצב הבא או לאותו מצב אפשרויות התווים

השורות הבאות:

! האם מוביל למצב הבא או לאותו מצב - 1-למצב הבא 0-לאותו מצב אפשרויות התווים.

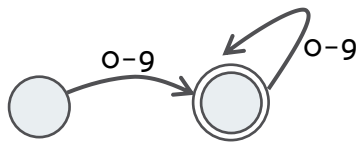
לדוגמא:

9	8	7	6	5	4	3	2	1	0	1	intValue
9	8	7	6	5	4	3	2	1	0	0	!

כאשר שם הביטוי הרגולרי הוא intValue התו הראשון יוביל למצב הבא וכל הספרות 0-9 יכולות להיות התו הראשון.

התו הבא יוביל לאותו מצב והוא גם כן יכול להיות אחד הספרות.

כך הגדרנו זיהוי ערך מספרי מסוג int.



הגדרת הדקדוק חסר הקשר:

כפי שציינתי, שמות הנון-טרמינל חייבים להתחיל באות גדולה.

הקובץ נראה כך:

איברי כלל הגזירה מופרדים במשבצות נפרדות שם הנון-טרמינל

כאשר יש יותר מכלל גזירה אחד נכתוב את כלל הגזירה הנוסף בשורה שמתחת ובעמודה הראשונה יהיה הסימן |.

כאשר נרצה להגיד שאחד האיברים הוא ערך של ביטוי רגולרי לא נכתוב את שם הביטוי אלא ניתן דוגמא.

לדוגמא:

S	A			
A	identifier	=	E	
	A	A		
E	E	+	T	
	T			
T	T	*	F	
	F			
F	identifier			
	5			
	3.5			

```

| 'str'
|
| true
|
| false
|
| "str"

```

הגדרת האילוצים הסמנטיים:

בראש הקובץ – בשורה הראשונה יש לציין 1 או 0. כאשר 1 = חובה להצהיר על משתנים
0 = לא חובה.

לכל אסימון שהוגדר כאופרטור צריכה להיות טבלה.

בשורה הבאה נכתוב במשבצות נפרדות את האופרטורים אליהם מתאימה הטבלה. (יתכן
ולשני אופרטורים תהיה טבלה זהה ולכן אין צורך לכתוב את הטבלה שוב).

השורות הבאות יהוו מטריצה ריבועית בגודל של מספר הטיפוסים שהגדיר המשתמש + 3
טיפוסים שהמנתח צריך. והם ok, op, err.

כל איבר במטריצה יהיה הטיפוס המוחזר מביטוי המכיל את שני הטיפוסים (בשורה
ובעמודה).

לדוגמא:

טבלה לאופרטורים + -:

	+ -								
	ok	err	op	int	float	string	char	bool	
ok	ok	err	ok	ok	ok	ok	ok	ok	
err	err	err	err	err	err	err	err	err	
op	ok	err	err	int	float	string	char	bool	
int	ok	err	int	int	float	err	err	err	
float	ok	err	float	float	float	err	err	err	
string	ok	err	string	string	err	string	string	err	
char	ok	err	char	char	err	err	string	err	
bool	ok	err	bool	err	err	err	err	bool	

חשוב לציין שהמילים המודגשות לא נכללות בקובץ אלא הוספתי אותם בשביל שהדוגמא
תהיה ברורה.

לאופרטורים הבאים נכתוב טבלאות באותה מתכונת בשורות הבאות.

למתכנת-משתמש

הקלד את הקוד בשפה שנבחרה.

בדיקות והערכה

במהלך בניית הקומפיילר ערכתי בדיקות שונות של מקרי קצה שיתכנו עבור דקדוקים שונים, ואילוצים שונים, נבדק הקוד וכאשר הופיעו בעיות הן תוקנו וכעת האלגוריתם הינו האופטימלי ביותר שיש באפשרותי להציע.

ניתוח יעילות

כאשר n מציין את האורך של כל קובץ נוכל להגיד שבסך הכל הפרוייקט (שהוא גנרי!) נכתב בסיבוכיות של n^2 כאשר זו הסיבוכיות הנמוכה ביותר שהצלחתי להפיק. בעוד שרוב הפונקציות כתובות בסיבוכיות של n .

אבטחת מידע

לא רלוונטי.

פיתוחים עתידיים

לפרוייקט יש פוטנציאל גדול מאד לפיתוח.

- פיתוח בניית הקומפיילר כך שיכלול הגדרת ספריות מצביעים ומבני נתונים מורכבים יותר.
- הוספת אפשרות בחירת שפת היעד - לא רק C# אלא כל שפה שתבחר על ידי המשתמש...

מסקנות

ביצוע הפרוייקט היה מועיל יותר מכל מבחן. הוא מעין סטאז' והכנה מעשית.

במחשבה לאחור, אז בהתחלה הפרוייקט היה נראה הר גבוה מאד. אך בסיעתא דישמיא עם תכנון נכון ומחשבה מעמיקה על כל הפרטים הוא נהיה פשוט יותר והפך לאתגר מהנה, נהניתי מכל רגע של פיתוח ומחשבה.

בניית הפרוייקט הזה הייתה מאתגרת ומתגמלת כאחד. בניית הקומפיילר בצורה גנרית היא משימה מורכבת, והצריכה כמות משמעותית של תכנון, מחקר ובדיקות.

למרות האתגרים הניסיון בבניית הפרוייקט הזה היה שווה ומשתלם. לראות את התוצאה הסופית גורם סיפוק גדול.

ביבליוגרפיה



Microsoft



StackOverFlow



GitHub



GeeksForGeeks



Java point



tutorials point

tutorials point

שלמי תודה

לפני כולם תודה גדולה ועצומה **לבורא עולם** שהוא – הוא עשה לי את הפרוייקט הזה כי מה אני בלעדיו?...

תודה מיוחדת **למשפחה היקרה** על הסבלנות ועל התמיכה לאורך הדרך.

תודה ענקית למנחת הפרוייקט **המורה רבקי נילסון** על הליווי המקצועי לאורך כל הדרך ועל המיומנויות הרבות שרכשתי – בזכותה!

תודה **לצוות המורות** של מגמת תכנות ובראשן **הרכזת חני גרשוני** על קורסים מקצועיים ועל ההשקעה התמידית בתלמידות.

ואחרונות חביבות ואהובות מאוד – תודה גדולה **לבנות המגמה שלי** על החוויה ועל כך שהייתן לי חברות לדרך. כי בינינו, בלעדיכן זו לא הייתה חוויה....