נובמבר 2024, תשפ״ה

ביה״ס תיכון ע״ש הרצוג כפ״ס

מגמת הנדסת תוכנה וסייבר

מסמך אפיון פרויקט גמר

אפיון פרויקט בנושא :

״קומפיילר ושפת תכנות Gravity״

פיתוח מהדר ויצירת שפת תכנות - פרסור, אנליזה וגנרציה

מגיש : אופיר הופמן

ת.ז : 217395094

בהנחיית המורה :

אופיר שביט

תוכן עניינים

ייזום

פונקציונליות

שפת הקוד

- שפת קוד מומצאת משלי (לבינתיים נקרא לה gravity, כנראה מבוססת ';')
- קומפיילר שמקבל בargs שם קובץ של קוד gravity (main.gy לדוגמה) ויוצר קובץ בסופו של דבר קובץ הרצה exe. שניתן להריץ.
- זמין בווינדוס בלבד.
- שפת הקוד תאפשר בהתחלה לעשות דברים פשוטים: קלט, פלט, השמה וכו'. לאחר מכן ארצה לשדרג ולהוסיף עוד ועוד פקודות ופונקציונליות כמו מבני נתונים (מערך, struct/מחלקה, מימוש מילון (hashmap) וכו') ומבני נתונים מורכבים (מערך של אובייקטים, מילון של מערכים ועוד).

הקומפיילר

- כאמור על קצה המזלג מקבל שם קובץ gravity (.gy) ויוצר קובץ הרצה
- יותר לעומק: תפקיד הקומפיילר יהיה להפוך שפת gravity לקוד אסמבלי ויעשה זאת בדרך המקובלת של פירוק השפה לטוקנים (tokens) או בלוקים של מילות מפתח, אופרטורים, שמות וכו' ואז parsing לכל טוקן שייפעל בהתאם לחוקי השפה (אשר יתועדו) והדפסת אזהרות ושגיאות בסוף במידה ונמצאו תקלות מסוגים שונים. משם כנראה עבודת תרגום לשפה low (IR), אופטימיזציה (אופציונלי לדעתי). למרות שתיאור זה קצת עמוק יותר הוא עדיין על קצה הקרחון ולכן אצלול מעבר למושגים ככל שאתקדם בפרויקט.
- <u>מעבר לשלב האסמבלר</u> - שימוש באסמבלר ולינקר קיימים בשוק על מנת להפוך את הקוד בשפת סף ל-object file ואז לקובץ הרצה (בהתאם לבחירת המשתמש).
- <u>דגלים (flags) – פונקציונליות נוספת</u>:

- יצירה ושמירת קובץ האסמבלי למשתמש בנוסף לקובץ ההרצה (הפלט העיקרי)
- יצירת הקבצים בשם מותאם אישית
- יצירת הקבצים בתיקייה שהיא לא התייקיה הנוכחית
- וכו'

תחומי מחקר

- <u>תקשורת</u>: אצור מערכת שרת לקוח שתאפשר למשתמש לבצע "קמפול מרחוק" באמצעות שרת מרוחק.
- <u>אבטחה</u>: הגנה מפני גישה בלתי חוקית לזיכרון, גלישה (overflow), injection וכו'.
- <u>מערכת הפעלה</u>: שימוש באיזורי זיכרון שונים, syscall, WinAPI, lock ועוד.

הגדרת הפרויקט

הפרויקט יעסוק ביצירת מהדר ("קומפיילר") בסיסי לשפת תכנות חדשה ובסיסית בשם gravity. באמצעותה יוכלו מפתחים ומשתמשים ליצור תוכניות הכוללות קלט, פלט והשמה ומבני נתונים מורכבים. לאחר מכן, יוכל המשתמש להזין את קובץ הgravity כקלט למהדר, ותפקיד המהדר יהיה להפוך את קוד הgravity לקובץ הרצה, שיוכל המשתמש להריץ.

השימוש במהדר יעשה באחת משתי דרכים:

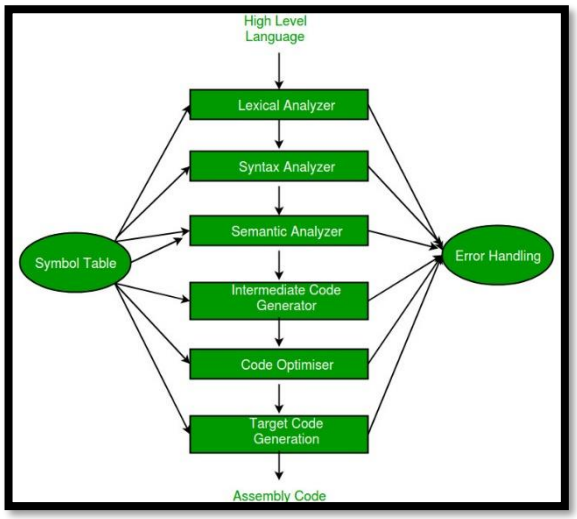
1. <u>התקנה במחשב המשתמש והרצה דרך command-line</u>: המשתמש יוכל להתקין את המהדר במחשב האישי שלו, ולאחר מכן להריץ אותו (דרך הcmd לדוגמה), כאשר בין הארגומנטים יהיו - שם התוכנית שאותה מעוניין המשתמש לקמפל (main.gy, לדוגמה) ודגלים שיוכלו למשתמש לשלוט על הפלט של המהדר, ועל תהליך הקמפול. המשתמש יוכל לבקש מהמהדר ליצור כפלט קובץ בשפת סף של התוכנית שלו, במקום קובץ הרצה. או לחלופין בקשה ליצירת הקבצים בשם מותאם אישית ו/או בתיקייה שונה מהתיקייה הנוכחית.
2. <u>"קמפול מרחוק" באמצעות שרת מרוחק</u>: אליו, יוכל המשתמש להעלות את קבצי התוכניות שכתב ולקבל קובץ הרצה מוכן להורדה, וכמובן שעדיין יוכל המשתמש לבחור באילו דגלים ואפשרויות של המהדר ירצה להשתמש. בנוסף, תינתן סביבת עבודה מרוחקת, בה יוכל המשתמש לכתוב קוד בשפה, לקמפל ולהריץ אותו ולקבל את הפלט, הכל בתוך הסביבה מבלי לקמפל ולהריץ במחשב המקומי.

קהל היעד של הפרויקט הוא מפתחים הזקוקים לשפת תכנות נוחה, קלה לשימוש ובעלת פונקציונליות בסיסית ושימושית, ויחד עם זאת מהירה ויעילה.

הפרויקט מתחלק לשני חלקים עיקריים – יצירת שפת התכנות: קביעת הדקדוק, החוקים, והסמנטיקה ותיעוד שלהם. והחלק השני – כתיבת קומפיילר בסיסי אשר יורץ דרך cmd או דרך שרת מרוחק ויקבל קובץ gravity כקלט. תפקיד הקומפיילר יהיה להפוך שפת gravity לקוד אסמבלי ויעשה זאת בדרך המקובלת הכוללת:

א. ניתוח לקסיקלי (מילוני או דקדוק, נקרא גם "Scanning")
ב. ניתוח תחבירי (parsing, למה שנקרא syntax)
ג. ניתוח סמנטי – בדיקת מבנה הקוד
ד. תרגום לקוד ביניים
ה. אופטימיזציה (אופציונלי)
ו. יצירת הקוד בשפת מכונה (קובץ exe.)

שלבי תהליך ההידור, מתוך אתר GeeksforGeeks



כל זאת תוך טיפול בשגיאות בכל אחד מהשלבים, אשר ארחיב עליהם ועל מימושם בהמשך.

חקר מוצרים

שוק הקומפיילרים – תמונת מצב

הקומפיילר הוא כלי קריטי בתהליך פיתוח התוכנה, המשמש לתרגום קוד שנכתב בשפות תכנות ברמה גבוהה לשפת מכונה או לקוד ביניים שהחומרה יכולה להבין ולהריץ. בהקשר של פיתוח קומפיילר בסיסי, המטרה היא ליצור תוכנה שמבצעת את תהליך ההמרה באופן מינימליסטי, אך יעיל, ללא הכלים המורכבים שמספקים הקומפיילרים המודרניים.

כיום בשוק יש מגוון רחב של קומפיילרים עוצמתיים ויעילים כגון GCC, Clang, LLVM, Go Compiler, אך הם עלולים לפעמים להעלות מורכבויות בנוגע לקונפיגורציה והתאמה, תלות בספריות חיצוניות וזמני הידור איטיים יחסית. בנוסף, רוב הקומפיילרים כיום בשוק מלאים בתכונות ופונקציות רבות שלמשתמש הממוצע יהיו מיותרות ואף עלולות לבלבל אותו.

לכן, הפרויקט שלי יתמקד ביצירת קומפיילר בסיסי, אך גם נוח, פשוט וברור שיענה על הצרכים של משתמשים שלא משתמשים בפונקציות המורכבות של מוצרים קיימים בשוק. לדוגמה, הקומפיילר שאצור לא יתמוך ב-preprocessor ולא יתמוך באופטימיזציות מורכבות כאלה ואחרות, אך הוא כן יתמוך בקמפול קבצים לקוד אסמבלי, או ל-object files בלבד, וכן קמפול לקבצי הרצה, וידאג להציג שגיאות שונות שיעזרו למפתח.

תהליך הקומפילציה – דגשים

הקומפיילר לשפת Gravity מהות תפקידו תהיה להפוך קובץ בשפת gravity לקוד אסמבלי. יחד עם זאת, לאחר שלב התרגום לקוד בשפת סף, ובהתאם לבחירת המשתמש, הקומפיילר ישתמש בכלים קיימים בשוק של אסמבלר ומקשר ("linker") כמו NASM ו-GNU linker על מנת להפוך את הקוד בשפת סף ל- object file ולאחר מכן לקובץ הרצה.

שוק שפות התכנות

שפות תכנות שונות

כיום קיימות שפות רבות בשוק שפות התכנות, כאשר כל לכל אחת מהן שימושים בתחומים אחרים. אחת החלוקות העיקריות שניתן להצביע עליה כיום היא ההבדל בין **שפות backend** – שמתמקדות ב-business logic, בביצועים, בניהול נתונים וכו', לבין **שפות frontend** – שמשמשות ליצירת צד הלקוח של יישומים: עיצוב, אינטראקטיביות, UX\UI וכל היוצא בזה. בין שפות ה-frontend ניתן למצוא את HTML, CSS, JavaScript ועוד, ומבין שפות הbackend ניתן למצוא את **C, C#, C++, python, Java, Go** ועוד הרבה.

Gravity תוגדר כשפת backend ותעוצב בצורה שתנסה לממש ולקיים כמה שיותר מהפונקציונליות הקיימת בשפות הקיימות בשוק, כמו השפות המצוינות לעיל. בין היתר היא תכלול משתנים, אופרטורים, השמה, קלט ופלט, מבני נתונים ומבני נתונים מורכבים (מחלקות, מערכים, מילונים ועוד, ושילוביהם – מערכים של מחלקות וכדומה).

שפות התכנות כיום מציעות מגוון רחב של פונקציונליות, כלים ואופני שימוש שונים, החל בפייתון שמציעה שפת high-level פשוטה להבנה, רב שימושית ומלאה בפעולות מובנות שימושיות וכלה בשפות C\C++ שמביאות לשולחן קוד low-level עוצמתי שמאפשר גישה מקרוב לזיכרון ולמערכת ההפעלה וכתיבת תוכניות מהירות. אמנם, שפות אלו ואחרות עלולות להימצא מורכבות עבור משתמשים ומפתחים, בעיקר חדשים, שעלולים להיבהל ולהתבלבל מכמות הכלים שיש להן להציע. לכן, gravity תיווצר במטרה לענות על הצורך לשפה פשוטה ובסיסית ולא גדושה בתכונות וכלים שיציפו את המתכנת המתחיל בשאלות, ובנוסף תגיע עם תיעוד ברור שיאפשר לכל מפתח ללמוד את השפה במהירות, תוך הבנה של מה שקורה "מאחורי הקלעים", ובכך יביא לפיתוח ידיעה עמוקה יותר של הקוד, בשפה עילית.

חקר פיתוחי

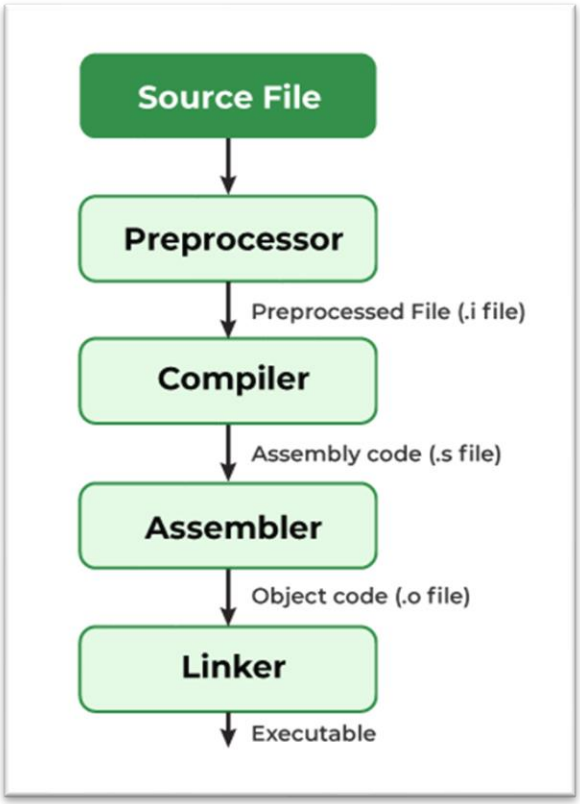
כאמור, שלב הקומפילציה שבו יעסוק המהדר שלי הוא התרגום משפה עילית / שפת ביניים לשפת סף.

Assembler ו-Linker

נתמקד לרגע בתמונה הגדולה יותר בשלבי תהליך הקומפילציה הסטנדרטית (כמוצג בתמונה).

כפי שציינתי, בפרויקט זה לא אממש preprocessor, ולכן נתעלם משלב זה כעת. המהדר מקבל קובץ שמכיל קוד (בשפה עילית או שפת ביניים) כקלט, והפלט הוא קובץ שמכיל קוד אסמבלי. אמנם בפרויקט זה מוטל על המהדר התפקיד להוציא כפלט סופי קובץ הרצה, ולכן המהדר שאבנה בפרויקט זה ייעזר בכלים אחרים שקיימים בשוק על מנת להביא את הקוד מהשלב שבו הוא מתורגם לשפת מכונה לשלב שבו ישנו קובץ executable, מוכן להרצה. תוכנת אסמבלר כגון NASM תסייע ביצירת object code/file מתוך הקוד בשפת סף, ותוכנת linker כמו GNU linker תסייע בקישור ה-object file ויצירת קובץ הרצה ממנו.

שלבי תהליך ההידור, מתוך אתר GeeksforGeeks



שלבי תהליך ההידור

לשם חוסר בלבול מכאן ואילך ועד לסוף הפרק, הכוונה בהידור/קמפול הינה לתהליך בו מתורגם קוד בשפת gravity (במקרה הזה) לקוד בשפת סף, מה שנקרא **מתרגם** או **מעבד שפה**.

כמו שכבר צוין, בפרויקט זה ההידור ייעשה פחות או יותר בדרך המקובלת כיום הכוללת 6 שלבים:

1. **ניתוח לקסיקלי (Scanning)**
2. **ניתוח תחבירי (Parsing)**
3. **ניתוח סמנטי**
4. **יצירת קוד ביניים**
5. **אופטימיזציה (לא ימומש בפרויקט זה)**
6. **יצירת הקוד בשפת מכונה**

כעת ארחיב על כל שלב בתהליך.

1. Lexical Analysis – ניתוח לקסיקלי

ניתוח לקסיקלי, או ניתוח מילוני הוא השלב הראשון של הקומפיילר, והוא מוכר גם כשלב ה-Scanning. הוא מקבל כקלט source code של תוכנית בשפת high-level ותפקידו **לקרוא** את הקוד **ולפרק אותו** לחלקים קטנים יותר, לכדי אלמנטים שנקראים **טוקנים (Tokens)**. טוקן מוגדר כרצף תווים שמיוחס כיחידה, בתוך חוקי הדקדוק של שפות תכנות. ישנם סוגים שונים של טוקנים ולכל סוג תפקיד אחר, וניתן לחלק את הטוקנים ל-3 סוגים עיקריים:

סוג Token	שימוש / הגדרה	דוגמאות בשפות קיימות
Terminal Symbols (TRM)	מילים או מזהים השמורים בשפה לשימוש מיוחד Keywords (מילות מפתח), אופרטורים punctuation (סימני פיסוק) וDelimiters (תוחמים)	Keywords: if, break, char, int, const, return, unsigned, static, void, Class, Namespace… Operators: +, -, *, / … Delimiters: Symbols used such as commas " , " semicolons " ; " braces `{}` etc.
Identifiers (IDN)	מספקים שמות לרכיבים בשפה שניתנו בדר"כ על ידי המשתמש, כגון משתנים, פונקציות, אובייקטים, פרמטרים, מחלקות וכו'	"int **x**" "void **myfunc**(int **num**)" "class **Person**"
Literals (LIT)	ערכים (כמו מספרים ומחרוזות וגם ערכים מורכבים כמו מערכים ואובייקטים) דגש חשוב: Literals, לעומת Identifiers הם **ערכים קבועים** שלא ישונו במהלך ריצת התוכנית. במילים אחרות, הם מוסיפים למושג "הקבוע" (Constant) את הרעיון שאנו יכולים לדבר עליו רק במונחים של ערכו. **Literal הוא אינו ניתן להתייחסות, מה שאומר שהערך שלו מאוחסן איפשהו בזיכרון, אבל אין לנו אמצעי גישה לכתובת זו.**	Variables: "if (a == **3**)" "string name = **"Itzik"**" "int y = **10**;" Compound Variables: "if lst == **[1, 2, 3]**:" "arr[i] = **new Person()**; Anonymous Functions: Lambda x, y: x+y

כחלק מתהליך הניתוח המילוני, הקלט עובר מספר שלבים הכוללים:

- עיבוד מקדים של הקלט: ניקוי טקסט הקלט והכנתו לניתוח מילוני – עשוי לכלול פעולות כמו הסרת הערות, רווח לבן ותווים לא חיוניים אחרים מטקסט הקלט.
- טוקניזציה (Tokenization): פירוק טקסט הקלט לרצף של טוקנים. נעשה בדרך כלל על ידי התאמת התווים בטקסט הקלט אל מול קבוצה של תבניות או ביטויים רגולריים (regular expressions) המגדירים את סוגי הטוקנים השונים.

- <u>סיווג הטוקנים לקטגוריות</u>: קביעת סוג כל טוקן. לדוגמה, סיווג לפי מילות מפתח, מזהים, אופרטורים וסמלי פיסוק כסוגי טוקנים שונים (בהתאם לשפת התכנות).
- <u>אימות/בדיקת הטוקנים</u>: בדיקה שכל טוקן תקף בהתאם לכללי שפת התכנות. לדוגמה, בדיקה האם משתנה הוא בעל מזהה חוקי, או שלאופרטור יש את התחביר הנכון. במידה ויימצא שאחד הטוקנים עובר על חוקי השפה או לא תואם לאף אחת תבנית של טוקן חוקי, המנתח הלקסיקלי "ייתקע" ולא יוכל להמשיך לבדוק את שאר הקוד (שאר הטוקנים). במקרה כזה הוא צריך "להתאושש", והוא יכול לעשות זאת באחת מכמה דרכים, בהתאם למצב. הדרך הפשוטה ביותר נקראת "**panic mode**" ובה מוחקים את התווים הבאים מהקלט עד למצב שבו המנתח המילוני מזהה טוקן חוקי בתחילת הקלט שנותר.
- <u>יצירת פלט</u>: בשלב סופי זה, נוצר הפלט של תהליך הניתוח המילוני, שהוא בדרך כלל רשימה של טוקנים, אשר ניתן לאחר מכן להעביר לשלב הבא של ההידור.

דוגמה לפירוק קטע קוד פשוט לטוקנים

נתון קטע התוכנית הבא:

```
int a = 10;
```

לאחר שלב הטוקניזציה, נקבל את רשימת הטוקנים הבאה:

Lexeme	Type
"int"	Keyword
"a"	Identifier
"="	Operator (assignment)
"10"	Numeric Literal
";"	Punctuation

<u>Lexeme</u>: רצף של תווים בקוד התואמים לאחת מהתבניות המוגדרות מראש ובכך יוצרות טוקן חוקי.

כאמור, הטוקנים הם הפלט של המנתח המילוני, והם משרתים כקלט של השלב הבא – הניתוח התחבירי.

<u>כלים</u>

כיום ישנם כלים שונים שבעזרתם ניתן ליצור Scannerים, ובניהם **Flex**, אשר ככל הנראה בוא אשתמש ליצירת פרויקט זה. Flex זהו כלי בעל שפת תכנות משלו אשר מאפשר באמצעותו להגדיר את החוקים המילוניים של השפה ולאחר מכן ליצור קובץ בשפת C אשר אחרי קמפול יחד עם קובץ הטוקנים ועם קובץ המain שלו ליצור Scanner (קובץ הרצה).

<u>סיכום שלב הניתוח המילוני</u>

קלט: קוד בשפה עילית.

פלט: רשימת טוקנים.

2. Syntax Analysis – ניתוח תחבירי

הניתוח התחבירי, או שלב ה-Parsing הינו השלב השני בתהליך הקומפילציה העוקב לשלב הניתוח המילוני, ומטרתו לקבל את הטוקנים שמייצר המנתח המילוני ולהפיק מהם "עץ תחבירי". על קצה המזלג, העץ התחבירי, או ה-**AST** (Abstract Syntax Tree) הינו ייצוג היררכי של ה-source code ומשקף את מבנה התוכנית. במילים אחרות, אם שלב ה-Scanning הוא כמו הרכבת מילים מאותיות, אז שלב ה-Scanning הוא כמו הרכבת משפטים ממילים. במהלך שלב זה בהידור נבדק לראשונה האם הקוד כתוב לפי חוקי הדקדוק שהוגדרו לשפת התכנות. במידה והמנתח התחבירי נתקל בתחביר שגוי, תפקידו לדווח על שגיאה.

<u>חוקי הדקדוק</u>

חלק חשוב בניתוח התחבירי הם חוקי הדקדוק. הדקדוק של שפת תכנות מורכב מרשימה של חוקים, שנקראים חוקי הדקדוק והם מסבירים כיצד חלקים שונים בשפת התכנות מורכבים. לפני שאצלול לכיצד בנוי סט חוקי הדקדוק של שפה, יש להרחיב קודם על דקדוק חסר הקשר, או **CFG**. CFG הינו סט חוקים אשר מתאר באופן רשמי את המשפטים המותרים בשפה, ובשונה מביטויים רגולריים, אשר לדעתי מהווים את הפתרון האינטואיטיבי יותר לשלב זה בקומפילציה, מאפשר להביא פתרון רקורסיבי לבעיה, ולכן הוא נחשב להרבה יותר עוצמתי. אמנם, CFG שרירותי יכול להכיל בעיות המקשות על כתיבת מנתח אוטומטי. לכן, ישנן 2 תתי קבוצות של דקדוק חסר הקשר אשר מהוות את שתי השיטות העיקריות לניתוח דקדוקי כיום – LL(1) ו-LR(2), כאשר ההבדל המהותי ביניהן הוא ש-LL(1) מבצע פרסור מלמעלה למטה (יורחב בהמשך), ומסוגל להחזיר הערכה עבור החוק הנוכחי, והטוקן הבא בקלט. לעומת זאת, LR(1) מבצע פרסור מלמטה למעלה, ונחשב מורכב יותר אך מסוגל גם לנתח מגוון רחב יותר של דקדוקים, כולל אלה ש-LL(1) אינו יכול להתמודד עימם.

בהגדרתה של מסגרת חוקי הדקדוק של שפה נשתמש בארבעה מושגים:

<u>Terminal</u>: סימול דיסקרטי שיכול להופיע בשפה, ידוע גם כטוקן מהשלב הקודם. לדוגמה - keywords, אופרטורים ו-identifiers. אלו יסומנו כעת באמצעות אותיות לועזיות קטנות. יש להדגיש שבשלב זה הייחוס הוא לטיפוס בלבד, לדוגמה, int ולא ערך (כמו 234).

<u>Non-Terminal</u>: מסמל מבנה שיכול להיות קיים בשפה, אך שהוא לא terminal, לדוגמה הצהרות (declaration) וביטויים. כעת הם יסומנו באמצעות אותיות לועזיות גדולות.

<u>Sentence (משפט)</u>: רצף חוקי של terminals בשפה.

<u>Sentential form – צורה משפטית</u>: רצף חוקי של terminals ו-non-terminals. יסומנו באמצעות אותיות יווניות (α, β, δ וכו'). לדוגמה: E + E, או E + ident (ident, כקיצור של identifier).

לפי העיקרון של דקדוקים חסרי הקשר, החלק השמאלי של כל חוק הוא תמיד non-terminal בודד, והחלק הימני של החוק הוא Sentential form שמתאר צורה חוקית, שהשפה מאפשרת, של החלק השמאלי.

לדוגמה, החוק $E = E + E$ קובע שביטוי (E – Expression) יכול להיות חיבור של כל שני ביטויים.

הצורות המשפטיות מרכיבות את סט חוקי הדקדוק של שפה, כאשר החוק הראשון מיוחד היות והוא ההגדרה העליונה ביותר של תוכנית, וה-non-terminal שלו ידוע גם בתור ה-**start symbol**.

דקדוק פשוט לדוגמה G_1 יכול להיראות כך:

Grammer G_1

1. $P \rightarrow E$
2. $E \rightarrow E + E$
3. $E \rightarrow ident$
4. $E \rightarrow int$

הדקדוק הזה יכול להתפרש בצורה הבאה:

1. תוכנית (שלמה) מורכבת מביטוי אחד (זהו ה-start symbol של השפה הזו).
2. ביטוי יכול להיות חיבור של כל שני ביטויים.
3. ביטוי יכול להיות identifier.
4. ביטוי יכול להיות ערך מטיפוס int.

* לצורך קיצור, ניתן לאחד חוקים שהצד השמאלי שלהם זהה באמצעות שער לוגי OR לדוגמה בדרך הבא:

$$E \rightarrow E + E \mid ident \mid int$$

הדקדוק הוא מה שיעזור לקבוע האם "משפט" בקוד הינו תקין או לא. כל שפה מכילה מספר אינסופי של משפטים אפשריים, ולכן משתשים ב"גזירת משפטים". העיקרון הבסיסי הוא שנעזרים באחד מחוקי הדקדוק שהוגדרו על מנת "לגזור" בכל פעם את המשפט ובכך לפתח/לפשט אותו (תלוי בשיטה). עלינו למצוא רצף של שימוש בדקדוק השפה אשר מקשר בין המשפט הנתון/הרצוי לבין ה-start symbol על מנת להוכיח שהמשפט אכן תקין וקיים בשפה. רצף שכזה של הפעלת חוקי דקדוק נקרא Derivation או גזירה. לדוגמה:

נפעיל את החוק	צורה משפטית – Sentential Form
$E \rightarrow int$	ident + int + int
$E \rightarrow int$	ident + int + E
$E \rightarrow E + E$	ident + E + E
$E \rightarrow ident$	ident + E
$E \rightarrow E + E$	E + E
$P \rightarrow E$	E
	P

בכך הראינו שכל משפט מהצורה המשפטית "ident + int + int" הינו משפט תקין בשפה, וזאת מכיוון שהצלחנו להוכיח באמצעות לוגיקה שנשענת על הדקדוק G_1 שקיים רצף של הפעלת חוקי דקדוק אשר מביא בסופו של דבר את הצורה המשפטית ההתחלתית ל-start symbol.

פרסר LL(1) לעומת פרסר LR(1)

ראשית, שתי השיטות הן תחת CFG והמשותף להן הוא ששתיהן פרסרים אשר עובדים משמאל לימין. עם זאת, כפי שצוין לעיל, ישנם שני הבדלים עיקריים בין דקדוקי LL(1) לבין דקדוקי LR(1).

LL(1) קלים לפרסור וניתוח באמצעות אלגוריתמים פשוטים. הם עובדים בשיטת "גזירה משמאל" (אשר לא ארחיב עליה), ופרסור מלמעלה למטה. דקדוק מסוים נחשב LL(1) אם ניתן לנתח אותו על ידי התחשבות בnon-terminal אחד והטוקן הבא בקלט.

עם זאת, LL(1) אינו מסוגל לייצג את כל המבנים שעלולים להימצא כיום בשפות תכנות. לכן, לשפת תכנות בעלת שימוש כללי, לדוגמת gravity, עליי להשתמש ב-LR(1) - קבוצת-על של LL(1) ויכולה להתאים לשיטות פרסור וניתוח שאינן מתאימות ועובדות ב-LL(1).

Parser Generator

היות ופרסרים מסוג LR(1) מסורבלים מדי לכתיבה ידנית, ישנם כלים ליצירת פרסר אשר מקבלים פירוט של דקדוק השפה ובאופן אוטומטי מייצרים קוד עובד. כלים כאלה הקיימים כיום בשוק לדוגמה הם GNU Bison אשר עובד עם flex ונחשב לדי עוצמתי, אך עלול להחסיר בכלים ותכונות נוחים שקיימים בכלים אחרים כמו ANTLR אשר תומך בשפות רבות יותר, מצוין לטיפול בדקדוקים מורכבים ויכול לייצר פרסרים במגוון שפות, אך מצד שני מסובך יותר ללמידה ושימוש.

AST – עץ תחביר מופשט

כמצוין, AST הינו מבנה נתונים חשוב בתהליך הקומפילציה אשר מייצג את המבנה העיקרי והיסודי של התוכנית. הוא מהווה את הפלט של שלב הניתוח התחבירי והקלט של השלב הבא בתהליך ההידור – הניתוח הסמנטי. כדי להבין את מהותו של הAST יש לציין שבמבנה הזה יש התעלמות מכל מה שנוגע לפרטים הקטנים של שפת תכנות: קידומות, סיומות וביטויים נוספים. כלומר שמדובר במבנה שבתיאור מסוים שלו יכול לייצג את רוב שפות התכנות הפרוצדורליות.

לשם יצירת AST, עלינו ראשית להגדיר בצורה מפורשת שלושה מושגים מופשטים:

1. Declaration – הכרזה: מצהיר על שם, סוג וערך של symbol. אם בשלב ה-Scanning היו identifiers שסימלו אך ורק את השם המייצג בקוד, ניתן לחשוב על symbols כישות השלמה של מה שייצג הidentifier. לדוגמה, בקטע הקוד הבא:

```
int a = 10;
```

בשלב ה-Scanning, הidentifier "a" סימל שם שמיוחס למשתנה. לעומת זאת, בשלב ה-Parsing (השלב הנוכחי), ההכרזה על ה-symbol "a" מכילה מידע כמו סוג המשתנה (int במקרה הזה), ה-scope שבו המשתנה הזה קיים (האם הוא גלובלי או מקומי), והמיקום או הכתובת שלו בזיכרון, היכן שנמצא הערך. באותה צורה, symbol יכול להיות קבועים ופונקציות. במידה וההכרזה הינה של פונקציה, תוכן ההכרזה יכיל גם את ה-statement שההכרזה מכילה.

2. Statement – הצהרה: מציין פעולה (הכוונה אינה לפונקציה, שעלולה להיות שם אחר לפעולה) בקוד אשר תשנה את מצב התוכנית, או המקום בתוכנית שכעת רץ. דוגמאות ל-Statements מוכרים הם לולאות, תנאים (if, else if, else וכו') וחזרה מפונקציות (return). בנוסף לאלו, הצהרות יכולות להיות גם

declaration ו-expression, ודברים בסיסיים נוספים כמו הדפסה (print). התוכן של statement יכיל את סוגו, ועוד ערכים אשר משתנים בהתאם לסוג ההצהרה. לדוגמה, ההצהרה הבאה:

```
if( x<y ) print x; else print y;
```

תהיה מסוג if-else והיא תכיל את הביטוי "x<y", את גוף ההצהרה "print x" ואת גוף ההצהרה "האחר", כלומר מה שייקרה להצהרה אם הביטוי לא מתקיים – "print y". יחד עם זאת, ההצהרה לא תכיל מידע שיכול להיות קיים בהצהרות אחרות כמו ביטוי התחלתי, שקיים במקרה של לולאות for שבהן יש ביטוי שמצהיר על משתנה int שעליו תתבסס הלולאה (לדוגמת "int i = 0"), וביטוי נוסף שמגדיר מה קורה לאותו משתנה בסוף כל ריצה של הלולאה (כמו "i++").

<u>Expression.3 – ביטוי</u>: שילוב של ערכים ואופרטורים שמוערכים יחד בהתאם לסט חוקים מסוים, ו"מניבים" ערך כגון מספר שלם, מספר עשרוני (לא שלם), מחרוזת וכדומה. הביטוי יכיל מידע כמו סוג הביטוי (השמה, חיבור, חיסור, כפל וכו' או ערכים בודדים כמו מספרים ומחרוזות ושמות משתנים), תת-הביטוי השמאלי, תת-הביטוי הימני ועוד. מעבר לכך, יכולים להיות ביטויים אונריים כגון not ושערים לוגיים נוספים, increment (++), decrement (--), +, - וכו'. לביטויים אלה יופיע ביטוי אחר אחד בלבד, לפני או אחרי, כך שיהיה ביטוי ימני ולא יהיה ביטוי שמאלי, או להפך. לבסוף, indexing יכולה גם היא להיחשב כביטוי של אופרטור בינארי אשר צידו השמאלי זה שם המערך וצידו הימני זה ה-index הרצוי.

<u>טיפוסים</u>

בעתיד ארצה שתהיה לשפה את היכולת להכיל טיפוסים מורכבים כמו מערכים של מערכים, פונקציות שמקבלות מספר ומחזירות מערך, פונקציה שמחזירה פונקציה ואף פונקציה שמחזירה מערך של פונקציות. כדי להגיע לשם, אצטרך ראשית להגדיר את הטיפוסים הפרימיטיביים: bool, int, char, string, void, מערכים ופונקציות. לאחר שהגדרנו את הטיפוסים הפרימיטיביים, נרצה שלכל טיפוס יהיו שתי תכונות נוספות מלבד סוג הטיפוס – תת-טיפוס ופרמטרים. אלו יעזרו ביצירת טיפוסים מורכבים, ובהעברה והחזרה של טיפוסים שונים בפונקציות.

כדי להתחיל לראות כיצד הכל מתחבר נבנה

3. Semantic Analysis – ניתוח סמנטי
4. Intermediate Code Generation – יצירת/תרגום לקוד ביניים
5. Code Generation – תרגום לקוד מכונה