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:

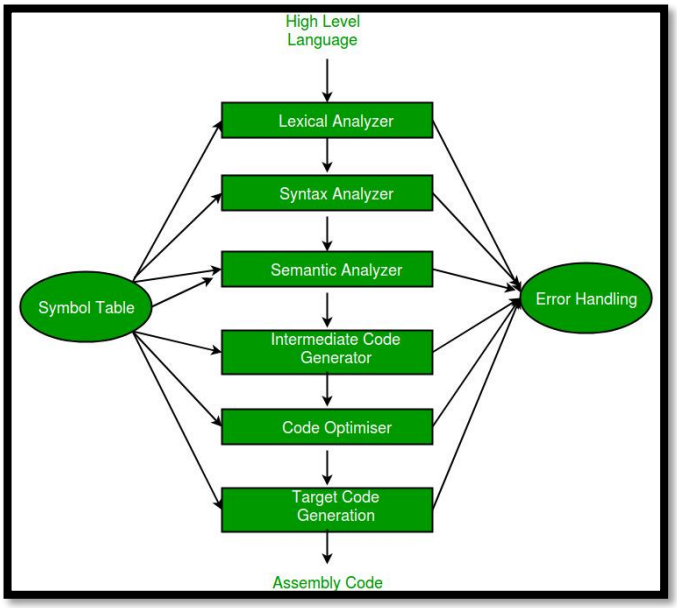
1. <u>התקנה במחשב המשתמש והרצה דרך command-line</u>: המשתמש יוכל להתקין את המהדר במחשב האישי שלו, ולאחר מכן להריץ אותו (דרך הcmd לדוגמה), כאשר בין הארגומנטים יהיו - שם התוכנית שאותה מעוניין המשתמש לקמפל (main.gy, לדוגמה) ודגלים שיוכלו למשתמש לשלוט על הפלט של המהדר, ועל תהליך הקמפול. המשתמש יוכל לבקש 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.

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 סריקה - "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.)

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



שלבי תהליך ההידור, מתוך אתר GeeksforGeeks

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

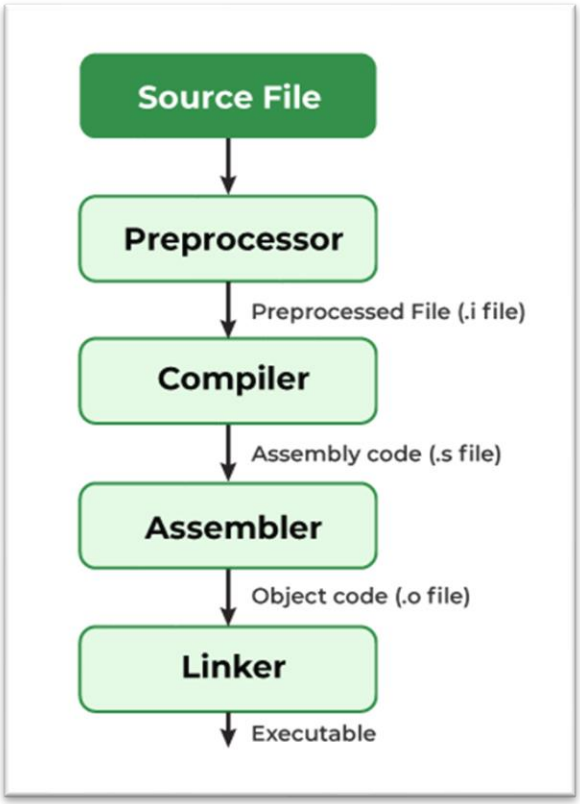
כאמור, שלב הקומפילציה שבו יעסוק המהדר שלי הוא התרגום משפה עילית / שפת ביניים לשפת סף.

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.

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