

הקדמה:

קומפילר מחולק ל 3 שלבים עיקריים:

1. חלק קדמי שמבצע ניתוח לקוד .
2. חלק שמבצע ניתוח סמנטי ומוודא תקינות הקוד.
3. חלק אחורי שלוקח ייצוג ביניים ויוצר ממנו קובץ הפעלה.

שיטה זו מאפשרת לנו לבנות קומפילרים לכל ארכיטקטורה ע"י שימוש בחלק קדמי ואחורי מתאים ולא לכתוב הכול מחדש.

בביצוע הפרויקט חילקנו אותו ל 5 שלבים:

1. ניתוח לקסיקלי: לקחנו את הקוד והמרנו אותו לרצף של טוקנים.
2. ניתוח סינטקטי: לקחנו את רצף הטוקנים והמרנו אותו ל AST.
3. ניתוח סמנטי: בנינו את טבלאות הסמלים וטבלת הטיפוסים וביצענו בדיקות סמנטיות ב AST.
4. תרגום לייצוג ביניים: המרנו את ה AST שלנו לשפת LIR, עם אופטימיזציות של רגיסטרים.
5. יצירת קוד סופי: המרנו את שפת ה LIR לאסמבלי.

תזכורת ביטויים רגולריים:

תבנית	מתאים ל..
x	התו x.
.	כל תו מלבד שורה חדשה
[xyz]	כל אחד מהתווים x,y,z
R?	R אופציונאלי
R*	אפס או יותר הופעות של R
R+	מופע אחד או יותר של R
R1R2	R1 ואחריו R2
R1 R2	R1 או R2
(R)	R בעצמו

כדי להכניס תווים שמורים (()*+ ועוד) לתוך ביטוי רגולרי יש להשתמש ב \, לדוגמא +\.

כדי להכניס מחרוזת יש להשתמש במרכאות לדוגמא "abcd".

ניתוח לקסיקאלי (Scanner):

בביצוע הפרויקט, שלב זה בוצע בעזרת הכלי האוטומטי JFLEX, שמשתמש בכללים המוגדרים בעזרת ביטויים רגולריים כדי לנתח את הקלט. הקלט בשלב הזה הוא טקסט, והפלט הוא רצף של טוקנים. בשלב זה מזהים בקוד אם יש דברים לא חוקיים בסיסיים (מילות מפתח לא חוקיות) מסירים רווחים והערות, ומטפלים במאקרו וב-Include. שלב זה מקל על הניתוח הסינטקטי. טוקן מוגדר ע"י השפה, ובעצם מהווה את היחידות הקטנות ביותר בשפה. אנו מגדירים טוקנים בעזרת ביטויים רגולריים. תיאור טוקן מכיל את השם שלו, ואת הביטוי הרגולרי שהוא מייצג. מנתח לקסיקאלי פותר מצבים של דו משמעות ע"י בחירת הטוקן הארוך ביותר. אם יש שני טוקנים מתאימים באותו אורך אז הוא בוחר בראשון שהתאים. בניית המנתח: הכלי יוצר אוטומט לא דטרמיניסטי לכל הביטויים, לאחר מכאן מתבצע איחוד של כולם. את האוטומט הלא דטרמיניסטי מעבירים לאוטומט דטרמיניסטי ואז עושים לו צמצום. לרוב השפות מנתחים לקסיקאליים יכולים להבנות בקלות ע"י כלים אוטומטיים. יוצאת מן הכלל למשל Fortran שבה לרווחים אין משמעות.

ניתוח סינטקטי (Parser):

בביצוע הפרויקט, שלב זה בוצע בעזרת הכלי האוטומטי JavaCup (יוצר דקדוק (LALR(1)), שמשתמש בכללים המוגדרים בעזרת דקדוק חסר הקשר כדי לנתח את רצף הטוקנים ולבנות AST. הקלט בשלב הזה הוא רצף טוקנים, והפלט הוא AST. בשלב זה מזהים בקוד אם יש שגיאות סינטקטיות. חסרים סוגריים, או שיש רצף לא חוקי של טוקנים וכולי.

בשלב זה מבחינת זיהוי שגיאות, מעבר למציאת שגיאה ראשונה אפשר לנסות לבצע "תיקון" כדי להמשיך לנתח את הקלט, למשל אם חסר נקודה פסיק, לבצע השלמה וכדומה.

נשתמש בדקדוק חסר הקשר כדי להגדיר סינטקס של שפת תכנות כיוון שכך אנו תופסים את מבנה התוכנית (היררכיה), נוכל להשתמש בתוצאות פורמאליות תיאורטיות ונוכל ליצור מנתחים "יעילים".

דקדוק דו משמעי הוא דקדוק שבו קיים קלט שלו שני עצי גזירה שונים. הפיכת דקדוק דו משמעי לחד משמעי במצגת 3, שקופית 25 ובתרגול 3 שקופית 11.

יש 2 סוגים של מפסקים:

1. Top-Down LL

עבור כל Non-Terminal וטוקן, נבצע תחזית לגבי הגזירה הבאה (לפי הרישא). לשם כך על הדקדוק אסור להכיל שני כללים עם רישא זהה. אסור לאפשר רקורסיה לשמאל (למשל $E \rightarrow E + T$). קריאה משמאל, גזירה משמאל. עובר על העץ ב Preorder.

2. Bottom-Up LR

אנו מנסים לבנות את העץ מלמטה למעלה, ואם נגיע לשורש אז הקלט חוקי. קריאה משמאל, גזירה מימין. עובר על העץ ב Postorder. בקבוצה זו נמצאים בין היתר: LR(0), SLR(1), LALR(1), LR(1).

דקדוק LR(0):

- אוטומט דטרמיניסטי המכיל רכיבי LR(0) ואין בו קונפליקטים.
את האוטומט ניתן להמיר לטבלת שליטה שבעצם תייצג את האוטומט.
כדי לבנות את טבלה לדקדוק LR(0):
- נוסיף כלל התחלה $S' \rightarrow S\$\$$.
 - נבנה אוטומט מתאים.
 - נבנה טבלה שמייצגת את האוטומט.
 - אם אין קונפליקטים הדקדוק הוא אכן LR(0), אחרת הוא לא.

דקדוק SLR(1):

מדובר בדקדוק שמשתמש גם בקבוצת Follow כדי לשבור Shift-Reduce Conflict שעלולים לצוץ ב LR(0). נשתמש באותה טבלה מ LR(0). כעת, מותר לנו לעשות Reduce רק אם התו הבא הוא בקבוצת ה Follow שלנו. במקרה שיש לנו חפיפה בין קבוצת ה Follow לחץ יוצא במצב שבו יש קונפליקט, דקדוק SLR(1) אינו מספיק ועדיין לא יפתור את הקונפליקטים.

חישוב קבוצת Follow: מדובר בקבוצת כל הדברים שיכולים להופיע אחרי הטרמינל.

קבוצת First: אם מופיע טרמינל בתחילת הכלל, מוסיפים אותו ($A \rightarrow aB$ מוסיפים את a). אם מופיע לא טרמינל – מוסיפים את קבוצת First שלו ($A \rightarrow Ba$ נוסף ל $First(A)$ את $First(B)$).
חישוב Follow: אם לאחר הלא-טרמינל מופיע טרמינל – נוסף (עבור $A \rightarrow Ba$ נוסף ל $Follow(B)$). אם הלא-טרמינל מופיע בסוף כלל, נוסף את ה $Follow$ של הלא טרמינל שבצד שמאל (עבור $A \rightarrow aB$ נוסף ל $Follow(A)$ את $Follow(B)$).

חישוב קבוצות First ו-Follow בסוף הערות שיעור 4. במצגת שיעור 4 החל משקופית 88.

דקדוק LR(1):

אוטומט דטרמיניסטי המכיל רכיבי LR(1), ובו אין קונפליקטים.
כל רכיב מכיל גם את מה התו הבא אחריו.

דקדוק LALR(1):

אוטומט דטרמיניסטי המתקבל מאוטומט של LR(1), שמבצעים בו איחוד של מצבים כאשר מתעלמים מה Look-Ahead. הצמצום הזה עלול ליצור Reduce\Reduce Conflict אבל לא יכול ליצור Shift\Reduce Conflict שלא היו קיימים ב LR(1).

- דוגמאות לכל הדקדוקים ניתן לראות בתרגיל הבית.

היררכיית הדקדוקים במצגת שיעור 4, שקופית 96.

Abstarct Syntax Tree

משמש ייצוג ביניים לתוכנית.
מיוצג ע"י עץ, מה ששומר על היררכיית התוכנית. נוצר ע"י המפסק.
מוגדר ע"י דקדוק חסר הקשר ולכן יחסית שטוח (לא משמש לפיסוק עצמו).
מילות מפתח וסמלי פיסוק לא נשמרים, מכיוון שאין צורך בהם.

ניתוח סמנטי:

בשלב הזה מבצעים על ה AST בדיקות סמנטיות.
הבדיקות כוללות Scope Checking, Type Checking ועוד כמה בדיקות פשוטות.
ישנם שני סוגים של בדיקות טיפוסים:

- בדיקות סטטיות שאפשר לבצע בזמן קומפילציה.
יתרון השיטה הוא שהוא נותן לנו דרך לתקן את הקוד לפני הרצתו.
יתרון נוסף הוא שאין בזבז זמן ב Runtime מכיוון שאין צורך לבדוק.
- בדיקות דינמיות שמתרחשות בזמן ריצה (נובע בגלל פולימורפיזם בשפות OO)
יתרון השיטה היא שהיא יכולה לעבוד עם טיפוסים שלא מוגדרים מראש.
השיטה הסטטית תכשל במערכת שבה יש פיתוח מהיר של טיפוסים.
הבדיקה הסטטית מאיטה את שלב הקומפילציה.

מימוש טוב לטבלאות סמלים יעשה ע"י Hash Tables. כאשר בכל רשומה בטבלה נשמור את הרמה שלה בתוכנית.

רמה 1: פונ' ומשתנים גלובאליים.
רמה 2: ארגומנטים שמעוברים לפונקציה.
רמה 3: משתנים מקומיים של פונקציות.
הרמות הבאות הם בלוקים פנימיים והמשתנים שמוגדרים בהם.
ניתוח סמנטי מתעסק גם בהמרת טיפוסים. זה כולל המרה מרומזת שבה אם אנחנו מבצעים צמצום הקומפילר יכול להתלונן על כך.

בדרך כלל הבדיקות מתבצעות במעבר או שתיים על העץ.

בביצוע הפרויקט אנו מבצעים שלב זה ע"י בניית טבלת טיפוסים, וטבלאות סמלים לכל SCOPE.

לאחר מכאן אנו יכולים לבצע את כל הבדיקות הנדרשות.

מפרשים:

בהינתן תוכנית בשפת מקור וקלט, המפרש מריץ את התוכנית על הקלט. זאת לעומת קומפיילר שמתעלם מהקלט.

ישנם שני סוגים של מפרשים:

- **מפרש רקורסיבי** – עובר על העץ בצורה רקורסיבית, מפרש כזה עובד לאט עד פי 1000 יותר איטי מקומפיילר. מעולה בגילוי שגיאות.
- **מפרש איטרטיבי** – פי 30 יותר איטי מהקומפיילר. צריך לעבור מהעץ למבנה שטוח. מפרש כזה טוב בגילוי שגיאות אבל פחות טוב מהמפרש הרקורסיבי בגלל המבנה השטוח.

במפרש צריך ייצוג כללי לכל סוג של נתון כי הוא לא יכול להתמודד עם טיפוסים שלא מוגדרים מראש (struct למשל). לכן הוא שומר לכל טיפוס נתונים 2 שדות: size ו type. והוא מייצג כל אובייקט בצורה שניתנת לחלוקה ל 2: החלק הייחודי לאובייקט הספציפי והחלק הכללי לכל האובייקטים מאותו הטיפוס.

כדי לדעת בכל שלב באיזה מצב הוא נמצא המפרש צריך Status indicator. ערכיו האפשריים הם: Normal, Error, Jump, Exception, Return.

שערוך חלקי: ניתן לבצע ניתוח חלקי של התוכנית, בו מבצעים פישוט לחלקים סטטיים בתוכנית. נקבל תוכנית הזזה למקור אך יותר פשוטה. לאחר מכן, ניקח את הקלט ונריץ אותו על התוכנית שקיבלנו. בשיטה זו ניתן לשפר את המשערך להיות גרוע רק פי 30-40 מקומפיילר במקום פי 1000.

לדוגמה תוכנית המחשבת e^n , בהינתן e ניתן לפשט אותה בעזרת שערוך חלקי.

מפרש איטרטיבי: מודל זה קרוב ל CPU. הוא עובר על ה AST, ובודק מי הצומת הבא לביצוע. המשערך עובד עם Switch ענק עם המון פעולות Case. ובנוסף הוא מתחזק מחסנית שבה יש חישוביים חלקיים ומשתנים לוקאליים. כדי להשתמש בו צריך להחזיק AST שבו כל צומת יודע מה הצומת הבאה (Threaded AST).

כדי ליצור את ה AST הזה, מבצעים מעבר Preorder על ה AST. לכל סוג של AST יש את הרוטינה שלו. אנו נתחזק משתנה גלובאלי של Last Node Pointer. תחילת הרצאה 7 בסיכומים של עידן. שקף 12 בשיעור 7.

הבעיה בשיטה זו היא בקריאות של לולאות בהם, אחרי חישוב התנאי הלוגי יש שתי אפשרויות, והצומת צריך להחליט לאן להמשיך.

יש 3 דרכים לתחזק Threaded AST:

1. תרשים זרימה (גרף).
2. מערך.
3. מערך עם פסאדו פקודות (קפיצות וכו).

יצירת קוד (Code Generation):

נותר לנו להמיר את ה AST שמשמש אותנו כייצוג ביניים לשפת מכונה. ניתן או לתרגם אותו ישירות לשפת מכונה, או לתרגם לאסמבלר, ואז קומפיילר של אסמבלר יתרגם אותו לשפת מכונה.

כדי לתרגם את ה AST ישירות לאסמבלר, נוסיף ב AST התייחסות לרגיסטרים וגישות לזיכרון. לאחר מכאן נחפש דפוסים מוכרים בעץ שניתן להמירם לפקודות אמסבלר.

בתרגום כזה יש מספר בעיות:

1. איזה חלק בעץ לתרגם? (איזה תת עץ ניתן לתרגם לפקודה)
2. חלוקה יעילה של רגיסטרים?
3. מהו סדר הפעולות הסופי?

בגלל שלושת הבעיות הללו בעיית בניית התרגום היעיל ביותר לתוכנית היא NPC.

לכן אנו מסתכלים על חלקים קטנים של התוכנית בכל פעם. בנוסף, נשמור מקדם ביטחון בשימוש שלנו ברגיסטרים (פישוט מכונת היעד). לסיום, כאשר מוצאים תבנית מתאימה ישר נצמצם אותה.

מודלים מופשטים של מכונת היעד כוללים את:

- Simple Stack Machine – מודל מחסנית פשוטה (מכונה שמכילה רק מחסנית). בנוסף למחסנית יש לנו מצביע לראש המחסנית ולבסיס המחסנית.
- Register Machine – מודל בו במכונה נתונים לנו קבוצת רגיסטרים, הוראות טעינה ושמירה מרגיסטרים וזיכרון, ופעולות אריתמטיות אותם ניתן לבצע רק על רגיסטרים.

לאחר התרגום למודל המופשט, הקוד שמתקבל בשפת היעד אינו בהכרח אופטימאלי, לכן מבצעים Post-Processing. למשל, שחרור רגיסטרים שהשימוש בהם מיותר, תרגום פקודות לא יעילות (משתנה מוכפל באפס או באחד וכו'). אופטימיזציות אלה מאריכות את זמן הקימפול אך מייעלות את התוכנית.

נשים לב שבתרגום הנחנו שמספר הרגיסטרים שלנו אינסופי, מה שכמובן לא נכון ויוצר קוד לא אופטימאלי. גם מבחינת שימוש יותר רגיסטרים משצריך וגם מבחינת שימוש בזכרון והרד דיסק כשנגמרים הרגיסטרים (Spilling).

אלגוריתם מרכזי לייעול הקוד שלנו הוא סטי אולמן.

אלגוריתם סטי אולמן:

תחילה ניתן משקל לכל צומת לפי כמה רגיסטרים היא צורכת, כאשר המעבר יהיה Bottem-Up. נעשה זאת ע"י שימוש בעקרון ה Labeling.

עיקרון ה Labeling של צומת בעץ:

- אם לשני הבנים יש משקל זהה n , אזי משקל הצומת הוא $n+1$.
- אם אחד מהם כבד מהשני אזי משקל הצומת הוא כמשקל הצומת הכבד.

לאחר שכל הצמתים יש משקל, במקומות שבהם ניתן לשנות את סדר השערוך (בעיקר ביטויים מורכבים), ניצור תחילה את הקוד לצד הכבד יותר, כאשר המעבר יהיה Top-Down.

הוכחת סטי אולמן:

אם עבור תת עץ a דרושים m רגיסטרים, ועבור תת עץ b דרושים n רגיסטרים, וכן $m > n$, אז לאחר חישוב תת העץ a , נשמור את התוצאה ברגיסטר בודד. כדי לחשב את תת עץ b נידרש ל n רגיסטרים. בתוספת רגיסטר התוצאה של a נקבל שהשתמשנו ב $n + 1$ רגיסטרים. נשים לב כי $m \geq n + 1$.

מהוכחה זו נובע גם עיקרון ה Labeling. נסמן $L(x)$ כמשקל של הצומת x . מכאן עבור כל צומת נקבע כי $L(\text{parent}) = \text{Max}(L(\text{left}), L(\text{right}))$ עבור $L(\text{left}) \neq L(\text{right})$

ו- $L(\text{parent}) = L(\text{left}) + 1$ עבור $L(\text{left}) = L(\text{right})$.

תוספות נוספות שייעלו את הקוד שלנו:

ניתן להוסיף פעולות אריתמטיות של זיכרון ורגיסטר (בניגוד לרק רגיסטר ורגיסטר). זהו מודל שקיים במציאות ובמודל כזה יש לנו פוטנציאל טוב לחיסכון נוסף במשאבים.

בשנת 1977 ניסויים הראו שלתוכנית שנכתבת ע"י אדם מספיקים 5 רגיסטרים. אבל בעזרת כלים אוטומטיים נשתמש כמעט תמיד ביותר.

Spilling:

כאשר הקוד שלנו משתמש ביותר רגיסטרים ממה יש לנו, עלינו להשתמש במשתני ביניים אליהם נעביר את תוכן הרגיסטרים, עד אשר נצטרך את תוכן הרגיסטרים שוב. ההחלטה אילו רגיסטרים כדאי לפנות היא בעיה NPC, אולם קיימים פתרונות יוריסטיים.

אחד האלגוריתמים הפשוטים במקרה של שפיכה, הוא לחפש את תתי העצים ה"כבדים" (הדורשים יותר רגיסטרים מאשר יש לנו), בתוכם נחפש תתי עץ "קלים" אותם יש לנו מספיק רגיסטרים לחשב. נחשב את תת העץ הקל, נשמור את התוצאה שלו בזיכרון ונעדכן במקום כל תת העץ הקל את המיקום שלו בזיכרון.

נשים לב שבשגרת שפיכה משתמשים רק אם מספר הרגיסטרים הנדרש גדול ממה שיש לנו בפועל. בעיית הקצאת הרגיסטרים דומה לבעיית צביעת הגרף. זו בעיה NPC אך לקלטים מסויימים יש לה פתרונות לינאריים.

בעיית קוד המיוצר ע"י מכונה:

קלט: AST, ותיאור של מכונה (מספר רגיסטרים, ופעולות ועלותם).

פלט: קוד אופטימאלי המתאים למכונה עבור ה AST.

הבעיה הזו היא NPC.

כדי לפשט את הבעיה, נסתכל על חלקים קטנים בעץ, כמו כן נשתמש במודל מצומצם של המכונה (לא ננצל את כל הפקודות והרגיסטרים).

בלוק בסיסי – חלקים בגרף ללא פיצולים, כלומר נרצה למצוא חלק שתמיד יתבצע בצורה סדרתית. בהינתן AST ננסה לאתר את הבלוקים הבסיסיים המקסימאליים.

בשלב הבא נגדיר גרף תלויות המתבסס על ה – AST. גרף התלויות מאפשר לנו לבצע הקצאה כמעט אופטימאלית של הרגיסטרים.

לקומפילר מותר לשנות את סדר הפעלות, כל עוד זה לא משפיע על התוצאה.

כדי ליצור את גרף התלויות מ AST:

1. צמתי ה AST הופכים לצמתי הגרף.
2. נחליף קשתות ה AST, בקשתות תלות, המחברות בין אופרטור לאופרנדים שלו.
 - Operator → Operand
3. ניצור קשתות מההשמות לשימוש שלהם.
4. ניצור קשת בין ההשמות של אותו שמתנה.
5. נבחר את משתני הפלט (שורשים)
6. נסיר את צמתי ה ; והחצים שלהם.

הגרף שיתקבל יהיה גרף אציקלי, הקשתות בגרף מסמנות תלויות.

מהרגע שנוצר גרף התלויות, יכול להיות שיהיה לנו יותר מנק' כניסת אחת, אבל כל עוד נשמור על התלויות, לא משנה במי נתחיל את החישוב.

כדי לפשט את הגרף, נחבר בין המשתנים לביטוי הסופי שלהם, ונחבר בין ביטוי לשימוש שלהם, בנוסף נמחק צמתים שלא ניתן להגיע אליהם מהשורש.

פתרון יוריסטי לא ברור בסיכום של עידן.

בעיית צביעת גרף – אלגוריתם יוריסטי:

בהינתן גרף לא מכונן נרצה לצבוע אותו במינימום צבעים (בלי שכנים באותו צבע).

תחילה נסתכל על דרגות כל הצמתים, ונתחיל להוציא צמתים מהגרף ע"פ דרגה (מהקטן לגדול), בכל הוצאה מעדכנים את דרגות הצמתים ומכניסים את הצומת לראש המחסנית.

עתה מסתכלים על ראש המחסנית, ובודקים האם קיים צבע שהשתמשנו בו כבר, שצביעה של הצומת הזה בו לא תפר את החוקיות. אם כן צובעים בו, אם לא נשתמש בצבע חדש.

אלגוריתם זה עובד טוב עבור 96 אחוז מהגרפים.

פונקציות (הקדמה ל Activation Records):

בכל שפה על כל פונקציה יש להכריז. ההכרזה אומרת לקומפילר שבפעם הבאה שהוא פוגש את הפונקציה הוא יוכל לעשות בדיקות תקינות סמנטיות של משתנים וערכים מוחזרים.

עבור כל פונקציה עליה הכרזנו, עלינו לכתוב מימוש. המימוש יכול לבוא עם ההכרזה או בנפרד. במקרה שהוא מופיע עם ההכרזה עליו לבוא לפני ה Main.

בהמשך ניתן לקרוא לפונקציות שנכתבו. אם הפונקציה אינה מוגדרת להיות Void היא מחזירה ערך. במהלך השגרה עשויות להיות קפיצות, קפיצה בתוך השגרה אל תוך השגרה אינה בעייתית, אך קפיצות אל מחוץ לשגרה יכולה לדרוש משאבים ומחייבת טיפול מיוחד.

בנוסף, בשפות מסוימות ניתן להעביר שם פונקציה כמשתנה, או להחזיר שגרה בפעולת Return.

פונקציות מקוננות – בשפות מסוימות ניתן להגדיר "פונקציה בתוך פונקציה". המשמעות היא שבתוך הפונקציה מוגדרת שגרה שיכולה להשתמש בכל המשתנים של השגרה החיצונית. צורת כתיבה זו יכולה לגרום לבעיות בעת הקימפול כיוון שעשוי להיות קשה לאפשר גישה למשתנים של פונקציה חיצונית באמצעות מודל מחסנית "אמיתי" (המכיל רק פקודות Push, Pop).

בעיה נוספת שקומפילר יכול להתקבל בו הינו בעת קריאה לפונקצית Goto. פקודות goto יכולה לגרום לקפיצה לא לוקאלית באמצע של פונקציה. למרות שאין לבצע את הפקודות שלפני התווית על הקומפילר לדעת ולהכיר את הסביבה בה הוא נמצא לאחר הקפיצה.

פקודות קפיצה ב C:

- Setjmp – זוכרת את המצב בו אנו נמצאים (מצב הרגיסטרים, השורה בה אנו נמצאים וכו')
- Longjmp – קופצים למקום אשר הוגדר לנו. בנוסף, מבצעת pop לכל הרשומות המיותרות במחסנית.

פרמטריים חלקיים ב C (Currying) – מצב בו אנו מגדירים פונקציה בתוך פונקציה, ומעבירים פרמטרים רק לפונקציה החיצונית. הפונקציה החיצונית מחזירה את הפונקציה הפנימית, עתה נוכל לקרוא לפונקציה הפנימית עם הפרמטר החסר (כאשר הפרמטרים ה"חיצוניים" כבר ידועים לנו).

משתנים (הקדמה ל Activation Records):

הנתונים הידועים על משתנה בזמן קומפלציה הם:

- שם
- טיפוס
- מה הסקופ שלו.
- עד מתי הוא מוגדר.
- מה הגודל שלו.
- מה הכתובת שלו בזיכרון.
- כתובת אמיתית וקבועה, כתובת יחסית (עם offset) או כתובת דינאמית.

מחסנית זמן ריצה (Stack Frames/Activation Records):

על קומפיילר לממש מחסנית זמן ריצה וכך לאפשר את ריצת התוכנית.

המחסנית מורכבת מ Frames, כאשר כל Frame מייצג סביבה בה אנו נמצאים.

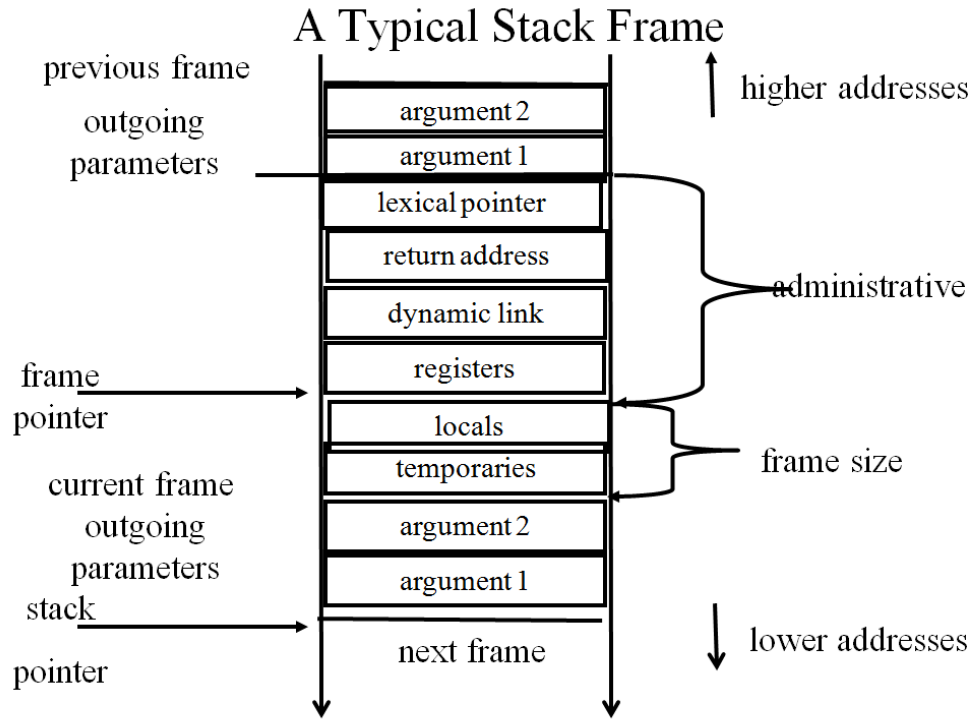
מדיניות המחסנית היא LIFO כצפוי.

בכל קריאה לפונקציה מוקצה מקום במחסנית.

נשים לב שמבנה זה מאפשר תמיכה טבעית יחסית ברקורסיה.

כמו כן, לא מדובר במחסנית רגילה (רק עם PUSH ו POP) אלא במבנה יותר מורכב שניתן לנוע עליו בעזרת אריתמטיקה ולגשת לאיברים פנימיים (נצטרך את זה במיוחד בגישות למשתנים לא לוקאליים שלא נמצאים ב Frame שלנו).

במקרים מסוימים לא נוכל להשתמש במקום שהוקצה במחסנית זמן הריצה בעת הקריאה כי הוא קבוע ונצטרך להשתמש בערימה. למשל הקצאת מערך דינאמי.



12

החלק ה"מנהלי" של ה-Frame מכיל את:

- Lexical Pointer – מצביע על הנתונים של הפונקציה שקראה לנו במחשנית זמן הריצה.
- Return Address – מכיל את הכתובת שאליה נקפוץ בקוד בסוף השגרה.
- Dynamic Link – מכיל מצביע ל-Frame הקודם במחשנית.
- Registers – מצב הרגיסטרים.

Frame Pointer יצביע לתחתית החלק המנהליואחריו יופיעו המשתנים הלוקאליים, משתנים זמניים ופרמטרים (חלק זה תלוי בפונקציה עצמה).

Stack Pointer יצביע לראש המחשנית.

קיימים מספר מימושים של מחשנית זמן הריצה, כאשר ההסכמה היא שבתחילת כל פריים יהיו משתנים מנהלתיים המחזיקים מידע חשוב.

כדי לגשת למשתנה לוקאלי, מחשבים בזמן קימפול את הכתובת במחשנית ע"י $Offset + FP$ שלילי כדי להגיע אליו.

נוצרת בעיה עם שגרות מקוננות, בהם יכול להיות שאנו קוראים למשתנה שמוגדר מחוץ לפריים שלנו.

כדי לפתור את הבעיה ולחשב את ה-OFFSET של המשתנה לאחר שמצאנו אותו, אחד הפתרונות הפשוטים יחסית הוא לתת ערך לכל סביבה כאשר מתחילים ב 0 וכל סביבה שנוצרת תקבל את הערך של הסביבה שיצרה אותה ועוד אחד. ולאחר מכאן לפי ההפרשים בין הסביבה הנוכחית לבין הסביבה שבה נמצא המשתנה ניתן לבצע חישוב שייתן לנו את ה-Offset ביחס ל FP הנוכחי.

Lambda-lifting: חוקרים מסוימים טוענים שעדיף לשטח קריאות מקוננות ע"י העברת כל המשתנים שזמינים לנו כמשתנים לפונקציה כך שהם יהיו זמינים בפריים הנוכחי. נשים לב שהעברה צריכה להיות לפי כתובת כדי ששינויים שיתבצעו בפונקציה ישמרו. פעולה זו מומלץ לעשות במחסניות בעם אין אפשרות לגשת ישירות אל תוך המחסנית.

גישות שונות בשמירת רגיסטרים: ישנן שתי גישות מבחינת אחריות שמירת רגיסטרים בעת קריאה לפונקציה:

1. **אחריות הנקרא:** הפונקציה הנקראת, שומרת את מצב הרגיסטרים לפני הפעולות שלה ובסופן היא משחזרת אותם. לעיתים נתמך בחומרה. הערכים נשמרים אוטומטית מקריאה לקריאה.
2. **אחריות הקורא:** הפונקציה הקוראת שומרת את מצב הרגיסטרים שבשימוש לפני הקריאה, ולאחר סוף הקריאה משחזרת אותם. הערכים לא נשמרים אוטומטית מקריאה לקריאה.

כותב הקומפיילר מחליט באיזה גישה להשתמש. לעיתים משלבים את שתי הגישות.

מצגת 9 שקפים 34 עד 43
Activation Record – תרגול 10.

ניתוח תוכנית סטטי:

ניתן להסתכל על התוכנית בשלב ה AST או ב IR, ולבצע ניתוח שיאפשר לנו לדעת האם מסלול מסוים הוא בלתי נגיש, או האם ניתן לשחרר רגיסטר (כי המשתנה אינו חי) וכו'.

הערת ביניים: הבעיה האם תוכנית בהכרח יבחר מסלול מסוים היא לא כריעה, כיוון שאפשר לעשות דוקציה מבעיית העצירה.

כל השיטות הללו הינן Sound אבל incomplete. בכל זאת הן יכולות להיות די מדויקות.

שיטות אלו עוזרות לנו לעשות אופטימיזציות לקומפיילר ולתכנן כלים שיבדקו את איכות הקוד (זיהוי באגים פוטנציאליים, להוכיח שלא יכולים להתחרש שגיאות זמן ריצה ובדיקות נכונות חלקית).

דוגמא: אם בקוד יש לנו קריאה לפונקציה עם ארגומנט והפונקציה מחזירה ערך קבוע בהינתן הארגומנט הזה ללא תלות במצב התוכנית, אפשר להחליף את הקריאה לפונקציה בתוצאה שלה. דוגמה נוספת: אם יש לנו מצב של IF שלא יכול להיכשל, אפשר לוותר על המסלול של ה ELSE, ולהמשיך לנתח את התוכנית.

אתגרי הניתוח הסטטי:

- הניתוח לא טריוויאלי.
- הניתוח לא מדויק – בקריאות לפונקציות בתוכנית מונחת עצמים, ניתן להיכנס למעגלים ולא לדעת מהי ריצת התוכנית.
- יעילות – ניתוח כזה לוקח זמן ולעיתים צריך מספר מעברים גדול על הקוד כדי לצמצם את הקוד.

שפת C: שפת C תוכננה להיות אופטימאלית ושלא יהיה בה צורך לניתוח סטטי. זאת מכיוון שבשפה זו יש גישה ישירה לרגיסטרים ולזיכרון. ולכן קומפיילרים של C לא צריכים כ"כ לעשות ניתוח סטטי ואופטימיזציות. קומפיילרים מודרניים של C כן מבצעים ניתוח סטטי וכתוצאה מכך הם עובדים קשה יותר איך משיגים תוכנית קצרה ומשופרת יותר.

בניתוח סטטי משתמשים ב Software Quality Tools כדי לגלות סכנות בקוד. למשל שחרור זיכרון וניסיון לגשת אליו לאחר מכאן, גישות מחוץ לגבולות המערך ונזילות זיכרון.

Control Flow Graph: גרף זרימה המכיל נקודת כניסה ויציאה מוגדרת היטב. זהו גרף מכון שמייצג זרימה ויכול להכיל מעגלים. כל שורה בקוד היא צומת בגרף וכיווני הקשתות מתארות את זרימת התוכנית. בניתוח סטטי משתמשים ב CFG.

Constant Propagation: תהליך של החלפת משתנים שערכם ידוע בזמן הקומפילציה בקבועים וכתוצאה מכך ניתן להגיע לגרף CFG משופר. במעבר על גרף הזרימה לגבי כל משתנה, ננסה לקבוע את ערכו אם הערך ידוע.

Constant Flooding: טכניקה נוספת, שבמקרה שבו ביטוי מכיל רק ערכים קבועים ניתן בזמן קומפילציה לעשות את החישוב ולהחליף את הביטוי בקבוע התוצאה. ניתן לבצע את זה בשלב ה AST או לאחר מכאן.

"חיות" של משתנה: בדיקה נוספת שניתן לעשות בשלב הניתוח הסטטי היא לבדוק עד איזה שלב משתנה הוא חי, כדי לשחרר רגיסטרים וכדומה. משתנה מוגדר חי בנקודה מסוימת אם נעשה בו שימוש בהמשך התוכנית באחד ממסלולי הריצה. הקומפיילר מבצע את הניתוח הזה מלמטה למעלה.

ניתוח סטטי הוא כלי טוב אבל אינו מדויק מכיוון שהבעיות שהוא מנסה לפתור הן לא כריעות. ולא ניתן תמיד לחשב את כל המסלולים האפשריים של תוכנית. השאלה האם ניתן להגיע לנקודה מסוימת בתוכנית היא בלתי כריעה.

לצורך השוואה הבעיה האם משתנה חי בנקודה מסוימת היא לא כריעה, אך ניתן לנסות לגלות אם משתנה הוא חי או לא. לגבי משתנים שבשיטה שלנו נמצא שהם חיים התשובה תהיה נכונה תמיד, אבל אנו עלולים לטעות ולסמן משתנה חי כמת.

בגלל שהבעיות שניתוח סטטי מטפל בהן אינן כריעות, כלי בדיקת האיכות הם Sound מה שאומר שאסור להם לפספס שגיאה, אבל הם עלולים גם להודיע על שגיאות בטעות (False alarm).

חישוב איטרטיבי של מידע סטטי:

1. בנה CFG
2. התחל עם הערך האופטימאלי בכל צומת (לחישוב חיות מתחילים מלמטה).
3. שערך כל Statement בצורה שמרנית.
4. עצור כאשר אין שינויים.

הניתוח נעשה פונקציה אחת בכל פעם (קיימים פתרונות מדויקים יותר).

הקצאת רגיסטרים:

אנו מנסים למצוא כמה רגיסטרים קוד צריך. ראינו כבר את סטי אולמן שמנסה למזער לנו את השימוש ברגיסטרים. אבל עדיין הוא פועל תחת הנחה של אינסוף רגיסטרים ויש מצבים של שפיכה.

כעת נדון באלגוריתם לצביעת גרפים והשימוש בו להקצאת רגיסטרים. השלב עליו נדבר הינו השלב האחרון שבו אנו מקבלים תוכנית אסמבלר ומחזירים תרגום סופי לאסמבלי המתחשב במספר הרגיסטרים במכונה.

הבעיה מבחינתנו הינה: בהינתן n רגיסטרים, האם ניתן לצבוע גרף המקביל לתוכנית שלנו ב n צבעים? אם לא, אנו יודעים שנצטרך לבצע שפיכה (ובמקרה כזה עולה השאלה "אילו רגיסטרים כדאי לפנות?"). כמוכן שהבעיה היא NPC. לכן, נשתמש באלגוריתם יוריסטי הנותן פתרון מקורב.

יש מספר אלגוריתמים בנושא, אבל העיקרון של כולם דומה.

1. בונים גרף תלויות.
2. מנסים לצבוע אותו במספר הצבעים שיש לנו, אם הצלחנו סיימנו.
3. אם לא הצלחנו, נבצע שפיכה של משתנה לזיכרון ונסה שוב.

גרף התלויות – גרף אינו מכוון. הצמתים בגרף הם כל המשתנים החיים במהלך התוכנית (לכל משתנה חי יש צומת אחד גם אם הוא חי במספר מקומות שונים). נוסף קשת בין 2 צמתים אם התוכן של המשתנים שהם מציינים צריך להיות ברגיסטרים בו זמנית. הגדרה נוספת לקשה הינה: "2 משתנים נחשבים קשורים אם אחד מהם חי שהשני מוגדר".

האתגר נובע מכך שבעיית הצביעה היא קשה חישובית, מספר הרגיסטרים במכונה קטן, ואנו רוצים להימנע מהרבה פעולות MOVE. ניתן לפשט זאת ע"י צביעה מראש למשל של צמתי הרגיסטרים בצבעים שונים.

כדי לפתור את זה נשתמש במשפט שאומר שבגרף לא מכוון, אם נסיר ממנו צומת עם פחות מ K שכנים, לגרף החדש יש צביעה ב K צבעים.

האלגוריתם לצביעת הגרף ב K צבעים:

1. כל עוד יש קודקודים בגרף בצע:
2. הסר קודקוד עם דרגה קטנה מ K . והוסף אותו למחסנית
3. אם נגמרו הקודקודים אזי הגרף K צביע, ואפשר להוסיף קודקודים מהמחסנית לגרף ולצבוע אותם. אחרת אם נתקענו. נגדיל את K ונתחיל שוב.

אם אנו מוצאים שלא ניתן לצבוע את הגרף עם מספר הרגיסטרים שיש לנו עלינו לפשט את גרף התלויות ע"י שפיכה של אובייקט והסרת הקודקוד שלו מהגרף.

כיצד נחליט למי לבצע את השפיכה?

ההחלטה היא יוריסטית. ישנן 3 שיטות:

1. לפי הצומת עם הדרגה הגבוהה ביותר.
2. לפי המשתנה שחי הכי הרבה זמן (כנראה גרם להכי הרבה תלויות).
3. לפי דירוג spill priority המחושב לפי מספר השימושים מחוץ לזולאה ועוד 10 כפול מספר השימושים בתוך זולאה. כל זה חלקי דרגת הצומת. נשפוך קודם את מי שהדירוג שלו יותר נמוך. ההיגיון הוא שככל שמתמשים במשתנה יותר, חשוב יותר שיהיה ברגיסטר. בנוסף, ההנחה היא שזולאה רצה כמה פעמים לכן מקבלת יותר ניקוד.

לאחר שהחלטנו על "שפיכה" של משתנה, עלינו להוסיף לאחר כל חישוב שלו פקודה המעבירה את התוכן שלו לזיכרון.

לאחר הרצת האלגוריתם אולי נקבל קוד עם שטויות כמו Move R1 R1, ולכן נוכל למחוק שורות מיותרות ונקבל קוד מצומצם יותר.

:Assembler/Linker/Loader

זהו השלב שבו לוקחים את האסמבלי ומקמפלים אותו באסמבלר לשפת מכונה (אפסים ואחדות).

תוכנית באסמבלי בנוי ב 4 חלקים :

1. Code segment – הקוד בשפת אסמבלי לא משתנה במהלך התוכנית.
2. Data segment – מתחיל ממצב התחלתי ומשתנה במהלך הריצה.
3. Stack – מחסנית זמן הריצה.
4. Registers.

בדר"כ Data segment עובד כמו Heap וה Code segment כמו Stack.

Loader הוא חלק ממערכת ההפעלה המבצע טעינה של Code segment לזיכרון (רצוי בלי לחלק אותו אך לא תמיד אפשרי), אנו מקבלים כמובן מאליו שהתוכנית נמצאת כגוש אחד בזיכרון.

הוא גם טוען את ה Data segment לזיכרון ומוסיף למבנה כתובת התחלה שכל הגישות הרלטיביות יעבדו לפיה.

Linker מבצע חיבור בין מספר סגמנטים של קוד (Include למיניהם), ה Linker צריך לקחת את כל החלקים וליצור תוכנית אחת עם Code/Data Segment אחד, מחסנית אחת, וטבלה שנקראת external table המכילה כתובות אמיתיות של המשתנה ב Data segment ונתונים על מיקום פונקציות בזיכרון. הוא צריך כמובן לעדכן את כל הכתובות בקוד. הוא עושה זאת בשני מעברים על האסמבלי. במעבר הראשון הוא מסתכל לאן קפצו ולאילו פונקציה, כדי לזכור את הכתובת האמיתית שלה. במעבר השני הוא מעדכן את הכתובות של הקריאות.