מכללת כפר סבא

הצעת פרוייקט - יג הנדסת תוכנה

MORA-C

Modern Optimizing Recursive Assembly Compiler

שם פרוייקט: MORA-C

סמל מוסד: 470112

מכללה: אורט דיזיין כפר סבא

מגיש: יאיר פוזננסקי , 209599802

מנחה: מיכאל צ'רנוביליסקי

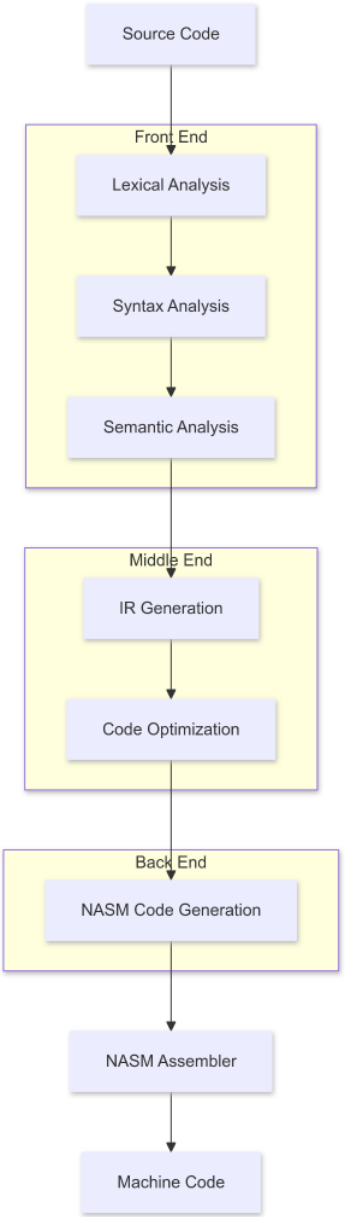
רקע תאורתי בתחום הפרוייקט

מה זה קומפיילר?

קומפיילר הוא כלי תוכנה חיוני המתרגם קוד מקור הכתוב בשפת תכנות עילית לשפת מכונה או אסמבלי שניתנת להרצה על המחשב. תהליך הקומפילציה הוא תהליך מורכב המורכב ממספר שלבים עיקריים, כאשר כל שלב מבצע טרנספורמציה ספציפית על הקוד.

הקומפיילר מחולק ל-3 חלקים עיקריים:

1. Front End - אחראי על קריאת קוד המקור והבנתו
2. Middle End - אחראי על אופטימיזציות וייצור קוד ביניים
3. Back End - אחראי על ייצור קוד המכונה הסופי



שלבי הקומפילציה:

1. ניתוח לקסיקלי (Lexical Analysis):
- מפרק את קוד המקור לאסימונים (Tokens)
- משתמש באוטומט סופי דטרמיניסטי (DFA) המיוצג על ידי מטריצת מעברים
- מזהה מילים שמורות, מזהים, מספרים, אופרטורים וכו'
2. ניתוח תחבירי (Syntax Analysis):
- בודק את המבנה התחבירי של התוכנית
- משתמש ב-Recursive Descent Parser
- מייצר עץ תחביר מופשט (AST)
3. ניתוח סמנטי (Semantic Analysis):
- בודק את נכונות התוכנית מבחינה לוגית
- בדיקת טיפוסים
- בדיקת הצהרות משתנים
- ניהול טבלת סמלים
4. ייצור קוד ביניים (IR Generation):
- מתרגם את ה-AST לייצוג ביניים
- מאפשר אופטימיזציות
5. אופטימיזציה (Optimization):
- משפר את הקוד מבחינת יעילות
- הסרת קוד מת
- פישוט ביטויים
6. ייצור קוד (NASM (CODE GEN:
- מתרגם את קוד הביניים לאסמבלי NASM x64
- מותאם למערכת ההפעלה Windows

סוגי קומפיילרים

ישנם כמה סוגים שונים של קומפיילרים, בהתאם למטרות השימוש:

קומפיילר למכונה (Machine Compiler):
מתרגם את קוד המקור ישירות לקוד מכונה.
דוגמה: GCC שמקמפל קוד C או C++ לארכיטקטורות x86 או ARM.

קומפיילר לקוד ביניים (Intermediate Code Compiler):
מתרגם את קוד המקור לקוד ביניים, ולאחר מכן מפרש או מקמפל אותו לקוד מכונה.
דוגמה: קומפיילר ה-Java שמפיק Java Bytecode, אשר מתורגם לקוד מכונה על ידי ה-JVM (Java Virtual Machine).

Just-In-Time Compiler:
קומפיילר שמקמפל את הקוד בזמן ריצה ולא מראש.
דוגמה: HotSpot JVM ב-Java או JIT ב-.NET CLR.

בפרוייקט שלי בחרתי לכתוב קומפיילר מכונה.

קומפיילר מכונה - היתרונות וחסרונות

יתרונות:

- **ביצועים גבוהים:** מאחר שהקומפיילר מייצר קוד שמותאם ישירות לארכיטקטורת x64, הקוד שנוצר יעיל במיוחד ומנצל את היכולות של המעבד בצורה מיטבית. הקוד רץ בצורה מהירה יותר בהשוואה לקוד שעובר דרך שכבת ביניים, כמו קוד Bytecode ב-Java.

- **גילוי שגיאות בזמן הקומפילציה:** מכיוון שהקומפיילציה נעשית מראש, ניתן לגלות שגיאות תחביריות וסמנטיות לפני הרצת התוכנית.

חסרונות:

- **חוסר ניידות**: הקוד שנוצר על ידי קומפיילר למכונה עבור x64 אינו ניתן להרצה על ארכיטקטורות אחרות, כמו ARM או x86. אם תרצה להריץ את התוכנה על מערכת אחרת, יש לקמפל אותה מחדש עבור הארכיטקטורה המתאימה.
- **זמן קומפילציה ארוך**: תהליך הקומפילציה ישירות לקוד מכונה דורש אופטימיזציות רבות, מה שיכול להאריך את זמן הקומפילציה, במיוחד בפרויקטים גדולים.

תיאור הפרוייקט

פרויקט זה מתמקד בקומפיילר בשם MORAC המיועד לתרגם שפה אשר דומה לשפת C לקוד אסמבלי NASM/x64. הקומפיילר תומך במאפיינים מתקדמים כגון מצביעים, מערכים ופונקציות, תוך שימוש בטכניקות מקובלות בתחום כמו ניתוח לקסיקלי מבוסס DFA וניתוח תחבירי רקורסיבי (Recursive Descent Parsing).

MORAC מתוכנן להיות קומפקטי ויעיל. הקומפיילר משתמש בטכניקות אופטימיזציה בסיסיות ועוקב אחר מוסכמות הקריאה שלWindows x64 , מה שמבטיח תאימות עם סביבות העבודה Windows.

יתרונו העיקרי של הקומפיילר הוא היכולת לתרגם קוד C בסיסי לקוד אסמבלי יעיל, תוך שמירה על פשטות המימוש ובהירות הקוד.

תכונות השפה:

1. טיפוסים בסיסיים:
- int - מספרים שלמים
- char - תווים
- float - מספרים עשרוניים
- מצביעים לכל הטיפוסים
- מערכים חד-מימדיים
2. מבני בקרה:
- לולאות (for, while)

- תנאים (if-else)
- פונקציות

3. פונקציונליות נוספת:

- פונקציות פלט בסיסיות
- ניהול זיכרון בסיסי
- קריאות מערכת בסיסיות

טיפול בשגיאות:

- שגיאות לקסיקליות (תווים לא חוקיים)
- שגיאות תחביריות (מבנה קוד שגוי)
- שגיאות סמנטיות (שימוש שגוי בטיפוסים)
- הודעות שגיאה ברורות עם מיקום השגיאה

פלט סופי:

- קובץ אסמבלי NASM x64
- מותאם ל-Windows
- ניתן להרצה ישירות על המעבד

תיאור בעיה אלגוריתמית:

ניתוח לקסיקלי (Lexer)

סקירת האלגוריתם
הלקסר מממש אוטומט סופי דטרמיניסטי (DFA) באמצעות מטריצת מעברים.

פרטי מימוש
מבנה נתונים: מטריצה דו-ממדית כאשר:
שורות מייצגות מצבים

דוגמת DFA
ממצב START:

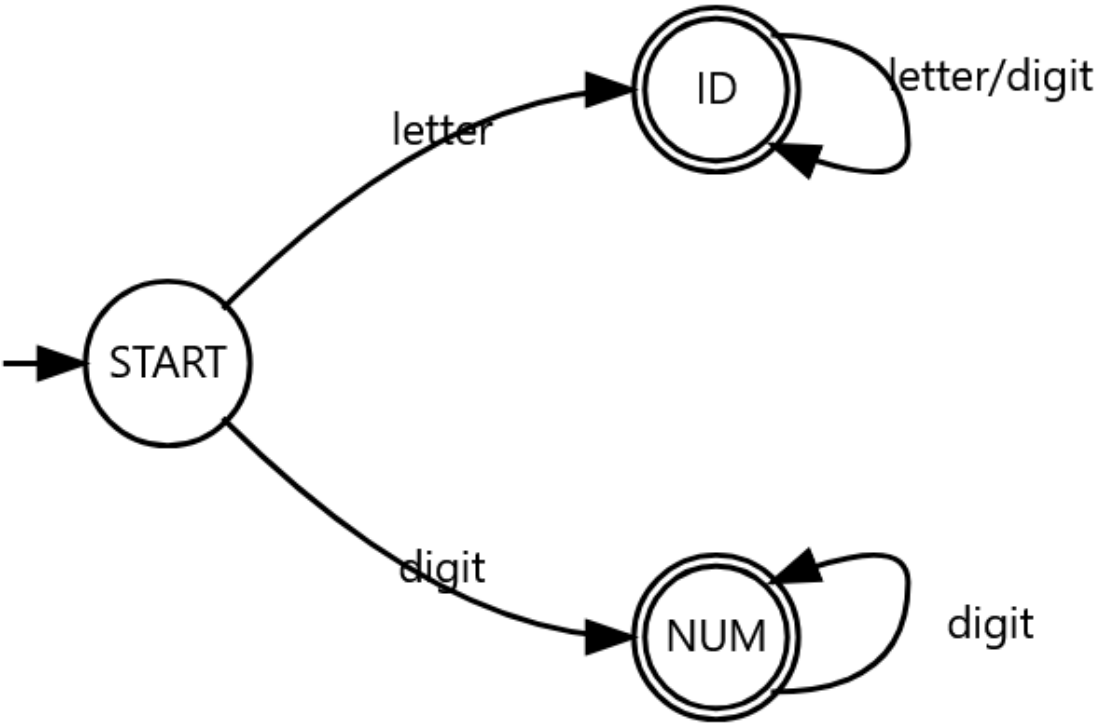
- אות -> מצב 1 ID
- ספרה -> מצב 2 NUM
- רווח -> חזרה ל-0 START

ממצב ID:

- אות/ספרה -> נשאר ב-1 ID
- כל השאר -> חזרה ל- (-1) START

ממצב NUM:

- ספרה -> נשאר ב-NUM 2
- כל השאר -> חזרה ל- START (-1)



מטריצת המעברים

State	letter	digit	operator	space	quote	/
START=0	1	2	3	0	4	3
ID=1	1	1	-1	-1	-1	-1
NUM=2	-1	2	-1	-1	-1	-1

מכללת כפר סבא

ניתוח תחבירי (Parser)

הפרסר יממש Recursive Descent Parser, שהוא פרסר מTOP - DOWN היוצר עץ פירוק על ידי עיבוד רקורסיבי של הקלט בהתאם לכללי הדקדוק.

דרישות לניתוח סמנטי

1. לבנות ולתחזק טבלת סמלים
2. לבצע בדיקות טיפוסים
3. לוודא הצהרות משתנים
4. לבדוק קריאות לפונקציות ופרמטרים
5. לאמת אינדקסים במערכים
6. להבטיח שימוש נכון במצביעים
7. לייצר הודעות שגיאה מתאימות

דרישות לייצור קוד ביניים

דורש עוד מחקר רב

1. לטפל ב:
   - הצהרות משתנים
   - פעולות חשבוניות
   - קריאות לפונקציות
   - גישה למערכים
   - פעולות מצביעים

דרישות לאופטימיזציה

דורש עוד מחקר רב

1. לבצע חישוב קבועים
2. לבטל קוד מת
3. לפשט ביטויים אלגבריים
4. לייעל לולאות
5. להסיר חישובים מיותרים
7. לייעל גישות לזיכרון

מכללת כפר סבא

דרישות לייצור קוד

דורש עוד מחקר רב

1. למפות קוד ביניים לאסמבלי NASM x64
2. לטפל ב:
   - הקצאת רגיסטרים
   - ניהול מחסנית
   - מוסכמות קריאה לפונקציות
   - שיטות גישה לזיכרון
   - קריאות מערכת
3. לייצר קוד תואם Windows
4. לעקוב אחר מפרטי x64 ABI

דרישות לטיפול בשגיאות

כל רכיב צריך:

1. לזהות שגיאות ספציפיות לשלב שלו
2. לייצר הודעות שגיאה משמעותיות
3. לכלול מיקום בקוד המקור
4. להתאושש משגיאות כאשר אפשר
5. להעביר שגיאות בלתי ניתנות לתיקון
6. לשמור על הקשר השגיאה

תהליכים עיקריים בפרוייקט

חקר וניתוח מבנה קומפיילר ושלביו השונים:

- למידת ארכיטקטורת קומפיילרים
- הבנת שלבי הקומפילציה
- בחירת אלגוריתמים מתאימים לכל שלב

בניית מנתח לקסיקלי (Lexer):

- מימוש DFA באמצעות מטריצת מעברים
- טיפול בטוקנים בסיסיים (מזהים, מספרים, אופרטורים)
- טיפול בשגיאות לקסיקליות

פיתוח מנתח תחבירי (Parser):

- מימוש Recursive Descent Parser
- בניית עץ תחביר מופשט (AST)
- טיפול בשגיאות תחביריות

מימוש ניתוח סמנטי והמרה לקוד ביניים:

- בדיקות טיפוסים
- ניהול טבלת סמלים
- יצירת ייצוג ביניים של הקוד

פיתוח מחולל קוד NASM ואופטימיזציות:

- תרגום לאסמבלי x64 NASM
- ביצוע אופטימיזציות בסיסיות
- יצירת קוד יעיל למערכת ההפעלה Windows

שפות תכנות

C/C++

סביבת עבודה

Visual studio code/ windows

לוחות זמנים

חודש 1 - למידה ותכנון

- שבוע 1-2: חקר וניתוח מבנה קומפיילר
- שבוע 3-4: תכנון מפורט ובניית תשתית

חודש 2 - Lexer ו-Parser

- שבוע 1-2: מימוש Lexer
- שבוע 3-4: מימוש Parser

חודש 3 - ניתוח סמנטי וקוד ביניים

- שבוע 1-2: ניתוח סמנטי
- שבוע 3-4: קוד ביניים

חודש 4 - מחולל קוד ואופטימיזציות

- שבוע 1-2: מחולל קוד NASM
- שבוע 3-4: אופטימיזציות ובדיקות

חתימת הסטודנט : _____

חתימת רכז המגמה :_____

אישור משרד החינוך :_____