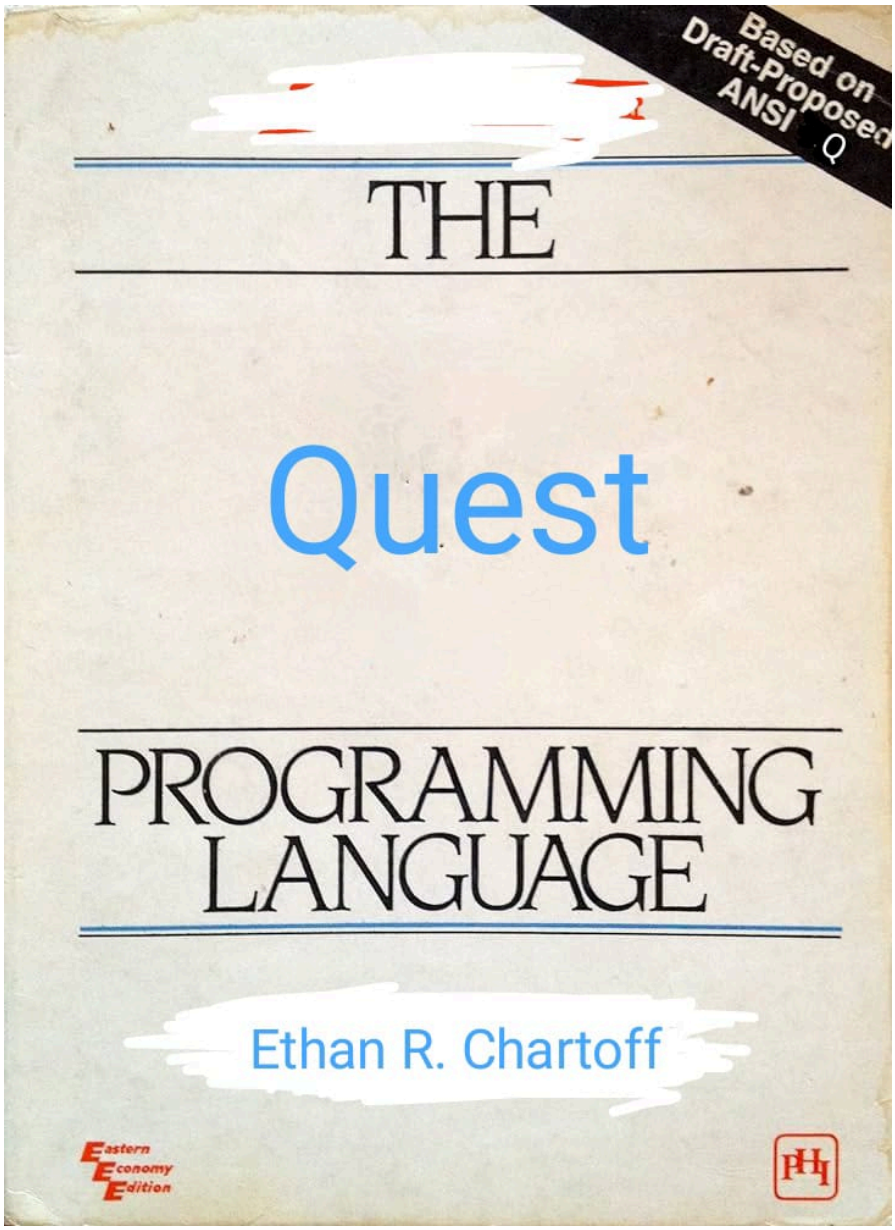
תכנון פרויקט - י"ג הנדסת תוכנה



סמל מוסד: 471029
שם המכללה: מכללת אורט הרמלין נתניה
שם הסטודנט: איתן רפאל צ'רטוף
ת"ז הסטודנט: 215310715
שם הפרויקט: The Quest Programming Language

ארכיטקטורת קומפיילר

מטרת הקומפיילר

מטרת הקומפיילר היא להפוך קוד הרשום בשפת התכנות Quest לשפת מכונה התואמת באלגוריתמים ומבני הנתונים הרשומים בקוד המקור. הקומפיילר מקבל קוד מקובץ (או כמה) ומתרגם את אותו הקוד לקובץ הרצה.

מבנה מופשט

קומפילציית קוד מקור של שפה עילית לשפת מכונה הינו אלגוריתם גדול ומסובך המורכב משבעה רכיבים עיקריים:

1. עיבוד מקדים (preprocessing)

השלב הראשון של המהדר הוא *עיבוד מקדים*. תכנית הקולטת נתונים מקדימים בשביל שהפלט שלה ישמש בתכנית אחרת. סוג תכנית זו תקרא תמיד לפני תכנית אחרת שתשתמש בפלט תכנית זאת, לכן השם עיבוד מקדים. דוגמה פשוטה לסוג תכנית זו אפשר למצוא בקומפיילר של C, שחלק אחד שה-preprocessor עושה הוא לקחת את כל השורות המתחילות עם '#' והופך אותם להוראות בשביל הקומפיילר.

2. ניתוח מילוני (lexical analysis)

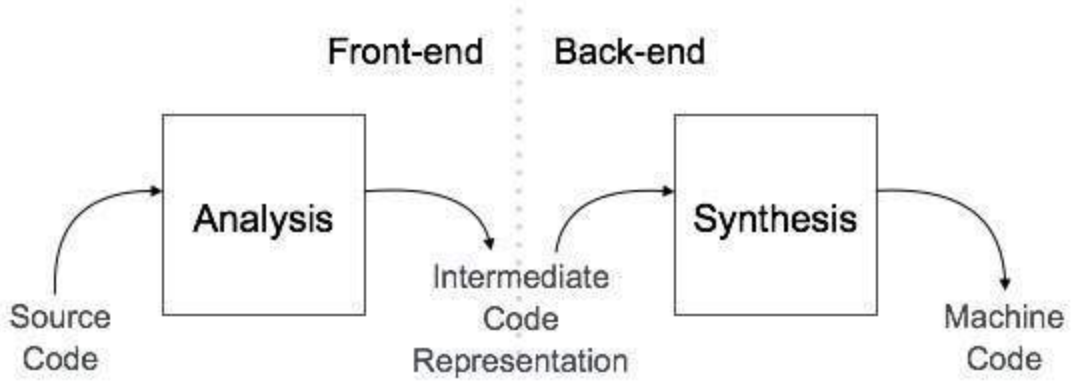
בחלק זה התכנית קוראת את הקוד והופכת אותו אל אסימונים או "lexical tokens", שהם כמו היחידות הבסיסיות של השפה. אפשר לקרוא לתכנית זו ה"lexer". החלק הזה מבטיח שכל המילים הנמצאות בשפה מותאמות למילון השפה.

3. ניתוח תחביר (syntax analysis/parsing)

חלק זה קולט את האסימונים שקיבלנו מהתכנית הקודמת ומחיל אליהם כללים מוגדרים כדי לקבוע עם הקוד נכון מבחינה תחבירית. קוראים לתכנית זו ה"parser". המנתח התחבירי מבטיח שאין לקוד המקור בעיה אם התחביר.

4. ניתוח סמנטי (semantic analysis)

בשלב זה התכנית בוחנת האם לקוד יש שגיאות סמנטיות כמו טיפוסים לא תואמים או משתנה שלא הוגדר. השלב הזה מבטיח שאין שגיאות סמנטיות בקוד המקור.



5. יצירת קוד ביניים (intermediate code generation)

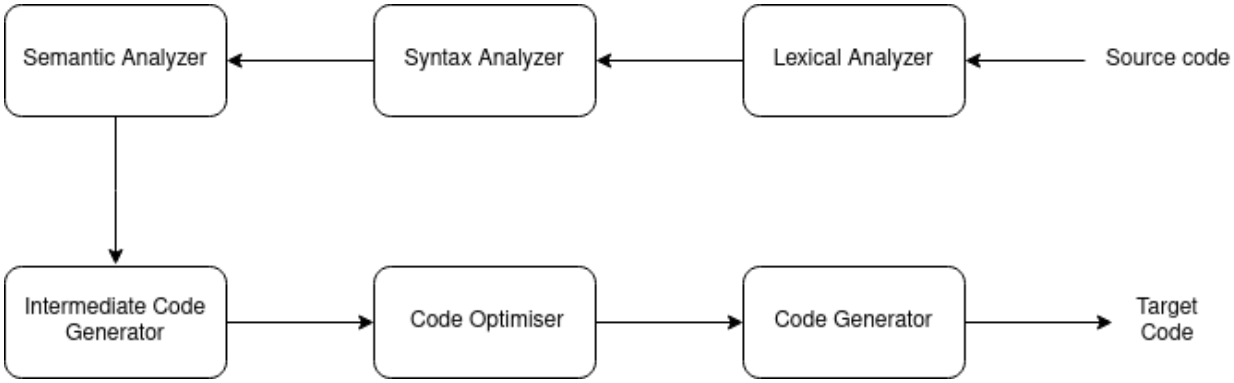
לאחר ניתוח הקוד, התכנית מייצרת פרזנטציה שונה של הקוד ופולטת אותה בשביל שהקומפיילר אשתמש בא לאופטימיזציה ותרגום.

6. אופטימיזציה (code optimization)

שלב זה משתמש בפלט של התכנית הקודמת ומשפר אותה. תכנית זו עושה דברים כמו מוחקת שורות קוד מיותרות ומסדרת את ההוראות בשביל למהר התכנית הסופית.

7. יצירת קוד (code generation)

בשלב זה התכנית קולטת את הקוד המשופר ומתרגמת אותו לשפת המחשב הרצויה.



כל רכיב זה הוא מבנה משל עצמו עם קלט ופלט שונה, והם משתמשים כמו שכבות בשביל להגיע לתוצאה הסופית, קוד מקומפל.

Symbol table

טבלת סמלים היא מבנה נתונים המשמש את המהדר לאחסון מידע על הסמלים השונים, כגון מזהים, קבועים, נהלים ופונקציות, בקוד המקור של תוכנית. היא שומרת מידע חיוני על כל סמל, כולל שמו, סוגו,

תחומו (scope) ותכונות אחרות שלו. טבלת הסמלים משומשת במהלך כל שלבי הקומפילציה, ונבנת בשלבי הניתוח. מטרות טבלת הסמלים:

- זיהוי סוג משתנים, כתובת הזיכרון ושמות של משתנים.
- ניהול משתנים בהתייחס לתחום.
- משומש בשביל להתמודד עם שגיאות.
- הטבלה מסדרת את הסמלים והתכונות שלהם מה שעוזר לניהול התכנית.
- משתמשים בטבלה בשביל ליצור את הקוד הסופי.

התמודדות עם שגיאות

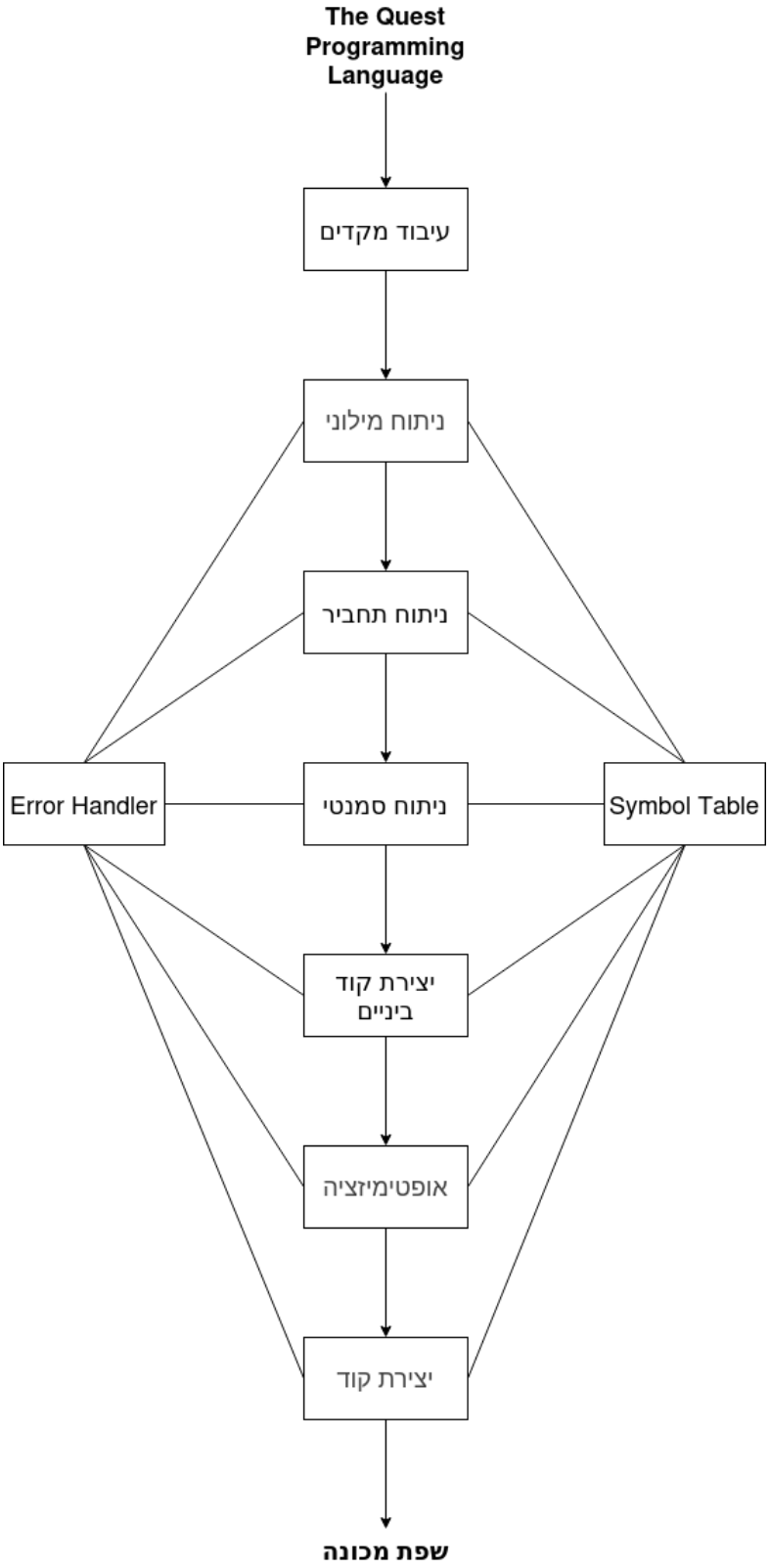
בנוסף לכל הרכיבים האלו, הקומפיילר צריך לדעת איך להתמודד עם שגיאות. לכל מבנה שונה יהיה שגיאות שונות, לדוגמא, במנתח המילולי, יכול להיות שגיאה לקסיקלית, שבא אותו מנתח מזה רצף תווים לא מזוהה ואינה יודעת איך להתמודד אם אותו רצף. נחלק את השגיאות שיכולות להיות לנו לארבעה קטגוריות:

1. **שגיאות לקסיקליות** - המנתח המילוני קורא רצף תווים לא מזוהה ולא יודע איך להתמודד עם אותו הרצף, לדוגמה, שם של סוג משתנה רשום בצורה שגויה.
2. **שגיאות תחביריות** - המנתח התחבירי מזהה בתחביר החומרה, כלומר אסימון במקום שהוא לא צריך להיות, לדוגמא, סוג משתנה במקום לא רצוי או סוגריים פותחות ללא סוגר מתאים.
3. **שגיאות סמנטיות** - המנתח הסמנטי מזהה שגיאה סמנטית, לדוגמא חיבור בין שני סוגי משתנה שונים בלי דרך למצוא תוצאה תואמת.
4. **שגיאות לוגיות** - שגיאות בלוגיקה של הקוד, לדוגמא לולאה אין-סופית.

בשביל לזהות את אותם שגיאות, נשתמש בעוד רכיב הנקרא **מטפל השגיאות (Error Handler)**.
למטפל השגיאות יהיה שלושה ייעודים:

- זיהוי שגיאות
- דיווח שגיאות (במידה ורצוי)
- טיפול בשגיאות (במידה ואפשר)

תרשים סופי


The Quest Programming Language
עיבוד מקדים
ניתוח מילוני
ניתוח תחביר
Error Handler
ניתוח סמנטי
Symbol Table
יצירת קוד ביניים
אופטימיזציה
יצירת קוד
שפת מכונה

שלבי הקומפילציה

מבני נתונים

במהלך פרק זה יתוארו כל מבני הנתונים, אציין אם הם כלליים בשביל כל שלבי הקומפילציה או שהם מיוחדים לשלב מסויים.

חשוב לזכור למה אנחנו בוחרים מבני נתונים אחד לגבי השני, אפילו שבסופו של דבר הם יעזרו לנו להגיע לאותה מטרה. מבני נתונים הופכים את האלגוריתמים שלנו ליותר יעילים, מפחיתים את זמן הריצה, חוסכים בזיכרון ויכולים להפחית את המורכבות והסיבוכיות של אלגוריתם ספציפי, ומבני נתונים מסוימים עושים את העבודה הזאת יותר טוב ממבני נתונים אחרים. כמובן שבחירת מבני הנתונים הכי טוב היא תלות האלגוריתם.

מבני נתוני המנתח המילוני

אוטומט דטרמיניסטי סופי (Deterministic finite automaton)

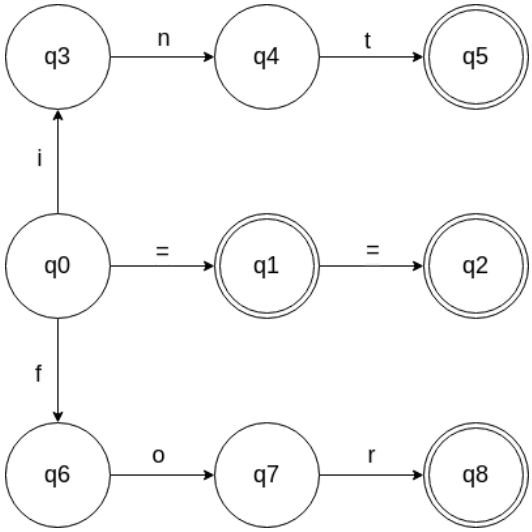
המנתח המילולי קורא את קוד המקור תו אחר תו בשביל להתאים את המילים לאסימונים המתאימים שלהם, לכן חשוב שההתאמה הזאת תהיה כמה שיותר אופטימלית. כלומר, שסיבוכיות הזמן של אלגוריתם ההתאמה בין מילה לאסימון תהיה בעל החסם האסימפטוטי העליון הקטן ביותר. לכן, בשביל שההתאמה תהיה כמה שיותר אופטימלית, בחרתי להשתמש ב*אוטומט דטרמיניסטי סופי*, ונשתמש בו להכין אלגוריתם שמתאים מילה לאסימון במילון השפה בסיבוכיות O(1) .

אוטומט דטרמיניסטי סופי (באנגלית Deterministic finite automaton או DFA בקיצור) הוא מודל מתמטי המגדיר שפה פורמלית. המודל מורכב מקבוצה סופית של מצבים בעל כללי מעבר ביניהם, כלומר חוקים המגדירים מה לעשות במצב כאשר נקלט אות מהקלט. הקלט יהיה אוסף של אותיות (מילה) מתוך הא"ב של השפה, שהיא בנויה מקבוצה של סימנים, לדוגמה, בשפת Quest, הא"ב יהיה כל תווי הascii. הסיבה שהאוטומט דטרמיניסטי היא שכל מצב ידוע מראש ומוגדר. בדר"כ מוגדר מצב בוא מתחילים, אנחנו נגדיר אותו כהמצב עם השם q0.

נממש מבנה זה בשפת C בעזרת מערך דו-ממדי, כאשר:

- כל 128 תווי ה-ascii יוצגו ע"י עמודות המערך הדו-ממדי, כאשר כל אינדקס מותאם לערך האות (לפי טבלת הascii).
- כל המצבים ייוצגו ע"י שורות המערך הדו-ממדי, המצב האחרון שנהיה בוא יהיה תואם לאסימון.
- כל ערך בעמודה התואמת לסימן ובשורה התואמת למצב יהיה שווה למצב הבא שהאוטומט יקפוץ אליו.

לדוגמא, נגדיר DFA שלוקח את המילים: int, for, ==, =



נמיר את זה למערך דו ממדי (נתעלם מזה שאינדקס העמודה מתאים לערך האות):

	=	i	n	t	f	o	r
0	1	3	-1	-1	6	-1	-1
1	-1	2	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	4	-1	-1	-1	-1
4	-1	-1	-1	5	-1	-1	-1
5	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	7	-1
7	-1	-1	-1	-1	-1	-1	8
8	-1	-1	-1	-1	-1	-1	-1

תיאור מערכת