# סיכומים למבחן בקומפילציה

## סמסטר א' תש"ע 2010 (ד"ר רינה צביאל-גירשין)

### מושגים בסיסיים:

**מהדר (compiler) מול מפרש (interpreter):**

| | Interpreter | Compiler |
|---|---|---|
| קלט | • תוכנית<br>• קלט עליו צריך להריץ את התוכנית | • תוכנית בשפת המקור בלבד |
| פלט | • פלט הפעלת התוכנית על הקלט שהתקבל. לא נוצר קובץ. בדיקת שגיאות תו"כ.<br>• כל הפעלה עושה הכל מהתחלה. | • מתרגם את התוכנית לקובץ executable בשפת מכונה או byte-code. תרגום משפת high-level לשפת מכונה.<br>• קובץ התוצר הוא תוכנית שיכולה לקבל קלט ולהוציא פלט ללא תרגום חדש כמו מפרש. |

מהדר ומפרש כתובים בשפת ביניים או השפה אותה מתרגמים. ה-Compiler וה-interpreter מבצעים בערך אותו דבר, רק שמהדר מבצע פעולות נוספות. מהדר מול מפרש:

- Compiler: $source\ code(txt) \rightarrow \boxed{\underset{analysis}{frontend} \rightarrow Semantic\ Representation \rightarrow \underset{synthesis}{Backend}} \rightarrow executable\ (exe)$
- Interpreter: $\left.\begin{matrix} source\ code(txt) \\ input \end{matrix}\right\} \rightarrow \boxed{\underset{analysis}{frontend} \rightarrow Semantic\ Representation \rightarrow interpretation} \rightarrow output$

בחלק הראשון (frontend), הם זהים. בחלק השני המפרש יוצר וייצוג סמנטי לתוכנית הרצה על הקלט. לבסוף המפרש מוציא פלט התוכנית על הקלט והמהדר מוציא קובץ בר-הרצה של תוכנית המקור.

| Compiler | Interpreter |
|---|---|
| דיווח שגיאות לפני שרואה את הקלט<br>יעיל יותר עבור תוכנית שאמורה לרוץ הרבה: מבצע front/backend פעם אחת בלבד. לעתים זמן קומפי+הרצה קטן מזמן פירוש | קל לפיתוח (אין התעסקות עם ה-backend)<br>ניידות: לא נוצרת תוכנית בשפת מכונה, ניתן להרצה בכל מכונה<br>דוח שגיאות מדוייק על הרצת תוכנית על קלט ספציפי<br>עבור הרצות בודדות – מהיר מהמהדר (לא בטוח עבור הרצות רבות)<br>מאובטח יותר (תוצר exe של המהדר עלול להכיל וירוס פוגעני) |

**כלים בשימוש:**

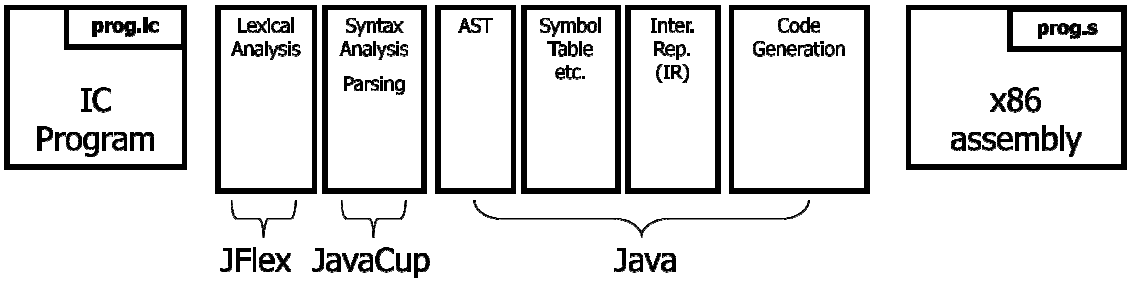
- Flex: כלי המקבל regular expressions ומייצר תוכנית scanner: lexical analysis – הבודק חוקיות קלטים לפי אותם ביטויים רגולרים. פלט התוכנית הוא זרם מילים – tokens.

**Abstract Syntax Tree – AST:**

יצירת עץ גזירה הוא השלב האחרון ב-frontend של יצירת קומפיילר. ייצוג סימבולי של התוכנית המשמר את ההיררכיה הפנימית בתוכנית. ה-AST מיוצר ע"י ה-parser: syntax analysis (המנתח התחבירי). בשלב זה מילות מפתח וסימני פיסוק מסולקים.

שלבי יצירת תוכנית: ניתוח לקסיקלי, ניתוח תחבירי, ניתוח סמנטי (context analysis) ← AST (intermediate code) ← יצירת קוד מכונה / פירוש.

**שלבי בניית IC Compiler:**



## Lexical Analysis: ניתוח לקסיקלי:

שלבי הקומפיילר הבסיסיים:

$$source\ code \underset{string}{\rightarrow} \boxed{Frontend \underset{\textbf{lexical analysis}}{} \xrightarrow{tokens} Syntax\ analysis \xrightarrow{AST} Semantic\ analysis} \xrightarrow{Annotated\ AST} Backend \rightarrow Assembly\ (exe)$$

Tokens: היחידה הבסיסית ביותר בפירוש קוד המקור. כל אסימון הוא מילה המייצגת איבר בסיסי, כגון ID עבור שמות משתנים, COMMA עבור פסיק, IF עבור "if", NUMBER עבור מספרים וכן הלאה. כל token יכול להיות בעל ערך (כמו שם ל-ID) או ללא ערך (כמו מילה שמורה if).

**ניתוח תחבירי**:

המנתח התחבירי מקבל כקלט את קוד המקור ומוציא כפלט רשימת tokens. ה-tokens מוגדרים ל-scanner (lexical analyzer) באמצעות regular expressions ע"פ הגדרות שפת המקור.

בעיות: בזיהוי token יכולה להיות עמימות מדו-משמעות. פתרון: לוקחים את ההתאמה הארוכה ביותר, ואם יש כמה התאמות באותו אורך, לוקחים את הראשונה מביניהם לפי סדר הופעת הכללים (בקובץ ה-lex).

**ניתוח Dotted Items**:

בניתוח הלקסיקלי מתייחסים לאובייקט הנקודה (.) כאובייקט המסמן את מיקום ה-scanner. Items שונים נקבעים לפי מיקום הנקודה:

- Shift items: הנקודה נמצאת לפי תבנית בסיסית, למשל: $A \rightarrow (ab)+.c$
- Reduce items: הנקודה נמצאת בסוף, למשל: $A \rightarrow (ab)+c.$

הזזות נקודה יכולה להתבצע בכמה אופנים:

- תזוזת תו (char move): מעבר פשוט של הנקודה אל מעבר לתו, למשל: $A \rightarrow a.bc \overset{b}{\Rightarrow} A \rightarrow ab.c$
- תזוזת $\varepsilon$: התקדמות עבור חזרות או בחירות מרובות, למשל:
  - $A \rightarrow a.(b)^*c$: אם אין הופעות b, יעבור ל-$A \rightarrow a(b)^*.c$. אם יש: $A \rightarrow a(.b)^*c$. הנקודה תעבור לסוף b (בתוך הסוגריים) וחזרה לתחילתו כך עד שכל ה-bים יכוסו, ואז תצא אל מחוץ ל-$(b)^*$.
  - $A \rightarrow a.(R_1|R_2)b$: יכול לעבור ל-$A \rightarrow a(.R_1|R_2)b$ או ל-$A \rightarrow a(R_1|.R_2)b$.

ה-scanner הוא למעשה DFA, ובנייתו נעשית ע"י בניית NDFA כדי לטפל בכל המקרים האפשריים השונים והפיכתו ל-DFA. שלבי בניית NDFA:

- לריבוי מעברים נוספת הגזירה $S \rightarrow T_1| \dots |T_n$, המצב ההתחלתי הוא: $S \rightarrow .(T_1| \dots |T_n)$
- לכל מעבר תו נוסיף מצב ומעבר על אותו תו באוטומט, למשל: $A \rightarrow .ab$ - נוסיף מעבר לפי $'a'$ ל-$A \rightarrow a.b$
- הוספת מעברי $\varepsilon$
- מצבים מקבלים: Reduce items (בהם הנקודה בסוף)

במעבר ל-DFA כל מצב הוא רשימה של items. כאשר במצב יש שני reduce items, כאמור בוחרים את זה שהחוק שלו הופיע ראשון.

מנתח לקסיקלי ל-IC:

ה-scanner שבנינו ל-IC מפרק את התוכנית ל-tokens ומוציא אותם כרשימה עם השורה בה מופיע כל token. תהליך העבודה:

$IC.lex \rightarrow JFlex \xrightarrow[creates]{} Lexer.java \rightarrow javac \xrightarrow[compiled]{} Lexical\ analyzer$- התוצר הוא שמקבל IC source code ומוציא tokens.

מבנה קובץ lex:

לקובץ ה-lex שלושה חלקים המופרדים ע"י "%%" ביניהם:

- קוד java של המשתמש, מועבר איך שהוא לקובץ ה-Lexer.java
- הגדרות macros ושמות מצבים (כמו $DIGIT = [0-9]$ או $YYINITIAL$)
- חוקים לקסיקלים: מורכב מחלק ראשון שהוא הביטוי הרגולרי והחלק השני שהוא הפעולה שיש לעשות (ב-java) במידה וזוהה.

## Syntax Analysis: ניתוח תחבירי:

- המנתח התחבירי מקבל זרם tokens ויוצר AST.
- בעת היצירה הוא בודק שגיאות תחביריות בלבד, ושגיאות סמנטיות (כמו ניסיון השמת מחרוזת למשתנה int) יבדקו בשלב מאוחר יותר. בגילוי השגיאה המנתח יכול להמשיך את הסריקה (ואולי לגלות שגיאות נוספות). המשך סריקה יכול להיעשות ע"י השלמת סימנים חסרים או דילוג על tokens בעייתיים.

**שימוש שב-CFG (דקדוק חסר הקשר):**

CFG משתמש ב-terminals שהם ה-tokens ו-non-terminals שהם חוקי הגזירה: $non\ terminal \rightarrow Symbol\ Symbol\ \dots Symbol$. בדיקה האם משפט הוא בדקדוק: מתחילים אם symbol ההתחלה ומחליפים לפי כללי הגזירה את ה-non-terminals עד קבלת משפט terminals בלבד. הגזירה יכולה להיות **ימנית** או **שמאלית**.

- תפיסה טובה של מבנה התוכנית.
- מאפשר יצירת מפסק יעיל:
  - משתמש באוטומט מחסנית דטרמיניסטי.
  - טיפול בשגיאות: בקבוצת דקדוקי LL טיפול בשגיאות פועלת מיד כאשר מתקבל סימן לא צפוי. שיטה לא מתאימה עבור דקדוק רב משמעי (טוב עבור דקדוקי LL).

**סוגי מפסקים:**

- Top down (LL): בניית העץ למעלה למטה, מפסק תחזית, סריקת pre-order; המפסק שמומש בפרוייקט הוא מפסק LL – סריקה משמאל לימין וגזירה משמאל. בשיטה זו לכל non-terminal מסתכלים על ההתחלה (רישא) ומוחלט איזה כלל גזירה מתאים. לכן אסורים שני כללים בעלי אותו רישא. בשיטה זו מתחילים מסימבול ההתחלה ומנסים לגזור אותו עד הגעה לקלט.
- Bottom up (LR): בניית העץ מלמטה למעלה, סריקת post-order. מפסק LR – סריקה משמאל וגזירה מימין. מתחילים מהתחתית ומטפסים מעלה עד הגעה לשורש. ההחלטה מתבצעת לפי ה-token בקלט, תוכן אוטומט המחסנית והכלל בו נמצאים. שיטה זו נמצאת בשימוש נרחב: LR(0), SLR(1), LR(1), LALR(1). בשיטה זו מתחילים מהקלט ומנסים לשכתבו עד לסימבול ההתחלה.

הערה: תוצר של ה-parser הוא parse-tree (עץ גזירה), ואם הריצה נכונה ומתאימה לדקדוק, נוצר AST (ב-parse tree הצמתים הם non-terminals והעלים הם terminals, ב-AST אין non-terminals – כל הצמתים והעלים הם terminals).

דקדוק רב משמעי: דקדוק הוא רב משמעי אם קיימת מחרוזת קלט עבורה יש שתי גזירות אפשריות (או יותר).

**מנתח תחבירי ל-IC:**

החוקים התחביריים של השפה מכתיבים CFG הנכנס ל-CUP וכך מיוצר Parser מסוג LALR(1). ה-parser מקבל tokens ומוציא AST. חלקים:

- החלק הראשון הוא חלק שנכנס לקוד ה-java שה-CUP מייצר (Parser.java).
- הגדרת terminals ו-non-terminals.
- הגדרת קדימויות (precedence) בסדר עולה (הקדימות הכתובה בסוף היא הקדימות החזקה ביותר).
- הגדרת חוקי גזירה.

המשתמש יכול להגדיר קדימות למקרים מיוחדים, למשל קדימות עבור "-" כסימן שלילה: ברשימת הטרמינלים יתווסף terminal UMINUS – לא מוחזר ע"י ה-scanner, נועד רק לקדימות, בקדימויות יתווסף precedence UMINUS כאחרון ברשימה (ראשון בקדימות), ובחוקי הגזירה של expr יתווסף MINUS expr %prev UMINUS.

הבנייה של bottom-up parser מתבצעת ע"י שמירת כל צומת ב-push-down stack (שמירת RESULT). כאשר צומת נבנה, כל ילדיו כבר בנויים.

הפגת עמימות:

- בקונפליקט shift/reduce משווים קדימויות של טרמינל (יביא ל-reduce) או פרודוקציה (שלב גזירה נוסף), ובהתאם פועלים.
- במקרה של קדימות שווה, החלטה תיעשה לפי קדימות left/right: left – יתבצע reduce; right – יתבצע shift.

**Bottom-up Analysis ומודל אוטומט המחסנית:**

- הקלט הוא סימנים ו-CFG, והפלט הוא parse tree אם הקלט תקין או הודעת שגיאה. כל רמה נבנית לאחר שהרמה מתחתיה נבנתה.
- Shift – משמעו התקדמות בקלט, reduce – משמעו לא להתקדם אלא לעשות משהו.

- מחסנית מחזיקה סימן $ המסמן את סוף הקלט, ובהתחלה תוכנה הוא טרמינלים ומשתנים דקדוקיים (בהמשך תכיל גם מצבים). הקלט יהיה זרם tokens המסתיים ב-$. קלט בשפה אם כאשר מגיעים בו ל-$ גם במחסנית מגיעים ל-$.
- טבלת המעברים (LR(0)):

| עמודת מעברי $\varepsilon$ – עבור לא טרמינלים בלבד. | עמודה לכל terminal, non-terminal | | | | פריטי הדקדוק – כל האפשרויות של כללי הדקדוק עם מיקומי הנקודה האפשריים |
|---|---|---|---|---|---|
| $\varepsilon$ | t \|\| n/t | t \|\| n/t | ... | t \|\| n/t | LR(0) items |

- בעמודת ה-items: לכל כלל בדקדוק יופיעו כל ה-items הקשורים אליו, למשל $S \to E\$$: $S \to .E\$, S \to E.\$, S \to E\$.$.

צריך להוסיף לדקדוק כלל $S' \to S\$$ ולהוסיף עבורו את ה-items המתאימים.

- עמודת הסימנים ומעברי ה-$\varepsilon$:
  - עבור פריט מהצורה $S \to .E\$$ (הנקודה לפני n/t): מספרי שורות למעבר יופיעו תחת כל עמודה אפשרית, למשל אם בשורה ה-2 מופיע $S \to E.\$$, אז תחת עמודת הסימן "E" יופיע 2 שמסמל מעבר למצב בשורה 2; בנוסף כל שורה של פריט מהצורה $E \to .< something >$, כלומר כלל גזירה ל-E, תופיע תחת מעבר ה-$\varepsilon$. תתכן רקורסיה.
  - עבור פריט מהצורה $S \to .i$ (הנקודה לפני t): כללי shift יופיעו בצורה s4 – ביצוע shift (מעבר לתו הבא) ומעבר למצב בשורה 4. לטרמינלים יהיה תמיד מעבר אחד אפשרי (לעומת n/t שיכולים להיות עם כמה).

עבור פריט סיום מהצורה $S \to < something >.$ יהיה סימן יחיד "r" תחת עמודת ה-$\varepsilon$ שמשמעו reduce.

**במקרה של קונפילקט**: הדקדוק אינו LR(0).

- Shift-reduce conflict: כאשר יש שני כללי גזירה בעלי אותו prefix, למשל: $T \to \alpha\$, T \to \alpha\beta\$$.
- Reduce-reduce conflict: כאשר יש שני כללי גזירה זהים הגוזרים שונה, למשל: $T \to \alpha, S \to \alpha$.

Control table: דומה לטבלה לעיל: במקום עמודת ה-items יש עמודת מצבים (המתאימים לאוטומט), ותחת כל תו (terminal או non-terminal) יש פקודת "s<state #>" או תחת כל השורה יש כלל יחיד מהצורה "r<reduce rule>", כאשר אם הכלל הוא "$rS' \to S\$$" זהו מצב מקבל (נכתוב בסוגריים "accept").

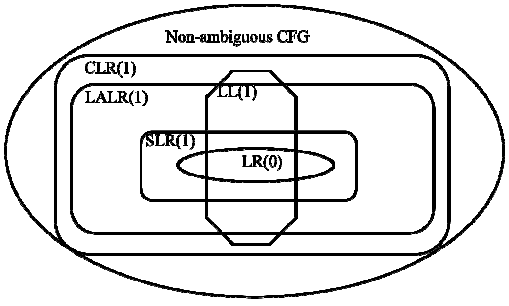
**דקדוק SLR(1)**:

בדקדוק זה פעולות reduce לא חייבות להופיע על כל השורה, אלא תחת תו מסויים בלבד. כך יתכנו גם shift וגם reduce מאותו מצב והקונפליקט נפתר. כדי לקבוע את הכניסות בטבלה עבור SLR(1) יש צורך להסתכל ב-Follow sets של הכללים: $Follow(A) = \{t | S \Rightarrow^* \beta A t \gamma\}$ - כלומר כל התווים שיכולים להופיע אחרי A.

**דקדוק LR(1) (או Canonical LR(1) - CLR(1))**:

- הפריטים בדקדוק הם מהצורה: $A \to \alpha.\beta\{t\}$, כאשר במצב זה $\alpha$ הוא בראש המחסנית ומצופה לראות $\beta t$ בהמשך.
- שימוש ב-look ahead של תו אחד קדימה, שהוא t בדוגמה לעיל (מכאן המספר 1 – מסתכלים צעד אחד קדימה).
- השפעת ה-look ahead היא רק על פריטים מהצורה $A \to \alpha.\{t\}$ או $A \to \alpha.\beta\{t\}$ כאשר $\varepsilon \in \beta$ – מתבצע reduce רק אם התו הבא בקלט הוא t.
- מצב באוטומט LR(1) הוא רשימת פריטים מהצורה לעיל.
- דקדוק הוא ב-LR(1) אם:
  - אין קונפליקט shift-reduce: לכל פריט מהצורה $A \to \alpha.x\beta\{t\}$ במצב מסויים אין באותו מצב פריטים מהצורה $A \to \alpha.\{x\}$.
  - אין קונפליקט reduce-reduce: בכל מצב אין שני פריטים מהצורה $A \to \alpha.\{t\}, B \to \beta.\{t\}$.

**דקדוק LALR(1)**:




מבצעים איחוד מצבים זהים ב-LR(1) תוך התעלמות מה-look ahead כאשר מאחדים (כל ה-look aheads נכנסים יחד ל-{}). מעבר מ-LR(1):

- איחוד כל המצבים בהם יש אותם פריטים עד כדי look ahead.
- פריטים מהצורה $A \to \alpha.\{t\}, A \to \alpha.\{s\}$ יאוחדו לפריט מהצורה $A \to \alpha.\{t, s\}$.

החלטת סוג דקדוק:

- כדי שדקדוק יהיה LR(0) הוא חייב להיות חד משמעי, אחרת אינו דקדוק LR ובפרט לא LR(0).
- עבור כל מצב מקבל (reduce) אסור שיהיו 2 כללי reduce או קשת יוצאת – shift-reduce.
- כדי להפוך דקדוק רב משמעי ל-LR(0) ניתן להוסיף non-terminals, ואז לבדוק שוב. יתכן שאינו LR(0) אלא SLR(1) – כאשר קונפליקטים נפתרים ע"י הסתכלות על ה-follow במקומות הבעייתיים.
- יתכן שהדקדוק אינו SLR(1) אלא LR(1) (ואולי LALR(1)) – קונפליקט שנוצר כאשר follow זהה שאינו פותר את הקונפליקט שאמור לפתור.

## Semantic Analysis: ניתוח סמנטי:

**פעולות ובדיקות הנעשות בעת ניתוח סמנטי:**

- נקבע הטיפוס לכל ביטוי ונעשית בדיקת semantic type (לרוב נעשה בשני מעברים נפרדים על ה-AST – אחד לקביעת הטיפוסים והשני לבדיקתם). טיפוסים יכולים להיות מוגדרים מראש במערכת או מוגדרים ע"י המתכנת.
- Scope rules: תחזוקת טבלת סמלים – symbol table עבור כל scope, כאשר כל טבלאות הסמלים מוחזקות בעץ. שורש העץ הוא טבלת הסמלים הגלובלית (מכיל מחלקות), תחתיו טבלאות מחלקה =לפי יחסי ירושה בין מחלקות. תחת כל מחלקה יהיו טבלאות מתודות של אותה מחלקה, ומתחת לזה טבלאות לבלוקים תחת כל מתודה (טבלאות מקוננות).
- העמסת מתודות.
- המרת טיפוסים כולל המרה מרומזת, למשל השמת ערך int לתוך float. הרחבה (כמו דוגמא זו) חוקית, צמצום (ניסיון השמה הפוך) לא חוקי. בדיקות המרת טיפוסים נעשית ע"י סכמה מונחית תחביר.
- בדיקת חוקים ספציפים, כמו למשל הופעה יחידה של מתודת main.

**סוגי בדיקות:**

- סטטיות: נעשות בזמן קומפילציה בשלב הניתוח הסמנטי. מאפשרות תיקון קוד לפני הרצה, מהיר יותר מאשר כשיש בדיקות זמן ריצה.
- דינמיות: בעת ריצת התוכנית. בשפות OO יש צורך בבדיקות זמן ריצה של טיפוסים בגלל פולימורפיזם של טיפוסים. בדיקות אלו מאפשרות בניגוד לסטטיות בדיקת טיפוסים שלא מוגדרים מראש.

## Interpreters:

**סוגי מפרשים – interpreters:**

מפרשים רקורסיביים:

- מעבר רקורסיבי על העץ. מפרשים אלו עובדים לאט, עד פי 1000 איטי יותר מקומפיילר.
- מפרשים מתמודדים עם טיפוסים המוגדרים בזמן ריצה ע"י החזקת שני שדות לכל טיפוס – type ו-size. למפרש יש status indicator שיכול להכיל ערכים: normal mode, errors, Jumps, exceptions ו-return. כך המפרש מחזיק בכל רגע נתון היכן נמצא בתוכנית.

Partial evaluation: שערוך חלקי:

- שערוך חלקי יכול לפשט את התוכנית המקורית, ולרוב יעשה על החלקים הסטטיים (בד"כ חישובים מתמטיים; דינאמיים הם לרוב לולאות).
- תוצר המשערך החלקי הוא תוכנית פשוטה יותר הרצה על קלט אחר/חלקי למקורי ומוציא את תוצאת החישוב.
- דוגמאות פישוט: לפשט חישוב רקורסיבי לכדי חישוב איטרטיבי, הורדת בדיקת מקרי switch שאינם רלוונטיים לקלט. שערוך חלקי יכול לשפר את התוכנית עד פי 30-40 גרוע מקומפיילר (במקום 1000).

מפרשים איטרטיביים:

- איטי פי 30 מקומפיילר; מעבר למבנה שטוח במקום עץ. במשערך איטרטיבי כל צומת ב-AST ישמור את הצומת הבא שיש לשערך: **control flow**.
- השערוך נעשה ע"י מבנה עזר מחסנית זמן ריצה: המחסנית תחזיק רצף סדרתי של חישוב הצמתים. לאחר החישוב רצה לולאה על רשימת הפעולות שנוצרה במחסנית.
  - יתרון: מעבר רקורסיבי אחד על ה-AST, לאחר מכן כל מעבר הוא איטרטיבי (מהיר יותר).
  - חסרון: בעייתי בלולאות.
- למפרשים איטרטיביים יש ייצוגים שונים (תרשים זרימה, מערך וכו').

## Code Generation:

עד כאן היה החלק האחורי של הקומפיילר. משלב זה ניתן להמשיך לבניית מפרש או מהדר. החלק הקדמי של הקומפיילר:

- תרגום מה-AST לשפת מכונה.
- בעיות: כיצד לחלק את העץ לפקודות, חלוקת רגיסטרים יעילה, סדר הפעולות הסופי – בעיית תרגום יעיל ביותר היא NP-שלמה.
- לרוב יעשו מעברים נוספים לאחר בניית הקוד לצורך אופטימיזציות: צמצום רגיסטרים מיותרים, הסרת פקודות מיותרות (כמו הכפלה ב-1) וכו'. זמן הקומפילציה מתארך אך התוכנית מתייעלת.

מודלים:

- Simple stack machine: מודל מכונה המכילה רק מחסנית עם מצביעים לראש המחסנית ולבסיסה.
- Register machine: מודל בו נתונים במכונה אוגרים, פקודות load, store (רגיסטרים / זיכרון) ופעולות אריתמטיות הניתנות לביצוע על אוגרים.

**אופטימיזציית צמצום מספר רגיסטרים – אלגוריתם Setti-Ullman:**

שני שלבים עיקריים לאלגוריתם:

- מעבר ראשון על העץ מלמטה למעלה וספירת הרגיסטרים הנצרכים ע"י כל צומת באופן הבא; יהי m מס' רגיסטרים לתת עץ ימני ו-n לשמאלי:
  - אם $m > n$ מספר הרגיסטרים הנצרך הוא m.
  - אם $n > m$ מספר הרגיסטרים הנצרך הוא n.
  - אם $n = m$ מספר הרגיסטרים הנצרך הוא $m + 1 = n + 1$.
- במעבר שני מתבצעות החלטות איזה תת עץ לתרגם קודם, והכבד מבין כולם הוא שיתורגם ראשון. הסיבה: את תוצאת החישוב שלו ניתן לשמור באחד הרגיסטרים שמשתמש לחישוב, ושאר הרגיסטרים נהיים פנויים ומספיקים עבור חישוב תת העץ הקטן יותר (מבחינת צריכת רגיסטרים).

**Spilling: שמירה בזיכרון:**

- כאשר מספר הרגיסטרים בו משתמשים חורג מהנתון, יש צורך לשמור ערכים בזיכרון.
- בעיית בחירת אילו רגיסטרים לשמור בזיכרון, כלומר לאילו רגיסטרים נזדקק שוב היא בעיה NP-שלמה, אך קיימים פתרונות היוריסטיים.

פתרון למשל:

- מחפשים תת עץ "כבדים" (דורשים יותר רגיסטרים משיש למכונה).
- בתוכם מחפשים תתי עץ "קלים" (משתמשים בכמות רגיסטרים שקיימת בפועל). על תת עץ זה יתבצע חישוב ותוצאתו תוכנס לזיכרון. הכתובת בה נשמרה התוצאה תחליף את כל תת העץ הקל.

הבעיה דומה לבעיית graph coloring: צביעת גרף במספר מינימלי של צבעים כך שאין שני צמתים צמודים בעלי אותו צבע.

## Code generation for basic blocks:

קוד המכונה נוצר תחילה בבלוקים בסיסיים. קלט הבעיה הוא AST ותיאור המכונה כאוסף פקודות, מספר רגיסטרים ועלות כל פעולה. המודל שנסתכל עליו הוא מכונה עם גישה לרגיסטרים וזיכרון, אך מוגבלת מבחינת הפעולות. שלבים:

- בלוק בסיסי: חלק בגרף ללא פיצולים, היכול להתבצע באופן סדרתי. נרצה למצוא ב-AST את בלוק בסיסי מקסימלי. בלוק הוא קטע קוד שלא מכיל קריאות לפונקציות או לולאות.
- הגדרת גרף תלויות ושינויים בגרף: על בסיס ה-AST, המאפשר הקצאה כמעט אופטימלית של רגיסטרים. תחילה ממירים את ה-AST ל-threaded AST, כלומר AST המכיל בכל צומת את הצומת הבא ב-flow התוכנית. הקומפיילר יכול לשנות את סדר ביצוע הפעולות עבור חלק מהקוד (חלק חייב להישאר בסדר המקורי). יתקבל גרף ללא מעגלים. תלויות:
  - Expressions: אופרנדים תלויים באופרטור שלהם, השמה תלויה בערך המושם.
  - Statements: השמות צריכות לבוא לפני השימוש בהן.

  לאחר יצירת הגרף, תתכן יותר מנקודת כניסה אחת, לא משנה באיזו מתחילים כל עוד התלויות נשמרות. ניתן לבצע אופטימיזציה נוספת: צמצום ע"י הסרת צמתים לא נגישים (קוד מת).
- יצירת קוד מגרף התלויות: לאחר בחירת שורש התחלה (כאמור יתכנו כמה) צריך למפות רגיסטרים סימבוליים לפיזיים תוך שימוש במס' מינימלי. לכך יש פתרונות היוריסטיים. מציאת מספר הרגיסטרים זהה למציאת צביעת גרף לא מכוון בו הצמתים הם רגיסטרים סימבוליים ואמיתיים והקשתות הם קישור בין סימבולי לאמיתי. לאחר היצירה יתכנו פעולות מיותרות (כמו השמה מרגיסטר אחד לשני וחזרה מהשני לראשון) שניתן להפטר מהן.

<u>אלגוריתם היוריסטי למציאת צביעת גרף</u>:

- מסתכלים על דרגת כל צומת בגרף, ומוציאים צמתים מהגרף לפי דרגתם (מהקטן לגדול).
- בכל הוצאה מעדכנים את דרגות הצמתים ושמים את הצומת שהוצא בראש המחסנית.
- מסתכלים על ראש המחסנית ובודקים האם קיים צבע שהשתמשנו בו וצומת זה לא מחובר לצומת שנצבע בצבע זה. אם מצאנו, נצבע אותו בצבע זה, אחרת נצבע אותו בצבע חדש.

## Activation Records:

**Nested routines in C**: שגרות מקוננות (ב-java יש גם inner classes): ב-C יכולות להיות שגרות מקוננות, אך הבעיה שיש להתגבר עליה היא שפונ' יכולה להשתמש בכל המשתנים שרואה עד עכשיו – גם מהפונקציות שמעליה (בתוכן היא מקוננת). כאשר משתמשים במחסנית זו בעיה – כדי להגיע לערך שיושב "גבוה" צריך לעשות הרבה pops ולחזור למיקום הנוכחי הרבה pushes חזרה. זה מהווה בעיה לקומפיילר.

**Non-local goto in C**: קפיצה (goto) לתווית שלא מוגדרת בפונ' הנוכחית: הקפיצה יכולה להיות לקוד שטרם עברנו בו, כלומר לא במחסנית זמן הריצה, אלא במקום עתידי. זה לא כמו קריאה רגילה לשגרה אלא קפיצה למקום שאולי הוא באמצע פונ' – וצריך לדעת שם כל מה שהוגדר באותה פונ' קודם לתווית שאליה קפצנו. מבחינת קומפילציה זהו מבנה מסובך.

**Setjmp ו-longjmp ב-C (בחבילת <setjmp.h>)**:

- Setjmp: זוכרת את הסביבה בה נמצאים כרגע: מצב רגיסטרים, שורה בקוד, שגרה בה נמצאים, מבנה מחסנית זמן הריצה והכל.
- Longjmp: קופצים למקום שהועבר לנו כפרמטר ב-longjmp. כתוצאה מהקפיצה בדרך כל הרשומות המיותרות במחסנית זמן הריצה עפות (pop).

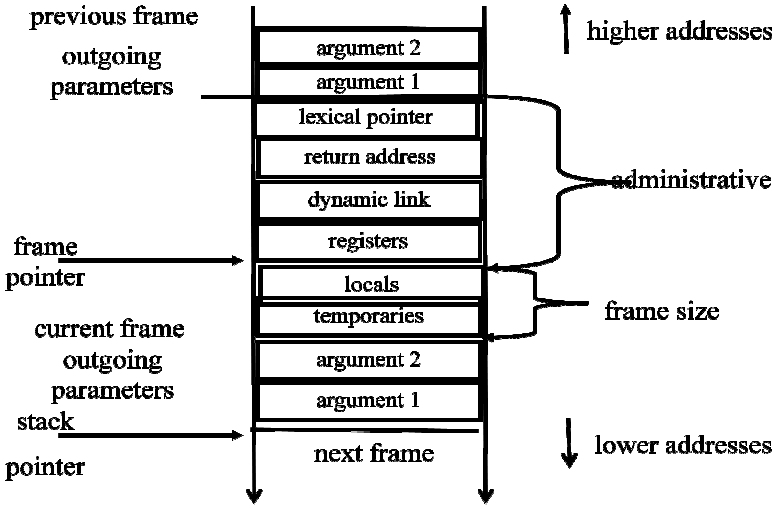
**Non local transfer of control in C**: שימוש בטיפוס מסוג jmp_buf יהיה בצורה הבאה: בקריאה למתודה ישלח מצביע ל-jmp_buf ע"י setjmp, ומתוך המתודה שנקראה, קריאה ל-longjmp על אותו טיפוס מסוג jmp_buf יחזיר אותנו לפקודה הבאה במתודה שממנה נקראה מתודה זו. ה-jump_buf עוזר לנו לזכור את המצב בו היינו, וכך המתודה שאליה קראנו יכולה לחזור לאותו מצב. ערך הפונקציה חוזר ע"י הקריאה מהצורה: return_value = setjmp(jmpbuf) (בודקים האם הערך הזה הוא 0 כתנאי להצלחת setjmp, ולא סביר שלא נצליח אלא במקרים קיצוניים כמו out of memory).

**Passing a function as parameter**: ב-C ניתן לשלוח פונקציות בארגומנטים, למשל: void foo(void (*interrupt_handler)(void)) - מאפשר קביעת איזו פונ' רוצים להפעיל בזמן ריצה. פעולה זו מורכבת מבחינת קומפילציה.

**Currying in C syntax**: העברת פרמטרים חלקיים לפונקציה: במקרה שיש לנו פונקציות מקוננות, ניתן להגדיר פונקציה שהיא הפעלת הפונקציה העליונה עם פרמטרים חלקיים. כך, אם יש לי פונקציה f המקבלת x ובתוכה מוגדרת פונקציה g המקבלת y ו-f מחזירה את ערך g (שמשתמש ב-x וב-y), ניתן להגדיר int (*)() f(int x) ואז אוכל להגדיר למשל int (*h)()=f(3), ואז הגדרנו את h להיות בעצם מה ש-g עושה על 3 ועל y – קבענו לה חלק מהערכים. מבחינת קומפיילר הבעיתיות היא בזיכרון הערכים החלקיים.

**Compile-Time information on variables**: משתמש מקומי דורש החזקת: שם, סוג, scope, duration, size (כמה בתים ידרוש בזמן ריצה), address – יכול להיות fixed, כלומר כתובת קבועה, כתובת יחסית לכתובת מסויימת או dynamic הדורש חישוב כתובת יותר מורכב.

**מבנה מחסנית זמן הריצה**:
בטיפול במחסנית מתחילים מכתובת גבוהה ויורדים מטה לכתובות נמוכות. דוגמא:




<u>החלק האדמיניסטרטיבי</u>: קבוע בגודלו לכל שגרה

- רשומת הפעלה מתחילה עם ארגומנטים שהשגרה צריכה לקבל, הנכתבים למחסנית ע"י ה-caller (פרמטרים ששלחה).
- Lexical pointer: מחזיר אותי לרשומת ההפעלה של ה-caller, מצביע לנתונים של הפונקציה שקראה לי במחסנית זמן הריצה.
- Return address: הכתובת אליה צריכה השגרה לחזור בתום פעולתה, הפקודה הבאה בפונקציה ממנה נקראנו.
- Dynamic link: מצביע לתחילת הפריים הקודם.
- Registers: שמירת מצבי הרגיסטרים לפני שמתחילים לבצע את השגרה הנוכחית. בסוף הפונקציה ניתן לשחזר את מצב הרגיסטרים שהיה קודם.

<u>גודל הפריים</u>: משתנה לכל פריים בהתאם ללוקאליים והזמניים בהם משתמשת

Frame pointer: מצביע על הנקודה בה נמצאים, לפריים הנוכחי – סוף המבנה הסטטי.

- Locals
- Temporaries

<u>ארגומנטים לשגרה הבאה הנקראת</u>:

- Arguments: פרמטרים שצריכים להעביר לשגרה הבאה אליה קוראים משגרה זו. הארגומנטים מוכנסים כשיש קריאה לשגרה הבאה, אז ידועים כמה ומה הארגומנטים שצריך.

Stack pointer: מצביע לסוף המחסנית – כלומר אחרי הארגומנט האחרון – סוף המבנה הדינאמי.

**גישה ישירה למשתנה ב-frame קודם**:

- לכל משתנה ניתן לשמור את הרמה בה הוא מוגדר.
- לפי העומק בו נמצאים ניתן לחשב את המיקום של המתשנה.
- אפשרי רק במחסנית שמאפשרת גישה ללא pop.

**Local values of local variables**: ה-offset של המחסנית ידוע בזמן קומפילציה. נניח ורוצים לשים לתוך x את הערך 5: L-val(x) = FP+offset(x) – ה-offset הוא שלילי (כי יורדים בכתובות, זו הקונבנציה). בקוד מכונה זה ימומש כך: פעולה ראשונה היא load_constant r, R3 – שמירת הקבוע ברגיסטר; פעולה שניה היא store R3, offset(x)(FP).

**Code Blocks**: ישנן שתי גישות: עבור בלוק פנימי מממשים רשומת הפעלה נפרדת, או עבור בלוק פנימי רק שומרים ברשומת ההפעלה הנוכחית את המשתנים הלוקאליים. extend base pointer – Ebp – מצביע לתחילת רשומת בלוק בתוך רשומת ההפעלה של המתודה בה מקונן. אין lexical pointer וכמעט כל שאר החלק האדמיניסטרטיבי – אך יש שמירת registers. לאחר מכן שומרים משתנים לוקאליים וזמניים של הבלוק, ובסוף ארגומנטים שרוצים להעביר ביציאה מהבלוק. האחרונים הם משתנים גלובליים מנקודת המבט של הבלוק. זו לפי האופציה שלא מחזיקים רשומת הפעלה נפרדת.

**Caller and Callee**:

<u>חובות הקורא</u>:

- שמירת ארגומנטים לשגרה אליה קוראים בסוף המחסנית.
- שמירת ה-lexical pointer – לתת לשגרה הבאה מצביע לעצמו.

<u>חובות הנקרא</u>:

- FP:= SP
- SP = SP – frame-size – ה-fs מורכב מחלק אדמיניסטרטיבי קבוע וחלק דינמי.

<u>למה צריכים גם SP וגם FP</u>: ה-stack pointer הוא מעבר דרך המבנה הדינמי, ה-frame pointer מסמן את סוף המבנה הסטטי; לכן אי אפשר לחשב את ה-stack pointer באופן יחסי ל-frame pointer, כי ההפרש ביניהם דינמי.

**Supporting static scoping**: ישנן שפות בהן מותר להגדיר שגרות מקוננות, תחת הנחה מקובלת שכאשר יוצאים מהשגרה הפנימית היא מתה, וכל הזיכרון שלה ניתן לשחרור. אם רוצים לממש static scoping: בכל קריאה לפונקציה זוכרים lexical pointer – מאיזו שגרה הגענו, ומעבירים אותה כארגומנט לפונקציה; פתחנו פריים לשגרה הנקראת, ובסופה ניתן לשחרר אותה ואת כל הזיכרון שלה כי מבחינתנו היא מתה ואינה נגישה יותר.

**Routine descriptor for languages with nested scopes**: שומרים שתי רשומות הפעלה, "שגרה כפולה". נניח ב-PC בשגרה הנוכחית הגענו לקריאה לשגרה R. כאשר נכנסים ל-R מחסנית זמן הריצה משתנה – return address זוכר לאן לחזור (ה-PC+1 של השגרה הקוראת). כמו כן dynamic link – מצביע לשגרה הקוראת ו-lexical pointer מצביע לפריים האב הקדמון של השגרה הקוראת. למרות ש-R נקרא מהשגרה הקוראת, ה-lexical pointer מצביע לאב הקדמון של השגרה הקוראת.

**Nesting depth**: אין הגבלה על עומק מקונן אליו אפשר להגיע. שגרות מקוננות הן בעייתיות לניתוח – כל משתנה בו פוגשים יכול להיות משתנה לוקאלי של הבלוק או בסקופ מעליו וכך הלאה עד עומק 0 (ושם אם לא נמצא תזרק שגיאה). יישומים אפשריים:

ניתן להוסיף לכל פונקציה ומשתנה נתון שיסמן את עומק המשתנה (0 לגלובליים והוספת 1 בכל כניסה לפונקציה חדשה). ניתן לבצע חישובים אריתמטיים על כתובות ומצביעים וכך ללא pop לגשת אחורה ל-FP+offset(lexical_pointer) ולחזור אחורה כמה רמות שצריך.

יישום נוסף הוא הפיכת פונקציות מקוננות לקינון רגיל – קריאה לפונקציה. בכל קריאה צריך להעביר את כל המשתנים שהיו ידועים עד כה כארגומנטים לפונקציה החדשה. מושג זה נקרא **lambda lifting**. שיטה זו נדרשת במידה ולא ניתן לקפוץ קדימה ואחורה במחסנית זמן הריצה בלי pops ו-pushes; שיטה זו מפרקת את הפונקציות לפונקציות נפרדות כאשר בכל אינבוקציה דואגים להעביר מצביעים למשתנים לוקאליים כארגומנטים.

**Machine registers**: מספר הרגיסטרים מאפשר גישה יעילה. בנוסף לרגיסטרים וזיכרון פנימי ישנו ה-cache memory שאיטי מהרגיסטרים אך מהיר מהזיכרון. חשוב אם כן לשאוף בכתיבת קוד להשתמש כמה שיותר ברגיסטרים.

**Caller-save and Callee-save registers**:

אם ה-callee שומרת את הרגיסטרים:

- לפני שינוי רגיסטרים תחילה שומרים את תוכנם – כל הרגיסטרים, כי לא יודע במה משתמש ה-caller ובמה לא.
- הערכים נשמרים אוטומטית לאורך הקריאות וביציאה הם ישוחזרו.

אם ה-caller שומרת את הרגיסטרים:

- שומר <u>אך ורק את הרגיסטרים שרוצה לשמור את תוכנם</u>.
- הערכים לא נשמרים אוטומטית בין קריאות.

לרוב הארכיטקטורה מגדירה רגיסטרים כאלו ורגיסטרים כאלו. בסופו של דבר הקומפיילר מחליט על אופן הטיפול.

**Callee-saved registers**: בתחילת העבודה ב-prolog (pre processing) נשמרים הרגיסטרים; בסוף העבודה הרגיסטרים ישוחזרו ב-epilog. תתכן תמיכת חומרה בשמירה ושחזור רגיסטרים באופן יעיל. באופן עבודה זה ערכי רגיסטרים נכונים ישמרו לאורך כל הקריאות.

**Caller-saved registers**: ה-caller לפני כניסה לפונקציה שומר את הרגיסטרים שהוא צריך בלבד. בתוך ה-callee ניתן להשתמש ברגיסטרים, ואח"כ אלו שחשובים לנו ישוחזרו ממה שנשמר ב-caller. בשיטה זו ערכי הרגיסטרים לא נשמרים באופן אוטומטי לאורך הקריאות. שחזור הרגיסטרים הרלוונטיים יהיה ביציאה משגרת ה-callee.

**Modern architectures**:

Return-address: בד"כ חוזרים למקום משם קראנו לשגרה, ואותו יודעים לפי PC שנשמר לפני הקריאה בשגרה. בשגרה שהיא non-leaf, כלומר קוראת לשגרות אחרות, חשוב לשמור מצב מחסנית ומקום מדוייק בקוד כיוון שיותר מאוחר נצטרך לחזור אליהם. בשגרה שהיא leaf ברגע סיום השגרה צריך רק לשים את ערך ההחזרה למקום הנכון (רגיסטר שמור או מקום בזיכרון שמור).

Function result/value: בד"כ ישמר ברגיסטרים. נשמור גם תוצאות ביניים ורקורסיה במחסנית, ואפשר גם ברגיסטרים.

**Limitations**:

- לפעמים הקומפיילר יוכרח לשמור במחסנית ולא ברגיסטר.
- לפעמים המחסנית לא יכולה להתמודד עם חלק מהשפות, כמו למשל קריאות לפונקציות כשאחד הפרמטרים הוא שם שנקבע בצורה דינאמית.

**Frame-resistant variables**: מתי אי אפשר לשמור משתנה ברגיסטר:

- גודל המשתנה לא מתאים לרגיסטר.
- משתנה שהוא מערך – לא יכול להכנס לרגיסטר אחד אלא יש צורך בסדרה.
- כל הרגיסטרים בשימוש ברגע הצורך לשמור את המשתנה שלנו.
- כאשר יש הרבה משתנים מקומיים, שימוש רחב ברגיסטרים ידרוש פעולות שמירה והוצאה מרגיסטרים רבות.
- משתנה המועבר by reference.
- מצביע למשתנה שמתקבל רק מתוך חישוב שצריך תחילה להיעשות.
- ישנה גישה למשתנה מתוך פרוצדורה פנימית – מסתכלים מנקודת מבט של שגרה חיצונית ולכן לא ניתן להסתכל על הרגיסטרים מהפרוצדורה הפנימית.

<u>Escape variables</u>: משתנים ניתן להעביר by reference: הארגומנט הוא כתובת, וניתן לגשת אליה דרך pointer, דרך קפיצה למקום מסויים ב-stack frame או דרך הפניה מתוך שגרה פנימית.

**Limitations of stack-frames**: ישנן פעמים בהם משתנה מקומי צריך להמשיך להתקיים גם ביציאה מהשגרה. למשל:

- הגדרה ב-C של משתנה static – משמעותו היא הגדרת משתנה גלובלי. ביציאה משגרה המכילה הגדרת משתנה גלובלי, המשתנה צריך להמשיך להתקיים, למרות שמחסלים את רשומת ההפעלה בה הוגדר. במקרים כאלו צריך להשתמש במבנה נתונים נוסף לשמור אותו בו (?heap).
- דוגמא נוספת היא כאשר שגרה מחזירה עבור משתנה לוקאלי את הכתובת שלו - &x.
- דוגמא נוספת היא הקצאה דינאמית של זיכרון: בקריאת malloc שומרים משתנה זמני במחסנית, ויש לשמור אותו גם מחוץ לפונ'.

**Compiler implementation**: צריך להסתיר חלקים תלויים במכונה וחלקים תלויים בשפה, וזאת ע"י שימוש במודולים מיוחדים. שלבי הקומפיילר הבסיסיים הרלוונטיים: ב-code generation וניתוח סמנטי יש צורך ב-frames.

**Hidden in the frame ADT**: כל מיני דברים חבויים ב-Abstract data type, כמו מספר המשתנים המקומיים שהוקצה להם זיכרון עד כה, התווית בה קוד המכונה מתחיל לרוץ וכו'.

**Invocations to frame**: בכל קריאה לפונקציה יש להכין רשומת הפעלה, להקצות מקום (מחסנית זמן ריצה היא גם מבנה נתונים דינמי), הקצאת מקום למשתנים מקומיים, להעביר פרמטרים, לקשר בין רשומה קודמת לנוכחית, ייצור קוד לשגרה שיהיה עצמאי ובלתי תלוי בקוד לפניו ואחריו (מקרה מיוחד בשגרות מקוננות), שמירה ושחזור רגיסטרים והחזרת ערך למקום הנכון במידת הצורך לאחר יציאה מהפונקציה, הזזות.

## Static Program Analysis:

בניתוח סטטי נרצה לבדוק האם יש קוד מת, כלומר מסלול שאינו ניתן להגעה. בעיית ההכרעה האם תוכנית תבחר במסלול מסויים היא בעיה לא כריעה. פורמלית: עבור תוכנית מסויימת בודקים את כל מסלולי הזרימה, וקוד שלא ניתן להגיע אליו מאף מסלול יוצא מהקוד שכן זהו קוד מת. הניתוח אינו טרוויאלי ובד"כ לוקח יותר ממעבר אחד על הקוד.

דוגמא: בדיקה האם ניתן להחליף תוצאת חישוב בקבוע; הסרת בלוק else ב-if שערכו תמיד יקבל true.

ניתוח סטטי עושה שימוש ב-software quality tools כדי לגלות סכנות, כמו שחרור זיכרון שיש בו שימוש בהמשך הקוד, חריגה מגבולות מערך וזליגות זיכרון שיכול להיעשות בשימוש לקוי במבנים דינמיים (למשל שינוי מצביע איבר ראשון ברשימה מקושרת הגורם לאובדן קשר לרשימה).

**Control flow graph**:

- גרף זרימה מכוון המכיל נקודת כניסה ונקודת יציאה מוגדרות היטב. הגרף עשוי להכיל מעגלים.
- כל שורה בקוד היא צומת בגרף וכיווני הקשתות מתארים את זרימת התוכנית.

ניתוח סטטי משתמש ב-CFG, למשל עבור ניתוח התפשטות קבועים: החלפת תוכן משתנים בערכים קבועים אם ניתן לבצע זאת כבר בזמן קומפילציה. ניתן כך להגיע לגרף משופר.

**Constant folding**: אם צד ימין של ביטוי מכיל רק קבועים, ניתן כבר בזמן קומפילציה "לקפל" את כולם לכדי קבוע יחיד. יכול להתבצע בשלב ה-AST או אחרי התרגום. אופטימיזציה זו מתבצעת בכל קומפיילר מודרני.

**גילוי משתנים חיים**: ניתוח סטטי בודק חיות משתנים, כלומר בהינתן CFG, האם אחרי כל שורה בקוד משתנה בו נעשה שימוש / חושב יהיה בשימוש בהמשך. לשם כך צריך לעבור על כל מסלולי הזרימה של התוכנית. קומפיילרים מבצעים את הניתוח מלמטה למעלה. בעית גילוי חיות משתנים אינה כריעה – עבור משתנה שנקבע שהוא חי, התשובה נכונה תמיד. עבור מקרה בו נקבע שמשתנה אינו חי, תשובה זו לא בהכרח נכונה.

עבור משתנה מת נשחרר את האוגר בו נמצא וה-garbage collector דואג לשחרור הזיכרון. ישנו איזשהו אלגוריתם איטרטיבי לגילוי מידע סטטי.

## Register Allocation: (שיעור 11)

שיטות לקביעת מספר רגיסטרים:

- Setti-Ullman: פורט קודם.
- Top down: יורדים מלמעלה למטה תוך בחירת המסלול הארוך ביותר.

**קביעת מספר רגיסטרים ע"י צביעת גרף**:

הבעיה: בהינתן n רגיסטרים, האם ניתן לצבוע גרף המקביל לתוכנית הנתונה באמצעות n צבעים. אם לא, נצטרך לבצע שפיכה של חלק מהרגיסטרים לזיכרון, ויש צורך להחליט אילו מהרגיסטרים יאוחסנו בזיכרון.

האלגוריתם הינו אלגוריתם היוריסטי שכאשר מחזיר תשובה חיובית, בודאות קיימת צביעה ב-n, ואם מחזיר תשובה שלילית עדיין יתכן שקיימת צביעה אך האלגוריתם לא יכול לגלות זאת.

שלבים: תחילה בונים גרף תלויות. לאחר מכן מנסים למצוא n-צביעה לגרף. אם אין n צביעה, נבצע שפיכה של משתנה לזיכרון וננסה שוב. נחזור על התהליך עד שהצלחנו לצבוע.

גרף תלויות: גרף לא מכוון בו הצמתים הם המשתנים החיים בכל שלבי התוכנית (לכל משתנה חי צומת יחיד גם אם חי במספר מקומות שונים), והקשתות הן בין שני צמתים אם תוכן המשתנים שהם מציינים צריכים להיות ברגיסטרים בו זמנית. **שקופית 11: דוגמא**. כך צמתים הצבועים באותו צבע הם משתנים שיכולים תיאורטית לשבת באותו אוגר.

מטרה: שימוש במספר הקטן ביותר של רגיסטרים, ובמקרה של שפיכה שימוש במספר הקטן ביותר של פעולות Move.

**משפט**: אם G גרף לא מכוון, $v \in V$ צומת בעל דרגה קטנה מ-k והגרף $G\backslash\{v\}$ הוא k-צביע, אזי G הוא k-צביע – כלומר לקודקוד $v$ אין השפעה על ה-k צביעות של הגרף כולו.

מציאת הצביעה: **(שקופית 19)**

יהי R מספר הרגיסטרים במכונה ו-G גרף התלויות. נבצע את האיטרציות הבאות עד סיום / עד שנתקעים:

- כל עוד ל-G יש צומת עם דרגה קטנה מ-R, נסיר את הצומת והקשתות הקשורות אליו מ-G ונדחוף את הצומת למחסנית S.
- אם כל הגרף הושם ל-S, אזי הוא R-צביע: כל עוד S לא ריקה, נוציא את הצומת בראש, נחזיר אותו ל-G וניתן לו צבע מ-R הצבעים תוך ניסיון לבחור צבע שכבר השתמשנו בו. אם כל שכניו צבועים בכל הצבעים בהם השתמשנו עד כה, ניתן לו צבע חדש.
- אחרת נפשט את הגרף ע"י בחירת אובייקט לשפיכה והסרת הצומת המתאים לו מהגרף. האובייקט נבחר לפי מספר ההגדרות והמצביעים: ניתן לבחור את הצומת עם הדרגה הגבוהה ביותר בגרף התלויות G, או המשתנה שחי את הזמן הרב ביותר, או בדיקת שימוש במשתנה בתוך לולאה ומחוצה לה (המשתנה עם המונה הגדול יותר נבחר).

יישום הדרך השלישית לבחירת אילו משתנים לזרוק לזיכרון:

- נחשב לכל משתנה spill priority באופן הבא: $\frac{\#times\ outside\ the\ loop+10\times\#times\ inside\ the\ loop}{node's degree}$.
- הצומת עם בעדיפות הנמוכה ביותר יועף ראשון. הרעיון בבסיס השיטה הוא לרצות לשמור משתנה שמשתמשים בו הרבה בתוך רגיסטר.

Pre-colored registers: חלק מהרגיסטרים לא ניתנים לשפיכה: stack pointer, frame pointer, parameters.

## Assembler/Linker/Loader: (הרצאה 12)

Assembler: קומפיילר המייצר קוד מכונה מ-assembly בתרגום אחד לאחד מקוד assembly לקוד בינארי, בהתאם לנתונים ול-opcode. בין היתר פותר בעיות הפניות חיצוניות ומשנה סדר קוד.

Loader: חלק מערכת ההפעלה שלא מסתמך על שפת התכנות. חלק זה בעל הרשאות מיוחדות ממערכת ההפעלה והוא מאתחל את מצב זמן הריצה של התוכנית. רשומת ההפעלה שלו אינה נגישה (לא נראית).

Linker: מאחד כמה executables תוך פתירת בעיות הפניות חיצוניות. כמו כן הוא יכול לשנות כתובות. הלינקר רץ ב-mode user (בניגוד ל-loader), מסופק ע"י מערכת ההפעלה.

- הקוד יכול לכלול הפניות למשתנים חיצוניים כמו פונ' ספריה או מידע חיצוני. אלו מאוחסנים בטבלת סימבולים חיצונית.

**תהליך היצירה**:

- ה-Assembler יוצר קוד בינארי.
- ה-Linker יוצר קוד executable.
- ה-Loader יוצר מצבי זמן ריצה (images).

טיפול בכתובות פנימיות ב-Assembler:

שני מעברים על הקוד:

- בניית טבלת לייבל לכתובת.
- החלפת לייבלים עם ערכים.

Backpatching: ניתן להחליף תהליך זה במעבר קוד יחיד תוך שמירה על רשימת לייבלים שלא טופלו.

טיפול בכתובות חיצוניות:

מחזיקים טבלת סימבולים חיצונית, מייצרים קוד בינארי עם שינוי כתובות (relocation bits). תוצר הפלט של ה-Assembler:

- Code segment
- Data segment
- Relocation bits
- External table

בעיות בעיצוב ה-Linker:

- שרשור כל חלקי פלט ה-Assembler
- ...