קומפיילרים ומתרגמים

שנת 01011010010100

מסוכם משיעורי מרן הרב

ד"ר נדב שוייצר שליט"א

ובתוספות תרגולי הבחור הנעים אב בתורה ורך בשנים

שלום רוכמן שליט"א

עם פירוש

"רי"ח טוב"

מאיתי הצב"י יוחנן חאיק

לסיכומים נוספים לכו לאתר שלי -
https://yohananha.wixsite.com/smellsgood

להערות, הארות ותיקונים:
yohananha@gmail.com
yohanan@ - בטלגרם
ניתן להשתמש בסיכום באופן חופשי לכולם!!

תוכן עניינים

תוכן עניינים 1

קומפיילרים ומתרגמים 5

הכרת מושגים בסיסיים 5

חשיבות הקומפילציה 7

מבנה הקומפיילר – תמונה כללית 8

קומפיילר – חלוקה גסה 8

מרכיבי הקומפיילר – חלוקה עדינה 9

Front-End 9

Lexical Analyzer (המנתח המילולי) – 9

Syntax Analayzer 10

Semantic Analyzer 10

טבלת הסמלים 10

הודעות שגיאה 10

קוד ביניים 11

אופטימיזציה של קוד 11

Back-End 12

כלים לקומפיילרים 12

ניתוח לקסיקלי 13

מושגים בסיסיים 14

תפקידי המנתח המילולי 14

Typical Tokens 15

טיפול במחרוזות שאינן ID 15

הקושי בניתוח הלקסיקלי 15

ניסוח ה-Tokens המותרים בשפה 16

Lookahead 17

טיפול בשגיאות 17

סיכום 17

המנתח המילולי מצגת דוגמאות מס' 1 18

מימוש המנתח המילולי בתכנות ידני 21

ניתוח תחבירי – Syntax Analysis 27

דקדוק חסר הקשר 27

מושגי דקדוק ח"ה 27

גזירה Bottom-Up 28
סוגי הניתוח התחבירי 29
גזירה Top-Down 29
גזירה Bottom-Up 30
Recursive Descent - ירידה רקורסיבית 30
הוספת פעולות במהלך הגזירה 32
FIRST 35
ניתוח תחבירי בירידה רקורסיבית מצגת דוגמאות מס' 2 36
כתיבת פונקציה איטרטיבית 36
הגדרת הפונקציה first 38
LL(1) 48
רקורסיה שמאלית 48
Left Factoring 50
הצבת גזירות במקום משתנים 50
LL(k) Parsers 52
טבלת המעברים 52
FOLLOW 53
גזירה בהינתן טבלה 54
טיפול בשגיאות 55
לסיכום 56
ניתוח תחבירי מונחה-טבלאות – LL מצגת דוגמאות מס' 3 57
דקדוקי LR 63
תהליך העבודה 63
פריטים ומצבי LR(0) 64
האינטואיציה לניתוח LR(0) 65
המחסנית 65
טבלת הפעולות 65
טבלת GOTO 66
הרצת דוגמה 66
בניית המצבים 66
בעיות עם דקדוק LR 72
SLR(1) 73
ניתוח תחבירי מונחה-טבלאות – LR מצגת דוגמאות מס' 4 75
ניתוח סמנטי 81
מושגים והגדרות 81

תרגום מונחה דקדוק ... 81
תלויות ... 83
תכונות נוצרות ונורשות ... 84
בניית גרף התלויות ... 84
דקדוקי S ... 85
דקדוקי L ... 86
קוד ביניים ... 88
ייצוג ביניים ... 88
קוד תלת מעני ... 89
סוגי המשפטים בקוד ביניים ... 91
ייצוג של קוד תלת מעני בזיכרון ... 94
הקצאת מקום בזיכרון ... 96
ניתוח משמעות : דקדוקי תכונות ופתירת שמות – מצגת דוגמאות מס' 5 ... 98
מתחמי הכרה וישויות ... 101
שמות מוסמכים ... 104
ניתוח משמעות: פתירת העמסה - מצגת דוגמאות מס' 6 ... 106
טיפול בהפניות בקרה ... 116
ביטויים בוליאניים בייצוג מספרי ... 117
הפקת קוד ביניים - מצגת דוגמאות מס' 7 ... 121
יעדי קפיצות והערך fall ... 123
פעולות סמנטיות להפניות בקרה ... 123
ייעול קודים בוליאניים ... 124
סביבת זמן ריצה ... 131
Dynamic vs. Static Scope ... 131
מימוש גישה למשתנים ... 133
רשומת הפעלה ... 135
פרוצדורות בתוך פרוצדורות ... 138
פרוצדורות בתוך פרוצדורות: שימוש ב-access link ... 139
סביבת זמן-הריצה - מצגת דוגמאות מס' 8 ... 140
ניהול access link באמצעות מערך display ... 147
יצירת הקוד ... 151
Flow Graph ... 152
אופטימיזציות ... 155
טיפול במשתנים ... 155
טבלת ה-Next Use ... 156

הפקת קוד משופרת: גושים בסיסיים - מצגת דוגמאות מס' 9 חלק א' 158
טרנספורמציות פשוטות בתוך בלוקים 164
תהליך יצירת קוד 164
מעקב אחרי משתנים ורגיסטרים 164
Code Generation 165
הפונקציה getreg 165
הפקת קוד משופרת: גושים בסיסיים - מצגת דוגמאות מס' 9 חלק ב' 169
הקצאה והשמת רגיסטרים 178
הקצאת רגיסטרים גלובלית 178
צביעת גרפים 179
אופטימיזציות 183
מה זה אופטימיזציות? 183
ממה נובע חוסר יעילות בקוד? 183
אופטימיזציה של זמן ריצה 183

קומפיילרים ומתרגמים

הקומפיילר הוא מרכיב חשוב בכל מערכת. תחום הקומפיילרים הינו רחב, ומכיל בתוכו כמעט מכל תחומי מדעי המחשב – שפות תכנות, בוודאי , אבל גם מערכות הפעלה (הקצאת מקום וכו'),הנדסת תכנה, מבני נתונים ואלגוריתמים (בהם נשמש בשביל לנתח את הכתוב), אוטומטים ושפות פורמליות, הכרת מבנה המחשב ועוד.

תחומי הידע אינם רק מעשיים, אלא גם מחקריים – חלק גדול מהמדענים שקיבלו פרסי טיורינג, קיבלו אותם על סמך העבודה שהם ביצעו וחקרו בנושא הקומפיילרים.

הכרת מושגים בסיסיים

בגדול, כל המטרה שלנו בבניית קומפיילר, הוא שתהיה לנו מערכת המקבלת קובץ מקור בשפת קוד כלשהי, לעשות על הקטע הזה מניפולציה מסוימת תחת היכולות השונות, ולייצא לנו קובץ הפעלה EXE אותו אנחנו יכולים לתת ללקוח שיכול הפעיל.

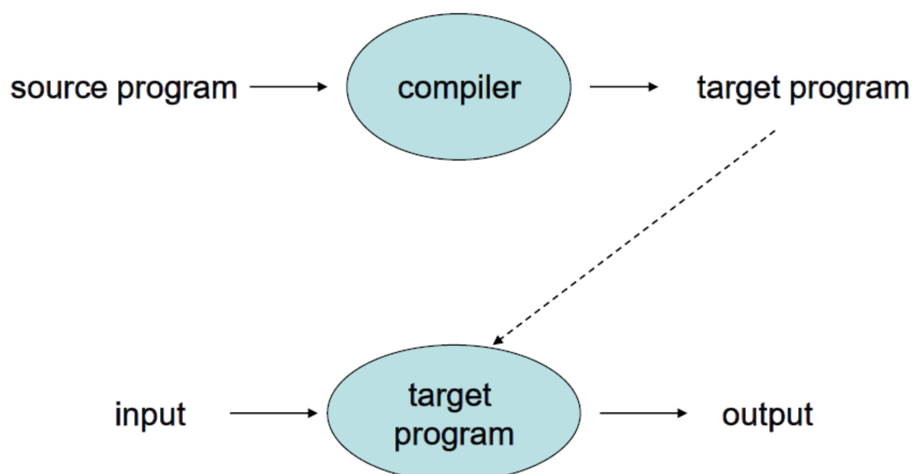
מכאן, נוכל כבר לשאוב שני מושגים ראשוניים:

זמן קומפילציה מתייחס לזמן העיבוד של הקובץ. כל פרק הזמן החל מקובץ הטקסט המקורי ועד קובץ ההפעלה המתקבל, שייך לזמן הקומפילציה.

זמן ריצה מתייחס להפעלת התוכנית עצמה. ברגע שהלקוח או אנחנו בעצמנו מפעילים את התוכנית, מכניסים אליה את הקלט, וממתינים לפלט מתאים.

חשוב לעמוד על ההבדל מבחינתנו בין השניים – הקומפילציה עצמה צריכה להתבצע פעם אחת בלבד, מה שבדרך כלל לא קורה, אבל נניח שהכל עבד חלק. ומכאן ואילך, כל שגיאה שתתבצע תהיה על זמן הריצה. עכשיו, אנחנו כמובן רוצים שהתכנה תרוץ באופן חלק אצל הלקוח, ולא תעמיס עליו וכו',וכמובן שלא תקרוס בגלל שגיאות. אבל חשוב לא פחות מזה הוא הקימפול בעצמו. אמנם הרבה פעמים קל לנו לחשוב שנקמפל, נראה איפה יש שגיאה, נתקן וחוזר חלילה, אבל זה גם בזבוז זמן עבודה שלנו, ובמידה שיש לנו תוכנית גדולה ועצומה, אנחנו לא יכולים לקמפל שוב רק בשביל לראות ששוב יש לנו בעיה באותו מקום בדיוק ולא באמת תיקנו את זה.

מבחינת השלבים, אנחנו יכולים להציג את זה בצורה חזותית באופן הבא –



אנחנו מקבלים את תכנת המקור (Source Program) ומעבירים אותה בקומפיילר, ומה שיוצא לנו היא תכנית המטרה (Target Program), שזה הקוד שהמחשב של הלקוח יודע לעבוד איתו. החלק העליון הוא מוגדר תחת זמן

קומפילציה, כאשר החלק התחתון הוא זמן הריצה – תוכנית המטרה עומדת עכשיו בפני עצמה ומתמודדת עם קלט ופלט.

נתקלנו גם בעבר בסוג אחר של שפה – שפות סקריפט. בשפות אלו (ופייתון הוא ייצוג מאוד בולט של השפות האלה), אין לנו קומפיילר, אלא מפרש (Interperter), המפרש לוקח את התכנית הכתובה בבת אחת עם הקלט ומנסה לקרוא הכל שורה אחרי שורה.

כמובן, שלכל שפה ומתודה יש יתרונות וחסרונות –

מבחינת פענוח הקוד בעצמו, כבר ציינו שקומפיילר עובד פעם אחת בלבד, ואז הוא לא צריך לחזור על התהליך, דבר שלא קורה כשאנחנו צריכים לפרש מחדש בכל פעם והפעלה של התוכנית.

יותר מזה, נלמד בהמשך על האופטימיזציה שהקומפיילר עושה לקוד הכתוב. ברגע שהקומפיילר מזהה חלקים מיותרים מכל מיני סיבות בקוד, הוא פשוט מוחק אותו, מה שכמובן גם מועיל לנו בזמן ריצה, האינטרפרטר מסתכל באופן מאוד מקומי על כל שורה שהוא רואה ולכן לא יכול לשפר קדימה.

מבחינת השגיאות, מרבית הדברים בהם ניתקל בקומפילציה הם שגיאות קומפילציה, ועד כמה שזה מבאס, זה עדיין ניתן לשליטה ברמה המקומית, ולא מדובר פה על שגיאות בזמן ריצה כשהתוכנית עצמה כבר לא אצלנו.

מהצד השני, היתרונות הגדולים בשפות סקריפט, הוא בכך שבדרך כלל מדובר בתוכניות שהן קטנות יחסית, כך שקל יותר לבדוק את המודולים השונים ואין צורך לחכות הרבה זמן קימפול בשביל תוכנית קטנה.

כמו כן, מאחר שהכל עובד בזמן ריצה, אנחנו יכולים אפילו לשנות את הקוד תוך כדי ריצה ולא להיות נעולים.

מבנה נוסף של שפה שחשוב לדבר עליה היא JAVA. עד ליצירת ג'אווה, כל שפה היתה צריכה לעבור בתהליך הקימפול לשפה שמתאימה לאותה מכונה שמריצה אותה, כך שהיינו צריכים להכיר את כל סוגי המכשירים השונים ולהתאים אליה את השפה.

החידוש של ג'אווה הוא ביצירת שלב ביניים. קוד המקור הנכנס לתוך המתרגם של ג'אווה יוצר מהצד השני בקובץ שנקרא Bytecode, הבייטקוד הוא שפת הביניים האחידה. כעת, כל מכשיר שרוצה להריץ את הבייט קוד הזה, לא צריך לדעת לפרש אותו בעצמו, יש לו מעין אינטרפרטר בצורה של מכונה וירטואלית JVM, שמתקשר למכשיר ויודע להריץ עליו את קוד הביניים, מה שחסך לנו הרבה בהתאמה השונה בין סוגי מכשירי הקצה.

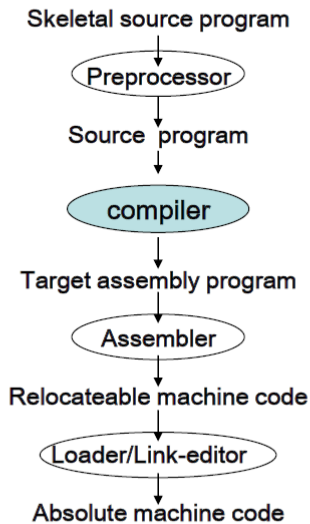
אם נכליל, נוכל לדבר על שלושה סוגים של הידור:

1. Source to Source – העברה של קוד משפה אחת לשפה אחרת. למשל: העברה של קובץ ב-++C לקובץ בשפת C.
2. Virtual Machine – המרה של תוכנית בג'אווה לקוד ביניים.
3. Pre-Processors – קדם -הידור. חלק שקורה לפני כל ההידור ומתייחס לפקודות מאקרו שמופיעות בקוד, וכן להכללות של ספריות ומודולים שונים.

אם נרחיב את המבט על קומפיילרים, נוכל להתייחס גם ל"וורד" בתור סוג של קומפיילר – אנחנו כותבים טקסט, ואם התוכנה מזהה שיש בעיה עם המילים שכתבנו, אנחנו מקבלים סימון אדום מתחת למילה. ואפילו הויז'ואל סטודיו שאנחנו משתמשים בו לכתוב קוד, עושה לנו הידורים לפני הקימפול בעצמו – בכל פעם שהוא מציע לנו את המילה הבאה לכתיבה, הוא בעצם "מהדר" לנו את מה שכתבנו.

נתמקד רגע שוב ב-Pre-Processor. בכל פעם שאנחנו כותבים איזה פקודת מאקרו שמתחילה ב-#, לפני שמתחיל כל הקימפול, הקומפיילר עובר על החלקים האלה ובודק מה כתוב שם – זה יכול להיות include#, ואז הוא מביא לנו את הספריות (מה שמוסיף כמות עצומה של טקסט לקובץ), וזה יכול להיות גם משהו בסגנון של define max 5#, ואז בכל מקום בקוד שמופיעה המחרוזת "max", המחרוזת תוחלף אוטומטית ב-"5".

כמו שאנחנו יכולים לראות באיור הצמוד, המעבר מה"שלד" של תוכנית המקור – כלומר קטע התוכנית שכתבנו, לשפת המכונה המוגמרת, הוא לא רצוף ופשוט. קודם כל, אנחנו נתקלים ב-Pre-Processor, שמשנה את הדרוש ומוסיף מודולים שונים.



לאחר מכן, יש לנו את תוכנת המקור הסופית הנכנסת לקומפיילר. משם יוצא לנו קוד מיועד אסמבלי, שעובר באסמבלר ומוציא לנו שפת מכונה. בשלב הזה, יש לנו כל מיני קטעי טקסט של הקוד שתפורים אחד לשני בצורה גסה, ובשביל להבטיח שהכל יעבוד בסדר הנכון, אנחנו משתמשים ב-Loader/Link Editor וייסדרו הכל לכדי קוד אחד קריא שיהיה קוד המכונה המושלם.

מה ההבדל בין Linker ל-Loader?

ה-Linker דואג שיהיו קישורים (להלן: לינקים) בין החלקים השונים בקוד. אנחנו מתחילים מהחלק הראשון, וברגע שמסיימים אותו הלינקר יודע להגיד לנו לאן לקפוץ בשביל להמשיך את הקוד.

ה-Loader רוצה שיהיה קוד רציף, בשביל זה הוא לוקח את כל הקטעים השונים וטוען אותם (להלן: Load) אחד אחרי השני בסדר הנכון.

מה טוב יותר? מה יעיל יותר? לא אכפת לנו כרגע.

חשיבות הקומפילציה

אז למה אנחנו צריכים בכלל לדעת על קומפילציה מעבר ל-F5?

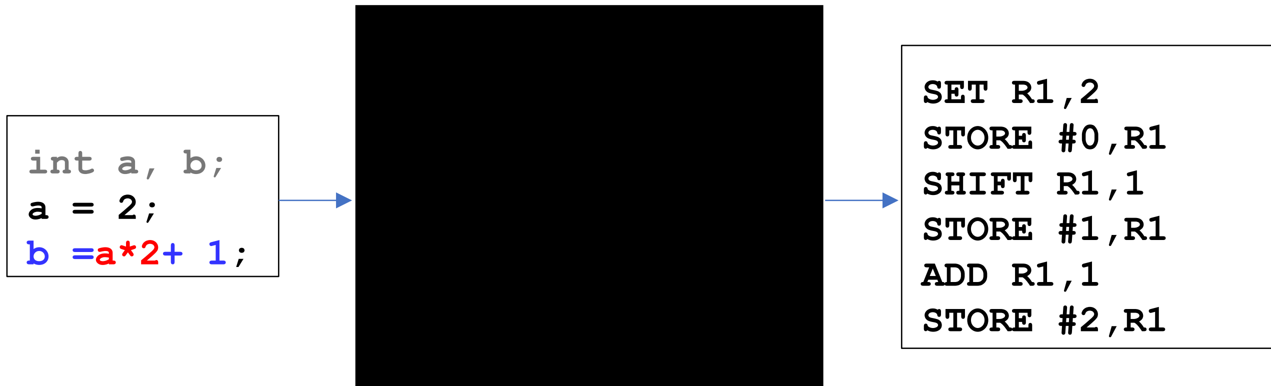
בפן התעשייתי, הקומפיילרים הם כבר לא רק מה שקורה מאחורי עיבוד התוכנה הכתובה. הזכרנו את הוורד ודומיו, אבל בעצם גם כל הבינה המלאכותית משתמשת באותם אספקטים – ניתוח של מלל והבנת המשמעות שלו.

בפן התכנותי, עלינו לדעת מה ההבדל בין סוגי השגיאות השונות – אנחנו רוצים וצריכים להגיב אחרת לשגיאות קומפילציה ושגיאות זמן ריצה. כמו כן, אם נבין ונפנים את דרך פעולת הקומפיילר, אולי זה עצמו יצליח לייעל את התכנית ולהמנע משגיאות מיותרות. רק לדוגמא, מבחינת הקומפיילר, אם נכתוב x++ זה הרבה יותר יעיל מאשר לכתוב x++. כמה יותר יעיל? כל פעם שנכתוב את הצורה הפחות טובה, הקומפיילר יחליף את זה לתצורה האופטימלית. אז מתכנת טוב כבר יידע לעשות את זה מראש בצורה נכונה.

מבחינת שפות התכנות החדשות, אם נרצה לבנות קומפיילר לשפה חדשה (דבר שעושים בקורס ההמשך "עקרונות שפות תכנה"), עלינו להכיר את הכללים ואת הדרך הנכונה לעשות זאת.

נכנס עכשיו לסקירה כללית של תהליך הקימפול עם דוגמא פשוטה.

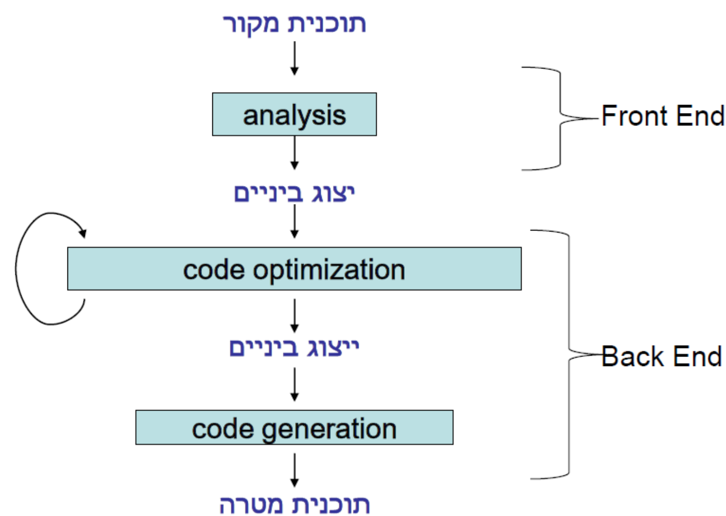
מבנה הקומפיילר – תמונה כללית



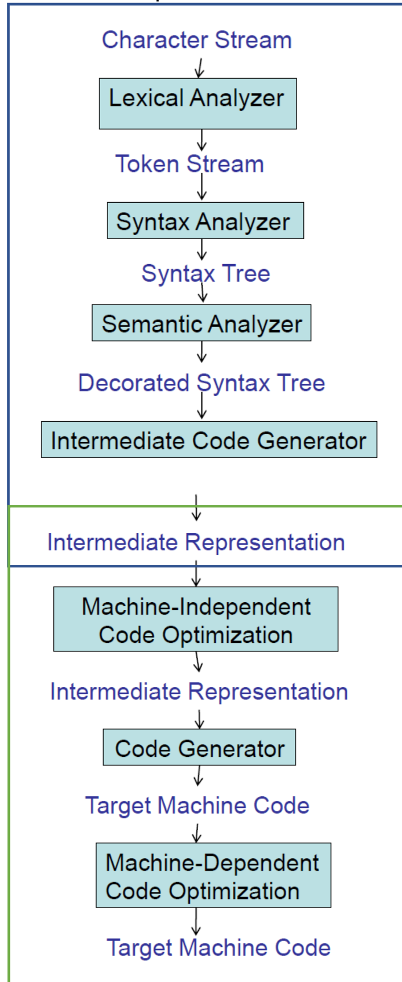
אם נסתכל על הקומפיילר כקופסא שחורה, בצד אחד ייכנס הקוד כמו שאנחנו מכירים אותו, ובצד השני תצא לנו שפת מכונה. כמו שאנחנו יכולים לראות שפת המכונה הרבה יותר מסובכת וארוכה (לפחות מבחינתנו), אבל אם מתייחסים למשמעות הטקסט עצמו, שני קצוות הטקסט הם בעלי אותה משמעות בדיוק.

על מנת להתייחס לחלקים השונים, הקומפיילר מחולק למקטעים מודולריים, אליהם ניתן להתייחס כל אחד בפני עצמו. את הסקירה הכללית הזאת נעשה בצורת Top-Down, וננסה לרדת מטה בסולם ולהבין מה הקופסה השחורה הזאת עושה.

קומפיילר – חלוקה גסה



נהוג לחלק את הקומפיילר לשני חלקים עיקריים Front-End ו-Back-End, כאשר הקצה הקדמי מקבל את הקוד שלנו, עובר עליו ומנתח את המשמעות שלו, ומוציא ממנו קוד ביניים. קוד הביניים הזה נכנס יותר עמוק אל החלק האחורי ועובר אופטימיזציות שונות. ניתן לראות שעל האופטימיזציות יש לולאה, כי אנחנו תמיד יכולים להכנס קצת יותר עמוק ולעשות עוד ועוד אופטימיזציות, כי תמיד יש מה לשפר בקוד, אך לרוב יהיה שלב שבו נחדול את האירוע ונתקדם הלאה. ומשם יצא הקוד בשפת שהמכונה מולה אנחנו עובדים יודעת לממש.



מרכיבי הקומפיילר – חלוקה עדינה

באיור משמאל, אנחנו יכולים לראות את כל השלבים השונים שהקוד עובר החל מ"זרם התוים" המקורי אותו אנחנו כתבנו, ועד שהוא יוצא כפלט מכונה. חילקתי פה את האזורים השונים במלבנים הצבעוניים על מנת להפריד בין החלק הקדמי לאחורי, כאשר יש לזכור שהפלט של אחד הוא הקלט של האחר.

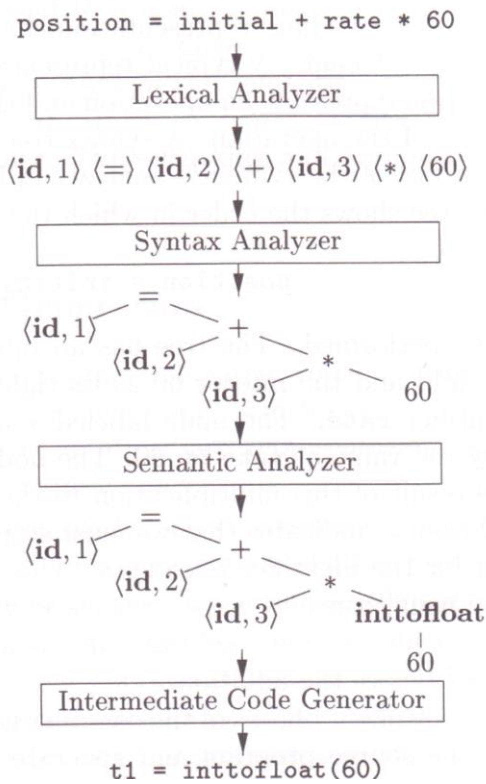
Front-End

ננסה לעבור כל חלק ולהפריד בגדול ולהסביר מה התפקיד העיקרי. כמובן שלא ניכנס ממש לעומק, אלא רק נסביר בגדול תפקידים ורעיונות –

Lexical Analyzer (המנתח המילולי) –

כשאנחנו כותבים פקודה פשוטה כמו Int a =3, אנחנו נוטים אוטומטית להתייחס לכל זה בתור מילים שמרכיבות משפט כלשהו. אך יש לזכור שמבחינת הקלט של המחשב יש פה רצף של תווים. לא משנה אם כתבנו "א", " ", או "3", המחשב "קורא" הכל באותו אופן.

מה שהמנתח המילולי עושה, הוא קריאה של רצף התווים ובדיקה האם הם עומדים בתקן של תווים תקינים בקוד. המנתח עובר אות אחרי אות, וכל עוד הוא מזהה שאנחנו מרכיבים מילים ומחרוזות עם צירופים הגיוניים, הוא ממשיך לעבוד. למה אנחנו מדגישים שמדובר על "צירופים הגיוניים"? כי אנחנו עוברים כרגע מ"אות" ל"מילה". אם יהיה רשום לנו int int a = 3, הקומפיילר ימשיך לרוץ ולא תהיה לו שום בעיה, כל המילים פה הם חוקיות.



איך אנחנו נדע איזה רצף תווים הוא חוקי? אנחנו נקבל/נגדיר אוטומט שמקבל את המילים שמותרות בשפה, ונשתמש בביטויים רגולריים שונים על מנת לאשר את המילים. בדוגמה אנחנו יכולים לראות את החלוקה של כל האסימונים שקשורים למשפט שכתבנו. יש לשים לב, גם התווים <+> נחשבים אסימונים, ולא רק המשתנים עצמם.

פלט: רצף אסימונים (Tokens).

Syntax Analayzer

המנתח הסינטקטי לוקח את רצף האסימונים, ובודק האם הוא כתוב בצורה חוקית. מה הכוונה? אנחנו עדיין לא מתייחסים למה שכתוב, אלא להירכיה של המשפט. אם למשל יהיה לנו סוגריים פותחים, אנחנו נצפה שבאיזשהו שלב הם גם ייסגרו.

המנתח עובד על שפות חסרות הקשר, ולמעשה משתמש באוטומט מחסנית בשביל לפענח את הסינטקס – מילים נדחסות לתוך המחסנית, וברגע שאנחנו מזהים שהחוקיות הנכונה מתקיימת, אנחנו יורים בצרור את כל המילים הקיימות ובונים מהם עץ גזירה.

Semantic Analyzer

הניתוח הסמנטי הוא קצת יותר מסובך – לאחר שאנחנו יודעים שכל המילים נכונות והסדר חוקי, אנחנו מתחילים להתעסק עם הסמנטיקה (המשמעות) של הכתוב. אנחנו נכנסים בצורה רקורסיבית, ועוברים על עץ הגזירה מעלה ומטה, ומוסיפים קוד לחלקים מסויימים של העץ. ניתן לראות למשל בדוגמה בצד, שהיתה לנו הצבה של מספר – ובשלב זה אנחנו רואים שכל המספרים שאנחנו מתעסקים איתם הם float ולא Int, ולכן אנחנו מעבירים את המספר בפונקציה שנקראת inttofloat, ותשנה את 60 ל-60.0.

טבלת הסמלים

אפשר לראות שהמנתח המילולי לקח את כל המשתנים, ובמקום להתייחס אליהם בשמם מספר אותם. למעשה, בשלב זה אנחנו בונים טבלת סמלים -איזה סוג של מבנה נתונים בו נאכסן את כל המידע החשוב של המשתנים, כמו הטיפוס שלו, המקום בזיכרון, באיזה רמה של scope הוא נמצא (יכולים להיות לנו מספר משתנים עם אותם שמות וטיפוסים, אך אם הם ברמות שנות אנחנו בסדר עם זה) ועוד.

המשתנים ייכנסו לפי הסדר של הגילוי שלהם בניתוח המילולי, כך ש-position שהשתנה ל-<id,1> יהיה הראשון בטבלה וכן הלאה. הטבלה הולכת ומתמלאת בכל שלב שאנחנו מתקדמים עם המידע החדש שמגיע, כך שכשאנחנו מסיימים הכל יש לנו את כל המידע הדרוש.

הודעות שגיאה

לכל שלב של ניתוח יש את הודעות השגיאה המתאימות לו. אם משתנה רשום בצורה לא חוקית, אנחנו נזרוק הודעה, וגם אם יהיה חסר לנו סוגר לסוגריים, אבל כל אחת מהשגיאות האלה מגיעות ממקום אחר לגמרי והטיפול בהם יהיה שונה.

יש להעיר, כשאנחנו כותבים תוכנית אנחנו נתקלים לפעמים במצב בו יש לנו המון שגיאות, ואנחנו משנים משהו קטן (שכחנו להוסיף ";" בסוף שורה), וזה הדליק לנו את כל המערכת ו-200 שגיאות. אבל כשאנחנו מתקנים את הבאג הזה, פתאום הכל משתחרר. למה זה קורה? הקומפיילר לוקח בחשבון שאנחנו עלולים לעשות טעויות, ולכן לפעמים

ינסה להמשיך הלאה ולקוות שהקוד יסתדר מעצמו. כמו שלמדנו על בשרינו לא פעם – קוד לא מסתדר מעצמו, ואנחנו צריכים לשבת ולתקן אותו.

קוד ביניים

הייצוג של הקוד היוצא לאחר כל הניתוחים שעשינו. קוד הביניים הוא נוח מאוד לקריאה ולאופטימיזציה. המבנה שלו מכונה "קוד תלת-מעני", שהמשמעות שלו הוא שהוא יכול להכיל עד שלושה מענים שונים – כלומר, בדרך כלל יהיו לנו איזה שלוש משתנים המחוברים ביניהם באיזשהו אופרטור, שיעשה את הפעולות וההצבה הדרושה.

מה טוב בקוד הזה? מאחר והוא כל כך קל גם לייצור וגם להמשך העבודה, הוא משמש בתור קוד אוניברסלי, והוא יכול לקשר בין המון שפות ומערכות שונות בצורה יותר קלה. למה הכוונה? אם יש לנו n שפות ו-m מכשירים שצריכים לקרוא אותו. אם אנחנו עובדים בצורה של אחד-לאחד, אז על כל שפה נצטרך לעשות m קומפיילרים שיעבירו את הקוד עבור כל המכונות האפשרות, כך שבסוף יהיו לנו n*m קומפיילרים, ובכל פעם שמישהו ירצה לכתב שפה חדשה זה יהיה לו מאוד קשה.

לעומת זאת, אם אנחנו יודעים שמנקודה מסוימת אנחנו לא צריכים להתאמץ, וכל הקומפייילרים לכל המכשירים החל מקוד הביניים כבר כתוב, אז יש לנו תקווה טובה לצמצם את המספר הזה לn+m קומפיילרים שבוודאי הרבה יותר נוח.

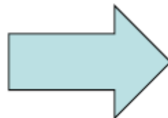
אופטימיזציה של קוד

כמה ניתן לייעל קוד? בעזרת כלים מתאימים אפשר לקחת תוכנית מסויימת ולהתחיל לשפר אותה, אבל איך נדע בוודאות ששיפור הקוד שעשינו הוא השיפור הכי טוב, או בכלל? איך נדע שאם נתאמץ עוד קצת נוכל להביא ייעול מטורף לקוד שכמוהו לא נראה בעולם?

למעשה, התשובה לשאלה זאת מוגדרת NP קשה, שאולי זה לא אומר הרבה, אבל בהמשך הסמסטר בקורס אלגוריתמים יובהר עד כמה זה קשה.

ניקח למשל את הקוד הבא -

```
temp1  :=  inttoreal(60)
temp2  :=  id3 * temp1
temp3  :=  id2 + temp2
id1 := temp3
```



```
temp1  :=  id3 * 60.0
id1 :=  id2 + temp1
```

החלק הצהוב הוא הקוד התלת מעני שהוצאנו. אנחנו מציבים במשתנה ערך מסוים ועושים עליו מניפולציות שונות, אך בשלב האופטימיזציה אנחנו עוברים מספר שלבים ומבינים שלמעשה אנחנו יכולים לצמצם הכל לשתי שורות, לשמור על חוקית הקת"מ, ולקבל בדיוק את אותה התוצאה. אושר ודמעות!

אילו דברים נוכל לשפר?

- Constant propagation – שימוש בקבועים במקום משתנים ופישוט של הקוד. דוגמא טובה לזה הוא מה שראינו בדיוק עכשיו, נפטרנו מכל המשתנים והשתמשנו במספרים.
- Common Subexpression – אם יש לנו קטע קוד שחוזר על עצמו הרבה פעמים או ביטויים משותפים כלשהם, אנחנו נשתדל למחוק את המיותר ולהישאר עם קוד נקי יותר.
- Dead Code Elimination – אם הקומפיילר מזהה קטע קוד אליו לעולם לא נכנס, למשל (if(false, לא ממש ברור לנו למה זה נכתב, אבל אנחנו לעולם לא ניכנס לשם, ולכן אנחנו מוחקים את כל הקטע הזה.

כל השיפורים האלה, בשאיפה, יעבדו על המהירות של הקימפול, המקום בזיכרון שעכשיו יהיה פנוי יותר, אנרגיה רבה של המחשב תיחסך ועוד. למעשה, בגלל שאנחנו יכולים לחזור ולשפר בכל פעם קצת את הקוד, זה החלק שלוקח הכי הרבה זמן, כך שאנחנו צריכים לקחת בחשבון איזה סוג קומפיילר אנחנו רוצים, אחד שעובד לאט אבל יסודי, או מהר ומסורבל. התשובה כמו תמיד נמצאת איפשהו באמצע -אין סיבה להסתבך מידי על "Hello World", אבל גם לא כדאי לזלזל בתוכניות מתוחכמות ומסיביות.

האתגר הנוסף היום, הוא איך להשתמש במערכות מרובי ליבות בצורה אופטימלית, ואיך ננהל את הירכיית הזיכרון בצורה נכונה. כל זאת ועוד בהמשך. או שלא. נראה כבר.

Back-End

החלק ה"אחורי" נקרא גם "שלב הסינתזה", ועיקרו לתת משמעות למילים ולטקסט שהוצאנו עד עכשיו. בעוד שעד עכשיו הצלחנו לחבר תוים לאותיות, מילים, ומשפטים, עכשיו אנחנו מנסים להבין מה אומר כל דבר, ואיזה פקודה אנחנו רוצים לעשות.

לטובת זה אנחנו עוברים על עץ התחביר שקיבלנו, מעטרים אותו, וממנו מפיקים את קוד הביניים התלת מעני, שהוא הרבה יותר מוכוון משימה (ללא הערות וחלקים מיותרים). בשלב הזה, אנחנו ננסה כמה שאפשר לעשות אופטימיזציה לקוד. כאשר בשונה מהאופטימיזציה הקודמת שהיתה "לא-תלוית מכונה", והוגדרה בעיקר להורדת כפילויות וכדו', כאן אנחנו מתעסקים יותר עם אופטימיזציה של מכונה – הקצאת רגיסטרים, תזמון הקצאה ועוד. למה זה כל כך חשוב לנו? כידוע, אנחנו לא רוצים לגשת כל רגע ולשלוף מידע מהזיכרון הראשי, כי זה לוקח לנו המון זמן, אלא מעדיפים לשמור את המידע החשוב קרוב אלינו. האופטימיזציות שנעשה כאן, ידאגו לשמור את המידע החשוב קרוב אלינו, כמה שניתן.

בשביל לעבוד בצורה יעילה על האופטימיזציה, אנחנו נחלק את הקוד לבלוקים (נראה בהמשך איך בדיוק) וננסה לעבוד על כל בלוק בפני עצמו, כך שבסוף נקבל את התוצאה האופטימלית הסבירה ביותר.

כלים לקומפיילרים

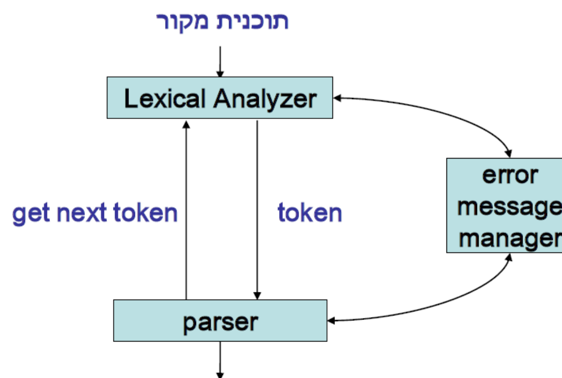
תכלס, אף אחד מאיתנו לא הולך לכתוב קומפיילר בחיים האמיתיים, לפחות לא מאפס. מי שבכל אופן רוצה / צריך לכתוב קומפיילר משתמש בהרבה כלים מובנים שיש וקיימים בשוק, ולהם עושה את ההתאמות הנדרשות. יש כלים שמתעסקים עם פארסינג, עם ניתוח לקסיקאלי ואפילו כלי אופטימיזציה שעושים עבודה של מחיקת קוד מת ועוד.

עכשיו, אחרי שראינו הכל מלמעלה, נתחיל לעבור על כל המודולים השונים, ונתחיל עם -

ניתוח לקסיקלי

חשוב מאוד שנבין מה אנחנו רוצים לעשות בכל שלב, מה מיוחד בו, מה אנחנו בדיוק דורשים שיקרה וכו' בשביל לעמוד על ההבדלים שבכל שלב ושלב. נזכיר רק, שהניתוח הלקסיקלי מקבל רצף של מחרוזות וממנו הוא מוציא את האסימונים השונים.

בצורה מפושטת מאוד, ניתן להראות את פעולת המנתח באופן הבא –



נכנסת תוכנית מקור, והמנתח פולט אסימון, ומחפש את האסימון הבא, תוך כדי הוצאת הודעות שגיאה. תכלס, הפארסר עובד רק בסוף ולא באמת ביחד עם המנתח, מאחר ובחלק מהמקרים אנחנו נצטרך מה שמכונה אצלנו Lookahead, שזה אומר להסתכל יותר מאסימון אחד בכל פעם. אבל זה הרעיון הכללי – מוצאים אסימון, מחפשים אסימון.

עכשיו נדבר על שאלה שעלולה לצוץ בהמשך בכל שלב נוסף – למה בכלל אנחנו מפרידים את המודולים השונים?

ספציפית לרגע הזה, נשאל את השאלה לגבי הניתוח הלקסיקלי והסמנטי – הניתוח הלקסיקלי מתקבל לנו על ידי אוטומטים, שהם כידוע ביטויים רגולריים המבטאים שפה רגולרית. ומהצד השני של הבמה, רמה מעליו בסולם השפות של חומסקי – הניתוח הסמנטי מגובה בדקדוק חסר הקשר. אממה? כל מי שעבר אוטומטים, ולא משנה באיזה ציון, אמור לדעת ששפה רגולרית היא מקרה פרטי, או מוכלת, בדקדוק חסר הקשר. ונחזור לשאלה – למה לא להרביץ הכל במכה אחת?

ברמה הרעיונית – כשאנחנו לוקחים את הניתוח הלקסיקלי, אנחנו מסתכלים על א"ב שמורכב מתוים שונים – א"ב ממש, סוגריים, מספרים וכיו"ב, והמילים של השפה הם בעצם האסימונים שאנחנו מחברים. ולעומת זאת, הניתוח הסמנטי מקבל תוים שהם אסימונים, והמילים של השפה הזאת זה בעצם תכנית (זה קצת קשה להבנה, אבל בעצם כמו שהתוים מרכיבים מילה בעלת משמעות כזו או אחרת, אוסף מילים מרכיבות משפט, או במקרה שלנו תכנית וזה עובד בדיוק על אותה הקבלה). כך שבעצם, אנחנו מחפשים שני דברים שונים לגמרי. אם כן, יבוא השואל וישאל – מכפתלי? נריץ הכל כדקדוק ח"ה, נוציא אסימונים ונריץ שוב, למה לשנות תהליכים?

התשובה לזה היא כבר ברמה הפרקטית יותר – כמו שלא כדאי להביא סכין לקרב אקדחים, ככה גם לנסות להרוג יתוש עם שוטגאן זה לא ממש יעיל. הכלי של האוטומט הוא מאוד פשוט וקל (רצף של תנאי if), לעומת הדקדוק ח"ה שהרבה יותר מסובך.

מעבר לזה – אנחנו כמובן רוצים להישאר מודולריים, ככל שנוכל לפרק את הדרך למספר תחנות, ככה נוכל להרכיב לנו כלים שנוכל לעשות הכל יותר יעיל, ולהשתמש במנתחים האלה גם במקומות אחרים.

דבר נוסף – מאחר וכל חלק מחפש דברים מסוימים יותר, כך אנחנו יכולים למקד את האלגוריתמים שיהיו מתאימים יותר בצורה מדויקת למה שאנחנו רוצים.

מושגים בסיסיים

אנחנו מתעסקים בניתוח הלקסיקלי עם שלושה מושגים עיקריים –

<u>Lexeme</u> – "סדרת אותיות המופרדת משאר התוכנית באופן מוסכם". כלומר, התוכנית שאנחנו מקבלים תגיע לנו רצף ענק של סטרינג, והמשימה שלנו היא להפריד אותה לחלקים השונים הקטנים ביותר האפשריים שבלתי ניתנים להפרדה, ועדיין בעלי משמעות. בדרך כלל, הפרדה תהיה עם רווח או ירידת שורה, אבל לפעמים זה יכול להיות גם צמוד למחרוזת רגילה – int a(max) לצורך העניין, יצטרך להפריד את המילה הארוכה לשלושה סמלים שונים), a, max, זה על ידי ההבנה הפשוטה שפתיחת סוגריים תקבל אחר כך משמעות בהקשר הסמנטי.

<u>Pattern</u> – "כלל המגדיר אוסף של מחרוזות". אנחנו רוצים למיין את כל הלקסמות שנקבל לפי משפחות שונות. כל לקסמה תמופה ישירות לקטגוריה מסוימת ותקבל יחס מתאים בהמשך. למעשה, בשביל לקבל מידע על כל התבניות, אנחנו נקבל חוקים או ביטויים רגולריים שיגדירו את כל המשפחות השונות. למשל, כל שמות המשתנים/פונקציות/מחלקות/ווטאבר יקבלו קיטלוג של TOK_ID, כלומר TOK כקיצור ל-Token, ויוגדרו כזיהוי של משהו. אנחנו בשלב הזה לא יודעים וגם לא אכפת לנו כל כך מה, אבל אנחנו מתייחסים אליו כשייך למשפחה הזאת.

<u>Token</u> – "זוג של שם (pattern) ותכונות". כדאי לשים לב, לא מדובר על תבנית ולקסמה, אלא על שם ותכונות, חלק מהתכונות זה הלקסמה, שזה המילה עצמה, אל כל השאר זה מידע רלוונטי על אותו אסימון. למשל – מיקום בשורה, טיפוס הנתונים שלו (יתווסף רק בהמשך הריצה), ועוד מידע שרלוונטי רק לגבי האסימון הספציפי הזה.

תפקידי המנתח המילולי

- קודם כל, להפריד את הטקסט המתקבל ללקסמות שונות. אילו לקסמות אנחנו עלולים לקבל? כמובן שיהיו לנו מחרוזות, מספרים, תוים שונים, אופרטורים של תו אחד או יותר ועוד. איך בעצם אנחנו יודעים להפריד בין 4 (מספר) ל-"4" (מחרוזת)?
  לפני שאנחנו מתחילים את הריצה על הקלט, אנחנו דוחפים לתוך טבלת הסמלים את כל המידע שיש לנו על מילים שמורות בשפה, ובהגיע תור לקסמה ולקסמה, אנחנו קודם כל בודקים בטבלה, האם זה מילה שכבר יש לנו, למשל אם נקבל את המילה int אנחנו נדע לשייך אותה ל-TOK_INT, ואם אנחנו מקבלים מילה חדשה שעוד לא הופיעה, כמו למשל "counter" אנחנו שומרים אותה כ-TOK_ID. בצורה זאת נוכל אחר כך לזהות שיש לנו מספרים שונים ומחרוזות על פי ההקשר.
- זיהוי התבניות השונות – על ידי שימוש בטבלת הסמלים, כאמור.
- טיפול וזיהוי של keywords ו-reserved words. יש לנו בכל שפה את אוסף המילים שאנחנו מגדירים כ"מילות מפתח" (Keywords), והם כל המילים שהם חלק מהסינטקס ומבצעים פעולות שונות. לעומתם יש לנו את המילים שמוגדרים כ"מילים שמורות", מאחר ויש להם משמעות כלשהי. ברוב המקרים, מילים שמורות הם גם מילות מפתח ולהיפך, אבל יש תמיד כמה יוצאים מן הכלל – למשל, המילה "goto", היא מילה שמורה בג'אווה, כך שלא נוכל לקרוא שם של פונקציה או משתנה בשם הזה, אפילו שזה לא פקודה שאנחנו באמת יכולים להריץ. בשפה Fortran לעומת זאת, אנחנו יכולים להשתמש בשם "if" בשביל לבטא שם של משתנה, על אף שהמילה הזאת משמשת גם כתנאי לכל דבר, והרי לנו מילת מפתח שאינה שמורה.
- מימוש של PreProcessing – דבר שקצת הזכרנו קודם – הכללה של קבצים שונים שביקשנו לעשות להם include, פתיחה של מקרואים (תזכורת define max 5# ישנה לנו כל מקום שכתוב בו max להיות 5).
- ספירת מספר השורות בתכנית – הזכרנו שחלק מהמידע השמור באסימון הוא מיקום המילה בקוד, ולכן יש לספור את השורות ובזאת לסייע לנו.
- דיווח על שגיאות – יש לשים לב, המנתח התחבירי לא עף לנו החוצה אם הוא מזהה מילה לא תקנית. הוא פשוט שולח אסימון שגיאה, ומקווה שזה יסתדר בהמשך (ספוילר: זה כנראה לא יסתדר בהמשך, אבל לפחות נדע איפה קרוב לוודאי שהתחלנו לטעות).
- הדפסת פלט – כל מיני אזהרות והודעות רלוונטיות שכדאי לעבור עליהם בשלב זה.

Typical Tokens

סוגי האסימונים השונים ימויינו למספר משפחות שונות. יש כמובן דיון בשאלה עד כמה אנחנו נכליל את המשפחות השונות – האם להגדיר את כל האופרטורים השונים באותה משפחה? הישבו חיבור וכפל כאחד? בגדול, אנחנו רוצים שהמשפחות יהיו מאותה הירכיה. למה הכוונה?

דיברנו על כך שאנחנו רוצים לעשות עץ תחביר, והצמתים בעץ יוגדרו על פי ההירכיה של האסימונים – אם יש לנו a+b, אז + יהיה האבא, ושני הבנים שלו יהיו המשתנים, אבל אם לא יהיה הפרדה בין הכפל והחיבור אנחנו עלולים לעשות בלאגן בעץ, ויותר חמור מזה – זה יגרור טעות בחישוב.

אם כן, הסיווג של האסימונים הוא כדלקמן –

- מילים שמורות בשפה – כל מילה שמורה תקבל Token משלה. מאחר ויש לנו משמעות שונה עבור כל מילה, נצטרך לזהות ישר על איזה מילה שמורה אנחנו מדברים.
- אופרטורים – כאן יש לנו לפעמים חלוקה לתתי משפחות כמו אופרטורי השוואה (=<, =! וכדו') שכולם תחת אותה הירכיה יהיו ביחד, ויהיו לנו גם אופרטורים שונים עבור האופרטורים המתמטיים.
- מזהים – כמו שהזכרנו קודם, כל שם של משתנה או מחלקה או כל הדומים להם יאוחדו תחת TOK_ID.
- קבועים – יהיו לנו מספר אסימונים שונים לסוגים שונים של קבועים – מספרים שלמים ורציונאליים, מחרוזות וכו'.
- סימני פיסוק – כל סימן פיסוק יקבל אסימון משל עצמו. אנחנו צריכים להיות מסוגלים לזהות גם פתיחת סוגרים וגם סגירת סוגריים, שיביאו לנו משמעויות שונות ופונקציונליות מסוימת, ולכן נפריד אותם.

טיפול במחרוזות שאינן ID

מעבר לכל הקוד החשוב, יש לנו הרבה פעמים קוד שלא נצרך לרמת המכונה. הערות שונות שהמתכנת כותב או פקודות Include שונות, וכן גם טאבים ורווחים. כל הדברים האלה נזרקים לפח. אמנם יש לנו טיפול של preprocessor על פקודות מאקרו, אבל אנחנו לא צריכים להשאיר את המילים בעצמם. לקומפיילר פחות חשוב אם שינית חלק מהטקסט בעקבות מאקרו, הוא יודע מה שהוא רואה ולפי זה הוא פועל.

אותו עניין לגבי הערות בקוד – זה שאנחנו מסבירים בהערה שהפונקציה sum עושה חיבור בין שני מספרים, כי ככה למדנו שעושים, לא באמת אכפת לקומפיילר. הוא רואה פתיחה של הערה וישר מתחיל לרוץ עד מה שהוא יזהה כסיום הערה.

לבסוף, כל המידע החשוב ישמר בטוקנים, וישמש בהמשך לתהליך הקימפול או להודעות השגיאה על הקוד שכתבנו. את כל התכונות אנחנו נשמור ביחד עם מספר הכניסה בטבלה – יכול להיות שנגדיר משתנה שנשתמש בו עשרות פעמים בתוכנית. במקום לשמור אותו בכל פעם מחדש, אנחנו פשוט בכל אסימון נשמור איזה רפרנס למיקום שלו בטבלה, ובכל פעם נוציא אסימון חדש על כל פעם שניתקל בו.

הקושי בניתוח הלקסיקלי

לפעמים אנחנו עלולים להתקל בבעיות מסוימות בחלוקה לטוקנים השונים. לפעמים אותם תוים עלולים לפי ההקשר להיות שימושים שונים לגמרי -קחו לדוגמא את המילה "לבנה", זה יכול להיות ירח, זה יכול תיאור צבע של שם עצם מסוים, זה יכול להיות "לבן שלה" או אפילו גבינה חמוצה, אבל מאחר שאנחנו לא מתעסקים פה עם סמנטיקה אנחנו עלולים לפספס פה את המשמעות ואת הניתוח. נראה מספר דוגמאות פשוטות בשביל להבין את הבעיה. ברוב השפות שאנחנו מכירים היום, אנחנו אוטומטית נפריד לקסמות ברגע שנקבל רווח, אבל יש שפות שמתעלמות מרווחים, כמו למשל השפה הנפלאה ורבת השימוש Fortran. בה אם נכתוב Do 5i זה עלול להתפרש גם כקריאת משתנה בשם 5i וגם כתחילת לולאה. או אם ניקח דוגמא נוספת מ-Ada או פסקל, שם הגדירו את הלולאה שתרוץ

בטווח מסוים שמוגדר מסביב שתי נקודות, כלומר 1..10 מגדיר טווח בין אחד לעשר. הבעיה שנוצרה היתה שאם אנחנו עכשיו מתירים לעשות שברים בצורה של 1. או 10. (שמבטא 0.1), הקומפיילר לא ידע האם הביטוי 1..10 בא לבקש את הטווח הנתון, או מדבר על שני מספרים עשרוניים שכתובים בצורה עילגת.

נראה בהמשך איך אנחנו מנסים לפתור בעיות שכאלה.

ניסוח ה-Tokens המותרים בשפה

הדרך הסטנדרטית והפשוטה ביותר להכריז על אסימונים, הוא על ידי אוטומט. האוטומט ייצג ביטוי רגולרי שסורק את הקלט הנתון ומזהה את הטוקנים השונים.

מן הסתם, אין צורך להתעכב על מה זה אוטומט, אבל כן נעצור רגע, וננסה להרחיב בקטנה על משפחות של ביטויים. בשביל לפשט כמה שניתן את הביטויים, אנחנו נתחיל ברמה הבסיסית ביותר של המשפחות ונגדיר את הקבוצות הבאות (שהן לא מחייבות, אבל די מוסכמות על כולם):

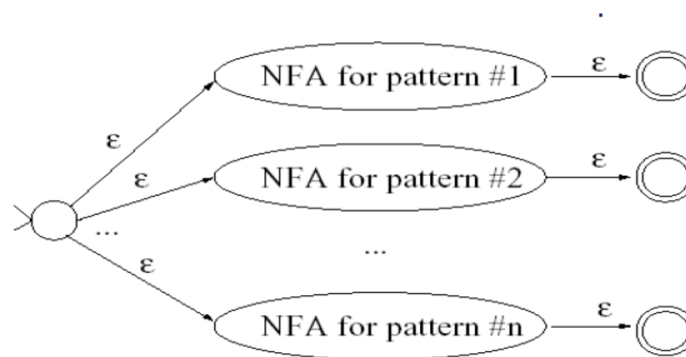
Letter = a | b | … z | A | B…Z
Digit = 0 | 1 |.. |9

כאשר בדרך כלל פשוט עושים תחום עם מקף (a-z).

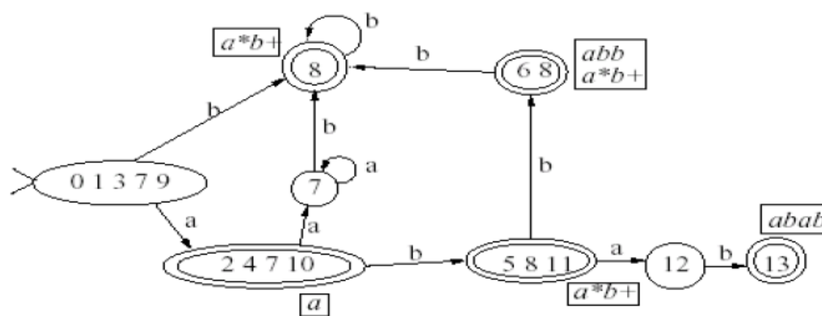
בדרך כלל, שתי הקבוצות האלה הן הבסיסיות ביותר, ועליהן עושים הרכבות ושינויים קלים, כך שלמשל נוכל לבקש רק אותיות גדולות, או מספרים רציונלים על פי המבנה שאנחנו כבר מכירים –

CAP_Letter = {A-Z}
Real = $\{digit\}^{+}$"."$\{digit\}^{+}$

את השפה שלנו אנחנו נגדיר בצורה של אוסף ביטויים רגולריים שכאלה, ובעבור כל אחד מהם נפתח אוטומט לא דטרמיניסטי, כך שהאוטומט הכללי של השפה – ייראה בערך כך –



ובהמשך נוכל להעביר את כל החלקים השונים לצורה דטרמיניסטית ולנסות לאחד הכל לאוטומט יפה וממצה –



Lookahead

כאמור, לפעמים אנחנו לא יכולים להסתפק רק בקריאה של אסימון בודד. נזכיר שוב את השפה Fortran, בה אנחנו יכולים להגדיר את המילה "if" גם כתנאי וגם כשם משתנה. במקרים כאלה אנחנו צריכים לקרוא קצת קדימה בשביל להבין הקשר ברמה המינימלית. הבדיקה הזאת נקראת Lookahead, והיא פשוט מעבר של מספר אותיות קדימה (בדרך כלל לא יותר מ-2 אותיות), וניסיון להבין משם לאן הקוד הזה מתקדם.

טיפול בשגיאות

בשלב הזה, קשה מאוד לזהות ולטפל בשגיאות. למשל, "fi" יכול להיות שם של פונקציה, וגם טעות בהקלדה. המנתח הלקסיקלי לא חושב על שיבושים של מילים ולכן פשוט יגדיר את זה כ-ID. אבל לפעמים הבעיה הרבה יותר מסובכת, וצריך למצוא איזה שיטה שעל ידה אנחנו נוכל לטפל בשגיאות.

הדרך הקלה ביותר לטיפול – השמטה של כל הקלט עד שנגיע לרמץ שברור לנו שמדובר בלקסמה חדשה.

דרך הרבה יותר מסובכת – לנסות לשחק עם הלקסמה השגויה, ולראות אם שינוי קטן מביא לנו איזה לקסמה בעלת היגיון כלשהו.

בגדול, אנחנו לא רוצים להתקע או לזרוק יותר מידי קוד, אלא לנסות כמה שאפשר לטפל בשגיאה בצורה מקומית ולהמשיך הלאה.

סיכום

על מנת לזהות לקסמות אנחנו כותבים ביטויים רגולריים. את הביטויים אנחנו ממירים לאוטומטים לא דטרמיניסטיים, ואז מאחדים הכל. לאחר מכן אנחנו מנסים לעשות את האוטומט הזה דטרמניסטי, וככה אנחנו יכולים לאסוף את הלקסמות על הדרך ובכל זיהוי של לקסמה, להוציא כפלט גם את התבנית המתאימה לאותה לקסמה.

המנתח המילולי מצגת דוגמאות מס' 1

מצגת הדוגמא הזאת מביאה לנו את הדרך להבנה של המנתח הסמנטי מההתחלה עד הסוף. כבר דיברנו על זה שהמנתח עובד עם ביטויים רגולריים ואוטומטים, ועכשיו נראה את הדרך השלמה להוצאת אסימונים, וכן את המימוש של האוטומט בצורת תוכנית.

[2]

נתחיל עם הביטויים הרגולריים שייצגו לנו את השפה –

CapLet = (‘A’ + ‘B’ + … + ‘Z’)
SmLet = (‘a’ + ‘b’ + … + ‘z’)
Underscore = ‘_’
IdComp = CapLet SmLet*
Id = IdComp (Underscore IdComp)*

ניתן לראות מעצם חלוקת הביטויים, שיש לנו כאן שפה שהיא Case-Sensitive, כלומר יש הבדל בין אות גדולה לקטנה (אחרת היינו עושים ביטוי רגולרי לכל סוגי האותיות כאחד). שם משתנה מורכב (IdComp) הוא בעצם רצף תווים שהאות הראשונה היא אות גדולה וכל השאר (אפס או יותר) הינן אותיות קטנות. והגדרה של ID יכולה להיות החל משם אחד, כאשר על כל שם נוסף נפריד ביניהם עם קו תחתון. עד כאן אין לנו חידוש ממשי.

כך שביטויים כמו הנידונים -Legal_Identifier, A_Good_Example הינם חוקיים בשפה, ויש לנו אוסף של ביטויים אחרים שאינם חוקיים מסיבות שונות ומשונות –

Illegal_identifier – השם השני מתחיל באות קטנה.

A_bad_Example – כנ"ל.

Not__This_Way – יש הפרדה כפולה אחרי השם הראשון (צריך להיות רק קו תחתון בודד).

Nor_That_One_ - קו תחתון מיותר אחרי השם השלישי.

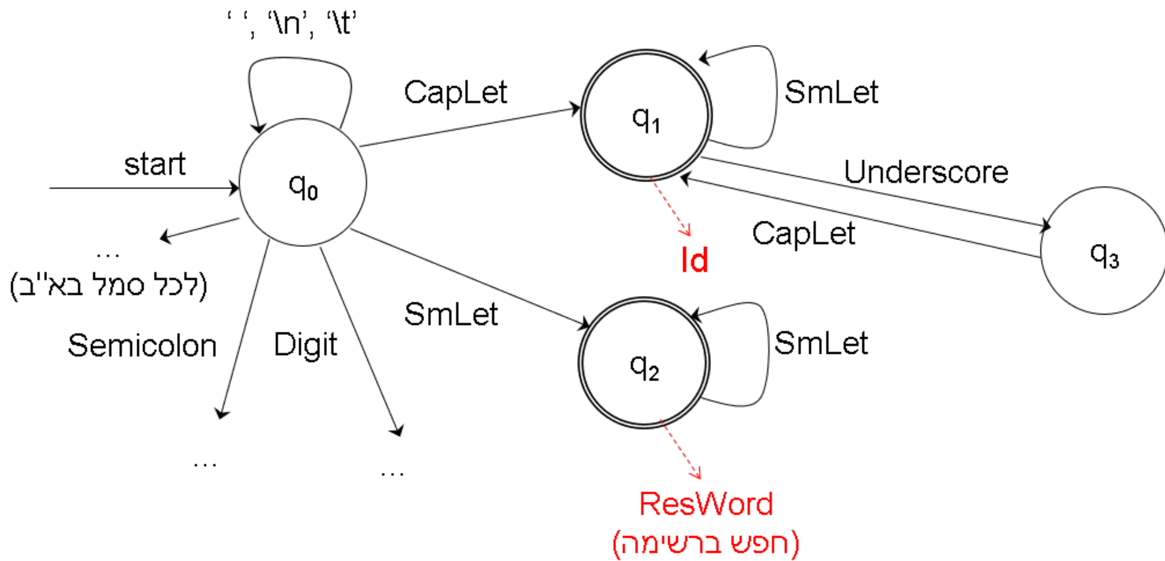
_surely_NOT_tHat – קו תחתון לפני השם הראשון, שם ראשון מתחיל באות קטנה, שם שני רק באותיות גדולות, שם שלישי מתחיל באות קטנה ולאחריו אות גדולה.

עד כאן הבנו את הגדרת שמות המשתנים בשפה, כאשר נצא מנקודת הנחה ששאר המילים השמורות הם כמו בשפת C++.

[3]

מתחילים לבנות את האוטומט – אנחנו מתחילים כרגיל מ-q0, ומגדירים כל מעבר על ידי רצף הביטויים הרגולריים כמו שאנחנו מכירים – כל תו שעלול להוביל אותנו לאסימון מסוים יפנה למצב אחר. אנחנו כרגע מתעסקים רק עם תווים של אותיות - משתנים ומילים שמורות.

הדבר שקצת שונה פה מאוטומטים רגילים שלמדנו, הוא שהבדיקה היא לא "האם המילה שייכת לשפה או לא?" ואם לא אז בעיה שלה, יש לנו פה טקסט שאנחנו אמורים להתייחס לכולו פחות או יותר, ולכן גם אם המילה לא "תתקבל" אנחנו צריכים להוציא פלט מסוים.



שני המצבים שאנחנו נתעסק איתם וחשובים לנו כרגע, זה קודם כל הגדרת ID – אנחנו מתחילים עם מצב מקבל q1, וכל עוד אנחנו עומדים בתנאים אנחנו בסדר. ברגע שתסתיים קריאת המילה הנוכחית – כלומר, נקבל איזה שהוא תו "זר" שאינו שייך לביטוי הרגולרי שהגדרנו, כמו מספר או רווח או ירידת שורה, אנחנו נעצור את הריצה ונבדוק אילו תוים בדקנו, ובאיזה מצב אנחנו עומדים – אם המצב שלנו מקבל, נוציא את האסימון המתאים, ואם לא נוציא אסימון שגיאה.

מבחינת q2, העבודה לא מסתיימת ברגע שהמילה בסדר, אלא אנחנו צריכים לבדוק שהמילה הזאת, שלכאורה שמורה, באמת נמצאת ברשימת המילים השמורות. מה קורה אם לא? ככל הנראה נוציא גם על זה הודעת שגיאה ונקווה שזה יטופל בהמשך.

[4]

כאן יש לנו ניתוח של קטע קוד פשוט מאוד – תנאי if קצר –

```
1. if (Var_A >= Var_B) {
2.   Var_C = Var_A;
3. }
```

שימו לב שמיספור השורות חשוב גם הוא- דיברנו על כך שכל אסימון מגיע עם מספר השורה הרלוונטית. זה נעשה למקרה שיש איזה שגיאות בהמשך, וככה נדע לאן לקפוץ לבדוק.

אם ככה, רצף האסימונים שיוצג לנו הוא כזה –

```
TOK_IF, (1)
TOK_LPAREN, (1)
TOK_ID, (1, Var_A)
TOK_RELOP, (1, GE)
TOK_ID, (1, Var_B)
TOK_RPAREN, (1)
TOK_LBRACE, (1)
TOK_ID, (2, Var_C)
TOK_ASGN, (2)
TOK_ID, (2, Var_A)
```

TOK_SEMICOL, (2)
TOK_RBRACE, (3)

המילים השמורות והאופרטורים יופיעו כאסימון פרטי (IF, ELSE, ASGN), יהיה לנו אסימונים לפתיחות וסגירות של סוגריים מסוגים שונים (LPAREN), ושמות ה-ID יופיעו כאסימון ID כללי. כזכור, האסימון מגיעים כל אחד עם תבנית מתאימה, ועם הפרטים, אך אם יש לנו מילים שמורות או סימנים רלוונטיים אין לנו מידע נוסף שאמור להירשם, אך אם יש לנו ID, אנחנו צריכים לדעת מה בדיוק ה-ID שאנחנו שומרים פה, ולכן אנחנו מכניסים אותו אחרי מספר השורה.

[5]

כאן יש לנו רצף של מילים שעלולים ליצור בעיות, ואיך אנחנו מתמודדים איתם ועם פליטת האסימונים –

VarA

זה מקרה קל יחסית – ה-A הגדולה שמופיעה לנו היא תו זר לאותו מצב, אבל יכול להיחשב כ-ID בפני עצמו, ולכן אנחנו מוציאים שני אסימונים "חוקיים" –

TOK_ID, (*line#*, Var)
TOK_ID, (*line#*, A)

אותו דבר גם במקרה הבא –

ifVar_A

כל עוד יש לנו אפשרות להפריד באופן חוקי, הכל בסדר. יש לזכור – אנחנו לא מחפשים להפריד עם רווח, ברגע שיש לנו תו שלא מתאים לתבנית, אנחנו עוצרים ומוציאים אסימון. –

TOK_IF, (*line#*)
TOK_ID, (*line#*, Var_A)

המקרה הבא מתחיל עם השגיאות –

Var_a

המשתנה יכול להיות מופרד עם קו תחתון, אבל צריך שיהיה שם אות גדולה אחר כך, כמו בדוגמא הקודמת. הדבר המעניין הוא, שזה מוציא לנו שני אסימונים של שגיאה ולא רק אחת –

TOK_ERR, (*line#*, Var_)
TOK_ERR, (*line#*, a)

_Var עדיין תקין מבחינתנו, אבל ברגע שמגיע ה-a שלא מתאים לתבנית, אנחנו עוצרים ובודקים מה יש לנו. אבל מאחר והמילה הנוכחית הקיימת לא מתאימה לביטוי הרגולרי, מבחינתנו זה שגיאה ונוציא אסימון. עכשיו נשארנו עם a. הוא הולך לרשימת המילים השמורות, והוא לא מוצא שם משהו מתאים, ולכן גם הוא יוציא אסימון שגיאה.

Var_AandB

הדוגמא האחרונה היא פשוטה למי שהבין עד עכשיו – B תקוע לנו בסוף. מצד שני, עד אליו הכל פרפרים וחדי-קרן, ואנחנו מקבלים את האסימון הזה כ-ID ורק נוציא שגיאה לתו B.

TOK_ID, (*line#*, Var_Aand)
TOK_ERR, (*line#*, B)

[6]
מימוש המנתח המילולי בתכנות ידני

עד עכשיו דיברנו בצורה די אמפירית על האוטומט והמצבים השונים. כל שנותר עכשיו הוא לדבר על איך כותבים את זה בקוד. בעצם בשביל להבין את הרציונל – כל מצב באוטומט בודק את האות בקלט שהוא עובד עליו, הוא יודע שיש לו אחת מכמה אפשרויות, ובעבור כל אחת מהן הוא עושה פעולה אחרת. Switch קלאסי.

כך שכל מצב הופך לרשימת switch, כל מעבר שיוצא מאחד המצבים האלה הוא בעצם אחד מהמקרים השונים עליהם הוא מגיב. כמו שאמרנו כבר, אנחנו צריכים גם לדאוג למצב של סיום מילה בקלט, ועל זה נכניס אפשרות מעבר ל-default, שיחזיר לנו את האסימון המתאים (מצב מקבל), או אסימון שגיאה (מצב לא-מקבל).

יש לזכור שכל תו שאנחנו קוראים נכנס ל-buffer ונשמר שם עד שנגיע לנקודה בה נצטרך להוציא אסימון, ואז נוציא כל מה שיש לנו.

מבחינת הקוד, אנחנו מתחילים עם הצורה הבסיסית ביותר – איך נגדיר token –

```
struct Token {
  Pattern pattern;
  unsigned int line_num;
  void *name;
}
```

קודם כל, אנחנו מדברים על סוג של תבנית מסוימת (לצורך העניין, נאמר שיש לנו Enum של כל התבניות האפשריות). בנוסף יש לנו את המידע הנוסף על כל אסימון – מספר השורה, ומצביע לטבלת השמות שם מאוכסן השם במקרה של ID.

```
# Global variables:
Token token;
unsigned int line_num = 1;

void set_token(Pattern pattern, void *name = NULL) {
  token.pattern = pattern;
  token.line_num = line_num;
  token.name = name;
}
```

השמה של אסימון מקבלת את התבנית שנמצאה, ואת שם המשתנה (שכברירת מחדל מוגדר NULL) ואז פשוט עושה השמה לכל המידע שיש לנו.

עכשיו הגענו למימוש המצבים (חשוב לשיעורי הבית ולמבחן) – נראה את המימוש של q0, וננסה להבין את הרעיון הכללי –

```
q0:
  switch (class(cur_char)) {
    case \n': line_num++;   // no break
    case ' ':
    case '\t': ignore_char(); goto q0;
    case CapLet: next_char(); goto q1;
    case SmLet: next_char(); goto q2;
    case Digit: …
    case Semicolon: …
```

```
  case EOF: set_token(TOK_EOF); return;
  case …
  …
  default: next_char(); set_token(TOK_ERR); return;
 }
```

אנחנו נכנסים לסוויץ' עם cur_char, כלומר התו הנוכחי אותו אנחנו מנסים לסווג. שלושת המקרים הראשונים הם די מנהלתיים ובדרך כלל לא מבקשים מאיתנו לממש אותם, אף חשוב להסתכל על הרעיון – קודם כל, '\n' (ירידת שורה) לא גורם לנו לבדוק משהו, אבל דואג להעלות את המספר של השורות. שתי השורות הבאות – ולמעשה גם ירידת השורה – מתעלמים מכל התווים האלה ועוברים ל-q0. כלומר, ממשיכים לחפש מההתחלה. השורות הבאות למעשה מתחילות להתעסק עם קלט רלוונטי, וברגע שאנחנו מזהים תו חשוב, אנחנו קודם כל מושכים את התו הבא (ומכניסים במקביל את הנוכחי ל-buffer), ואז עוברים למצב הבא.

נדלג על שורת ה-EOF, ונעבור רגע לשורת ה-default, נגיד שקיבלנו תו שלא רשום לנו ברשימת המקרים האפשריים. כזכור q0 הוא לא מצב מקבל, ולכן לאחר לקיחת התו הבא אנחנו מוציאים TOK_ERR בשביל לדווח על חריגה בקובץ.

כל המעבר בין המצבים ממשיך עד שאנחנו מגיעים ל-EOF, סוף הקובץ, ואז אנחנו סוגרים את כל מה שיש לנו, ומכריזים על סיום הקריאה.

נראה גם את המימוש של שאר המצבים –

```
q1:  // identifiers
 switch (class(cur_char)) {
  case SmLet: next_char(); goto q1;
  case Underscore: next_char(); goto q3;
  default:
   address = SymTable.insert_name(buf);
   set_token(TOK_ID, address);
 return;
 }

q2:  // reserved words
 switch (class(cur_char)) {
  case SmLet: next_char(); goto q2;
  default:
   pattern = ResWordTable.find(buf);
   set_token(pattern);
   return;
 }
q3:
 switch (class(cur_char)) {
  case CapLet: next_char(); goto q1;
  default: set_token(TOK_ERR); return;
 }
```

כמובן שיש עוד מצבים, אבל אנחנו לא מתעסקים איתם כרגע. המצב הראשון שמתעסק עם המשתנים מוגדר כמצב מקבל, ולכן בדיפולט שלו אין הכרזה על TOK_ERR אלא אם יש לנו תו שהוא לא אות קטנה או קו תחתון, מבחינתו הוא סיים את הקריאה. יש לזכור, שהאותיות הגדולות נכנסות לבאפר במצב 3 ולא פה. איך הוא מטפל באסימון שמתקבל? קודם כל, הולך לטבלת הסמלים ומכניס לשם את השם של המשתנה – כל התווים שנמצאים בבאפר. אחרי זה הוא מגדיר TOK_ID שאותו הוא שולח עם address, שזה בדיוק המצביע לשם שהגדרנו לפני רגע.

המצב השני, רץ על האותיות הקטנות, וכשהוא עובר לדיפולט, הוא קודם כל בודק האם התוים שבבאפר מרכיבים לנו איזה תבנית מוכרת – הוא שולח את הבאפר לפונקציה שתחזיר לנו את שם התבנית. אם לא יימצא פאטרן מתאים, יחזור כמובן TOK_ERR, ומה שלא יהיה – זה ייכנס להגדרת האסימון, מוצלח או פחות מוצלח.

המצב השלישי באמת פשוט – או הוא חוזר ל-q1, הוא שהוא מעיף ERR.

[14]

ברוב השפות אין הבדל ברמת האוטומט בין קריאת מילה שמורה ל-ID כלשהו, תחשבו על כל שפות ה-C שלמדנו והלאה, אין חובה לרשום משתנה עם אות גדולה כמו שהיה בדוגמא פה, אלא אפשר באופן חופשי לרשום הכל עם אותיות קטנות, ואז האוטומט שעשינו לא ממש רלוונטי.

לכן, את הסיווג לתבניות נעשה דרך טבלת הסמלים. מה הכוונה? קודם כל, בשביל לחסוך לנו זמן, נמלא את כל התבניות האפשריות (IF,ELSE), כאשר לכל TOKEN נכניס גם את ה"שם" שלו, למשל (TOK_IF, “if”), ואז כל מה שיידרש לנו הוא לרוץ על הטבלה ולראות אם יש לנו התאמה. אם יש התאמה – מה טוב, נחזיר את ה-pattern המתאים. אם לא, נכניס לטבלה את הלקסמה שמצאנו (כי הגיוני שניתקל בה בהמשך), ונחזיר TOK_ID.

כמובן, שאם יש הבדל כמו בדוגמה שעשינו, זה הרבה יותר קל ופשוט לשים שתי טבלאות שונות עבור המילים השמורות ועבור ה-ID השונים.

[17]

על מנת לחסוך לנו בחיפוש המילים, אנחנו משתמשים בטבלת מעורבלת, המכונה Hash Table. צריך לזכור, שבעבור שפות אמיתיות ולא הקקמייקה שאנחנו מתעסקים איתן בדוגמאות יש מאות מילים שמורות, וכמה שנעשה חיפוש יעיל, זה עדיין מלא זמן. לכן Hash יעזור לנו פה פשוט להציץ לנקודה בטבלה בגישה של O(1) ולבדוק אם מדובר במילה שמורה או ID חדש.

איך זה מתבצע, מגדירים פונקציית ערבול עבור התוים השונים. את המילים השמורות אנחנו קודם כל מכניסים לתוך טבלת שמות (מעין מערך ענק), ובין כל מילה דוחפים תו פשוט לרפד. כל מילה שמכניסים לטבלת השמות עוברת ערבול, ומוכנסת גם לטבלת הסמלים, עם מידע נוסף של המיקום של המילה בטבלת השמות. כמובן, שאם במקרה אותו תא בטבלה כבר מלא, נבנה רשימה ונשרשר הלאה את השמות.

לצורך הדוגמה, יש לנו שפה שנראית כמו C, עם המילים השמורות הבאות - float, int, return.

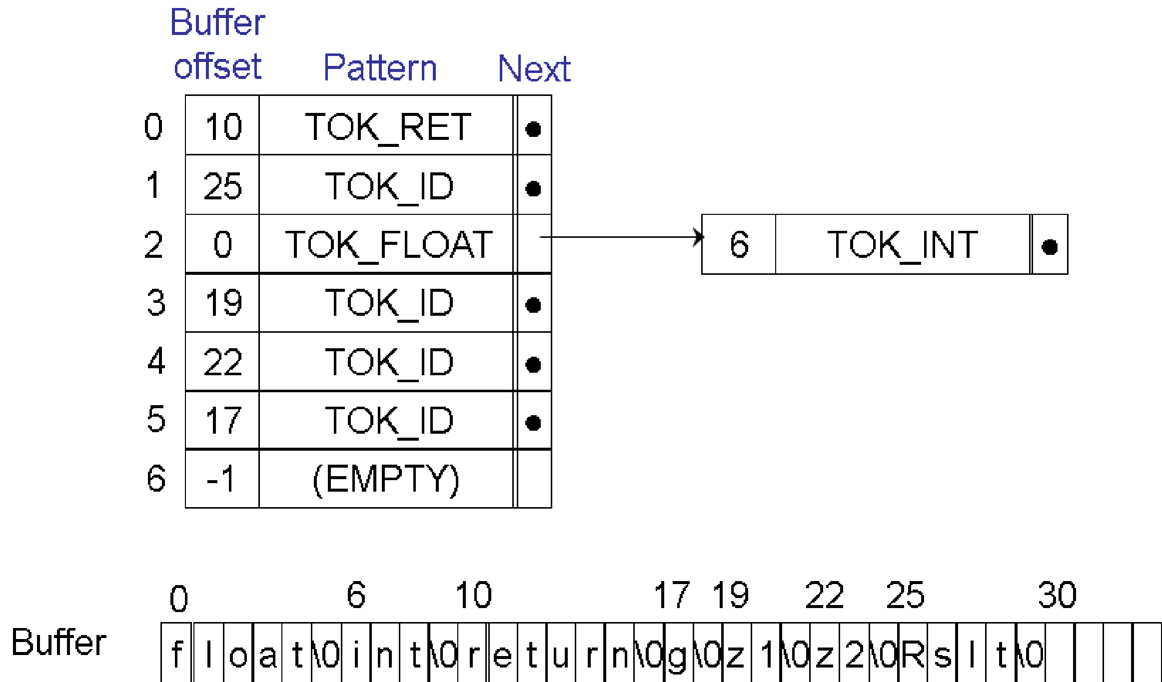
בעבור השפה הזאת קיבלנו קלט לניתוח –

```
float g(int z1, int z2) {
  float Rslt;
}
```

קודם כל, נגדיר את פונקציית הערבול – בעבור כל רצף תווים נעשה סכימה של ערך ה-ASCII שלהם, ונחלק אותו מודולו בגודל הטבלה (שתוגדר כמובן כמספר ראשוני בשביל לצמצם התנגשויות). למשל – עבור השם z2 המופיע בסוף השורה הראשונה נעבוד בסדר הנ"ל –

```
ASCII_SUM(“z2”) = ASCII(‘z’) + ASCII(‘2’) = 122 + 50 = 172
hash_func(“z2”) = ASCII_SUM(“z2”) % TABLE_SIZE = 172 % 7 = 4
```

אחרי שעשינו את כל זה, נאתחל את טבלת השמות, ונתחיל לשלב את המילים השמורות, ולערבל אותם לתוך הטבלה, ואז לעבור על הקלט ולהכניס את כל ה-ID שאנחנו מוצאים שם. בסוף הטבלה תיראה באופן הבא –



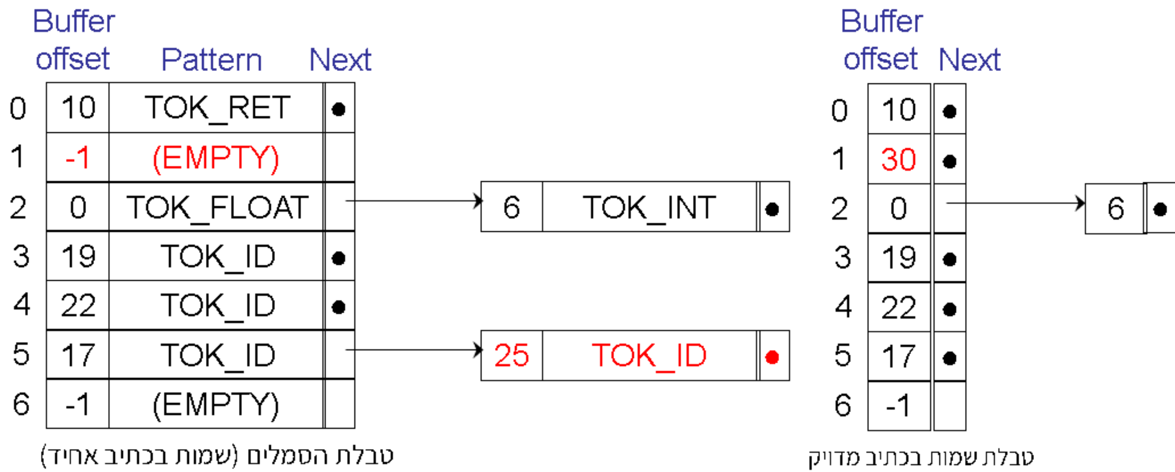
שימו לב שמשמאל לכל Pattern מופיע האופסט – מה המיקום של המילה בתוך המערך. השורה האחרונה האופסט מוגדר להיות 1-, שזה כמובן האופסט הדיפולטבי שאנחנו מכניסים בתחילת הריצה כסימון של מקום ריק.

[21]

הנושא האחרון עליו יש להעיר – שפות שאינן רגישות לגודל האות. רוב השפות המודרניות לא יתייחסו ל-ADDR ו-Addr כאותה מילה. אבל שפות ישנות יותר (שכל פעם שמזכירים אותם בקורס הזה, הן משתעלות בקבר), לפעמים לא היו רגישות לזה ומבחינתם הכל אותו דבר.

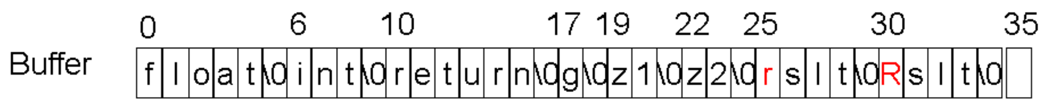
זה כמובן יוצר לנו בעיה – אם אנחנו מקבלים את כל הווריאציות האפשריות עבור כל מילה, איך נוכל לדעת כשנקבל מילה האם ראינו אותה כבר או לא? אפשרות נפלאה היא פשוט לדחוף כל פעם את כל הווריאציות האפשריות וככה לסתום את הזיכרון (ותחשבו שאנחנו מדברים לפני 30 שנה שזיכרון במחשב היה משאב שלא זלזלו בו). הפתרון שהוצע, אמר כך – הכתיב השונה של הלקסמה חשוב רק למשתמש/מתכנת. ברמת התוכנה לא אכפת לנו אם יש לנו שם נכון או בכלל, אלא רק שזה יסווג נכון. לכן, בעבור האסימון עצמו, שגם ככה מיוצר עבור כל מופע של המילה נכניס שני דברים – הכתיב של השם, כמו שהופיע בקוד, והכתיב הנכון – שאותו אנחנו מגדירים מראש להיות בצורה אחידה – או כתיב באותיות גדולות או בקטנות.

עכשיו ניצור שתי טבלאות סמלים – הכתיב האחיד, והכתיב המדויק (איך שהופיע בקוד), וכל לקסמה כזאת תישמר בשני המקומות כאחד –



טבלת הסמלים (שמות בכתיב אחיד)

טבלת שמות בכתיב מדויק



נוכל גם לראות פיענוח פשוט שמתאים לאיור –

1. float g(int z1, int z2) {
2. float Rslt;

ואת רצף האסימונים המתאים –

TOK_FLOAT, (1, -, 0)
TOK_ID, (1, &entry(g), 17)
TOK_LPAREN, (1, -, -)
TOK_INT, (1, -, 6)
TOK_ID, (1, &entry(z1), 19)
TOK_COMMA, (1, -, -)
TOK_INT, (1, -, 6)
TOK_ID, (1, &entry(z2), 22)
TOK_RPAREN, (1, -, -)
TOK_LBRACE, (1, -, -)
TOK_FLOAT, (2, -, 0)
TOKID, (2, &entry(rslt), 30)
TOK_SEMICOL, (2, -, -)

כאן האסימונים שיזוהו כ-ID ייכנסו בשני אופנים - &entry שיישלח להכנסה את המילה בכתיב אחיד (ראו השורה המודגשת), והמיקום אופסט שיוגדר לו יהיה הכתיב המדויק שהופיע בקוד, וכולם שמחים ומאושרים.

ניתוח תחבירי – Syntax Analysis

עד עכשיו ראינו מה המנתח הלקסיקלי עושה – לוקח תווים ומחלץ מהם לקסמות והפלט הסופי שלו הוא אסימונים (tokens). הניתוח התחבירי לוקח את זרם האסימונים שהגיע מהמנתח המילולי, ומוציא לנו את התוכנית בצורה של עץ גזירה. יש לזכור – אנחנו מתעסקים פה עם הטוקנים עצמם, כלומר מבחינתנו משפט כמו a = b + c, אנחנו נקרא אותו בתור ID ASGN ID OP_ADD ID, או משהו בסגנון. בשלב הזה אנחנו פחות מתעסקים עם מה זה כל ID כי זה פחות רלוונטי עבורנו. הבדיקה כאן היא רק חוקיות של מבנה המשפט.

בהשאלה משפות אנושיות, דיברנו על כך שמבחינת המנתח המילולי אם נקבל רצף של מילים כמו "משה" "הלך" "הלך" "לים", זה תקין לגמרי מילולית כי כל מילה עומדת בפני עצמה. אך מבחינת התחביר של המשפט "משה הלך הלך לים" זה לא משפט שמחובר כמו שצריך. באותו אופן אם אנחנו נקבל ID ID ID ID TOK_INT, זה תקין לקסיקלית, אבל בוודאי שאין בזה נכונות תחבירית.

ברמה קצת פחות תיאורטית, אנחנו יודעים שבעבור הגדרה של פונקציה יש לנו מספר דרישות מינימליות כמו סוג הערך המוחזר, שם הפונקציה וערכים מוכנסים, או ברמת האסימונים נרצה לקבל רצף של type name (args[]). על מנת שנוכל להגדיר לנו מראש את הערכים המתקבלים, אנחנו משתמשים בדקדוק חסר הקשר, ואז בעבור כל קלט שנקבל (תוכנית מסויימת) אנחנו ננסה לעשות את רצף הגזירה באחת מהשיטות השונות, ואם זה יסתדר לנו בצורה נכונה, נוכל להוציא את עץ הגזירה התחבירי.

אנחנו דורשים ושואפים להשתמש במחלקות של דקדוקים חסרי הקשר בצורה היעילה ביותר. מה הכוונה? אם אנחנו נקבל מחלקת דקדוקים שיכילו את שני הכללים הבאים –

A → ID = ID + ID
B → ID = ID * ID

אנחנו עלולים להתקל בבעיה – אם נקבל רצף טקסט ונתחיל לקרוא אותו, ומולנו יהיה a = b. בשלב הזה אנחנו עדיין לא יודעים על איזה כלל גזירה מדובר, אנחנו נצטרך לקרוא 4 אסימונים רק בשביל להגיע להכרעה של כלל הדקדוק המתאים לו. דבר זה כמובן מאוד בעייתי, כי אנחנו רוצים לעשות את המסלול שלנו בצורה דטרמניסטית, ואנחנו לא יכולים לרוץ על משהו שנחשוב שהוא כלל A רק בשביל לגלות שהוא בכלל B. כמובן, שכבר דיברנו על האפשרות של יצירת Lookahead, ולרוץ קדימה בשביל לקרוא את המקרים הבעייתים, אבל ברמת זמן הריצה זה מעלה לנו הכל בצורה ממש רצינית – אם על כל כלל נצטרך לרוץ ארבעה אסימונים זה ייסבך לנו את כל העבודה. מה שאנחנו נעשה (ונפרט על זה בהמשך) הוא לבנות את הדקדוק עצמו בצורה יעילה כזאת שלא ניתקל בשאלות והתחבטויות כאלה.

דקדוק חסר הקשר

דיברנו כבר בקורס אוטומטים על המחלק הזאת, וכרגע נעשה איזה תזכורת ונרחיב בדברים הרלוונטיים לנו. דקדוק ח"ה מורכב מהרביעייה G=(V, T, P, S), כאשר –

V– nonterminals, משתנים – יוגדר בתור אוסף של משתני הגזירה.

T – terminals, טרמינלים, tokens – מבחינתנו, רצף האסימונים עליהם אנחנו עובדים.

P – חוקי גזירה – המעבר מהמשתנים ל(משתנים אחרים ו)טרמינלים.

S – משתנה תחילי.

מושגי דקדוק ח"ה

- גזירה – סדרה של החלפות של אותיות לא טרמינליות תוך שימוש בחוקי הגזירה. שימו לב – לא דיברנו על החלפה למשתנים או לטרמינלים, כי שתי האופציות אפשריות ורלוונטיות.

- שפה - אוסף ביטויים הנגזרים מהמצב התחילי והמכילים טרמינלים בלבד. לאחר שנסיים לגזור את כל האפשרויות, אנחנו נגיע לשפה עצמה שתהיה מורכבת אך ורק מטרמינלים.
- תבנית פסוקית - תוצאת סדרת גזירות בה נותרו (אולי) לא-טרמינלים. למעשה, זה איזה "סנאפשוט" של מצב ביניים כלשהו החל מהשלב המוקדם ביותר של המשתנה התחילי, ועד למצב שיש לנו כבר שפה, וכמוכן כל מה שבאמצע.
- גזירה שמאלית - גזירה בה מוחלף בכל שלב הסימן השמאלי ביותר (באופן דומה – גזירה ימנית). גזירה לכיוון מסוים אומרת שאם החלטנו על כיוון אליו נגזור (לצור העניין – שמאל), ונתקל בתבנית פסוקית מסוג aAbBc, אנחנו נשאף לגזור קודם כל את ה-A, ואם תהיה לנו אפשרות אז אפילו נגזור את aA.

גזירה Bottom-Up

עד עכשיו אנחנו הסתכלנו על גזירה מלמעלה למטה – התחלנו מהמשתנה התחילי, וירדנו למטה עד שהיה לנו עץ שכולו טרמינלים. אבל כזכור, אנחנו מקבלים עכשיו רצף של אסימונים – כלומר, את הטרמינלים כבר יש לנו, ואנחנו רוצים לבדוק אם יש לצמצם הכל להגיע בחזרה למשתנה התחילי. בשביל לעשות את זה, אנחנו נסתכל על כללי הגזירה ונתייחס אליהם כצד ימין (החלק אליו גוזרים – בסוף מדובר בטרמינלים), וכצד שמאל (משתני הגזירה), וברגע שנזהה בקלט חלק שהוא צד ימין של כלל גזירה כלשהו, אנחנו נחליף אותו בצד השמאלי – במשתנה. נראה דוגמא לטובת העניין –

$S \rightarrow a\ A\ c\ B\ e$
$A \rightarrow A\ b\ |\ b$
$B \rightarrow d$

נגיד וקיבלנו את כללי הגזירה שלמעלה (למשתנה A יש למעשה שתי כללים – A→ Ab, A→b, הם מאוחדים ביניהם עם | רק לצורך הנוחות), ובעבורם נקבל גם קלט מסוים –

a b b c d e

נסתכל עכשיו, ונחפש את האופציות למשתני הגזירה השונים –

a b b c d e

כל אחד מהטרמינלים b יכול להיות A, והטרמינל d עלול להיות B. ניצמד לעבודה על האפשרות השמאלית, ונבחר את ה-b השמאלי ביותר. כעת יהיה לנו המופע הבא, עם האפשרויות השונות המוקפות הבאות –

a A b c d e

b יכול להפוך שוב ל-A, אבל אנחנו רואים שמופיע לידו גם אפשרות להעביר את Ab ל-A, ומאחר שזה מופיע שמאלית יותר לתחילת הטקסט, אנחנו נבחר בו ונקבל את המופע הבא –

a A c d e

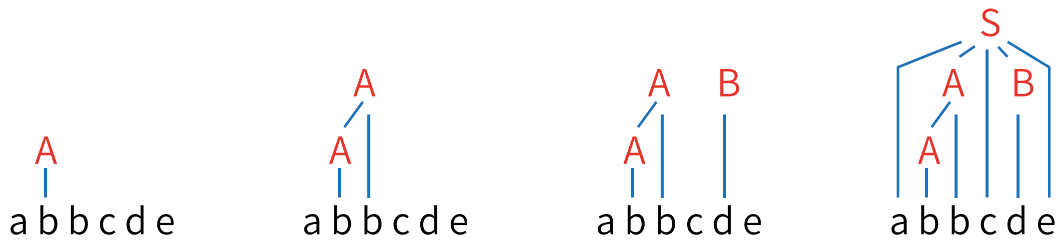
לזה יש לנו רק אפשרות אחת-

a A c B e

שתיהפך בתורה להיות כולה החלק הימני של S –

S

ומבחינתנו סיימנו את העבודה – הצלחנו לגזור אחורה למשתנה התחילי, וזה אומר שהקלט כתוב נכון מבחינה תחבירית. עכשיו ננסה להבין את בניית העץ מלטה למעלה, בשביל זה נראה את מהלך הבניה הבא –



בסוף הריצה הגענו לעץ גזירה המושרש ב-S, ועשינו זאת מהעלים כלפי מעלה.

האם הגזירה שעשינו היא ימנית או שמאלית? ראינו קודם שבחרנו את ההפיכה השמאלית יותר, אבל אם נעיין עכשיו בעץ שיצא לנו, בעצם ניתן לומר שהגזירה עצמה היא דווקא ימנית – אם היינו מתחילים מ-S כלפי מטה, היינו צריכים לגזור קודם את B ורק אחר כך את A, ולכך יש לשים לב.

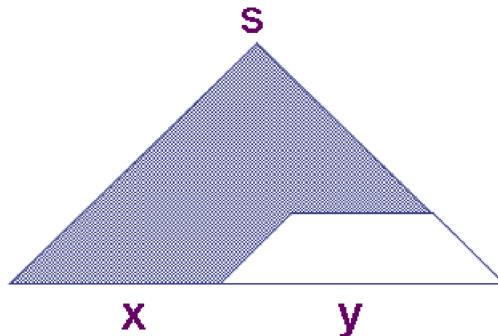
לצורך העניין, והמשך הבנת המושגים, נציין כי גזירה באופן שעשינו נקראת גזירת LR, כלומר עברנו על הקלט משמאל לימין (L), ויצרנו עץ גזירה ימני (R). נראה בהמשך גם אפשרות לעבור מלמעלה למטה וליצור עץ גזירה שמאלי, שנכנה אותו בהתאמה LL.

סוגי הניתוח התחבירי

כזכור, על מנת לעבוד עם דקדוקים חסרי הקשר, אנחנו יכולים להשתמש באוטומט מחסנית, אבל מבחינת זמן ריצה זה עלול לעלות לנו עד $O(n^3)$, שעלול להיות הרבה יותר מידי. ולכן אנחנו משתמשים בגזירות שראינו. לא משנה באיזה אופן נבחר, נזכור שאנחנו תמיד קוראים את הקלט משמאל לימין, כל שנוכל לחלק אותו באופן של x y, כאשר x יהיה החלק שכבר קראנו, ו-y יהיה החלק שעוד לא נקרא.

מאחר שאנחנו רוצים להגיע בסוף לעץ תחביר מסוים, נוכל להראות בצורה אבסטרקטית את דרך העבודה באופן הבא –

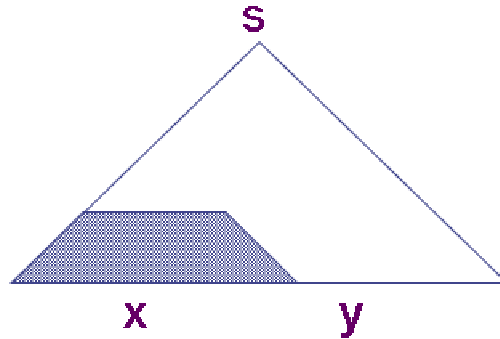
גזירה Top-Down



נעבור החל ממשתנה הגזירה התחילי S, שהוא יהיה תמיד קבוע, וביחס לקלט אנחנו ננסה להתחיל לגזור על מנת להגיע אליו. הצורה הזאת נקראת גם predictive (תחזיתית), כי אנחנו בעצם "מנחשים" מה תהיה הדר אותה נעבור עד שנגיע לטרמינלים. בגלל שאנחנו עושים את הדרך בצורה כזאת, אנחנו חייבים לדאוג שהדקדוק שלנו יהיה הרבה יותר מדויק, אחרת אנחנו נגזור לקלט אחר שהוא בוודאות לא משהו שאנחנו רוצים.

כאמור – הגזירה הזאת מוגדרת LL והעץ היוצא ממנה הוא גזירה שמאלית (שזה די הגיוני בהתחשב בכך שאנחנו עוברים משמאל וגוזרים באופן מסודר.

גזירה Bottom-Up



אנחנו מתחילים מהשכבה התחתונה ביותר, ומתחילים "לגזור" הפוך כלפי מעלה. את הגזירה הזאת אנחנו דווקא כן נעשה עם מחסנית אליה נכניס את הקלט ואת כל הצמצומים השונים, עד שנגיע למצב בו יהיה לנו במחסנית רק את המשתנה S.

נלמד 3 שיטות שונות – 2 Top-Down ואחד Bottom-up.

Recursive Descent - ירידה רקורסיבית

Recursive Descent (להלן RD) היא השיטה הראשונה מבין שיטות ה-Top-Down שנלמד.

ככלל, מה שאנחנו עושים הוא להסתכל על משתני השפה, והאות הבאה בקלט, ומנסים לבחור את המסלול אליו אנחנו רוצים לגזור. בשביל לעשות את זה אנחנו נגדיר סדרת פעולות שיסדרו לנו את הגזירה –

- עבור כל משתנה בדקדוק נגדיר פונקציה. בדומה למה שעשינו קודם עם המצבים, כאן אנחנו נכתוב פונקציה עבור כל משתנה גזירה. הפונקציה תיקח את האסימון הנוכחי (הטרמינלי), ותסיט את מסלול הגזירה לפי הכיוון המתאים.
  - אם האסימון עובר מול טרמינל – שולחים את שניהם לבדיקת התאמה, ומקדמים את האסימון הבא.
  - אם האסימון מתאים למשתנה – מפעילים את כל הפונקציות המתאימות למשתנה הגזירה.
- אם יש כמה חוקי גזירה לאותו טרמינל, בודקים עם Loookahead לאן יש לפנות.

לפני שנסביר את כל הפעולה ונביא דוגמא, נציין רק את פונקציית העזר match –

```
void match(token t) {
        if ( current == t )
                current = next_token();
        else
                error;
}
```

הפונקציה מקבלת אסימון, שמתאים למה שאמור להופיע עכשיו, ובודקת האם מדובר גם באסימון הנוכחי עליו אנחנו עובדים. אם מדובר באסימון מתאים, אז אנחנו עוברים לאסימון הבא, אחרת אנחנו מוציאים ERROR. כדאי לשים לב, שאם אכן יש התאמה, הפונקציה רק מקדמת את הטוק הבא ולא יותר מזה. כדאי לזכור את זה להמשך.

ניקח דוגמת משתנים, נראה כיצד אנחנו כותבים את הפונקציות ונראה גזירה פשוטה –

E → LIT | (E OP E) | not E
LIT → true | false

OP → <u>and</u> | <u>or</u> | <u>xor</u>

כאן יש לנו רק שלושה משתנים, נגדיר עבורם פונקציות מתאימות –

```
void E() {
        if (current ∈ {TRUE, FALSE})                      // E → LIT
                LIT();
        else if (current == LPAREN)                       // E → ( E OP E )
                match(LPARENT); E(); OP(); E(); match(RPAREN);
        else if (current == NOT)                          // E → not E
                match(NOT); E();
        else
                error;
        }
}
```

שימו לב, שאנחנו לא שולחים פה את ה-token שאנחנו עובדים עליו, אלא אנחנו מכירים אותו ב-scope. כל מה שנותר לנו הוא בעצם להסתכל על האפשרויות השונות – אם נסתכל בפונקציה של E, יש לנו בעצם מספר אפשרויות של טרמינלים אליהם נגזור בסופו של דבר. גם אם ניקח את הכלל E→LIT, בסופו של דבר LIT עצמו גוזר לשני טרמינלים true\false, ולכן מבחינתנו גם E גוזר אליהם. ולכן אם נקבל ב-currrnt אסימונים של true\false, אנחנו נעבור לפונקציה של LIT. כדאי לזכור שיכול להיות שהדרך תהיה יותר ארוכה, ו-LIT יגזור למשהו אחר שרק בסוף הדרך יהיה טרמינל, אבל כל עוד יכולים למצוא דרך סלולה ממשתנה לטרמינל, מבחינתנו זה מספיק.

נמשיך הלאה – אם האסימון הנוכחי הוא פתיחת סוגריים (LPAERN), זה בעצם אומר שאנחנו בתחילת הכלל השני. עכשיו, אמנם פה זה מסודר בשורה אחת יפה, אבל אנחנו עובדים פה בצורה רקורסיבית, ועוברים כל אפשרות מתוך מה שאנחנו גוזרים ובודקים אותה – את כל המשתנים אנחנו שולחים לפונקציות שלהם, ואת הטרמינלים אנחנו שולחים לפונקציית match. כאן כדאי לחזור ולהעיר לגבי הפונקציה הזאת – תכלס אנחנו עושים בדיקה אם מדובר בפתיחת סוגריים, ואז אנחנו נכנסים לפונקציית התאמה עם אותו אסימון שבודק גם עבור פתיחת סוגריים. למה כפל עבודה? מאחר והפונקציה דואגת לקדם את האסימון בפקודה ;()current = next_token אז נוח לנו לשלוח את כל האסימונים לאותה פונקציה, ולא להתחיל לחלק אם מדובר באסימון שכבר נבדק או לא – אתה אסימון, לך תיבדק ותתקדם הלאה.

כמובן שבסוף, אם לא מצאנו שום כלל גזירה מתאים נוציא שגיאה. נעבור עכשיו גם על שאר הפונקציות (שלא יחדשו לנו הרבה, אבל נו מילא –

```
void LIT() {
        if (current == TRUE)
                match(TRUE);
        else if (current == FALSE)
                match(FALSE);
        else
                error;
}

void OP() {
        if (current == AND)
                match(AND);
        else if (current == OR)
                match(OR);
        else if (current == XOR)
```

```
            match(XOR);
    else
            error;
}
```

פשוט עושים בדיקות אסימונים ומחזירים תשובות.

הוספת פעולות במהלך הגזירה

הפונקציות שכתבנו קודם מאוד נחמדות, אבל אנחנו אמרנו שאנחנו רוצים להגיע לעץ גזירה, ואין פה שום דבר דומה לזה. אבל מאחר וכבר עשינו כאן מעין תבנית, אנחנו יכולים על הפונקציות האלה "להלביש" פעולות נוספות וככה ליצור את העץ:

- קודם כל נחליף את הערך המוחזר של הפונקציה מ-void ל-Node. מאחר ואנחנו רוצים ליצוא עץ, אנחנו בכל כניסה לפונקציה ניצור תת עץ, ונחזיר אותו בסיום העבודה. כאן כבר אפשר לראות את הרקורסיה בצורה משמעותית יותר – אנחנו מתחילים עם קריאה לפונקציה של שורש עץ, ומה שחוזר לנו הוא עץ בנוי.
- בכניסה לפונקציה נגדיר את שורש העץ החדש השייך לפונקציה/משתנה הגזירה הנוכחי, ונגדיר את ה"שם" של העץ בתור שם המשתנה.
- בכל כניסה רקורסיבית לפונקציה חדשה, אנחנו לא סתם ניכנס אליה, אלא נגדיר את הפונקציה הזאת כבן של העץ עליו אנחנו עובדים.
- בסיום הפונקציה, נחזיר את העץ שבנינו.

נראה את שינוי הקוד על הפונקציה של המשתנה E –

```
Node E() {
    result = new Node();
    result.name = “E”;
    if (current ∊ {TRUE, FALSE})                  // E ⟶ LIT
            result.addChild(LIT());
    else if (current == LPAREN)                   // E ⟶ ( E OP E )
            result.addChild(match(LPARENT));
            result.addChild(E());
            result.addChild(OP());
            result.addChild(E());
            result.addChild(match(RPAREN));
    else if (current == NOT)                      // E ⟶ not E
            result.addChild(match(NOT));
            result.addChild(E());
    else error;
    return result;
}
```

כמובן שגם שאר המשתנים יהפכו להיות בסגנון דומה לפונקציה הזאת, וניתן לראות שכל מה שדרשנו לשנות אכן נעשה.

עכשיו נראה דוגמה של עבודה על קלט – ניקח את השפה שעבדנו עליה עד עכשיו, וננסה לנתח את הקלט הבא, ולראות אם הוא חוקי –

not (not true or false)

קודם כל, נתחיל עם המשתנה התחילי – E, ועם הקלט הראשוני – not.

נכנס לפונקציה E, שקודם כל תשתול לנו את שורש העץ –

not (not true or false)

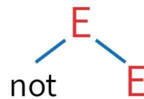
E

עכשיו, אנחנו נשווה את האסימון not, ונראה אם יש לנו מה לעשות איתו – במפתיע, יש לנו מה לעשות. נסתכל מה הקוד אומר לנו לעשות -

```
else if (current == NOT)
                    result.addChild(match(NOT));
                    result.addChild(E());
```

להוסיף ילד לאסימון הנוכחי ולבדור שהוא אכן not, ולהוסיף עוד ילד עבור הפעלת הפונקציה E(), שכמובן כל זה בעצם מפעיל את כלל הגזירה E→not E.

not (not true or false)

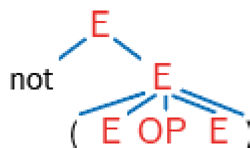


אנחנו נכנסים שוב ל-E(), ועכשיו בודקים את האסימון פתיחת סוגריים. הקוד המתאים –

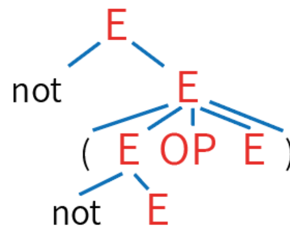
```
else if (current == LPAREN)
          result.addChild(match(LPARENT));
          result.addChild(E());
          result.addChild(OP());
          result.addChild(E());
          result.addChild(match(RPAREN));
```

אנחנו מוודאים שאכן פתחנו סוגריים, לוקחים את האסימון הבא, ומוסיפים את רשימת הבנים המתאימה –

not (not true or false)



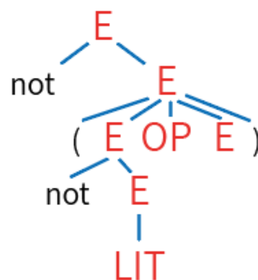
עכשיו אנחנו (שוב) בתוך E, מול אסימון של Not, ועכשיו אנחנו גם יודעים כבר מה צריך לעשות –



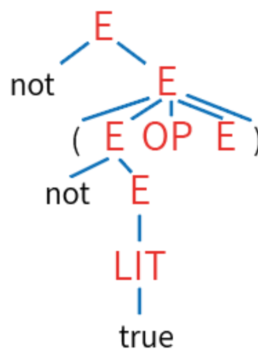
אנחנו בונים תת-עץ חדש עם E, ובודקים את האסימון הבא true.

```
if (current ∈ {TRUE, FALSE})                    // E → LIT
        result.addChild(LIT());
```

אנחנו נכנסים לפונקציה של LIT, כמובן בלי לקדם עדיין את האסימון-



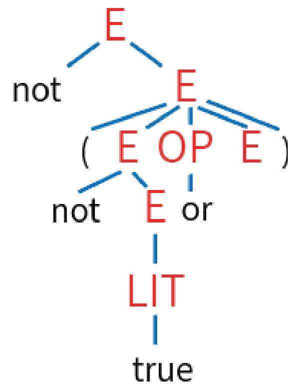
עכשיו, אמנם לא כתבנו את הפונקציה הזאת עד הסוף, אבל אנחנו יודעים מה קורה פה, אנחנו בונים את הבן של true, עושים לו התאמה ומחזירים את תת-העץ הקטנטן שבנינו –



שימו לב, שעכשיו זה הפעם הראשונה שהחזרנו משהו מהרקורסיה – רואים את הסוף!

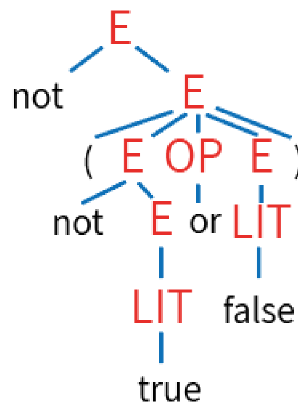
בשלב הזה, אנחנו חוזרים מ-LIT, חוזרים מ-E, ועומדים אל מול OP, עם קלט or, כלומר בונים בן חדש וממשיכים הלאה –

not (not true or false)



עכשיו יש לנו את false מול E, אנחנו כבר יודעים את הדרך מול LIT, ובונים את תת-העץ –

not (not true or false)



כל מה שנשאר לנו הוא לאשר שאנחנו סוגרים את הסוגריים האלה והכל סבבה.

FIRST

כל הדוגמא היפה שראינו עד עכשיו היתה מאוד נחמדה, אבל החיים בדרך כלל הרבה יותר מסובכים מזה, ועלולים ליצור בעיות. דבר ראשון יש לציין, וכבר הזכרנו את זה לרגע שבעבור כל משתנה גזירה, אנחנו צריכים להחליט לאיזה כיוון אנחנו הולכים לגזור, אבל איך נדע שאם יש לנו את E ובקלט יופיע לנו true זה יהיה תקין?

בשביל זה אנחנו "מפעילים" פונקציה שנקראת first לפני כל הפעולות שעשינו. הפונקציה הזאת בעצם משוטטת לעומק המשתנים ומחזירה לנו רשימה של כל הטרמינלים שעלולים להגיע מאותו משתנה. כך שמתי שנכתוב את הפונקציות, אנחנו נדע ישר להגיד מה המסלול שאנחנו צריכים לעבור.

כל זה טוב ויפה, אבל רק בהנחה אחת פשוטה- שאין לנו חפיפות בין ה-first של המשתנים השונים. אם פתאום גם E וגם LIT גוזרים לאותו טרמינל (ובהנחה שאין גזירה מאחד לשני), אז איך נדע האם אנחנו נמצאים במסלול הנכון, או שחלילה אנחנו צריכים להוציא הודעת שגיאה? לזה יש לנו שתי אפשרויות – או לתקן את הדקדוק בשביל שלא יהיו לנו כפילויות, או לעשות Loookahead לאסימונים הבאים.

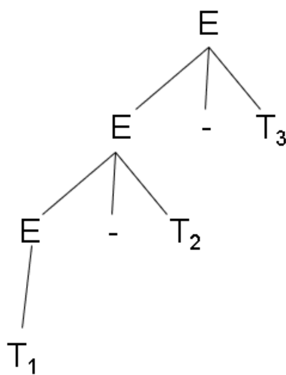
ניתוח תחבירי בירידה רקורסיבית מצגת דוגמאות מס' 2

כתיבת פונקציה איטרטיבית

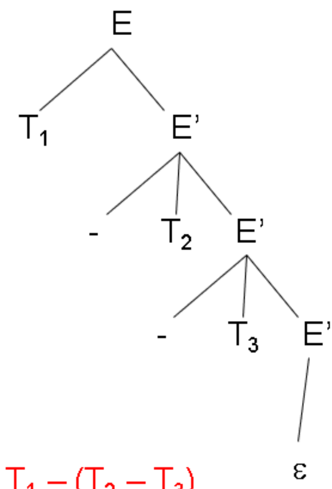
בעיה ידועה עם טיפול ברקורסיות הוא התמודדות עם פעולות לא אסוציאטיביות. מה הכוונה "לא אסוציאטיביות"? נניח ויש לנו את התרגיל הבא – 10-1-10. על פי מה שלמדנו בבית-הספר היסודי, נבצע קודם כל את 10-1 ואז מהתוצאה הזאת נחסר 10. כלומר 9- = 10-(10-1). אבל אם אנחנו נעבוד לפי הטיפול ברקורסיה שראינו, אנחנו עלולים ליצור מצב של 19 = (10-1)-10, וזה כמובן מאוד בעייתי לנו. נוכל לראות את ההשלכות בעץ הגזירה, כפי הדוגמא במצגת התרגול –

$$E \rightarrow E - T \mid T$$

$$E \rightarrow TE'$$
$$E' \rightarrow -TE' \mid \varepsilon$$



$$T_1 - T_2 - T_3$$



$(T_1 - T_2) - T_3$

$T_1 - (T_2 - T_3)$

כאן נגיד ש-T גוזר למספר כלשהו (TOK_INT) ואנחנו יכולים להבין שיש פה בעיה.

בשביל שנוכל להתמודד גם עם מצבים כאלה, אנחנו יכולים במקום לטפל ברקורסיות האלה בצורה שלמדנו, פשוט לכתוב את הפונקציה עצמה בצורה איטרטיבית ולא רקורסיבית – אם נסתכל על הכלל של E אנחנו מבינים שבסוף הוא גוזר לשרשרת של *{T-}T. מה שאנחנו רוצים לעשות, הוא כזה – אנחנו נשנה את העץ לצורה שהצמתים יהיו אופרטורים, והבנים יהיו האופרנדים (משתנים ולבסוף טרמינלים). איך נבצע את זה? אנחנו מקבלים T ופותחים לו עץ. ואז, כל עוד יש לנו באסימון הבא אפשרות שפותחת לנו צומת חדשה, אנחנו יודעים שהאסימון הבא הוא גם T, ולכן נצרף את שלושתם תחת תת-עץ בודד, ונחזיר את ה"ראש" שהוא ה"-". וכך חוזר חלילה עד שנסים את הגזירה הנוכחית. איך זה ייראה בקוד?

```
Node *E() {
 node1 = T();
 while (token.pattern == TOK_MINUS) { // כל עוד אנחנו מזהים פתיחת "תת עץ" חדש
  next_tok();                        // אוספים את המשתנה שהאחרי המינוס
  rnode = T();                       //הכנסה של  האסימון הקודם ימינה
  node2 = new Node('-', node1, rnode); // בניית הצומת והבנים
  node1 = node2;
 }
 return node1;
}
```

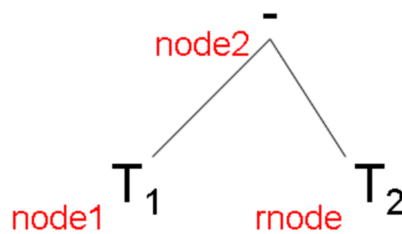
זה קצת מבלבל, אבל בין ה-T שמופיע פעם ראשונה (שמוגדר כ-node1), ל-T שמופיע פעם שניה (שיוגדר rnode) החלפנו אסימון, ועכשיו זה אחד אחר. על מנת להבין את זה בצורה קצת יותר ברורה, אפשר לראות דוגמת הרצה פשוטה לקלט שראינו קודם, ולראות איך אנחנו בונים את העץ.

בגדול, אנחנו משחקים פה על שלושה nodes (צמתים), node1 איתו אנחנו מתחילים ומסיימים כל לולאה, ואותו אנחנו גם מחזירים בסוף הריצה. rnode – ברגע שאנחנו מזהים שאנחנו בונים פה תת עץ, אנחנו מגדירים בן ימני. node2 שמהווה איזה צומת זמנית ליצירת אב לפני שנעביר אותו להיות node1. עכשיו נראה את התהליך –

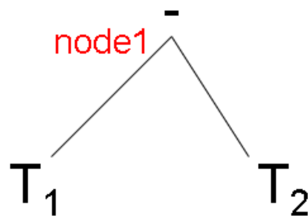
ברגע שנראה את ה-T הראשון אליו אנחנו מתייחסים, אנחנו נגדיר אותו בצומת node1, וזה בגלל שאם לא נכנס ללולאה בכלל (אם היה לנו סתם T בודד), זה מה שנצטרך להחזיר.

עכשיו, ברגע שהגדרנו את T, אנחנו אוטומטית מושכים את האסימון הבא, ויכולים לבדוק את התנאי ב-while. אם יש לנו שם TOK.MINUS אז זה אומר שאנחנו בונים פה עץ קטן, ולכן אנחנו נגדיר את האח של T_1 להיות צומת ימנית –

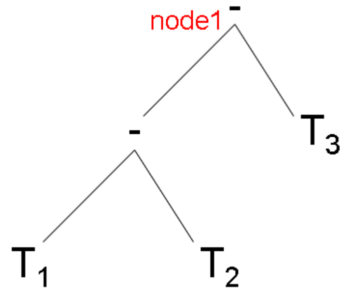
עכשיו אנחנו מגדירים את node2 שבונה לנו את העץ – הארגומנט הראשון יהיה השורש, ולאחריו שני הבנים – שמאל וימין –



עכשיו אנחנו סוגרים את הפינה של העץ הזה, ומגדירים ששורש העץ, שעד עכשיו היה node2 יהיה node1 ובהתאם גם זה שיכול לחזור ביציאה מהפונקציה –



עכשיו יש לנו את החלק עם T_3. אנחנו עושים בדיוק את אותו התהליך כמו קודם, רק שהבן השמאלי הפעם יהיה כל תת העץ עליו כבר עבדנו. נדלג על כל התהליך ונקבל את העץ הבא –



עכשיו אנחנו צריכים לבדוק מה הוא האסימון הבא, אנחנו בודקים ורואים שמדובר ב-$, סוף הקלט. אנחנו מבינים שסיימנו את העבודה ומחזירים את node1 שזה עתה סיימנו ליצור.

לעיתים אם ניתקל בגזירות שהן בעייתיות מבחינת כתיבה רקורסיבית, יבקשו מאיתנו לכתוב אותם בצורה איטרטיבית, אז כדאי לזכור איך זה נעשה.

נסתכל עכשיו על דקדוק כלשהו, נראה איך אנחנו בונים את כל הפונקציות וכל הדרוש, ואיך אנחנו קוראים את הקלט ובונים את העץ. אך בל נקדים את המאוחר. להלן, דקדוק:

G = ({A, N, E, T}, {**id**, **.**, **=**, **+**, **(**, **)**}, P, A)

על פי ההגדרות שהבאנו קודם, יש לנו ארבעה משתנים, שישה טרמינלים, ואת קבוצת כללי הגזירה P –

1. A → N **=** A | E
2. N → N **. id** | **id**
3. E → E **+** T | T
4. T → N | **(** A **)**

הגדרת הפונקציה first

אמנם הזכרנו את זה כבר קודם, אבל עכשיו נראה בצורה קצת יותר מפורטת מה זה אומר ואיך מתייחסים ל- firstש כללי גזירה. ההגדרה הרשמית אומרת כך –

יהיו $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$ כל כללי השכתוב ב-P אשר אגף שמאל שלהם הוא A, אזי: $first(A) = \bigcup_{i=1}^{k} first(\alpha_i)$

כלומר, ה-first(A) יהיה כל האפשרויות השונות להגיע מ-A לטרמינל כלשהו מכל גזירה שלא תהיה. אם יש לנו מספר כללי גזירה שנובעים מ-A אנחנו פשוט נעשה איחוד בין כולם ונרשום אותם, נשמע פשוט אבל נראה כמובן איפה זה מסתבך. לכל זה יש 3 כללים קטנים שעלינו להוסיף, כל אחד מהם מתייחס למקרה מסוים שעלול להקשות עלינו –

- $FIRST(\varepsilon) = \{\varepsilon\}$ – אם יש לנו אפסילון, שלכאורה לא באמת נקרא בקלט, אנחנו בכל זאת נכניס אותו לתוך קבוצת הטרמינלים המתאימה, נראה בהמשך כיצד זה עוזר לנו.
- $FIRST(a\alpha) = \{a\}$ – ה-a שיופיע בצמוד לטרמינלים או משתנים אחרים יוגדר בתור ה-first. זה יחסית מקרה קלאסי

$FIRST(B\alpha) = \begin{cases} FIRST(B) - \{\varepsilon\} \cup FIRST(\alpha), & \text{if } \varepsilon \in FIRST(B) \\ FIRST(B), & \text{otherwise} \end{cases}$ – כאן זה כבר קצת יותר מסובך. אם הגזירה עוברת למשתנה ולאחריו עוד משהו (סימון ה-α אומר שזה לא משנה לנו אם זה משתנה או טרמינל), במקרה רגיל פשוט נצרף את כל ה-first(B), כי כל טרמינל שאנחנו יכולים להגיע אליו מ-*B*, אנחנו יכולים גם ממה שגוזר אליו. אבל מה קורה אם אחד הטרמינלים האלה הוא ε?

במקרה כזה, אנחנו נוסיף גם את כל ערכי ה-first של α, אם זה טרמינל נוסיף אותו, ואם זה משתנה אז את כל מה שגוזר ממנו.

עכשיו אנחנו יכולים להתחיל ולחשב את ערכי ה-first של המשתנים –

First(N) = First(id) = **{id}**
First(T) = First(N) ∪ First(‘(‘ A ‘)’) = **{id, (}**
First(E) = First(T) = **{id, (}**
First(A) = First(N ‘=‘ A) ∪ First(E) = First(N) ∪ First(E) = **{id}** ∪ **{id, (}** **= {id, (}**

אמנם כולם גוזרים פחות או יותר לאותם טרמינלים, אבל מה שמדאיג אותנו הוא דווקא הכלל גזירה של A. לא רק שהוא גוזר ל-id כמו כל המשתנים עד עכשיו, הוא עושה את זה משני מקומות שונים. גם N וגם E עלולים להגיע לטרמינל זה, ואם אנחנו ננסה לחפש את הדרך לגזור במקרה ונראה את הטרמינל הזה ברצף הקלט, אנחנו לא נדע לאן אנחנו אמורים ללכת. במקרה כזה אנחנו צריכים לפתור את בעיית ההכרעה.

אנחנו מסתכלים שוב על הגזירה מ-A ורואים שבעצם כך או כך, אם אנחנו גוזרים ל-id אנחנו עוברים דרך N. השאלה היחידה זה רק אם אנחנו גוזרים ישירות מ-N או עושים איזה סיבוב קטן לפני זה. לכן, אנחנו ננסה לשכתב את הגזירה בצורה שתהיה יותר ברורה.

קודם כל, אנחנו יודעים שבשביל לגזור ל-')' יש לנו רק אפשרות אחת – דרך E. כלומר הכלל עליו אנחנו עובדים הוא בוודאות A→E. E יגזור בזמנו החופשי ל-T ואז ימשיך הלאה ל(A) אבל זה כבר יקרה מתי שבא לו.

עכשיו, מה שנאמר הוא שאם אנחנו רואים id אנחנו נגזור ישר ל-N, כי גם אם הגזירה היתה ל-E הוא יגזור ל-N, אז אנחנו יכולים להיות בטוחים ששם נעבור, כך שמה שיישאר לנו הוא לבדוק את המסלול. איך נעשה את זה? אם יש לנו '=', אז אנחנו כנראה בכלל הגזירה A = N →A ואז אנחנו יכולים להמשיך עם זה. אחרת, כנראה שעשינו סיבוב דרך E. לכן אנחנו נפעיל את E ונעביר לו את כל תת העץ שכבר עשינו ל-E.

נראה איך זה יופיע בקוד –

```
Node *A() {
 switch (token.pattern) {
  case TOK_LPAREN:   // A → E
   result = E();
   break;
  case TOK_ID:  // A → ?
   lnode = N();  // A ⇒* N … ?
   if (token.pattern == TOK_ASGN) {   // ‘=‘
     // A → N = A
    next_tok();
    rnode = A();
    result = new Node(ND_ASGN, lnode, rnode);
   }
   else {  // not ‘=‘: A → E, E ⇒* N [ + T + … + T ]
    result = E(lnode);
   }
   break;
  default:  // nor identifier neither ‘(‘
   error(“Identifier or ‘(‘ expected before %”);
   break;
 }  // end switch
 return result;
}
```

אנחנו השתמשנו פה במספר פונקציות עזר עליהן לא ממש הרחבנו, נראה את המימוש של כל אחת מהן, ונראה מה היא עושה ומה מיוחד בהן –

```
void next_tok() {   // Advance to next token
  token = LexAnalyzer.next_tok();
}
void verify(Pattern p) {   // Make sure current token is p
  if (token.pattern != p)
    error("Token of type $p$ expected before %");
}
void get_token(Pattern p) {   // Make sure next token is p and advance to next token
  verify(p);
  next_tok();
}
```

בפונקציה כמו A שבנינו עכשיו, אנחנו צריכים להתקדם לאט, ולכן לא השתמשנו בפונקציה get_token. אך בפונקציות הפשוטות יותר אנחנו עושים ווידוא וישר לוקחים את האסימון הבא.

```
error(String s) {
  print "Line ", token.location, ": ";
  print_msg_with_name(s);   // Inserts name of current token where '%' appears within given message
  print '\n';
  exit();
}
```

לצורך הבנת הריצה, נעבור גם כן על שאר הפונקציות של המשתנים –

```
Node *E(Node *node1 = NULL) {   // Iterative
  if (node1 == NULL)   // if leftmost operand not passed as argument
    node1 = T();
  while (token.pattern == TOK_PLUS) {
    next_tok();
    rnode = T();
    node2 = new Node(ND_ADD, node1, rnode);
    node1 = node2;
  }
  return node1;
}
```

האפשרויות שלנו לגזירה מ-E הן שתיים –

E→E+T
E→T

ראינו קודם שהפונקציה A יכולה לקרוא ל-E בשני דרכים שונות – האחת, ללא כל ארגומנט, כאשר מדובר בגזירה ישירה A→E, והשניה כאשר אנחנו עושים גזירה עקיפה, ואז אנחנו מעבירים גם את חלק העץ שכבר בנינו. לכן עלינו לדאוג לשתי האפשרויות האלה. ומאחר ויש לנו אפשרות לשרשור של גזירה עצמית, אנחנו צריכים לדאוג שהפונקציה תהיה איטרטיבית. לכן, אם אנחנו גזרנו ישירות ל-E, ולא קיבלנו ב-node1 שם מידע, אנחנו קודם כל יוצרים לנו תת עץ של T. עכשיו אנחנו בודקים האם יש לנו שרשור – במידה וכן, האסימון הבא יהיה +, ואז נרוץ בלולאה עד שנסיים איתו, ונחזיר את תת העץ שבנינו עד כה.

```
Node *T() {   // Recursive
  switch (token.pattern) {
    case TOK_ID:
        result = N();
        break;
    case TOK_LPAREN:
        next_tok();
        result = A();
        get_token(TOK_RPAREN);
        break;
    default: error("Identifier or '(' expected before %");
        break;
  }
  return result;
}
```

הפונקציה Tעובדת בצורה רקורסיבית פשוטה מאחר ואין לה בעיות הכרעה. אם הטוקן שאנחנו עובדים הוא id זה אומר שגזרנו ל-N, אם זה פתיחת סוגריים, אנחנו בונים את מה שחשוב ל-A, ומוודאים שהאסימון הבא הוא באמת סוגר סוגריים, ואז ממשיכים הלאה.

```
Node *N() {   // Iterative
  verify(TOK_ID);
  node1 = new Node(ND_ID, token.location, token.name);
  next_tok();
  while (token.pattern == TOK_DOT) {   // '.'
    next_tok();
    verify(TOK_ID);
    rnode = new Node(ND_ID, token.location, token.name);
    next_tok();
    node2 = new Node(ND_SELECT, node1, rnode);
    // Field-selection operation
    node1 = node2;
  }
  return node1;
}
```

הגזירה מ-N תביא לנו id, ולפעמים גם גישה פנימית יותר, כלומר ...id.id.id ולכן גם פה אנחנו נתעסק עם פונקציה איטרטיבית. בהתחלה אנחנו נוודא שאנחנו על id, ובאסימון הבא נבדוק האם יש לנו '.' ואז אנחנו עושים עוד סיבובים עד שנסיים את רצף הגזירה הזה, ובסוף נחזיר את כל העץ שייבנה לנו.

עכשיו אחרי שראינו איך כל פונקציה עובדת, אנחנו יכולים להתחיל לעבוד על הריצה. אנחנו נעבוד מול טבלה שתכיל מצד אחד את "שארית הקלט" – בתור התחלה יהיה לנו את כל הקלט, ובכל פעם שנתקדם באסימונים נוריד את החלקים עד שנגיע לסיום הקלט, ומהצד השני של הטבלה תהיה לנו מחסנית הקריאות – שם נסדר את כל הפונקציות שאנחנו נכנסים אליהם, וברגע שנסיים עם כל פונקציה נוציא אותה מהמחסנית.

דבר חשוב נוסף, אנחנו מתחילים לבנות את העץ מתחילת הריצה, אם נסיים את הריצה לא בטוח שנוכל לשחזר את הצעדים שלנו בצורה נכונה.

נתחיל – נכניס בהתחלה את כל הקלט, ואל מולו את A שהוא משתנה הגזירה הראשון לפי ההגדרה –

מחסנית קריאות	שארית הקלט
A()	a = b + (c = d) + e $

אנחנו רואים שבקלט יש לנו אסימון של id, אנחנו בודקים מה הפונקציה ()A אומרת בנידון –

```
case TOK_ID:
    lnode = N();
```

כלומר, קודם כל נבנה לנו צומת שמאלית (כי אנחנו מבינים שתהיה כנראה גם ימנית) תחת הפונקציה N. לכן אנחנו נכניס את N לתוך מחסנית הקריאות, ונראה מה עושים שם –

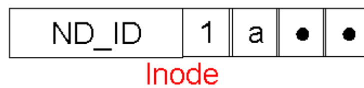
מחסנית קריאות	שארית הקלט
A()N()	a = b + (c = d) + e $

```
verify(TOK_ID);
node1 = new Node(ND_ID, token.location,  token.name);
next_tok();
```

קודם כל, אנחנו מוודאים שמדובר בטוקן id, תם כדי להיות בטוחים, ואז אנחנו בונים את node1 שיורכב מהצומת עצמה, הסימון של המיקום שלו והשם (a). בהמשך יש בדיקה האם הגישה היא יותר פנימית, אך מאחר שלא, אנחנו פשוט מחזירים את הצומת בתור node1 והיא עוברת אוטומטית להיות lnode של A.

מה שייתן לנו את המצב הבא –

מחסנית קריאות	שארית הקלט
A()	=b + (c = d) + e $



עכשיו אנחנו עומדים אל מול סימן '=' כלומר השמת ערך. אנחנו מבינים מזה שאנחנו עכשיו בצד השני צריכים ליצור את החישוב שיושם לתוך a שייצרנו. נראה איך אנחנו פועלים בקוד –

```
if (token.pattern == TOK_ASGN) {
     next_tok();
     rnode = A();
```

כלומר, אנחנו עכשיו מתחילים בצורה רקורסיבית להתחיל לבנות את הבן הימני. אנחנו ממש מתחילים מהתחלה ומפעילים את הפונקציה A –

מחסנית קריאות	שארית הקלט
A()A()	b + (c = d) + e $

שוב אנחנו רואים שיש לנו את ה-id(b) ולכן אנחנו בונים לו צומת בתוך הפונקציה N -

מחסנית קריאות	שארית הקלט
A()A()N()	b + (c = d) + e $

הכניסה החוזרת לפונקציה N מחזירה לנו בתור התחלה את אותו הדבר כמו קודם – בן (שמאלי) בודד בתוך הפונקציה ()A השניה –

מחסנית קריאות	שארית הקלט
A()A()	+ (c = d) + e $

עכשיו בעודנו ב-A, אנחנו בודקים מהו האסימון הבא, ומאחר שלא מדובר על סימן השמה, אנחנו שולחים את ה-lnode שבנינו זה עתה לתוך הפונקציה E –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)	+ (c = d) + e $

כזכור, אנחנו קודם כל עושים בדיקה לראות האם קיבלנו ערך אמיתי בקריאה לפונקציה (ברירת המחדל למקרה ולא נשלח כלום היא null), ואז אנחנו מתחילים לעבור ב-while ולשרשר את כל סימני החיבור. מאחר ופה יש רק אחד כזה, זה די קצר –

```
while (token.pattern == TOK_PLUS) {
 next_tok();
 rnode = T();
 node2 = new Node(ND_ADD, node1, rnode);
 node1 = node2;
}
```

אז מה שיש לנו לעשות הוא כדלקמן – להגדיר חלק ימני של העץ דרך T, ולחבר את שני החלקים שנקבל תחת האבא שיהיה סימן החיבור –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()	(c = d) + e $

```
case TOK_LPAREN:
      next_tok();
      result = A();
      get_token(TOK_RPAREN);
      break;
```

כאשר אנחנו ב-T ורואים בקלט פתיחת סוגריים, אנחנו יוצרים תת עץ חדש בשם result. קודם כל אנחנו נקדם את הקלט, ואז נכנס ל-A, ובסוף נוריד את סגירת הסוגריים –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()	c = d) + e $

מה קורה כש-A רואה id? שולח אותו לבנות צומת ב-N –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()N()	c = d) + e $

N מייצר לנו צומת בעבור c, ומחזיר אותה ל-A שמתייחסת אליו בתור lnode, עד שיוכח אחרת. המצב שלנו עכשיו ייראה כך –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()	= d) + e $

ND_ID 1 a • •

ND_ID 1 b • •

ND_ID 1 c • •
lnode

שימו לב שבינתיים יש לנו רק בנים שמאליים, ולא חיברנו שום דבר לשום מקום. עכשיו A מזהה שיש לנו כאן סימן '=', ולכן פועל בצורת השמה, כלומר יוצר לנו בן ימני דרך קריאה מחודשת ל-A –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()A()	d) + e $

אנחנו כבר יודעים מה קורה עכשיו – A קורא ל-N, ויוצר בן ימני, שהוא שמאלי ביחס ל-A שקרא לו –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()A()N()	d) + e $

ומזה יוצא לנו שאנחנו במצב הבא –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()A()	) + e $

ND_ID 1 a • •

ND_ID 1 b • •

ND_ID 1 c • •   ND_ID 1 d • •
lnode

עכשיו, מאחר וכאן אין סימן השמה, אנחנו שולחים את הצומת שיצרנו לתוך E –

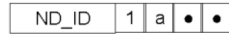
מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()A()E(lnode)	) + e $

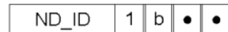
העניין הוא, שהצומת לא ריקה, והאסימון הבא הוא גם לא '+', לכן אנחנו ישר יוצאים מהפונקציה –

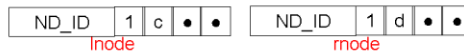
מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()A()	) + e $

עכשיו, אין עוד פעולות שאנחנו יכולים לעשות בתוך ה-A הפנימית ביותר, ולכן נצא מהפונקציה והצומת האחרונה שעשינו תוגדר סוף כ-rnode –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()	) + e $

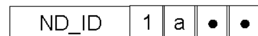


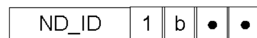


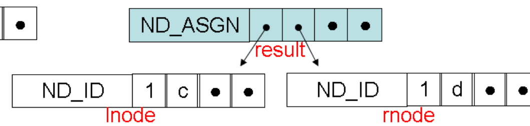


הראינו שם למעלה את הקוד של הטיפול של A בפתיחת סוגריים – ניתוח החלק הפנימי, והחזרת התוצאה. עכשיו נסיים את הפעולה הזאת – נקדם את הקלט, ונבנה צומת אב לרמה התחתונה ביותר בה אנחנו נמצאים –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()A()	)+ e $







עכשיו אנחנו חוזרים לפונקציה T, את כל מה שבנינו עכשיו בתוך A אנחנו מעבירים כ-result, ויוצאים מ-A –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()	)+ e $

עכשיו בתוך T אנחנו מקדמים את האסימון של סגירת הסוגריים –

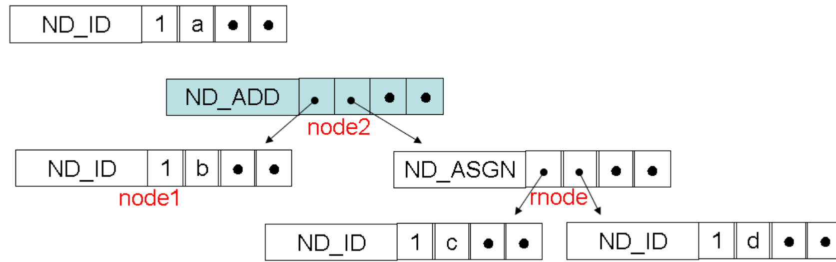
מחסנית קריאות	שארית הקלט
A()A()E(lnode)T()	+ e $

ויוצאים מ-T

מחסנית קריאות	שארית הקלט
A()A()E(lnode)	+ e $

עכשיו בתוך E, אנחנו סוגרים את תת העץ הנוכחי בעזרת צומת של סימן חיבור –

מחסנית קריאה	שארית הקלט
A()A()E(lnode)	+ e $



עכשיו אנחנו נכנסים ללולאת ה-while בתוך E. אנחנו לוקחים את הסימן + –

מחסנית קריאה	שארית הקלט
A()A()E(lnode)	e $

ועכשיו אנחנו נשלחים לתוך הפונקציה T בשביל לבנות לנו בן ימני לפעולת החיבור –

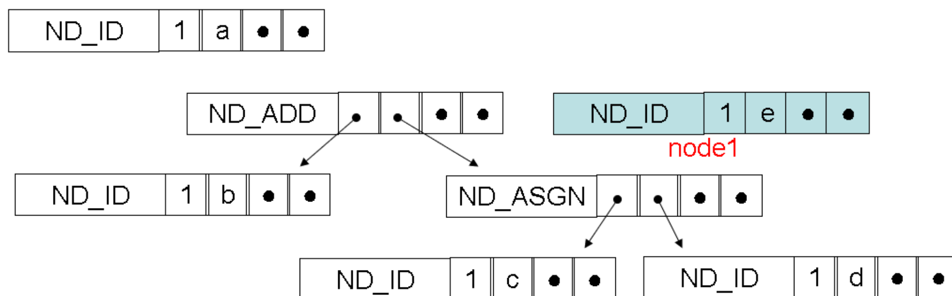
מחסנית קריאה	שארית הקלט
A()A()E(lnode)T()	e $

T מזהה שמולו עומד id, ולכן הוא קורא ל-N –

מחסנית קריאה	שארית הקלט
A()A()E(lnode)T()N()	e $

N מוודא שמדובר ב-id, ובונה לו צומת –

מחסנית קריאה	שארית הקלט
A()A()E(lnode)T()N()	$



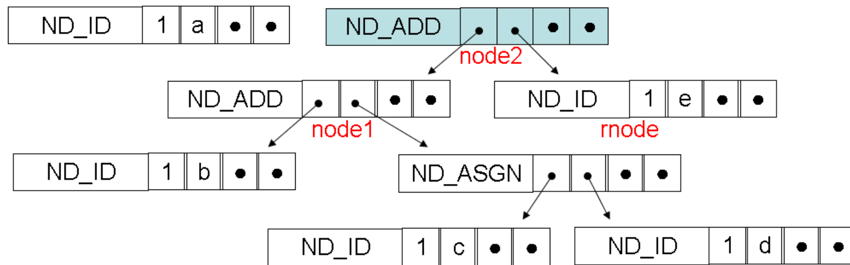
אנחנו יוצאים מ-N, מה שמעביר את הצומת החדשה להיות ה-result של T –

מחסנית קריאה	שארית הקלט
A()A()E(lnode)T()	$

T סיים את הפעולה שלו, ועכשיו הוא מחזיר את הצומת ל-E בתור בן ימני –

מחסנית קריאות	שארית הקלט
A()A()E(lnode)	$

E בתורו, בונה אב לשני הבנים תחת סימן החיבור, ואז יוצא –

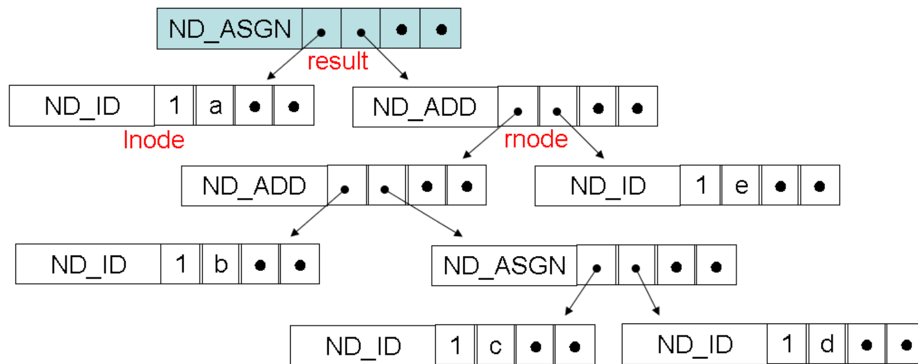


מחסנית קריאות	שארית הקלט
A()A()	$

פונקציית ה-A הנוכחית סיימה את הפעולה שלה, ומחזירה את כל תת העץ (בתור result) שבנינו לקריאה המקורית, שם היינו תחת סימן השמה –

מחסנית קריאות	שארית הקלט
A()	$

אחרי שחזרנו מכל הסיבוב הזה, אנחנו בונים צומת שתאחד את הצד הימני (החישוב) לצד השמאלי (משתנה ההשמה) –



ועכשיו, כל שנותר לנו הוא לצאת מהפונקציה A ולהחזיר את ה-result –

מחסנית קריאות	שארית הקלט
	$

ובאושר גדול אנחנו מזהים שסיימנו את הריצה על הקלט, ואין לנו קריאות במחסנית וכולם מרוצים ושמחים.

LL(1)

"המחלקה LL(1) היא המחלקה של דקדוקים שניתן לגזור עם look-ahead של אסימון אחד. יש דקדוקים כאלו לשפות תכנות רבות".

קודם כל, ניתן את הפרמטרים העיקריים לדקדוקי LL(K)-

- Top-down – כלומר, אנחנו מתחילים מהשורש ובונים את העץ כלפי מטה עד העלים.
- סריקת הקלט משמאל לימין – ה-L הראשונה.
- מניבה גזירה שמאלית ביותר -ה-L השניה.
- זקוקה ל-Lookahead בגדול K – מותרת "הצצה" קדימה בקלט עד K אסימונים מהמקום הנוכחי, יכול להיות גם פחות.

שפה LL(K) היא כזאת שיש לה דקדוק LL(K) מתאים.

אנחנו נתעסק בעיקר עם המקרה הפרטי שזה שפות שהן LL(1).

רקורסיה שמאלית

כשאנחנו מחפשים את ה-First של משתני גזירה כלשהם אנחנו עלולים ליפול לתסבוכות שונות, חלקן קלות לתיקון וחלקן קצת פחות. נראה בעיה המוכרת בשם "רקורסיה שמאלית", ונבין כיצד אפשר לטפל (כמעט תמיד).

לצורך הבנת הבעיה, נראה את כלל הדקדוק הבא –

A -> AaB | aC

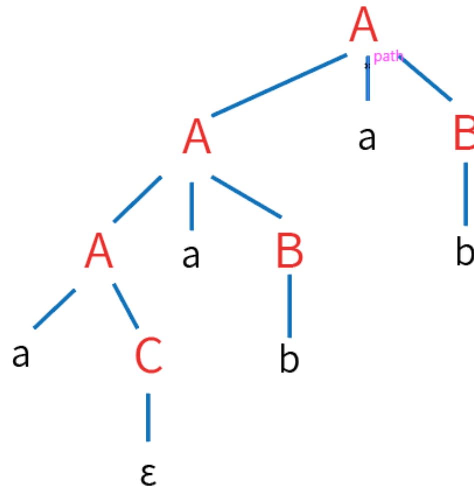
לכאורה דקדוק תמים, אנחנו לא חושדים בו, מה אנחנו חשדניסטים? ואז! אנחנו מחפשים את ה-first שלו – יש לנו את "a" שהוא יחסית פשוט, ובאפשרות השניה יש לנו משתנה – A. אז אנחנו נכנסים לבדוק מה ה-first שלו – יש לנו את "a"שהוא יחסית פשוט, ובאפשרות השניה יש לנו משתנה – A. אז אנחנו נכנסים לבדוק מה ה-first שלו – הבנתם את הקונספט. נתקענו בלולאה אינסופית.

אבל נשאל "מה הבעיה?" אנחנו יכולים לגזור את המשתנים ואז להמשיך הלאה! מסתבר שזה לא כזה פשוט. לצורך העניין נניח שיש לנו עוד את שני כללי הגזירה הבאים –

B -> b
C-> c | ε

ואנחנו מקבלים את הקלט הבא – aabab

עכשיו נעבור תו תו, ונראה מה אנחנו עושים – קודם כל יש לנו a – ברור לנו שאנחנו צריכים לגזור לאחת האפשרויות הימניות של A, אבל איזה מהם נבחר? אם נבחר את aC, אנחנו ניתקע בעוד רגע. נבחר את AaB ואז יהיה לנו בשורש A ולו שלושה בנים, כשהראשון הוא A. ואז אנחנו חוזרים שוב על אותו סיפור. מתי אנחנו עוצרים? במבט שלנו אנחנו יודעים שאחרי פעמיים שגוזרים ל-A, אנחנו יכולים להמשיך הלאה בשאיפה שנגיע לעץ הבא –



אבל לקומפיילר עצמו, אין שום אינדיקציה או קאונטר שיוכל לסמוך עליו ולהגיד לו שהוא יכול להמשיך. ולכן אנחנו מבינים שאנחנו צריכים לתקן את הדקדוק הזה.

אנחנו רואים שהחלק הבעייתי שלנו הוא המשתנה A בראש החלק הימני, והוא ממשיך וגוזר לעצמו עד שהוא מגיע ל-a. למעשה, אנחנו מבינים שיהיה מה שיהיה, בסוף יהיה לנו aC בראש המילה ואחר כך רצף של aB שעלול לחזור כמה פעמים. ולכן נכתוב את הדקדוק השקול הבא –

A -> aCA’
A’ -> aBA’ | ε

עכשיו, אנחנו קודם כל נגזור מ-A את ה-a הראשון, נוכל להכנס ל-C באופן חופשי, וכשנסיים שם נוכל לשרשר כמה aBA’ שבא לנו, איך נדע מתי לעצור? כשיסתיים הקלט, אנחנו נוכל לגזור לאפסילון ולסיים את העץ.

עכשיו באופן פורמלי יותר –

בהינתן כללי הגזירה הבאים –

$A \longrightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

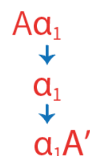
כלומר יש לנו משתנה שגוזר לשני סוגי גזירות –

1. טרמינלים או משתנים שעומדים לאחר המשתנה המקורי ($A\alpha_i$) כאשר α יכול להיות גם משתנה וגם טרמינל, אבל הם פחות מדאיגים אותנו.
2. טרמינלים או משתנים שעומדים בפני עצמם (β) כלומר שבסוף הרקורסיה, המשתנה ייגזר לאחד מאלה וימשיך הלאה.

התיקון שנבצע הוא כזה – קודם כל נעבוד רק על הגזירות מהסוג השני, נעביר אותם בתור האפשרויות היחידות לגזירה מ-A, ובסופם נצמיד ’A.

$A \longrightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$

עכשיו נתמודד עם הסוג השני – את כל הגזירות האלה נעביר תחת הכלל שהוספנו ’A, כאשר מכל כלל גזירה נשמיט את ה-A המקורית, ונוסיף בסוף שרשור נוסף ל-’A. לצורך הדוגמה –



נבצע את אותה הפעולה לכל הכללים השונים ונקבל –

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_n A' \mid \varepsilon$

חשוב מאוד להוסיף בסוף את ה-ε שיסמן לנו את סוף הקלט.

Left Factoring

בעיה נוספת שעלולה להתרחש ובעייתית למקרים של RD, הוא במקרה בו יש לנו התנגשות ב-first של כללי גזירה שונים. הבעיה היא שאנחנו צריכים למצוא באופן חח"ע איזה טרמינל גוזר לאיזה משתנה ולפי זה לעבוד, ואם יש לנו התנגשות אנחנו בבעיה.

נניח ויש לנו את כללי הגזירה הבאים –

S → if E then S else S
| if E then S
| T

אותו משתנה גוזר לשתי אפשרויות שמתחילות בדיוק אותו הדבר – "if E then S", כאשר כלל אחד עוצר בנקודה הזו, והכלל השני ממשיך הלאה ל-"else S". עכשיו כשניתקל בקלט באסימון "if", איך נדע לאיזה כלל ללכת? כדאי לזכור שאנחנו רוצים מסלול דטרמניסטי, ואנחנו לא יכולים לעבור באמצע ולהגיד טוב, הגענו ל-else אז כנראה זה הכלל הראשון. מה שנעשה פה הוא לגזור רק לחלק המשותף הגדול ביותר, ומשם לשלוח לכלל גזירה חדש שיכריע בין השניים על פי מה שיופיע, כלומר –

S → if E then S S' | T
S' → else S | ε

ועכשיו כל הגזירה תהיה חד משמעית.

באופן פורמלי יותר – בהינתן כללי גזירה מהצורה הבאה –

$A \rightarrow \alpha b_1 \mid \alpha b_2 \mid \ldots \mid \alpha b_n$

כאשר ה-α הוא משתנה או טרמינל שונה מ-A, נעביר את כל החלק שמשאל לאיזור המשותף לכלל גזירה חדש שיהיה חד משמעי –

$A \rightarrow \alpha A'$
$A' \rightarrow b_1 \mid b_2 \mid \ldots \mid b_n$

הצבת גזירות במקום משתנים

החלק הזה לא ממש מופיע במצגות, אבל חשוב לציין אותו[1]. בעצם עבור כל כללי גזירה שנקבל, אנחנו נצטרך לבדוק האם יש לנו רקורסיה או התנגשות ב-first. הבעיה היא, שאחרי שאנחנו משכתבים דקדוק, אנחנו עלולים להתקל בבעיה שאם לא נעשה בדיקה חוזרת, אנחנו עלולים לחזור למצב המקורי הבעייתי, או להחליף בעיית רקורסיה בהתנגשות עם first של משתני גזירה שונים.

לכן, לאחר כל סבב תיקונים, יש לבדוק האם יש רקורסיה שמאלית, שזה יחסית קל ואינטואיטיבי. ובנוסף, יש לבדוק בכל פעם את כל ה- first של המשתנים ולבדוק האם יש התנגשות. לפעמים אנחנו עלולים להתקע במצב בו המשתנים החדשים שהוספנו (בדרך כלל הם יהיו הבעיתיים) יוצרים התנגשויות חדשות שלא ניתנות לתיקון על ידי פקטורינג רגיל. במקרה כזה אנחנו נציב פשוט את כל הכלל הימני במקום המשתנה וננסה להמשיך משם.

למורך הדוגמה – נראה את הדקדוק שהביאו לנו בתרגיל 2-

[1] ותודה לרפי קנול שהעלה את זה בצורה מפורטת

F → foreach var (L) | foreach var (R)
L → L , I | I
I → list | var
R → var .. var

אנחנו יכולים לראות באופן די פשוט שיש לנו התנגשות של first ב-F, וגם רקורסיה שמאלית ב-L. נפעיל את הסדר הנכון על מנת לתקן אותם, ונשנה את כללי הגזירה לצורה הבאה –

F → foreach var (F'
F' → L) | R)
L → I L'
L' → ,IL' | ε
I → list | var
R → var .. var

לכאורה, פתרנו את הבעיות שהעלינו, אבל אם נבדוק עכשיו את ה-first של כל המשתנים, אנחנו נראה שיש לנו התנגשות –

First (F) = {foreach}
First (F') = first(L) ∪first(R) = {list, var}∪{var}= {list, var}
First(L) = first(I) = {list, var}
First (L') = {‘,’,ε}
First (I) = {list, var}
First (R) = {var}

קל לראות, שה-var הזה עושה לנו בעיה רצינית – F' גוזר הן ל-L והן ל-R, ושניהם גוזרים ל-var, כך שאם נראה var בקלט, אנחנו לא נדע לאיזה מסלול ללכת. בשלב הזה אנחנו נתחיל עם ההצבה – במקום להשאיר את L ו-R, אנחנו נכתוב בתוכם פשוט את כלל הגזירה הימני שלהם, ונקבל את הדבר הבא –

F → foreach var (F'
F' → listL') | varL') | var..var)
L → I L'
L' → ,IL' | ε
I → list | var
R → var .. var

נסתכל שוב על מה שיש לנו – קודם כל יש התנגשות מאוד בולטת ב-first של F', שאותה אנחנו יכולים לתקן עם left-factoring, אבל מעבר לזה, אנחנו עכשיו לא מגיעים בכלל למשתנה L, כי כלום לא גוזר אליו, וגם R עכשיו כבר לא רלוונטי . לכן נעשה את התיקון הנדרש ונוריד את המיותר, ונקבל את הכללים הבאים –

F → foreach var (F'
F' → listL') | varR')
L' → ,IL' | ε
I → list | var
R' →.. var | L'

ועכשיו הכל בא על מקומו בשלום.
לסיכום – יש לבדוק בכל איטרציה האם אנחנו עדיין שומרים את כל הדרישות שלנו לאחר התיקונים שלנו. אם יש בעיה – עלינו לתקן את הכללים עד שנגיע למצב שאנחנו מסודרים ואין התנגשויות.

LL(k) Parsers

דיברנו כבר על אלגוריתם של Recursive Descent שמתעסק עם שפות LL(1), והראינו את כל התהליך. הבעיה היא שאנחנו אף פעם לא ששים במיוחד להשתמש ברקורסיה, מהסיבה הפשוטה שזה תוקע לנו את המחסנית ועלול להוביל לעומס על המערכת. מעבר לזה, כתיבת הקוד בעצמה היא קצת מסורבלת עם כל בניית העץ והבנים.

אנחנו מחפשים דרך לעשות את זה קצת יותר קל, ולכן אנחנו נעובר לשיטה שמקצרת לנו את הרקורסיות – טבלה. כעת כל מה שנצטרך לטפל בו יהיה מחסנית שתעבוד עם הקלט. אלגוריתמים שכאלה המבוססים על טבלה, נקראים LL(k) Parsers.

טבלת המעברים

על מנת לחסוך לנו את החיפוש הארוך, מה שנעשה הוא לדאוג מראש שתהיה לנו טבלה מתאימה הבודקת את כל מקרי הקצה האפשריים. ואז בזמן שאנחנו רצים על הקלט, אנחנו פשוט מסתכלים בטבלה, ויודעים מה המצעד הבא שלנו. אז, אם יש לנו את הטבלה הזאת, למה אנחנו צריכים בכלל את כל הקוד שלמדנו? מהסיבה הפשוטה שטבלאות זה אולי נוח וכיף, אבל אנחנו לא לומדים קומפיילרים בשביל להנות. המטרה שלנו היא לתת פלט של עץ גזירה בשביל השלב הבא, ואם נעבוד עם טבלאות לא יצא לנו אפילו שיח קטן של נענע.

אנחנו נבנה את הטבלה בצורה הבאה –

- שורות הטבלה – משתנים – כמובן שנעשה את זה לאחר שנפתור עד כמה שאפשר את כל ההתנגשויות של כללי הגזירה השונים.
- עמודות הטבלה – טרמינלים – כל הטרמינלים האפשריים, אחד אחרי השני.
- תוכן הטבלה – חוקי גזירה – אנחנו ממספרים את כל כללי הגזירה שיש לנו – אם למשל יש לנו כלל שיש לו שלוש אפשרויות גזירה, כל כלל יהיה מספר נפרד. מה שאנחנו עושים, הוא עבור כל משתנה גזירה (שורה מסוימת) בודקים לפי ה-first איזה טרמינלים יכולים להיות רלוונטים אליו, ואז רושמים באותו תא את מספר כלל הגזירה.

ניקח למשל את כללי הגזירה הבאים –

(1) E → LIT
(2) E → <u>(</u> E OP E <u>)</u>
(3) E → <u>not</u> E
(4) LIT → <u>true</u>
(5) LIT → <u>false</u>
(6) OP → <u>and</u>
(7) OP → <u>or</u>
(8) OP → <u>xor</u>

שימו לב, שאת כללים 1-3 היינו פשוט רושמים עד עכשיו בתור כלל אחד – E-> LIT | (E OP E) | not E, אבל תכלס מדובר פה על שלושה מסלולים שונים ולכם שלושה כללי גזירה שונים.

עכשיו, אנחנו נוציא את ה-first של כל משתנה –

First(E) = {(, not} ∪ First(LIT) = {true, false} = { (, not, true, false}
First(LIT) = {true, false}
First(OP) = {and, or, xor}

עכשיו ניצור את הטבלה המתאימה –

	(	)	not	true	false	and	or	xor	$
E	2		3	1	1				
LIT				4	5				
OP						6	7	8	

כמה דברים שרצוי להעיר – קודם כל, לא סידרנו את החוקים כמו שצריך, כי כמו שאפשר לראות לטרמינלים true ו-false יש שתי אפשרויות להגיע אליהם – כלומר, הם first של שני משתנים. חוץ מזה – הוספנו את **כל** הטרמינלים. גם את) שאף אחד לא מגיע אליו בכלל, וגם את $ שמסמן לנו את סיום הקלט. למה עשינו את זה? למעשה, נרחיב את השאלה בכלל למה אנחנו עושים עם כל התאים הריקים? נראה בהמשך איך אנחנו עובדים עם המחסנית – אנחנו שמים שם משתנים ואת הגזירות שלהם, ואנחנו עובדים על ראש המחסנית אל מול ראש הקלט. אם אנחנו עכשיו עומדים על E בראש המחסנית, ומקבלים בראש הקלט (, זה אומר שמשהו פה לא נכון, ואנחנו צריכים לזרוק שגיאה.

FOLLOW

לפני שנמשיך, יש להרחיב רגע בעניין ה-follow – העוקב.

ראינו כבר קודם את ה-first, וגם עכשיו נשתמש בו, אבל נראה בהמשך, שלפעמים לדעת רק מה הראשון לא מספיק לנו, ואנחנו גם רוצים לדעת מה יהיה הטרמינל הבא שיופיע לנו בסיום הגזירה של המשתנה הנוכחי, או במילים אחרות – מה ה-first של המשתנה הצמוד.

נראה בהמשך מה יהיה השימושים שלנו ב-follow הזה, אבל יותר חשוב לנו עכשיו זה להבין איל מוצאים אותו.

נתחיל במקרה הפשוט ביותר –

S -> ABc
B -> b

בשביל למצוא מה ה -follow(B), אנחנו צריכים למצוא את כל המופעים שלו בכללי הגזירה, ואז לקחת בכל פעם את הטרמינל הצמוד לו. במקרה שלנו, נגיד והכל מסתדר כמו שאנחנו רואים, אז מה שצמוד ל-B הוא c, ולכן הוא יהיה ה-follow. זה קל, אבל מה ה-follow(A)? מה שצמוד אליו הוא משתנה אחר, ולכן אנחנו ניקח את ה-first(B) ונגדיר שהוא יהיה גם follow(A). הרציונל מאחורי זה הוא כזה – בכל פעם כשיש לנו משתנה מסוים, הוא נגזר לעץ/תת-עץ כלשהו, גדול ככל שיהיה. מה הטרמינל אחריו? נלך למשתנה שצמוד לו, שפורס גם הוא עץ נחמד מתחתיו, וכאשר נסתכל מה יצא לנו בטרמינלים צמוד לסוף המשתנה הראשון יהיה ה-first של הצמוד לו.

לסיכום הרעיון הזה, נגדיר את הרעיון הכלל הראשון –

if A →**a**N**b**, Follow (N) includes First (**b**)

חשוב לציין – בהגדרה זו ובכל מה שיבוא אחריה, אנחנו לא אומרים שה-follow של אחד הוא ה-first של אחר, אלא שהוא **מכיל** אותו. אנחנו צריכים לזכור שאנחנו עוברים על כל המופעים של המשתנה N, ובכל פעם הוא יוסיף לנו עוד קצת לאוסף העוקבים.

עכשיו כשהבנו את המקרה הבסיסי, נעבור לחלקים המסובכים יותר –

if A → **a**N, Follow (N) includes Follow (A)

אם אנחנו נמצאים בסוף משתנה גזירה, אין לנו שום first להישען עליו. אבל זה לא אומר שאין אחד כזה, תמיד יש אחד כזה. אם לא מדובר במשתנה שנמצא צמוד, אז מדובר ב-follow של האבא – הצד השמאלי של הגזירה. למה? כי אם אנחנו עכשיו בקצה הקלט הנגזר ממשתנה מסוים, האסימון הבא אחריו יהיה שייך לעץ הבא, הוא יהיה הראשון של העץ הבא שהוא צמוד למשתנה העליון, כלומר אנחנו מדברים פה על אותו follow.

if A →**a**NB , B→**ε**, Follow (N) includes Follow (A)

הכלל הזה הוא בעצם נגזרת של הכלל הקודם – אם המשתנה Bגוזר ל-ε, אנחנו בבעיה. כי אנחנו אף פעם לא נראה אותו בקלט. עצם המהות של זה הוא שהוא לא יופיע, כך שלא נדע לומר מה העוקב שלו. אבל מה שאנחנו אומרים, הוא שאנחנו מתייחסים לסיום הקלט כמו שעשינו בכלל הקודם, ובודקים מה הוא העוקב של הצד השמאלי – האבא.

if A →**a**NB , B→b | **ε**, Follow (N) includes First(B)-{ε}∪Follow (A)

הכלל הזה לא מופיע במצגת, אבל ראוי להתייחס אליו. אם יש לנו משתנה שגוזר גם לטרמינל וגם ל-ε, אנחנו צריכים להתייחס לשתי האפשרויות האלה כשאנחנו בודקים את העוקב של N. קודם כל יש לנו את first(B), ואת כל הטרמינלים הקשורים לזה, אבל מאחר שאנחנו מתייחסים גם ל-ε אנחנו צריכים לזכור שהוא לא באמת קיים, ולכן לקחת את העוקב של B, אבל אין לנו שום דבר כזה כי הוא כבר בקצה המשתנה. ולכן נלך כמו בכלל הקודם, שנגזר מהכלל לפני, ונלך ל"דוד", המשתנה הצמוד לאבא.

דבר אחרון שאנחנו צריכים לדעת – אם יש לנו משתנה התחלתי S, אז Follow(S) includes $. כלומר האסימון שבא אחרי שאנחנו מסיימים לגזור את כל הקלט הוא ה-$ שמסמן לנו את סוף הקלט.

גזירה בהינתן טבלה

ברגע שסידרנו את הטבלה, איך אנחנו מתחילים לעבוד? קודם כל, אנחנו מתחילים עם מחסנית אליה אנחנו מכניסים קודם כל את $ כביטוי לסוף הקלט, ואת המשתנה ההתחלתי. כזכור, אנחנו עובדים מלמעלה למטה, אז יהיה לנו משתנה שיוגדר בתור שורש העץ. בשלב הבא, אנחנו בודקים את האסימון הראשון בקלט (משמאל לימין כמובן), ואל מולו את ראש המחסנית ופועלים בהתאם –

- **<u>אם בראש המחסנית יש משתנה</u>** – מה שבעצם אמור לקרות בשלב הראשון – יש לנו בראש המחסנית את המשתנה ההתחלתי / שורש העץ. אנחנו בודקים כרגע את ראש הקלט ועוברים לתא המתאים בטבלה -
  - <u>אם מסומן לנו כלל גזירה</u> – שולפים החוצה את משתנה הגזירה (המכונה "צד שמאל" של כלל הגזירה), ומכניסים במקומו את צד ימין של הגזירה – כל מה שאמור להיות שם, משתנים וטרמינלים כאחד.
  - <u>אם התא ריק</u> – שגיאה – על פי מה שהגדרנו מדובר בתו שהוא לא חלק מה-first של הכלל.
- **<u>אם בראש המחסנית יש טרמינל</u>** – דבר שבעצם עלול לקרות החל משלב מסוים, אחרי שהוצאנו משתנה ושמנו את הצד הימני, חלק ממנו אמור להיות טרמינלי. שוב, נבדוק את ראש הקלט –
  - <u>אם ראש הקלט וראש המחסנית שווים</u> – עוברים לאסימון הבא. למעשה אנחנו עוברים הלאה גם בקלט, וגם שולפים החוצה את הטרמינל מהמחסנית.
    - <u>אם מדובר ב</u>-$ - סיימנו את הריצה על הקלט.
  - <u>אם ראש הקלט וראש המחסנית לא שווים</u> – שגיאה.

נראה דוגמא מאוד פשוטה, רק בשביל לטעום את החוויה. בגדול במבחן ובתרגילים נצטרך לעשות טבלאות דומות וגם להראות את הריצה, ולכן כדאי לשים לב –

ניקח כלל גזירה יחיד – A -> aAb | c. נבנה עבורו טבלה פשוטה –

c	b	a	
2		1	**A**

ועכשיו נראה את הריצה על הקלט aacbb

הקלט	המחסנית	כלל הגזירה	הערות
aacbb$	A&	1	אנחנו מוציאים את A, ומכניסים במקום את aAb
aacbb$	aAb$		טרמינל מתאים – מוציאים את a מראש הקלט והמחסנית
acbb$	Ab$	1	
acbb$	aAbb$		התאמת טרמינל
cbb$	Abb$	2	מוציאים את A ומכניסים את c
cbb$	cbb$		התאמת טרמינל
bb$	bb$		התאמת טרמינל
b$	b$		התאמת טרמינל
$	$		התאמת טרמינל וסיום

בתרגול יש דוגמה יותר ארוכה ומורכבת, שתביא את הפתרון בצורה שמספיקה להבנה מלאה.

טיפול בשגיאות

דיברנו על כך שאנחנו עלולים להתקע אם שגיאות תוך כדי הריצה, אבל גם אם יש לנו שגיאה, אנחנו רוצים לדעת שיש לנו דרך וודאית לצאת ממנה, וגם אם לא לצאת. לפחות להמשיך לעבוד בצורה שתשביע רצון.

באילו סוגי שגיאות אנחנו עלולים להפגש?

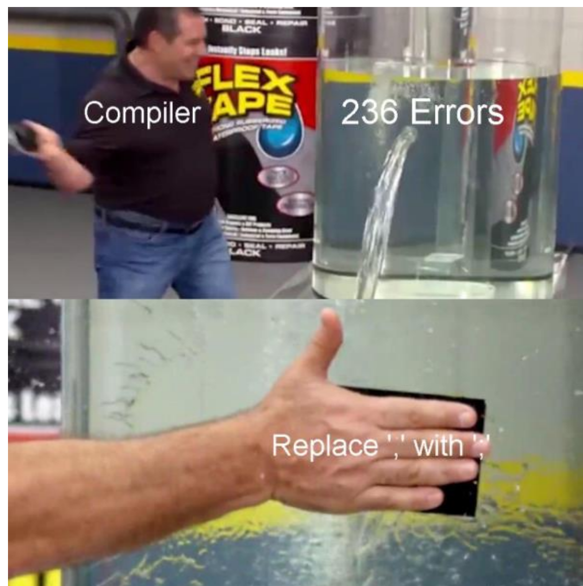
- שגיאות לקסיקליות – טעויות דפוס.
- שגיאות תחביריות – למשל, סוגריים לא מאוזנים. זה מבחינתנו מאוד בעייתי, כי אנחנו רוצים ליצור פה עץ גזירה שדי נשען על תחביר מאוזן ונכון.
- שגיאות סמנטיות – אופרנדים מטיפוס לא מתאים (float שמוגדר בתוך int ודומיהם)
- שגיאות לוגיות – גם כאלה שנתקעים בלולאה אינסופית, וגם סימן השמה (=) במקום השוואה (==).

בכל מקרה כזה, אנחנו רוצים לנסות ולתקן ברמה המינימלית, בשביל שנוכל להמשיך ולרוץ. אבל אנחנו לא מתקנים וממשיכים, אלא גם זורקים שגיאה, שהמתכנת יוכל לבדוק בעצמו ואולי לתקן את זה באופן שונה ממה שאנחנו עשינו.

בשביל להיחלץ משגיאות, יש כמה מתודות שהקומפיילר יכול להשתמש –

Panic Mode – אם חלק מהקוד לא עובד, אנחנו עושים את הדבר הפשוט וההגיוני ומתעלמים מהחלק הנגוע. אנחנו קופצים ישר לסוף השורה או סוף הקטע וממשיכים לנסות לרוץ משם. אילוסטרציה:

Phrase-Level Recovery – גישה קצת יותר ריאלית – מנסים לעשות תיקונים מקומיים, כמו החלפת פסיק (,) בנקודה-פסיק (;) או להוריד אחד כזה ו"לאחד פקודות". כמובן שאם השגיאה קרתה לפני ופה זה רק הסימפטום, זה לא ממש יעזור לנו. אילוסטרציה:



Error Production – טיפול בשגיאות צפויות – החלפה של אותיות קרובות או טעויות נפוצות אחרות. למשל: מי שמתכתב הרבה דרך המחשב, יודע לקרוא באופן אוטומטי את המילה "xccv" בלי שיצטרכו לתקן לו אותה.

Global Correction – מעבר על הקוד ובדיקה של מה תיקון הטעות שיעלה לנו הכי פחות. לא יעיל בעליל ויקר מאוד. הרבה יותר זול פשוט לזרוק שגיאה ולתת למתכנת לאכול את הדייסה שהוא בישל.

לסיכום

אנחנו לוקחים את הפלט של הניתוח הלקסיקלי – האסימונים, ומנסים ליצור ממנו עץ גזירה. אנחנו עושים את זה על ידי דקדוק חסר הקשר. בשיטות LL(1) השונות אנחנו עובדים מלמעלה-למטה בעץ, על ידי רקורסיה או טבלה עם Lookahead של אסימון יחיד.

ניתוח תחבירי מונחה-טבלאות – LL מצגת דוגמאות מס' 3

בתרגול הזה אנחנו נעבוד על אותם כללי גזירה שראינו קודם, אך עכשיו זה יהיה ניתוח של LL(1) – גזירה top-down, מעבר משמאל לימין בקלט, ויצירת עץ גזירה שמאלי.

לצורך התזכור נראה את הדקדוק וכללי הגזירה –

G = ({A, N, E, T}, {**id**, **.**, **=**, **+**, **(**, **)**}, P, A)

כאשר כללי הגזירה P הם כדלקמן -

A ⟶ N **=** A | E
N ⟶ N **.** **id** | **id**
E ⟶ E **+** T | T
T ⟶ N | **(** A **)**

אנחנו רוצים לוודא שהכללים עליהם אנחנו עובדים יהיו כאלה שלא יצרו לנו בעיות והתנגשויות. לטובת זה, אנחנו קודם כל בודקים את הדבר הראשון שיעיד לנו על בעייתיות – בדיקת ערכי ה-first של כל משתנה. באופן הכי פשוט כדאי בדרך כלל לבדוק ערכי first מלמטה למעלה. כלומר, אם הוגדר לנו ש-A הוא המשתנה ההתחלתי, אנחנו נבדוק את ערך ה-first שלו אחרון. כזכור, אנחנו משתדלים לכתוב את כל הדרך באופן ה"ארוך" ביותר, כי זה יכול להציג לנו דברים ובעיות שאנחנו ניתקל בהם:

First(N) = First(id) = {id}
First(T) = First(N) ∪ First('(' A ')') ={id, (}
First(E) = First(T) = {id, (}
First(A) = First(N '=' A) ∪ First(E) = First(N) ∪ First(E) **={id} ∪ {id, (}** = {id, (}

מהסתכלות ראשונית, אנחנו כבר יכולים לזהות בעיה ב-A, כאשר אנחנו רואים שיש לנו איחוד בין שני מסלולים שונים (שני כללי גזירה שונים) ששניהם יובילו אותנו ל-id, ואיחוד ביניהם אמנם אפשרי אבל הפוטנציאל של זה הרסני.

מעבר לזה, כל משתני הגזירה מגיעים ל-id. לכשעצמו זה לא בעיה, שכולם יגזרו ל-id. הבעיה היא שכולם מגיעים לשם דרך N, וזה יכול לעשות קצת בעיות בדרך – אם ניתקל ב-id, אין לנו שום דרך לדעת מי המשתנה שגזר אליו.

על מנת לראות הכל בצורה מסודרת יותר, נסדר הכל בטבלה ונראה איפה אנחנו נופלים –

	Id	.	=	+	(	)	$
A	N = A, E				E		
N	N . Id, id						
E	E + T, T				E + T, T		
T	N				(A)		

החלקים המסומנים באדום, הם החלקים הבעייתיים – A יכול לגזור ל-id. הבעיה היא, שהוא יכול להגיע אליו משני כללי הגזירה. עכשיו כשיהיה לנו id כלשהו בקלט, ו-A במחסנית הפעולות, אנחנו לא נדע מה אנחנו אמורים לעשות ובאיזה דרך לבחור. נזכור – אנחנו רוצים שכל הדרכים שנלך בהם יהיו דטרמניסטיות וחד משמעיות, ואם לא נוכל לעשות את זה אנחנו לא נבצע את הדרוש.

למעשה, הבעיה שלנו קיימת קודם כל בשני משתני הגזירה N ו-T, שלכל אחד מהם יש רקורסיה שמאלית בחוקי הגזירה שלו, נחזור שוב על הבעיה ונראה איך אנחנו פותרים אותה. לאחר מכן ננסה להתעסק גם עם חוסר ההחלטיות בכללי הגזירה של A.

מאחר ואנחנו עוברים על הקלט משמאל לימין, ובונים את העץ מלמטה למעלה אנחנו עלולים להתקל במצב הבא – נניח והקלט המוצג לנו הוא כזה:

a.b.c

כל אחת מהאותיות האלה היא כמובן משתנה, או id. עכשיו, כאשר נגיע למשתנה N שירצה לגזור את הקלט, יעמדו בפנינו שתי אפשרויות – האחת, לבחור את כלל הגזירה N→N.id ואז שוב את אותו כלל, ורק בסוף לאחר שיהיה לנו N.id.id, נגזור את ה-N הזאת ישירות ל-id. כל זה טוב ויפה כשאנחנו עושים את זה בעצמנו, אבל המחשב לא יודע מה יש לפנינו, ולכן אנחנו צריכים לחשוב על זה אחרת.

אנחנו רוצים למצוא את הדרך בו אנחנו נוכל לבחור בצורה חד משמעית מה שקורה, ולכן ננסה לבחון את כלל הגזירה ולראות מה הוא נותן לנו. כשיש לנו את הכלל הבא –

N → N **. id** | **id**

בעצם, יהיה מה שיהיה, אנחנו יודעים שבסוף בשביל לעצור את הגזירה הזאת, N יגזור ישירות ל-id ויעצור שם כמו בכלל השני שלו. מה שיעקוב אחרי אותו id יהיה שרשור של id. מספר פעמים (או כלום). אם ננסה להסתכל על ביטוי רגולרי מתאים, זה בעצם יוצר לנו משהו כזה – id(.id)* אז עכשיו אנחנו רק צריכים לראות איך נבטא אותו בגזירה. הכלל שמנחה אותנו, הוא להשאיר במשתנה המקורי את כל הדברים שאנחנו בטוח גוזרים ישירות אליהם, טרמינלים ומשתנים (שאינם המקורי), ולהצמיד אליו יציאה למשתנה חדש. במקרה שלנו, אנחנו נגדיר כך:

N→ idN’

עכשיו נטפל במשתנה החדש (אחרי זה אני אעבור על הכל שוב בצורה הפורמלית יותר). אנחנו רוצים לבטא פה את האפשרות לשרשור של (.id)* לצורך זה יש לנו בעצם שתי אפשרויות, קודם כל – השרשור עצמו – אנחנו ניתן למשתנה לגזור לחלק שהוא רוצה ואז לחזור על הגזירה הזאת, באופן הבא:

N’→.idN’

עכשיו, כמה עמוק שנרצה להכנס בתוך ה-id נוכל להמשיך ולשרשר בלי לחשוש שמה אנחנו עושים את זה יותר מידי או פחות מידי. איך נדע מתי לעצור את השרשור? אם נראה פתאום בקלט, איזה טרמינל שלא קשור ל-id, נבין שעברנו למשתנה אחר. הדבר האחרון שנצטרך הוא לוודא שאנחנו סוגרים את השרשור הזה, לכן את ה-N’ האחרון שיהיה לנו נגזור לאפסילון ככה שיהיה לנו סגירה בטוחה –

N’→ ε

למעשה, הרווחנו פה דבר נוסף. כזכור, סימן הכוכבית מבטא גם אפשרות של אפס פעמים. אם אנחנו גוזרים לאפסילון, אנחנו מוסיפים בעצמנו את האפשרות לא להמשיך ולגזור אחרי ה-id הראשוני.

על מנת לראות שוב את החוקים של ביטול רקורסיה שמאלית אפשר להסתכל קצת למעלה בסיכום [כאן](#) (רקורסיה שמאלית, למי שזה לא מודפס לו).

עד עכשיו טיפלנו בכלל גזירה אחד, נעשה את אותו סיפור ל-E, ונקבל את כללי הגזירה הבאים –

E → T E’
E’ → **+** T E’ | ε

זה היה החלק הקל.

במצגת אנחנו רואים שעשו לנו גם הרחבה של הדקדוק של A –

A → N A’ | **(** A **)** E’
A’ → **=** A | **+** E | ε

איך הגיעו לסיפור הזה, ומה בכלל קורה פה?

ראשית, נבין את הבעייתיות שעדיין קיימת. גם לאחר שטיפלנו ב-N וב-E, עדיין A גוזר לשניהם, ואנחנו לא יודעים לאיזה מהם ללכת. נסתכל על החוקים הקיימים לנו שכבר טיפלנו בהם, ובטבלה המתאימה להם –

A ⟶ N **=** A | E
N ⟶ **id** N’
N’ ⟶ **. Id** N’ | ε
E ⟶ T E’
E’ ⟶ **+** T E’ | ε
T ⟶ N | **(** A **)**

	Id	**.**	**=**	**+**	**(**	**)**	**ε**	**$**
A	N=A,E				E			
N	idN’							
N’	.idN’						ε	
E	T				T			
E’				+TE’			ε	
T	N				(A)			

אם כן, עדיין יש לנו בעיה עם הכללים של A. לכן, אנחנו נפנה לשלב הפיתרון הבא, וזה הצבת גזירות במקום המשתנים. כלומר, במקום לגזור למשתנים אחרים שעושים בעיות, נציב שם את ה-first של משתני הגזירה ומשם נפתור את הבעיה.

A→N = A
A→TE’

אבל גם T מביא לנו id, אז נמשיך להציב –

A→N = A
A→NE’
A→(A)E’

עכשיו הגענו למצב חדש – יש לנו שני כללי גזירה של A שמביאים לנו את אותו המשתנה. לכן נעשה left-factoring, וניצור משתנה גזירה חדש A’ –

A→NA’ | (A)E’
A’→=A | +E |ε

עוד כמה תחנו שדולגו בדרך – E’ למעשה הוא +TE’ שמקביל למה שצומצם שם +E. ה-ε הגיע לשם כי A יכול לגזור בצורה עקיפה ישירות ל-N ואז אנחנו צריכים לבלום את A’. תכלס, יש פה קצת יותר מידי דילוגים שיהיה אוד קשה לחשוב עליהם לבד. בכל אופן, נמישך.

עכשיו עלינו ליצור את ערכי ה-first החדשים עבור המשתנים שהוספנו, ומאחר ובדרך העבודה הוספנו לנו גם את ε אנחנו צריכים לדאוג גם ל- followשל כל משתנה. למה נוצר לנו המצב הזה? אנחנו הולכים לבדוק את הקלט אל מול ערכי ה-first שאנחנו חישבנו (במצב נורמלי), אבל אנחנו עלולים פתאום לראות שאנחנו בכלל גזירה מסוים ומקבלים קלט אחר לגמרי ממה שציפינו לו. המצב הזה עלול לקרות בגלל שאולי הגענו לאפסילון, ונהיה פה איזה דילוג שאנחנו לא יכולים לראות (כי אפסילון זה אוויר), ומה שאנחנו רואים אל מולנו הוא כבר ה-follow של אותו משתנה. לכן, נבדוך גם את האפשרויות האלה.

First(N) = {**id**}
First(N’) = {**.**, ε}
First(T) = First(N) ∪ {**(**} = {**id**, **(**}

First(E') = {**+**, ε}
First(E) = First(T) = {**id**, **(**}
First(A') = {**=**, **+**, ε}
First(A) = First(N) ∪ {**(**} = {**id**, **(**}

Follow(A) = {$, **)**}
Follow(A') = Follow(A) = {$, **)**}
Follow(E) = Follow(A') = {$, **)**}
Follow(E') = Follow(A) ∪ Follow(E) = {$, **)**}
Follow(T) = First(E') = {+, **ε**} ➔ {**+**} ∪ Follow(E') ∪ Follow(E) = {$, **)**, **+**}
Follow(N) = First(A') ∪ Follow(T) = {$, **=**, **+**, **)**, **ε**} ➔ {$, **=**, **+**, **)**} ∪ Follow(A) = {$, **=**, **+**, **)**}
Follow(N') = Follow(N) = {$, **=**, **+**, **)**}

שימו לב אל הכללים שראנו קודם – אם המשתנה הופיע בסוף הגזירה, לקחנו את העוקב של מי שגזר אליו, ובמקרים בהם העוקב גוזר ל-ε הוספנו עוד דור של follow.

לסיכום, נקבל את הטבלאות הבאות –

S	A	A'	N	N'	E	E'	T
First(S)	id, **(**	=, +, ε	id	., ε	Id, (	+, ε	**id, (**
Follow(S)	$,)	$,)	$,),+,=	$,),+,=	$,)	$,)	**$,), +**

ניצור את טבלת הפעולות. כזכור, השורות הן המשתנים כולל אלו שיצרנו בעצנו, ובעמודות אנחנו נכניס את הטרמינלים. בכל תא בטבלה בו יש מפגש של טרמינל ומשתנה, אנחנו נכתוב בו את החלק הימני של כלל הגזירה איתו אנחנו עובדים.

	id	.	=	+	(	)	$
A	N A'				**(** A **)** E'		
A'			= A	+ E		ε	**ε**
N	Id N'						
N'		. Id N'	ε	ε		ε	**ε**
E	T E'				T E'		
E'				+ T E'		ε	**ε**
T	N				**(** A **)**		

עד עכשיו היו לנו רק ההכנות לכל העבודה האמיתית. כעת נעשה דוגמת הרצה עבור הקלט הבא –

a = b . c . d + (e = f) $

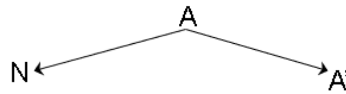
אנחנו ניצור לנו טבלה בעלת שתי עמודות, כאשר בעמודה אחת אנחנו נכניס את הקלט ועליו נעבוד, ובעמודה השניה תהיה לנו מחסנית הפעילות בה נכניס את כל המשתנים שאנחנו גוזרים אליהם. במקביל, אנחנו נדאג לבנות את העץ מלמעלה כלפי מטה על פי כללי הגזירה השונים.

שארית הקלט	מחסנית
a = b . c . d + (e = f) $	A $

כאמור, מאתחלים את המחסנית עם המשתנה ההתחלתי (במקרה שלנו A). ועכשיו דרך הפעולה שלנו הוא לראות את הטרמינל בראש הקלט, ולהשוות בטבלה מה אנחנו עושים אל מול מה שקיים בראש המחסנית. כאן אנחנו רואים a(id) ואנחנו במשתנה A, לכן אנחנו נגזור את ה-A ל-NA'. כלומר, נוציא את ה-A ונחליף אותו בחלק הימני של כלל הגזירה –

מחסנית	שארית הקלט
NA' $	a = b . c . d + (e = f) $

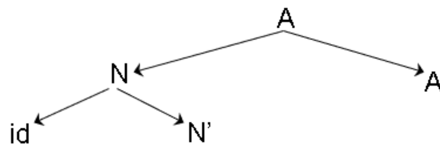
בנוסף, אנחנו יכולים להתחיל ולבנות את העץ. אם אנחנו פיצלנו את המשתנה לשנים חדשים, אנחנו יכולים ליצור את ראש העץ עם שני בנים, באופן הבא –



עכשיו נשווה (N,a) שזה סימון שהרגע המצאתי ולא רשמי או פורמלי בשום אופן, ונחליף את N ב-idN' –

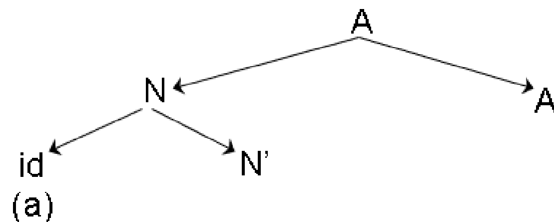
מחסנית	שארית הקלט
idN'A' $	a = b . c . d + (e = f) $

כמו כן, אנחנו נעדכן את העץ –



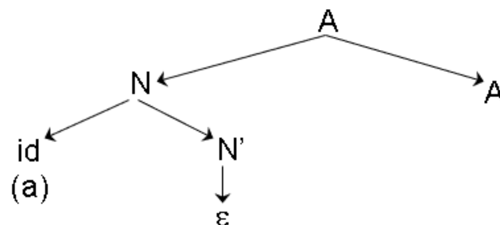
עכשיו אנחנו משווים (id,a) כאשר a הוא רק שם של id כלשהו, ואנחנו מזהים שיש לנו התאמה. במקרה הזה אנחנו פשוט מורידים את הטרמינל משארית הקלט, וכן את סימון ה-id מהמחסנית. בנוסף, נהוג לרשום את שם הטרמינל מתחת ל-id –

מחסנית	שארית הקלט
N'A' $	= b . c . d + (e = f) $



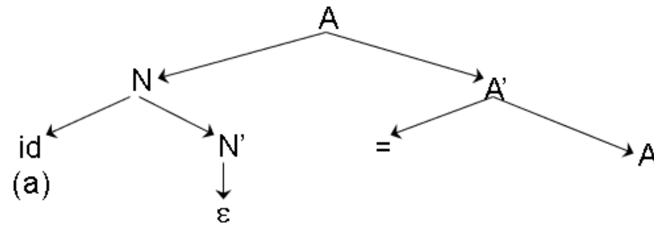
אם הגענו לטרמינל, זה אומר שסיימנו עם אותו החלק של תת העץ, ונעבור למשתנה הבא -N'. ε = (N',=), כלומר כאן יש לנו גזירה יחידה, אנחנו פשוט "סוגרים" את N' עם הטרמינל הריק, וממשיכים הלאה –

מחסנית	שארית הקלט
A' $	= b . c . d + (e = f) $

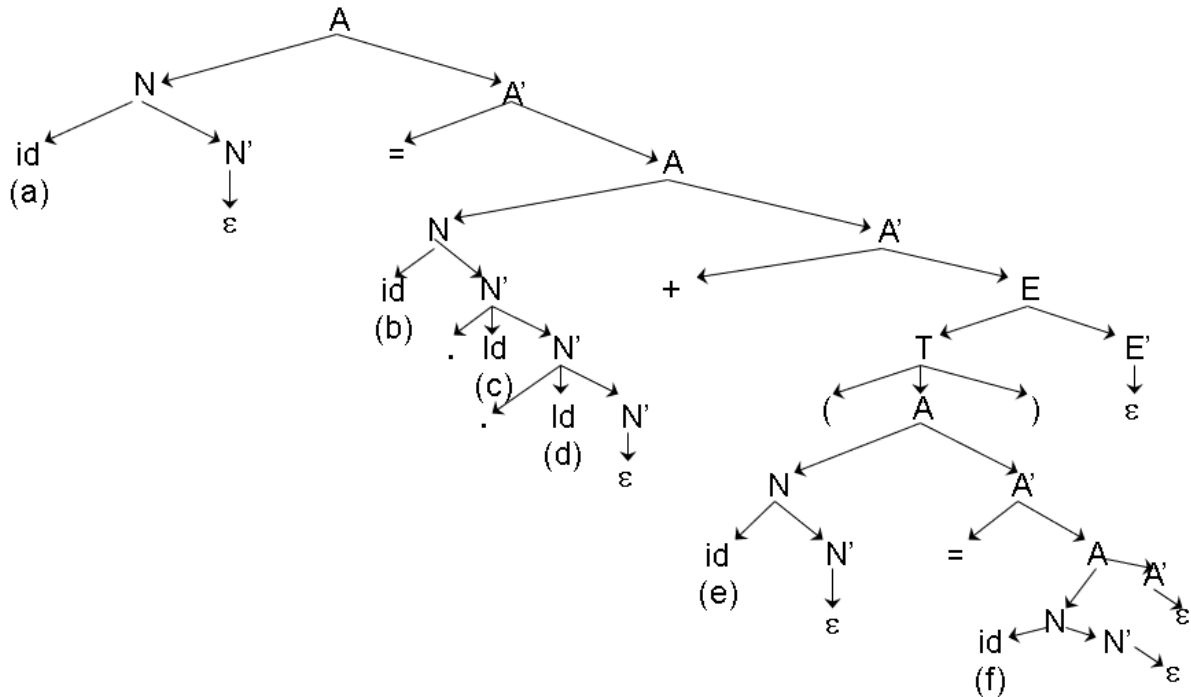


כאשר נשווה את (A',=) נקבל A=. נחליף במחסנית ובגרף –

מחסנית	שארית הקלט
=A $	= b . c . d + (e = f) $



תכל'ס, יש עוד כמעט 30 שלבים עד שאנחנו מסיימים את כל הריצה על הקלט. אני לא אעשה את כולם כי אין פה באמת איזה אתגר מסוים – ברגע שבנינו את הטבלה נכון, כל הבנייה של העץ מלמעלה למטה היא מאוד אינטואיטיבית ופשוטה, ואפשר לרוץ על זה מאוד בקלות ולקבל בסוף את העץ הבא –



איך נדע שאכן סיימנו בהצלחה? כאשר הקלט יסתיים, וכן גם המשתנים, וכל מה שיישאר לנו הוא ההשוואה הבאה –

מחסנית	שארית הקלט
$	**$**

אם הגענו לזה, אנחנו מורידים את הטרמינל הסוגר אל מול חברו וממשיכים הלאה בשמחה רבה.

דקדוקי LR

דקדוקי LR מביאים גישה שונה להתמודדות עם גזירת קלט, ונראה איך אנחנו עובדים עם הניתוח של הדקדוקים האלה .

דבר ראשון, ניתן כמה פרמטרים בדומה למה שראינו עד עכשיו, שיביאו לנו את ההבדל בין שני סוגי הדקדוקים –

- Bottom-up – הגזירה שלנו מתרחשת הפוך לגמרי מכל מה שעשינו ב-LL. בעוד שב-LL התחלנו בשורש (משתנה הגזירה התחילי), וניסינו לרדת למטה עד שנגיע לגזירה של הקלט הנתון לנו, כאן אנחנו בונים על גבי הקלט את העץ, או ליתר דיוק "יער של עצים", שלאט לאט יתחברו לנו אחד לאחד עד שנגיע לשורש העץ. **טיפ חשוב לעבודה**: בזמן שאנחנו מתרגלים, חשוב מאוד לבנות את העץ במקביל לעבודה על המחסנית. המחסנית עצמה מתרוקנת ומתמלאת, ואנחנו בטוח לא נזכור בעל פה את הדרך.
- מבוסס טבלאות – כאן אנחנו לא רק לוקחים את הטבלאות ככלי עזר, אלא בתור העבודה המרכזית איתה אנחנו מתעסקים.
- סורק את הקלט משמאל לימין – ה-L הראשונה.
- מניב עץ דקדוק <u>ימני</u> – ה-R.
- זקוק ל-Lookahead בגודל K.

גם כאן, שפה נקראת LR(K) אם יש לה דקדוק LR(K) שיקיים אותה.

המקרה הפשוט ביותר של LR(K) הוא LR(0). ה-Lookahead של משפחת ה-LR הוא שונה מהותית ממה שראינו עד עכשיו. ב-LL ה-lookahead התייחס לאסימון הבא שאנחנו רואים בקלט, ולכן מקרה הבסיס שלנו היה LL(1) כי אנחנו חייבים לראות משהו מינימלי. ב-LR לעומת זאת, אנחנו עובדים על הקלט מלמטה, ולכן בכל שלב יהיה לנו כבר בראש, או מול העיניים, את כל מה שכבר טיפלנו בו, ואנחנו שואפים שזה בעצמו יספיק לנו (ברוב המקרים) להחליט כבר מה נעשה במצב הבא. במקרים קשים יותר, כמו למשל LR(1), אנחנו נצטרך לראות את כל הקלט שעברנו עד עכשיו, ועוד אסימון אחד קדימה. מכאן אנחנו יכולים להבין שדקדוקי LR הרבה יותר חזקים מדקדוקי LL, עד כדי הקביעה כי $LR(K) \subset LL(K)$ מוכל ממש ולא מוכל/שווה – הLL(K) מסתכל רק ימינה K אסימונים, וה-LR(K) מסתכל שמאלה על כל האסימונים שעברו + K.

תהליך העבודה

על מנת להבין את תהליך העבודה, אנחנו ניתן דוגמה לאוסף כללים, ונראה איך עובדים ומחילים עליהם את הדקדוק והגזירה –

$E \rightarrow E * B \mid E + B \mid B$
$B \rightarrow 0 \mid 1$

כמו כן, נניח שנקבל את הקלט הבא –

0+0*1

צורת העבודה תהיה בכל פעם לעבור משמאל לימין, ולחפש אפשרויות שונות לצמצום למשתנים. בשביל שזה יהיה לנו יותר קל, ולמצוא איזו שפה משותפת, אנחנו מחלקים את הכללים וממספרים אותם –

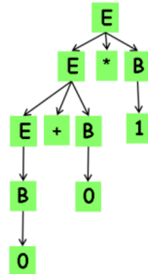
(1) $E \rightarrow E * B$
(2) $E \rightarrow E + B$
(3) $E \rightarrow B$
(4) $B \rightarrow 0$
(5) $B \rightarrow 1$

אנחנו מתחילים ב-0 הראשון, אנחנו יודעים שכלל 4 הוא $B \rightarrow 0$ ולכן אנחנו מחליפים את 0 –

B+0*1

במקביל, אנחנו מרימים מעל 0 קשת, ויוצרים עץ קטנטנן, בו B יהיה האבא של 0, כמובן שאז אנחנו בודקים שוב את ראש הקלט, ומעבירים את B להיות E (כלל 3). עם + לא נוכל לעשות יותר מידי חוץ מלדעת שאנחנו כנראה באיזור של כלל 2, וכאשר נגיע ל-0 השני, קודם כל נעביר אותו להיות B, ואז נשתמש בכלל ונאחד את כל מה שיש לנו עד כה תחת E, ואז נמשיך הלאה. בסוף, אם עבדנו נכון, נקבל את רצף עבודה , והעץ המקביל לו הבא –

0 + 0 * 1
B + 0 * 1
E + 0 * 1
E + B * 1
E * 1
E * B
E



עכשיו כל מה שנשאר לנו הוא להסביר מה עשינו ואיך עבדנו.

אנחנו עובדים בשיטה שנקראת "Shift & Reduce", שזה בעצם כל הפעולות שאנחנו עושים במהלך הריצה עצמה – קודם כל "Shift" – דחיפת קלט לתוך מחסנית העבודה. ואז "Reduce" – צמצום הקלט למשתנה (מלמטה למעלה!). בכל פעם שאנחנו עושים shift אנחנו מכניסים את הקלט הנוכחי ואת המצב בו אנחנו נמצאים (נרחיב על זה תיכף), וכשאנחנו מצמצמים, אנחנו מוציאים את כל מה שקשור לאותו כלל גזירה ומכניסים את השתנה החדש והמצב.

פריטים ומצבי LR(0)

אנחנו נגדיר עכשיו בעצם שני דברים שקשורים ל"הכנה" של הכללים לקראת הטבלאות וכל העבודה שתגיע בעקבותיהם –

<u>פריט</u> – הנקודה המסוימת בה אנחנו נמצאים בקריאה ביחס לכלל. אנחנו לוקחים את כל הכללים שעומדים בפנינו, ומסתכלים על כל האפשרויות בהם נעמוד במהלך קריאת הכלל. למשל, אם ניקח את הכלל –

$E \rightarrow E * B$

אז למעשה ב"אפשרויות" שאותם נראה כשייכים אליו, יופיעו לנו הדברים הבאים –

$E \rightarrow \bullet E * B$
$E \rightarrow E \bullet * B$
$E \rightarrow E * \bullet B$
$E \rightarrow E * B \bullet$

כאשר הנקודה האדומה עומדת **לפני** מה שאנחנו אמורים לקרוא עכשיו. כך שככל, ניתן לומר שעל כל כלל המכיל n אסימונים בצד שמאל, יהיו n+1 פריטים מתאימים – 1 לפני כל אסימון, ועוד 1 כאשר נסיים לקרוא את כל הצד הימני.

כך שכל כלל גזירה עם סימון מקום, ייחשב **פריט LR(0)**.

<u>מצב</u> – אובייקט שזוכר איזה כללים רלוונטים אלינו עד שהגענו לפה, ומה האפשרויות שלנו להמשיך. לצורך הדוגמה הפשוטה, אם ניקח את כללים 1,2, ועבדנו כל חלק מהקלט וכרגע במחסנית יש לנו E. המצב "יודע" שעברנו במסלול שהכיל את האפשרויות לשני הכללים האלה, וכל עוד לא נראה את הקלט הבא, שני הכללים עדיין רלוונטיים באותה מידה (מה שלא יקרה ברגע שנראה "+" למשל, ויהיה לנו מוגדר מאוד על איזה כלל גזירה אנחנו עומדים).

למעשה, **מצבים** הם אוסף של פריטים בנקודה מסוימת. עבור הכללים שראינו קודם, יוגדר המצב באופן הבא –

$S_{13} = \{ E \rightarrow E \bullet * B ,\ E \rightarrow E \bullet + B \}$

המספור בדרך כלל יהיה עוקב (לצורך נוחות בלבד) ואנחנו קוראים אותו באופן של "קראנו עכשיו קלט מסוים (E) ועכשיו יכול להיות שיהיו לנו אחד משני קלטים אחרים (+,*)".

האינטואיציה לניתוח LR(0)

נחזור על הפעולות שאמרנו קודם, אך עכשיו אנחנו יכולים קצת יותר להרחיב בעזרת ההגדרות שהבאנו.

כאשר אנחנו עושים **shift**, מעבר להכנסה של הקלט למחסנית אנחנו עוברים למצב חדש – במצב בו נהיה יהיו לנו כל הפריטים שאומרים לאן אנחנו יכולים להמשיך מפה, ומה האסימון האפשרי הבא שאנחנו אמורים לראות בקלט. כל מעבר שכזה יקדם אותנו ויעביר בין המצבים השונים. מתי נדע לעצור את העבודה? ברגע שנגיע לפריט של n+1, כלומר ברגע שסיימנו "לקבל" כלל ימני. בשלב הזה אנחנו עושים **reduce** ומצמצמים הכל למשתנה – החלק השמאלי של הכלל.

דבר זה בעצם יסביר לנו משהו שנראה קצת בעייתי – כשאנחנו עבדנו על הקלט הקודם, התחלנו לעבוד על מה שנדמה שייסגר ככלל הגזירה E+B, אבל בשלב מסוים, עמדנו מול E+1, ואז עצרנו את התהליך והתחלנו לעבוד על 1. בעצם, אנחנו הגענו למצב מסוים אחרי שעברנו את ה"+", ואנחנו מצפים לראות B, אבל בקלט עצמו לא יופיע B, אלא יופיעו 1 או 0, ולכן במצב המתאים לפריט הזה, אנחנו נכניס גם את האפשרויות של לעמוד ולראות את B, או למעשה את ה- first שלו. בהמשך נראה איך בדיוק בונים הכל.

דבר נוסף שיש לציין לפני שממשיכים – אמרנו שלכל כלל יש n+1 פריטים ביחס לכמות האיברים מצד שמאל. אך מה עם A→ε? יש לנו למעשה 0 איברים, ולכן יהיה לנו רק פריט בודד -

A→•

המחסנית

את כל רצף העבודה אנחנו עושים במחסנית. דיברנו על כך שאנחנו כל פעם מכניסים את הקלט ואת המצב למחסנית, למרות שנראה בהמשך שאין לנו צורך אמיתי בהכנסה של הקלט. מבחינת מה שאנחנו באמת צריכים זה רק המצבים – אם נבנה את הכל נכון, לא תהיה סיבה לראות מולנו את הקלט. אבל רק לצורך הקריאות עצמה, שיהיה לנו קצת יותר נוח לעבוד, נכניס גם את הקלט.

דבר נוסף שחשוב לשים לב - ה"קלט" שאנחנו מדברים עליו הוא לא רק אסימונים, אלא גם משתנים לאחר צמצום. כך שמה שיהיה לנו במחסנית יהיו צמדים של מצב-משתנה או מצב-אסימון. כל זה לא כולל אסימון ראשוני שנכניס לקלט $.

טבלת הפעולות

הטבלה תחולק באופן הבא –

- שורות – מצבים (על פי מספרים) – לאחר שנחלק את כל המצבים האפשריים, אנחנו נעשה את מספר השורות המתאים.
- עמודות – אסימונים אפשריים בקלט. גם פה אנחנו נכניס את **כל** האסימונים האפשריים, כולל $ (רק אפשר לשים לב שכאן יש שוני מ-LL כי אנחנו חייבים להיות מסוגלים להגיע לכל אסימון בשפה).
- תוכן הטבלה – פקודות עבודה –
  - Shift – מסמנים בתור sn. מעבירים את האסימון הבא לקלט ועוברים למצב המיוצג כ-n.
  - Reduce – מסומן בתור rn. זיהינו את כל הגזירה n ועלינו לצמצם על מנת להגיע אליו (שימו לב ש-n מבטא דבר שונה בכל פעם).
  - Accept – מסומן acc – סימון על גזירת הקלט בהצלחה.
  - Error – כל דבר אחר. פשוט תא ריק.

טבלת GOTO

בטבלת הפעולות אנחנו מסמנים מתי אנחנו צריכים לצמצם ולאיזה כלל עלינו להתייחס, אבל מה שחסר לנו זה מידע שאומר לנו לאיזה מצב אנחנו אמורים להמשיך עכשיו. ובכן, לזה יש לנו טבלה מיוחדת עם השם המקורי GOTO בה יהיה רשום לאן לעבור עכשיו.

שורות – מצבים – מה שיקל עלינו פשוט להצמיד את זה לטבלת הפעולות.

עמודות – משתנים – צד שמאל של כל הכללים.

תוכן – מצבים.

הרצת דוגמה

בשביל להבין יותר טוב את הרצף, אנחנו נעבור בדוגמת הרצה על בניית טבלה , ואחרי זה גם נראה איך אנחנו רצים על קלט נתון. ראשית נסתכל על כללי הגזירה:

(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

השלב הראשון הוא להכניס **דקדוק מורחב**. זה נראה קצת מוזר ושם מפוצץ לפעולה אחת פשוטה – אנחנו מוסיפים כלל גזירה 0, ש"יתחיל" לנו את כל העבודה. הדקדוק המורחב ייראה כך –

(0) S → E
(1) E → E * B
(2) E → E + B
(3) E → B
(4) B → 0
(5) B → 1

למה זה כל כך קריטי? כי אנחנו בונים את העץ מלמטה למעלה, ואנחנו צריכים בשלב מסוים לזהות שאנחנו יכולים לסיים את העבודה. השלה הזה יהיה המצב בו E יעבור להיות S, ואנחנו יוגים בוודאות שמכאן אין לאן להמשיך. רק הערה חשובה – שימו לב שאתם נותנים שם נכון לכלל ה-0. לפעמים הכלל הראשון יהיה של משתנה S, ואז נהוג להכניס S’.

עכשיו אנחנו מתחילים לבנות את המצבים השונים.

בניית המצבים

כמו שכבר אמרנו - המצבים הם אוסף של פריטים, כאשר כל פריט מציין מבט של הקומפיילר בנקודה נתונה בגזירה.

בשביל שנוכל להגדיר את המצבים בצורה המלאה ביותר, עלינו לדבר על ה-Closure. כשאנחנו עומדים לפני משתנה מסוים, אנחנו לא עומדים רק לפניו, אלא לפני כל בניו ובני בניו עד לסוף כל הדורות. למה הכוונה? באופן הכי פשוט, אם אנחנו עומדים עכשיו אל מול E, כלומר בפריט הבא:

S → •E

אז מה אנחנו מצפים לראות? את E, אבל E הוא משתנה, כלומר, אנחנו נראה את מה שנגזר ממנו, ואם נלך עד העומק נראה שבסוף הוא נגזר ל-0/1. ולכן מה שעלינו לעשות הוא להכניס למצב 0, בו S → •E או כמו שאנחנו

אוהבים להגיד "S רואה את E", את כל מה שייגזר מזה שרואים את E. אבל כדאי לשים לב, אנחנו לא מדברים פה על ה-first של E, אלא על כל הדרך עד לשם. כלומר –

$q_0 = Closure(\{S \rightarrow \bullet E\}) = \{S \rightarrow \bullet E , E \rightarrow \bullet E * B , E \rightarrow \bullet E + B, E \rightarrow \bullet B , B \rightarrow \bullet 0 , B \rightarrow \bullet 1\}$

כל זה, בעיתו ובזמנו יגדיר לנו מצב חדש – מצב q_0. כלומר, זה המצב ההתחלתי, ובהרבה מקרים הוא יהיה מפורט וארוך כמו שיש לנו פה. עכשיו איך אנחנו ממשיכים? אנחנו מסתכלים על כל פריט ופריט במצב, מזיזים את ה"סמן" (•) שיעבור את האיבר עליו הוא יסתכל, ואז אנחנו עושים סגוֹר (closure) על אותו פריט. הסימון הנהוג לשלב הזה הוא $q|x$ כשהכוונה היא "אנחנו נמצאים במצב q_i כלשהו, ומסתכלים על x". בדרך כלל נהוג פשוט לעבור פריט פריט מבין המצבים ולעבוד עליהם, אבל כאן בדוגמא עשו זאת קצת שונה, אז נזרום איתם. נתחיל דווקא מהטרמינלים –

$q_0|0 = \{ B \rightarrow \bullet 0 \}$
$q_1 = \{ B \rightarrow 0 \bullet \} = clos(q_1) = \{ B \rightarrow 0 \bullet \}$

מה נעשה פה? קודם כל, השורה הראשונה תיארה לנו את המצב הקודם q_0 רואה 0. בשורה הבאה אנחנו מתחילים ממש את העבודה. אנחנו מגדירים את q_1, הפריט ה"ראשי" שלו הוא השלב אחרי שראינו את 0, כלומר הנקודה אחרי הספרה. אנחנו שולחים את המצב הזה לפונקציית סגור, אך מאחר ואין לנו לאן להמשיך, אנחנו לא צריכים להמשיך לחפש ועוצרים כאן. המצב הבא הוא מאוד דומה רק עם 1 במקום 0 –

$q_0|1 = \{ B \rightarrow \bullet 1 \}$
$q_2 = \{ B \rightarrow 1 \bullet \} = clos(q_2) = \{ B \rightarrow 1 \bullet \}$

אין הרבה מה להרחיב.

השלב הבא – אנחנו חוזרים לתחילת הרשימה של הפריטים –

$q_0|E = \{S \rightarrow \bullet E , E \rightarrow \bullet E * B , E \rightarrow \bullet E + B\}$
$q_3 = \{S \rightarrow E \bullet , E \rightarrow E \bullet * B , E \rightarrow E \bullet + B\} = clos(q_3) = \{S \rightarrow E\bullet, E \rightarrow E\bullet*B , E \rightarrow E \bullet + B\}$

פה זה מתחיל להיות שונה – מה קורה כאשר q_0 רואה E? לא יודעים. בזמן שאנחנו רואים את E זה יכול להיות אחת מכל שלושת כללי הגזירה האפשריים, כשאנחנו רואים רק את E, אין לנו שום דרך לדעת בוודאות האם מדובר ב-E, או ב-E*B. ולכן חשוב לזכור – בכל פעם שאנחנו מסתכלים על משתנה או טרמינל באותו מצב, אך עם שני פריטים שונים, עלינו להכניס את שניהם למצב הבא ולראות מה קורה איתם.

למזלנו, אחרי שאנחנו מסיימים את E, יהיה שם מה שיהיה, אנחנו יכולים להסתכל על אחד משלוש טרמינלים, וזה סביר כי אין לנו לאן להמשיך פה, וסגרנו את המצב השלישי.

המצב הבא הוא לראות את B –

$q_0|B = \{ E \rightarrow \bullet B \}$
$q_4 = \{ E \rightarrow B \bullet \} = clos(q_4) = \{E \rightarrow B\bullet\}$

עכשיו סיימנו את כל המצבים הקשורים ל-q_0. אך האם סיימנו? q_1 ו-q_2 נמצאים כל אחד מהם בסוף הקלט, וכך גם q_4, אבל מה קורה ב-q_3? ששון ושמחה לכל המשפחה!

יש לנו שני פריטים שהם לא בסוף הקלט, אלא מול טרמינלים. מה יקרה אחרי שנקרא אותם? נעמוד מול B, זה ייצור לנו מצב נוסף עבור כל אחד מהפריטים הנ"ל –

$q_3|* = \{ E \rightarrow E \bullet * B \}$
$q_5 = \{ E \rightarrow E * \bullet B \} = clos(q_5) = \{ E \rightarrow E * \bullet B , B \rightarrow \bullet 0 , B \rightarrow \bullet 1 \}$

$q_3|+ = \{ E \rightarrow E \bullet + B \}$
$q_6 = \{ E \rightarrow E + \bullet B \} = clos(q_5) = \{ E \rightarrow E + \bullet B , B \rightarrow \bullet 0 , B \rightarrow \bullet 1 \}$

שני המצבים האלה הם כמעט אותו דבר, אבל יש הבדל בפריט המוצא שמשתנה לפי הפעולה, ולכן אנחנו מפרידים בין המצבים. האם סיימנו?

אנחנו יכולים לשים לב, שאנחנו יכולים עוד להתקדם קצת, מה קורה אם $q_5|0$? הפריט השני שם, גם הוא משהו. אבל בעצם אם נעשה לו סגור, נקבל את המצב הבא –

$q_5|0 = \{ B \rightarrow 0 \bullet \} = clos(q_1) = \{ B \rightarrow 0 \bullet \}$

שזה בדיוק מצב 1, ויש לנו מספיק בלאגן גם בלי להכניס מצבים כפולים. אבל מה כן יש לנו? כל אחד מהמצבים האלה שרואה את B –

$q_5|B = \{ E \rightarrow E * \bullet B \}$
$\mathbf{q_7} = \{ E \rightarrow E * B \bullet \} = clos(q_7) = \{ E \rightarrow E * B \bullet \}$

$q_6|B = \{ E \rightarrow E + \bullet B \}$
$\mathbf{q_8} = \{ E \rightarrow E + B \bullet \} = clos(q_8) = \{ E \rightarrow E + B \bullet \}$

ועכשיו, האם סיימנו? כן!

נשאר לנו לעשות את הטבלה עצמה, יש לנו את מספר המצבים שכבר קבענו, ואנחנו יודעי מה המשתנים, ועכשיו צריך להתחיל למלא את התוכן – נתחיל עם שני שלבים בסיסיים ביותר –

1. עבור כל מפגש של מצב-אסימון/משתנה אנחנו נרשום רק כמספר את המצב הבא אליו אנחנו עוברים. למשל, כשאנחנו במצב 0 ורואים את המספר 1 בקלט, אנחנו יודעים שאנחנו צריכים לעבור למצב 0. במצב 6 כשאנחנו נפגשים עם B, נעבור למצב 8.
2. כל מצב שבקבוצת הפריטים יש לנו את סיום קריאת הקלט, לצורך העניין - $S \rightarrow E \bullet$, אנחנו יודעים שאנחנו עלולים לסיים את הקריאה, ולכן אם נתקל ב-$ הקלט עבר בהצלחה, ונוסיף acc(ept).

מצב	טבלת הפעולות					טבלת goto	
	*	+	0	1	$	E	B
q0			1	2		3	4
q1							
q2							
q3	5	6			acc		
q4							
q5			1	2			7
q6			1	2			8
q7							
q8							

המצב הנתון לנו עכשיו, הוא שאנחנו יודעים לאיזה מצב אנחנו עוברים בהינתן כל קלט נתון. אבל עדיין לא דאגנו לפעולות עצמן. מה שנעשה הוא כזה – עבור כל מפגש בטבלת הפעולות, אנחנו נגדיר את זה כפעולת **Shift** ונסמן s ליד כל מספר.

מצב	טבלת הפעולות					טבלת goto	
	*	+	0	1	$	E	B
q0			s1	s2		3	4
q1							
q2							
q3	s5	s6			acc		
q4							
q5			s1	s2			7
q6			s1	s2			8
q7							
q8							

עכשיו עלינו לקבוע איפה נשים צמצומים **reduce** – נעבור על כל הכללים (המקוריים, לפני הרחבת כלל 0), ובכל מקום שאנחנו מגיעים למצב בו אנחנו מסיימים לקרוא חלק ימין של כלל גזירה, כלומר הנקודה האדומה עברה את כל הקלט, או באופן פורמלי יותר - • A → a, אז אנחנו נוסיף **לכל** השורה של אותו מצב את פקודת הצמצום, שבצידה יהיה כלל הגזירה אותו אנחנו מצמצמים. למשל, מצב 7 מסיים את כלל גזירה מספר 1, ולכן כל השורה שלו תהיה r1.

מצב	טבלת הפעולות					טבלת goto	
	*	+	0	1	$	E	B
q0			s1	s2		3	4
q1	r4	r4	r4	r4	r4		
q2	r5	r5	r5	r5	r5		
q3	s5	s6			acc		
q4	r3	r3	r3	r3	r3		
q5			s1	s2			7
q6			s1	s2			8
q7	r1	r1	r1	r1	r1		
q8	r2	r2	r2	r2	r2		

הגענו לטבלה הסופית, ועכשיו נותר לנו לראות איך אנחנו רצים על הקלט[2] –

בשביל זה נבדוק את הקלט - 0 + 0 * 1$

q_0									

כזכור, אנחנו מתחילים תמיד עם המצב 0 שמייצג לנו את התחלת העבודה. עכשיו אנחנו מסתכלים בטבלה שלנו על q0|0 ואנחנו רואים שיש לנו s1. כלומר, אנחנו עושים שיפט לתוך הטבלה, ומכניסים בנוסף את q_1. זה המקום להזכיר שאנחנו לא באמת צריכים להכניס את הקלט בשביל הרצה תקינה, אלא זה סתם מטעמי נוחות נטו. דבר נוסף וחשוב

[2] מצגת השיעור עושה את הריצה הקטנה הזאת לפני ההסבר על בניית הטבלה. אני העדפתי לשנות את הסדר, ולהריץ את הדוגמא הזאת עד הסוף.

לא פחות – אנחנו כבר עכשיו מתחילים את בניית העץ מלמטה למעלה! העבודה פה יחסית מאוד מסובכת עם הרבה הוצאות והכנסות ויהיה לנו מאוד קשה לשמור על הסדר. לכן נציג את הטבלה ואת בניית העץ –

q_0	0	q_1							

0

עכשיו אנחנו עומדים אל מול הקלט $1*0+, אבל כשאנחנו במצב 1, אין לנו באמת לאן להמשיך, אנחנו חייבים לעשות צמצום של כלל 4 – B→0. ולכן נוציא שני פריטים מתוך המחסנית (כזכור, פי 2 מאורך הצד הימני של כלל הגזירה, במקרה הזה 1), ונכניס את המשתנה אליו אנחנו גוזרים. כמו כן, נעדכן את העץ –

q_0	B								

B
0

מה עושים עכשיו? אנחנו בודקים את שני הפריטים האחרונים במחסנית, יש לנו משתנה ויש לנו מצב, עכשיו אנחנו יודעים ש $q_0|B$ מעביר אותנו למצב 4. לכן נוסיף גם אותו למחסנית –

q_0	B	q_4							

אבל! q_4 גם הוא דורש מאיתנו לצמצם בהוראת הכלל השלישי! נעדכן את הדרוש – נוציא את הפריטים מהמחסנית, נכניס את המשתנה אליו גוזרים, נבדוק את המצב של ה-GOTO ונעדכן את העץ –

q_0	E								

q_0	E	q_3							

E
B
0

נמשיך – $q_3|+ = s6$. נעדכן את העץ בתת העץ החדש המכיל את +

q_0	E	q_3	+	q_6					

E
B
0 +

המשך הקלט – $1*0

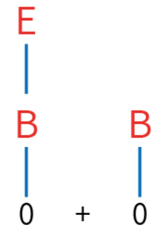
$(q_6|0) = s1$

q_0	E	q_3	+	q_6	0	q_1			

q_1 מפעיל צמצום, אנחנו בודקים את כלל 4 – B->0, ולכן אנחנו מצמצים ומעדכנים את העץ –

q_0	E	q_3	+	q_6	B				

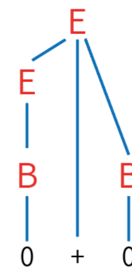
q_0	E	q_3	+	q_6	B	q_8			



המצב אליו הגענו עכשיו, q_8, גם הוא דורש לצמצם לכלל 2 - E → E + B. ולכן, נוציא 6 פריטים מהמחסנית ונעבוד בצורה הידועה, כאשר הפעם עם עדכון העץ, נקשר הכל למשתנה החדש-

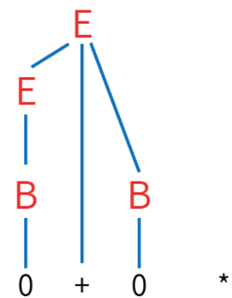
q_0	E								

q_0	E	q_3							



ננסה לעבוד קצת יותר מהר – $(q_3|*) = s5$

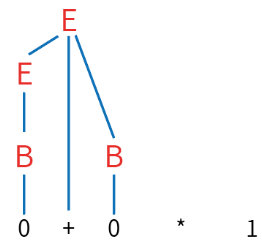
q_0	E	q_3	*	Q_5					



שימו לב שיש לנו כאן תת עץ חדש של *.

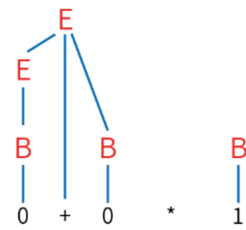
נעבור הלאה $(q_5|1) = s2$

q_0	E	q_3	*	q_5	1	q_2			



במצב 2, אנחנו מצמצמים את הטרמינל להיות B -

q_0	E	q_3	*	q_5	B				



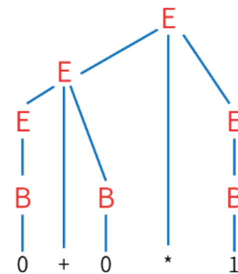
השלב הבא הוא לבדוק את ה-$GOTO(q_5|B)$. ואז אנחנו עוברים למצב 7 -

q_0	E	q_3	*	q_5	B	q_7			

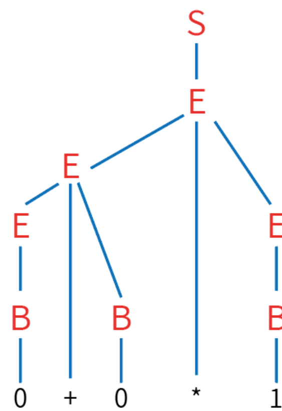
מצב 7, מצמצם לנו את שלושת תתי העצים להיות תחת אותו שורש, או במילים אחרות r1 –

q_0	E								

q_0	E	q_3							



עכשיו כל שנותר הוא להשוות את ($ | q_3). אנחנו מקבלים acc. כלומר סיימנו בהצלחה, ואנחנו יכולים לבנות את הקדקוד האחרון שיסגור לנו את העץ –



סיימנו.

בעיות עם דקדוק LR

עד עכשיו עבדנו עם LR(0), וגם בדוגמא שראינו הכל עבד חלק. אבל כבר דיברנו על כך שיכולים להיות כל מיני קונפליקטים, שבעבור אותו פריט, יש לנו שתי אפשרויות של פעולות. מאחר ואין לנו דרך לקבוע באיזה משתי

הפעולות אנחנו יכולים לבחור, אנחנו עלולים להתקע. יש לנו שני קונפליקטים עיקריים שעלולים לקרות, והזכרנו אותם קודם: Reduce/Reduce, כאשר על אותו פריט אנחנו יכולים לצמצם לשני כללים שונים. לדוגמא נראה את כללי הדקדוק הבאים –

E → A 1 | B 1
A → 1
B → 1

לא נבנה את כל הטבלה וכל התהליכים הקשורים אליה, רק נדבר באופן כללי – קל לראות, ש-1 הוא first(E), הוא אפילו מגיע משתי אפשרויות שונות – הן מ-A והן מ-B. כלומר הפריט של E|1 (ולא נעכב בדיוק איזה מצב זה) שייך גם לB→1, וגם לA→1, וכמובן שאחרי שנעבור אותו, יהיה לנו צמצום. הבעיה היא שהצמצום יכול להיות לשתי המשתנים האלה ואין שום חוק שיכול להכריע לנו על איזה מכללים אנחנו מדברים.

מה אנחנו עושים במקרה כזה? לא משתמשים ב- LRכי הוא לא מצליח לפתור את זה.

אפשרות נוספת לקונפליקט, היא מצב שמכונה Shift/Reduce. אפשר כמובן לנחש מה קורה שם, אך בכל אופן נראה דוגמת דקדוק –

E → 1 E | 1

E יכול לראות את 1 גם אם הוא יגזור ישר אליו (ולאחריו נוכל לצמצם), וגם אם הוא גוזר ל-1E ואז יהיה לנו רק פעולת שיפט. איך נוכל להכריע בין השניים? אם נבנה את הטבלה ניתקל בבעיה בצורתה הבאה –

	פעולות		goto
	1	$	E
q0	s1		2
q1	r2/s1	r2	3
q2		acc	
q3	r1	r1	

איך הגענו למצב הזה? כזכור, אנחנו קודם כל מכניסים את המפגשים שהופכים לשיפט, ולכן הוכנס שם s1, אך בהמשך אנחנו מזהים שיש לנו פעולת צמצום (כלומר, שהנקודה סיימה כלל גזירה ולא רואה לפניה שום דבר), ואנחנו מכניסים לכל השורה את הצמצום המתאים, מה שיוצר לנו התנגשות.

הבעיה הזאת נוצרה לנו מהסיבה שאין לנו Lookaheadקדימה – אם היה לנו, היינו יכולים להסתכל על האסימון הבא ולדעת בדיוק איזה פעולה התאימה לנו – כל עוד אנחנו לא באות האחרונה בקלט אנחנו יכולים לרוץ על 1E, וברגע שאנחנו מזהים את סוף הקלט נגזור רק ל-1 ונסגור את הסיפור.

SLR(1)

SLR(1) או Simple LR(1) נועד לטפל בדיוק בבעיה הזאת. אנחנו עושים את כל הבניה וכל הצעדים בדיוק כמו שעשינו עד עכשיו, רק שאנחנו משנים דבר אחד בהכנסת הערכים בטבלה –

בדקדוק LR(0)אנחנו אמרנו שאם אנחנו מזהים סיום קלט, אנחנו מכניסים צמצום לכל השורה. אבל כמו שראינו, זה מה שיוצר לנו את הבעיה. לכן אנחנו נצמצם את הכלל הזה ונאמר כך – אנחנו נכניס פעולת reduce אך ורק לאסימונים באותה שורה ששייכים ל-Follow שלה. מה תרמנו פה?

בעצם, פעולת הצמצום אמורה להגיע אך ורק אחרי שאנחנו כבר עוברים על כל מה שקשור לאותו משתנה. ולכן אם אנחנו באמת צריכים לצמצם, מה נראה בראש הקלט? את האסימון של הגזירה הבאה, כלומר Follow(A), ולכן רק במקרים אלה יש לנו מה להכניס פעולת reduce.

כמובן שגם זה לא מושלם ויש מקרים בהם עדיין אנחנו ניתקע עם קונפליקטים של shift/reduce. מה אנחנו נעשה כשניתקל בכאלה? לא ברור. יש פיתרון ואנחנו לא נראה אותו.

ניתוח תחבירי מונחה-טבלאות – LR מצגת דוגמאות מס' 4

הרצת הדוגמה שראינו קודם התעסקה יותר בדקדוק LR. העבודה הנוכחית שלנו בתרגול, ולמעשה גם בתרגיל הבית עצמו (ובמבחנים) יתמקד יותר ב-SLR(1). רק כתזכורת, ה-SLR בא לפשט את כללי מילוי הטבלאות של LR. הכלל היחיד שהשתנה היה שבמקום למלא שורה שלימה בפעולות צמצום (reduce), מה שהיה נהוג ב-LR, אך עשה בעייה של התנגשויות, הוחלט שאת פעולות הצמצום נפעיל רק על טרמינלים שהם ב-follow של כללי הגזירה אותם אנחנו מצמצמים.

הערה: גם פה, כמו בתרגול הקודם, לא אראה את כל השלבים. בניית הפריטים והטבלה הוא תהליך ארוך ומייגע, ואין צורך אמיתי לעבור ולראות את כולו, אלא נראה את ההתחלה של העבודה וכמה שלבים קדימה, ומשם נדלג.

הדקדוק עליו אנחנו עובדים, הוא זה המוכר לנו משני התרגולים הקודמים –

A → N **=** A | E
N → N **. id** | **id**
E → E **+** T | T
T → N | **(** A **)**

כאשר השלב הראשון אותו יבקשו מאיתנו הוא פשוט להציג דקדוק מורחב. נעיר רק שברגע שנתחיל לעבוד, אנחנו נפריד את הכללים השונים ונמספר אותם החל מ-1, והדקדוק המורחב ייתן לנו את כלל 0 – S→A, או במקרה שלנו A'→A. השם של המשתנה עצמו לא באמת חשוב.

המשמעות של כלל ה-0 הוא כזה – אנחנו עלולים במרוצת הגזירה ללכת ולחזור לכלל הראשון, ואנחנו רוצים שתהיה לנו איזה דרך לזהות שהצמצום שלנו לכלל A היה הצמצום האחרון. אי לכך, אנחנו נוסיף את משתנה הגזירה הטרום-התחלתי, שיוכל להביא לנו משוב שברגע שצמצמנו אליו, אנחנו לכאורה סיימנו את הגזירה והמעבר על הקלט.

אם כן, הדקדוק המורחב יוצג כך –

0. A’ → A
1. A → N **=** A
2. A → E
3. N → N **. Id**
4. N → **id**
5. E → E **+** T
6. E → T
7. T → N
8. T → **(** A **)**

עכשיו נתחיל לבנות את הפריטים והמצבים השונים. את כל זה אנחנו נתחיל מהמצב הראשוני, או אפילו לפני זה – A'→A.

S_0 = { [A’ → ·A], [A →·N = A], [A → ·E], [N → ·N . id], [N → ·id], [E → ·E + T], [E → ·T], [T → ·N], [T → · (A)] }

כזכור, הנקודה האדומה מבטאת לנו את הראש הקורא ומה הוא "רואה" לפניו בקריאה. מסיבה זאת, זה אכן הגיוני שהמצב הראשוני יכיל את כל כלל הגזירה השונים, מאחר וכולם בסוף אמורים להגיע ממנו.

יצירת המצב הבא, בדרך כלל תתייחס לסדר ולאופן שהוגדר לנו בתרגיל. בדרך כלל מדובר על חלוקה לקבוצות של משתנים-טרמינלים, כאשר הסדר הפנימי ביניהם הוא הסדר שמוגדר בהגדרת השפה G. כאן אנחנו עוברים קודם כל על משתנין ולאחר מכן על טרמינלים. לכן, המצב הבא שניצור, הוא עבור המקרה בו A’ כבר ראה את A ו"סיים לקרוא אותו. כלומר –

S_1 = { [A’ → **A**·] }

לכאורה, אנחנו צריכים לעשות פה closure, אבל אין באמת עניין.

במקביל, אנחנו מתחילים למלא את הטבלה – עבור כל מצב שנפתח, אנחנו נרשום כיצד הגענו אליו ביחס המשתנה-טרמינל. כלומר, במקרה שלנו, היינו במצב 0, וראינו A, לכן נעבור למצב 1. להלן הטבלה –

	id	=	+	.	(	)	$	A	E	T	N
0								1			
1											
…											
15											

אז, יש לנו את הטבלה המוכנה עם 15 שורות, במקרה, ואנחנו מכניסים את ערך המצב לתוך התא המתאים.

עכשיו אנחנו בודקים מה קורה במצב 0 עבור המשתנה הבא – N. אנחנו מחפשים את כל המופעים בהם הקורא היה לפניו, ומעבירים אותו לאחריו.

S_2 = { [A → **N** ·= A], [N → **N** ·. id], [T → **N**·] }

אנחנו עושים בדיקה האם יש לנו איזה closure שמוסיף לנו פריטים, רואים שאנחנו עומדים אל מול טרמינלים, ויכולים לעדכן את הטבלה.

	id	=	+	.	(	)	$	A	E	T	N
0								1			2
1											
…											
15											

אנחנו ממשיכים לשאר המשתנים, יש לנו את E, אך אין לנו את T, לפחות לא ישירות מ-S, ולכן אנחנו מתעלמים ממנו. עכשיו אנחנו עברנו בעצם על כל מה שאנחנו רואים ישירות מהמשתנה A. השלב הבא יהיה לעבור על שאר הפריטים לפי סדר היצירה שלהם (כאמור, בדרך כלל יגידו לנו מה הסדר הנדרש).

עד S_5 אין פה הרבה מה להראות. כשאנחנו מגיעים לפריט האחרון של S_0, יש לנו את T → ·**(** A) ואם נעביר את הקורא לאפשרות הבאה, כלומר T → **(**· A), אז יתחיל הכיף האמיתי. אם אנחנו רואים עכשיו גם את A, אז אנחנו מביאים איתנו את כל מה שאנחנו יכולים לראות דרך A, ואז בעצם אנחנו יכולים לגרור איתנו את כל הפריטים שיש לנו במצב 0 –

S_6 = { [T → **(** ·A)], [A →·N = A], [A → ·E], [N → ·N . id], [N → ·id], [E → ·E + T], [E → ·T], [T → ·N], [T → · (A)] }

עכשיו אנחנו יכולים לומר שחישבנו את כל מה שרלוונטי למצב 0, ולכן נמלא את הטבלה (מזכיר רק שדילגתי על שלבים לא מעניינים)-

	id	=	+	.	(	)	$	A	E	T	N
0	4				6			1	3	5	2
1											

...											
15											

נקפוץ לסוף יצירת המצבים. הגענו ל-15 המצבים הבאים –

S_0 = { [A' → ·A], [A → ·N = A], [A → ·E], [N → ·N . id], [N → ·id], [E → ·E + T], [E → ·T], [T → ·N], [T → ·(A)] }
S_1 = { [A' → A·] }
S_2 = { [A → N ·= A], [N → N ·. id], [T → N·] }
S_3 = { [A → E·], [E → E ·+ T] }
S_4 = { [N → id·] }
S_5 = { [E → T·] }
S_6 = { [T → (·A)], [A → ·N = A], [A → ·E], [N → ·N . id], [N → ·id], [E → ·E + T], [E → ·T], [T → ·N], [T → ·(A)] }
S_7 = { [A → N = ·A], [A → ·N = A], [A → ·E], [N → ·N . id], [N → ·id], [E → ·E + T], [E → ·T], [T → ·N], [T → ·(A)] }
S_8 = { [N → N . ·id] }
S_9 = { [E → E + ·T], [T → ·N], [T → ·(A)], [N → ·N . id], [N → ·id] }
S_{10} = { [T → (A ·)] }
S_{11} = { [A → N = A·] }
S_{12} = { [N → N . id·] }
S_{13} = { [E → E + T·] }
S_{14} = { [T → N·], [N → N ·. id] }
S_{15} = { [T → (A)·] }

שימו לב שבמקרה והגעתי למצב שבו הסגור הביא לנו מצב שכבר ראינו, למשל: {s_6|(} אז הבדיקה של הסגור הביאה לנו בדיוק את מצב 6 בחזרה, אז אין סיבה לפתוח לנו מצב חדש, אלא רק לעדכן בטבלה, שבמקרה ומצב 6 רואה (שיחזור בחזרה למצב 6.

נראה כעת את הטבלה הסופית –

	id	=	+	.	(	)	$	A	E	T	N
0	2	5	3	1			6				4
1											
2								8		7	
3									9		
4											
5											
6	2	5	3	10			6				4
7	2	5	3	11			6				4
8											12
9	14	13					6				4
10						15					
11											
12											

13											
14								8			
15											

על מנת לסיים את מילוי הטבלה, עלינו לחשב את ערכי ה-first וה-follow. אם היינו עובדים על LR רגיל, לא היה בזה צורך, אבל בגלל שמדובר ב-SLR אנחנו רוצים לדעת את ה-follow של כל משתנה גזירה –

X	A’	A	E	T	N
First(X)	Id, (	Id, (	Id, (	Id, (	**id**
Follow(X)	$	$,)	$,), +	$,), +	$,), +, =, .

עכשיו אפשר לסיים את הטבלה – כל המספרים שנמצאים בחלק של הטרמינלים יהפכו לפעולת shift, ובכל המצבים בהם סיימנו לקרוא כלל גזירה, נוסיף ל-follow שלהם פעולת reduce. נבדוק על אילו מצבים מדובר –

$S_1 = \{ [A' \rightarrow A\cdot] \} \Rightarrow \{\$\} = acc$
$S_2 = \{ \dots [T \rightarrow N\cdot] \} \Rightarrow \{\$,), +\} = r7$
$S_3 = \{ [A \rightarrow E\cdot] \dots \} \Rightarrow \{\$,)\} = r2$
$S_4 = \{ [N \rightarrow id\cdot] \} \Rightarrow \{\$,), +, =, .\} = r4$
$S_5 = \{ [E \rightarrow T\cdot] \} \Rightarrow \{\$,), +\} = r6$
$S_{11} = \{ [A \rightarrow N = A\cdot] \} \Rightarrow \{\$,)\} = r1$
$S_{12} = \{ [N \rightarrow N\ .\ id\cdot] \} \Rightarrow \{\$,), +, =, .\} = r3$
$S_{13} = \{ [E \rightarrow E + T\cdot] \} \Rightarrow \{\$,), +\} = r5$
$S_{14} = \{ [T \rightarrow N\cdot] \dots \} \Rightarrow \{\$,), +\} = r7$
$S_{15} = \{ [T \rightarrow (A)\cdot] \} \Rightarrow \{\$,), +\} = r8$

ונמלא את הטבלה –

	Action							GOTO			
	id	**=**	**+**	**.**	**(**	**)**	**$**	**A**	**E**	**T**	**N**
0			s6								4
1	acc						acc				
2	r7	r7		s8	r7	s7	r7	8		7	
3	r2	r2			s9		r2		9		
4	r4	r4		r4	r4	r4	r4				
5	r6	r6			r6		r6				
6			s6								4
7			6s								4
8											12
9			s6								4
10		s15									
11	r1	r1					r1				
12	r3	r3		r3	r3	r3	r3				
13	r5	r5			R5		r5				
14	r7	r7		s8	r7		r7	8			
15	r8	r8			r8		r8				

עכשיו, כל שנותר לנו הוא רק לרוץ על הקלט תוך כדי שנבנה את העץ.

לצורך כל, ניצור טבלה חדשה של קלט ומחסנית פעולות. לטובת הנוות שלנו גם נכניס עמודה קטנה שם נכתוב בכל פעם איזו פעולה עשינו -

פעולה	מחסנית	שארית הקלט
	0	**a = b + (c . d = e) + f $**

אנחנו מאתחלים את המחסנית עם 0, בשביל לבטא את המצב ההתחלתי בו אנחנו נמצאים, ומתחילים לרוץ על הקלט. בתור התחלה אנחנו בודקים על q_0|id (כזכור כל שם משתנה פה מתבטא כ-id), ומסתכלים בטבלה מה הפעולה המתאימה -

פעולה	מחסנית	שארית הקלט
s4	0 id 4	**= b + (c . d = e) + f $**

הכנסנו את ה-id ודחפנו אחריו את מצב 4. מבחינת העץ, אנחנו עכשיו בונים עלה בודד –

id
(a)

עכשיו אנחנו רואים שיש לנו את מצב 4 בראש המחסנית, ואת = בקלט. הפעולה המתאימה היא r4, כלומר צמצום על י כלל 4, שהוא id→N. אנחנו מעבירים את אנחנו בודקים כמה איברים יש בצד שמאל של כלל הגזירה (1), ומוציאים מהמחסנית פי שניים איברים (2) ובמקומם מכניסים את N –

פעולה	מחסנית	שארית הקלט
r4	0 N	**= b + (c . d = e) + f $**

בשביל לדעת לאן אנחנו ממשיכים מפה, אנחנו בודקים את המשתנה שצמצמנו אליו, ואת המצב הצמוד לו. ואז בודקים בטבלת ה-GOTO עבור המצב הנתון ואותו משתנה לאיזה מצב זה מעביר אותנו –

פעולה	מחסנית	שארית הקלט
s4	0 N 2	**= b + (c . d = e) + f $**

אנחנו ממשיכים עכשיו, אחרי שעברנו למצב , וממשיכים לרוץ על הקלט. הדבר הבא שיש לנו זה s2|= ואנחנו בעבור זה עושים s7 –

פעולה	מחסנית	שארית הקלט
s7	0 N 2 = 7	**b + (c . d = e) + f $**

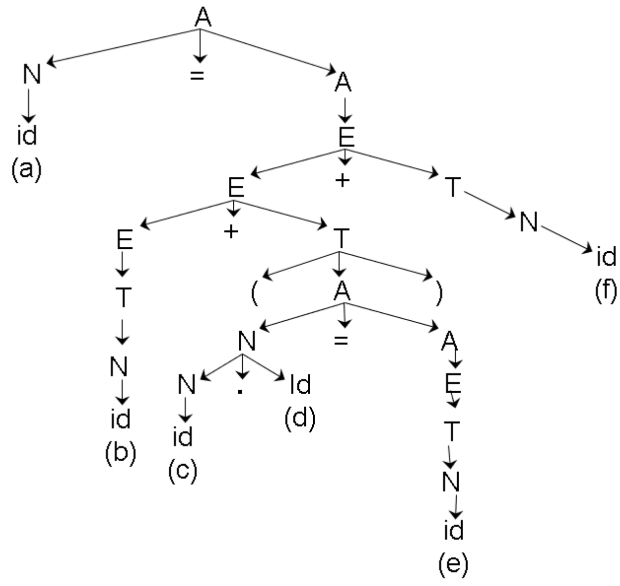
מבחינת העץ, אנחנו עכשיו דואגים להוסיף עץ חדש וקטנטן ביער שאנחנו נוטעים –

N =
↓
id
(a)

שוב, אני לא הולך להריץ פה את כל העץ, כי זה סתם טכני, ואין פה חידושים אמיתיים. נדלג יש לשלב האחרון -

פעולה	מחסנית	שארית הקלט
acc	0 A 1	**$**

שימו לב, שכאן אין לנו צורך לרוקן את המחסנית, אלא ברגע שהגענו למצב 1 ובבדיקה מול ה-$ קיבלנו acc, אנחנו ישר סוגרים את הבאסטה ומניחים את העץ כפי שהוא -



תם ונשלם הקונדס.

ניתוח סמנטי

הגענו לשלב משמעותי – הניתוח הסמנטי.

עד עכשיו התעסקנו עם המילים עצמם והמשפטים השונים, אבל עכשיו אנחנו מתחילים לדבר על הסך-הכל – האם יש היגיון בכל מה שכתבנו והאם זה חוקי. אנחנו מקבלים את העץ התחבירי שעשינו בחלק הקודם, ומוסיפים לכל אורכו וגובהו "עיטורים" – כל מיני הערות חשובות לגבי המשמעויות השונות, ובסוף העץ הזה על שלל עיטוריו יוציא לנו את קוד הביניים המופלא.

מה ההבדל בפועל בין החלק הסמנטי לסינטקטי?

הסינטקס – איך התוכנית אמורה להיראות – מה מגיע לפני מה, אילו מילים יכולות להופיע ביחד וכו'.

הסמנטי – מה כל גיבוב המילים הזה אמור לעשות.

כשדיברנו על דקדוק חסר הקשר במשמעות הסינטקטית שלו, התייחסנו בגדול לקשרי אב-בן. אם היה לנו כלל גזירה מהסוג E→ int x, אנחנו יודעים ש-E הוא האבא והחלק הימני הם הבנים שלו. אבל מבחינת הקשר הזה, כאן אנחנו עוצרים. אנחנו לא מתעסקים בהשמה של אותו x או מה אנחנו עושים איתו בהמשך. אם הקוד שלנו כתוב בצורה הבאה:

```
Int x;
{ Some Code}
x = 3;
```

זה יהיה תקין ברמה הסינטקטית ולא משנה מה יש לנו באמצע. אבל בדיוק כמו שבמנתח המילולי אנחנו לא התייחסנו למשפט אלא רק למילים, גם פה אנחנו התייחסנו למשפטים ולא לתכנית. לאן אנחנו חותרים? אם נגיד בין שתי השורות האלה היה בהמשך פקודה ;string x, אז עכשיו המשתנה הוא בכלל לא מספר, והניסיון להציב בו אחג כזה, הוא ניסיון חסר ערך. יותר מזה, אין שום תצורה של דקדוק חסר הקשר בו נוכל לדאוג לכך שלא משנה מה יהיה, ההצבה תהיה נכונה.

כלומר, יש אפשרות כזאת, אבל זה אומר לכתוב לכל התוכנית כללי גזירה פרטיים. לעשות את זה לתכנית בודדת זה אולי קשה, אבל בטוח זה לא מתודה שאנחנו רוצים לעבוד איתה באופן קבוע.

כדוגמא, אם התייחסנו קודם לכלל הגזירה E→E+T אז החלק השמאלי ייצור לנו עץ עם הימני. לאן יגזור ה-E החדש? מה קורה עם ה-T? לאן הם מתקדמים? לאן שתוביל אותם הרוח, אבל מבחינתנו זה פחות משנה.

ואז הגיע הניתוח הסמנטי. כאן אנחנו נתייחס לבנים/נכדים/אחיינים עד לסוף כל הדורות, וננסה להבין מהם את המשמעות (בהנחה שיש כזו) של התכנית.

נתייחס רגע לדוגמה שיש במצגת – עבור משתנה כלשהו S יש לנו שני חלקים חשובים – הצהרה והשמה. אנחנו רוצים את האפשרות להעביר מידע ממקום אחד למשנהו – אנחנו צריכים גם את מידע ההצהרה – סוג הטיפוס וגם את ההשמה – הערך. אחד מהם לא רלוונטי ללא האחר, ועלינו למצוא את הדרך ולהעביר את המידע בין הרמות השונות.

מושגים והגדרות

תרגום מונחה דקדוק

דיברנו כבר בחלקים הקודמים על כך שתוך כדי הגזירות, אנחנו מוסיפים "פעולות סמנטיות" לעצי הגזירה. הגיע הזמן לטפל בפעולות האלה. רק תזכורת – כשאנחנו בונים עץ אנחנו למעשה מייצרים קוד, שמלבד בניית העץ יכול לגשת לטבלת הסמלים, להדפיס לנו פלט מסוים ועוד.

מה המשימות של הפעולות האלה?

- לבדוק תכונות תחביריות שלא ניתן לבדוק על ידי דקדוק תחבירי רגיל.
- בנית ייצוג של התוכנית בשבת ביניים שלא תלויה בשפת המקור – הזכרנו כבר שאם אנחנו מסוגלים להגיע מכל סוג של קוד לאיזה קוד-ביניים שמכונות אחרות יכולות לקרוא אותו אנחנו חוסכים לנו את עלויות הפיתוח השונות עבור כל מכונה ומכונה. נעשה את זה פעם אחת כמו שצריך ולא נזדקק לעשות זאת שוב.

הדרך בה נעשה את זה – נשתמש ב**דקדוקי שדות ערך** – הרחבה של הדקדוק ח"ה, בו אנחנו לא סתם גוזרים ממקום למקום, אלא מוסיפים שדות של ערך/תכונות לטרמינלים והמשתנים השונים. בנוסף, נצמיד פעולות סמנטיות לגזירה הדקדוקית. את שני הדברים האלה, בנפרד או ביחד, נעשה לכל צומת בעץ- בין אם מדובר בטרמינל ובין אם מדובר במשתנה.

תכונות – התכונות יביאו לנו את המשמעות של כל צומת. מה הכוונה? לכל ערך כלשהו יש אטריביוטים שישמשו אותנו בפעולות השונות, כמו למשל סוג, גודל, ערך וכדו'. אנחנו נוסיף את השדות הרצויים למשתנים שנפגוש ונוכל למלא אותם בהמשך.

פעולות סמנטיות – את הפעולות האלה אנחנו נוסיף לכל כלל גזירה, והמשמעות של זה תהיה הכנה לקראת כתיבת קוד הביניים. למשל –

$E \rightarrow E_1+T$

יהפוך עם הפעולות הסמנטיות לדבר הבא –

E.code = E_1.code || ‘+’ || T.code

כלומר, יש לנו את המשתנה E, והקוד שלו, כלומר הערך שלו יקבע לאור הפעולה שהיא שרשור של הקוד (ערך) של E_1 עם פעולת חיבור ועם T. את הפעולה הזאת אנחנו כמובן נוסיף בשורש העץ – לאחר שיסתיים החישוב מה שיעלה לשורש של אותו תת עץ יהיה תוצאת המשוואה המבוקשת. הוספת החלקים האלו נקראת "עיטור העץ". כך שפלט של הניתוח הסמנטי יוסיף לעץ תכונות ופעולות לכל קדקוד וכן את הקוד שאומר איך להשתמש בכל תכונה.

כדאי לשים לב – כבר במצגת אפשר לראות שאנחנו ממספרים את כל המשתנים שאנחנו נתקלים בהם. זה נעשה מאחר ואנחנו לא יודעים אם $E \rightarrow E+T$ מדבר על אותו E. ולכן אנחנו נוסיף מספר קטן לכל משתנה, בפועל אנחנו לא נמספר את הפעם הראשונה שאנחנו נתקלים באותו משתנה, לכן אפשר לשים לב שבכלל הגזירה שהבאנו קודם מה שמוספר היה רק ה- Eבחלק הימני של כלל.

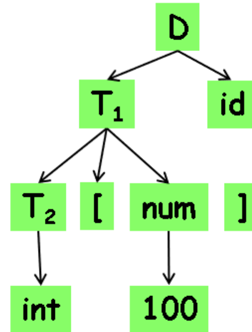
כמובן, שיש לנו גם חלקים שאינם "מוכרחים" מעצם הדקדוק, אבל תמיד נחמד להוסיף אותם – הדפסות שונות, עדכון של טבלת הסמלים במידה שמגיע אלינו ועוד כל מיני side-effects שונים ומשונים. את כל אלה אנחנו נוסיף עם הפעולות הסמנטיות השונות.

ברגע שאנחנו נכתוב פעולה כמו מה שראינו קודם – חישוב ערך של E, ברור לנו שאנחנו יוצרים גם איזה סדר פעולות. אם אנחנו רוצים את הערך של E, אנחנו נצטרך לחשב קודם את הערך של E_1 ואת T, וכל אחד מהם עלול להסחף למספר פעולות שונות. בעצם הדבר הזה יוצר לנו איזה סדר מוכרח של הפעולות השונות. כל חישוב של פעולה יהיה כמובן לפני השמת התוצאה שלו וכו', מה שייצור לנו כאן איזה "גרף תלויות" – אם חישוב של פעולה A תלוי בתוצאה של פעולה B כלשהיא, אז אנחנו אומרים ש-A תלוי ב-B.

ניקח לדוגמה את כללי הגזירה הבאים:

D -> T id
T -> T [num] | int | real | char

אם נרצה עכשיו להגדיר מערך של int בגודל 100, נצטרך ליצור את העץ הבא-



אין צורך להראות את כל הדרך שעברנו עד העץ, כי זה פחות חשוב לנו כרגע לדיון. אבל מה שכן חשוב זה שאלה אחרת- אנחנו רוצים להגדיר את גודל המשתנה הזה. מה החישוב שנעשה בשביל להגיע לגודל המתאים?

T_2.size = int.size
num.value = 100
T_1.size = T_2.size * num.value
id.size = T_1.size

כלומר – קודם כל אנחנו נגדיר את גודל הטיפוס – T_2 הוגדר (נגזר)לנו בתור int. ולכן הגודל שלו יוגדר להיות הגודל של int. במקביל, אנחנו רואים שיש לנו מספר שהוגדר בתור ערך קבוע כלשהו. אנחנו אוספים את שני הנתונים האלה, ועל ידיהם אנחנו יכולים עכשיו להגדיר את T_1 - אם הוא מערך של 100 int-ים אז הגודל שלו פשוט יהיה הגודל של T_2 (int.size) מוכפל בערך של num. עכשיו הגדרנו את גודל המערך הרצוי, אבל עדיין לא ביססנו את זה ביחס ל-id. הפעולה האחרונה תציב את מה שהתקבל גם בתוך שדה הגודל של id.

מתוך כל הדיון הזה, ניתן לומר כי יש תלות של id.size שעוברת את כל הדרך המפותלת עד ל-int.size ועם כל הפיצולים הנדרשים.

תלויות

לאור כל האמור, אנחנו יכולים להגדיר את ה"תלויות" באופן הבא – המשוואה-המגדירה יוצרת תלות של a ב-$b_1,\dots,b_m$. ע"מ לחשב את הערך a יש לחשב תחילה את ערכי $b_1,\dots,b_m$. נסמן זאת – $a \leftarrow b_i$.

כלומר, כל פעולה b שמוכרחת לקרות לפני a היא פעולה שתולה אותה. אם $b \leftarrow a$ אז b קורה לפני a.

לאחר שאנחנו נאסוף את כל התלויות השונות של כל העץ, אנחנו נוכל להגדיר לנו איזה סדר פעולות כלשהו, שיתחיל באחד העלים, ויסתיים בשורש העץ. השאיפה המוחלטת שלנו הוא להגיע למצב שבו עבור דקדוק מסוים G אנחנו יכולים לבנות גמ"ל (גרף מכוון ללא מעגלים) שבמעבר עליו אנחנו נוכל לקבוע את סדר הפעולות של התכנית. כמובן שלאחר שיש לנו את סדר הפעולות, ואת כל הפעולות הסמנטיות השונות, המעבר לקוד ביניים הוא מאוד פשוט.

הבעיה תיווצר לנו אם אנחנו נראה תלות מעגלית. למשל:

E.s = T.i
T.i = E.s + 1

במקרה זה, אם נרצה להגדיר את E, נצטרך להגדיר קודם את T. אבל בשביל להגדיר את T אנחנו צריכים את E+1. אך מה הוא E?? וכן על זה הדרך. מצב כזה ברור לנו שהוא בעייתי. כדאי לשים לב, שכשניצור תלויות שונות לא תמיד נגיע למשהו חד משמעי – יכול להיות שיהיו פעולות b_1, b_2, b_3 שיתלו את פעולה a, אך ביניהם אין איזה הכרח לעבוד על אחד לפני שעובדים על אחר, ופשוט עלינו לדאוג שהם ייעשו בסדר מסוים.

בהשאלה, אנחנו מתייחסים לכל הפעולות הסמנטיות האלו כמערכת משוואות ששדות הערך זה הנעלמים השונים אותם אנחנו צריכים למצוא, אם פתרנו את כל המשוואות, הכל בסדר. אם לא, פחות.

תכונות נוצרות ונורשות

כשאנחנו מדברים על התכונות השונות, יש להפריד בין שני סוגים – נוצרות ונורשות.

<u>תכונות נוצרות –</u> תכונות שונות של הצמתים המגיעים מאותה צומת בעצמה או מאחד הבנים. הבסיס של התכונות האלה הוא בין השאר כל מיני תכונות שנוצרות עוד על ידי המנתח הלקסיקלי, וכך, בעת מעבר מהבנים לכיוון האב, המידע הזה וכן מידע אחר שמגיע מצטרפים ועולים כלפי שורש העץ.

<u>תכונות נורשות</u> – תכונות העוברות מהאב או מאחד האחים.

איך נדע להפריד בין התכונות הנורשות לנוצרות בקוד שאנחנו כותבים? בגדול, אנחנו רוכשים את התכונות השונות על ידי השמות כלשהן – L.size = T.size וכל מיני דברים דומים שכאלה. אם מדובר על השמה של ערך סופי בצד הימני, כמו למשל: T.type := integer, ברור לנו שזה תכונה נוצרת. כל השאר, עלינו לבדוק לאן ההפניה הזאת מובילה אותנו מבחינת היחס ברמות העץ – אם מדובר על רמה נמוכה יותר, כלומר אחד הבנים הביא לנו את המידע, אז מדובר בתכונה נוצרת. ואם מדובר במידע שעבר מאותה רמה או מהרמה מעל, מדובר בתכונה נורשת.

בניית גרף התלויות

לאחר שהחלטנו מה מחושב על ידי מה, אנחנו צריכים להתחיל לסדר את הדברים לקראת יצירת גרף התלויות. לכל תכונה שבעבורה נצטרך לחשב משהו אחר לפני, נעביר את הפעולות האלה לצורה של $b:=(c_1,c_2..)$. כך שאת כל הביטויים הסמנטיים שכתבנו, נחליף במשוואות, ואת הפעולות שהגדרנו שהן רק לטובת ה-side-effects אנחנו מחליפים בהשמה לתכונות דמה. כלומר, במקום פעולות מסוג

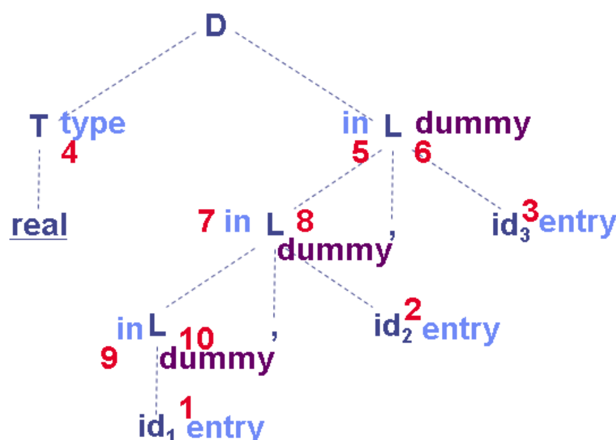
S → E { print (E.val) }

נכניס את הפעולות בצורה כזו –

S → E { S.dummy := print (E.val) }

עכשיו נותר לנו להתחיל לעבוד על הגרף בעצמו – לכל צומת אנחנו נוסיף עוד "תתי-צמתים". כל תכונה חשובה עליה אנחנו עוברים תוגדר כצומת חדש, ועכשיו אנחנו נוסיף את התלויות לפי כל המשתנים והצמתים החדשים. אם היה לנו תלות של משוואה מהסוג הזה - $b:=(c_1,c_2..)$ שכבר הצגנו כמה פעמים, אז לכל c תהיה לנו קשת שתוביל בחזרה ל-b – כלומר, (c_i, b).

אם כך בעבור העץ שכבר ראינו קודם, אנחנ נוכל לראות את הוספת התכונות לצמתים הרלוונטיים באופן הבא –



production	semantic rules
D → T L	L.in := T.type
T → <u>int</u>	T.type := integer
T → <u>real</u>	T.type := real
L → L_1 , <u>id</u>	L_1.in := L.in addtype (<u>id</u>.entry, L.<u>in</u>)
L → <u>id</u>	addtype (<u>id</u>.entry, L.<u>in</u>)

אנחנו יכולי לראות שעל כל צומת יש לנו גם את התכונות השונות (נוצרות ונורשות כאחד), וגם מיספור. לפי מה קבענו את המספור? כמובן, על פי התלויות שראינו עד עכשיו. העץ מתייחס לגזירה של קלט $real\ id_1, id_2, id_3$. כאשר אם נעבור על המספור, נוכל לראות שקודם כל אנחנו מתחילים עם הפעולות הנוצרות – addtype שכמובן אינו קשור לשום דבר שנעשה לפני – פשוט מכניסים את ה-id לטבלת הסמלים. לאחר מכן אנחנו מגדירים את הטיפוס real של T, ואותו מורישים לאח, שאחראי להעביר את זה הלאה. כעת כל בן יוכל להכניס את המשתנים לטבלת הסמלים, תוך כדי שעבר לנו כבר הטיפוס של המשתנים מהדוד – T.

יש לציין – פעולות ה-addtype הן פעולות דמה. אין להם שום השפעה על הקלט או תכונות כאלה ואחרות בעץ, אלא רק מבצעות פעולה מנהלתית ומסתיימות.

עכשיו, אם ניקח רק את הקוד הרלוונטי על פי סדר הפעולות שהגדרנו, נוכל לקבל את קטע הקוד הבא –

$$a_4 := real;$$
$$a_5 := a_4;$$
$$a_6 := addtype\ (id_3.entry,\ a_5);$$
$$a_7 := a_5;$$
$$a_8 := addtype\ (id_2.entry,\ a_7);$$
$$a_9 := a_7;$$
$$a_{10} := addtype\ (id_1.entry,\ a_9);$$

בשביל להבין מה קורה פה, כדאי לעקוב אחרי המספור המקורי שעשינו. אמנם אמרנו שאנחנו רוצים לטפל קודם במה שהוגר כ1-3, אך בשביל להכניס אותם לטבלה נצטרי גם לדעת את הטיפוס שלהם. שאותו אנחנו משיגים מהפעולה הרביעית, הלא היא a_4 כאן בקוד. הערך של זה עובר ממקום למקום, וזה די פשוט לעקוב אחריו ולראות איך כל הכנסה של טיפוס לטבלה נעשה בעזרת המידע המועבר.

אז, אפשר לעשות את כל הסיבוב שעשינו וליצור עץ גזירה ולהתחיל מיון טופולוגי, אבל לפעמים זה מאוד קשה – כבר ראינו את הבעיה של מעגלים בגרף. האם אנחנו יכולים לנסות ולדאוג שלא יהיה כאלה? כן. כמה זה יעלה לנו? הרבה. המעבר על כל האפשרויות השונות בגרף הוא עלות גבוהה ברמה מעריכית ביחס לקלט, וזה משהו שאנחנו לא רוצים לנסות להסתבך איתו. ולכן ישנה שיטה נוספת של הגדרת סדר וגזירה, וזה פשוט לדאוג שהדקדוק שלנו יהיה פשוט מבחינה סמנטית. למה הכוונה? אנחנו מגדירים סדרה של חוקים שכללי הגזירה צריכים לעמוד בהם, ואם נעמוד בהם, אנחנו יכולים לחסוך לעצמנו מצב של מעגלים בגזירה או בעיות דומות. כמובן שלא כל דבר ניתן לפתור עם המחלקות האלה, אבל במקרים מסויימים זה מאוד מקל לנו על העבודה, ויהיה יותר בטו.

דקדוקי S

המחלקה הזאת הינה מחלקה שעובדת **ללא ירושות**.

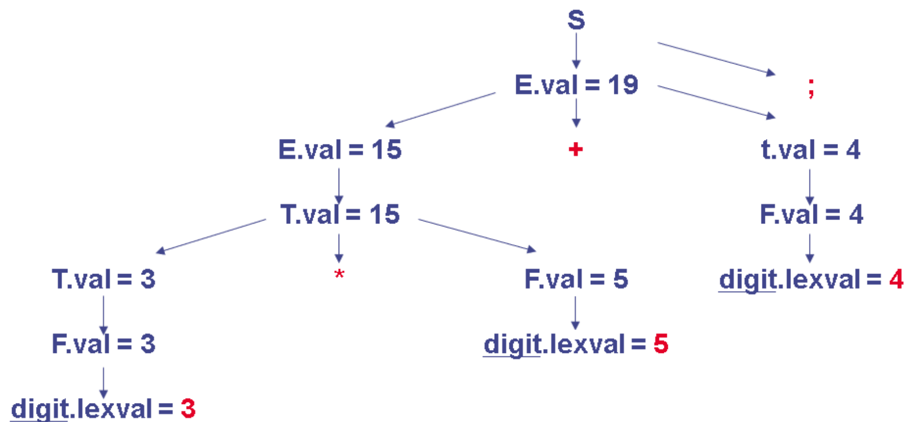
במה זה מסייע לנו? כל המידע עובר אך ורק מלמטה למעלה, ומתבסס על דקדוקי LR. מה שזה נותן לנו, הוא שאם אננו ממילא בונים את העץ מלמטה למעה, אנחנו יכולים על הדרך לעשות כבר את הניתוח הסמנטי ולהרוויח שניים במכה אחת. הרי כשאנחנו עושים את הצמצום של הבנים, אנחנו כבר יודעים את כל המידע הרלוונטי לגביהם, ואז כל המידע הזה עובר אל האב, שבתורו מעלה את זה הלאה, ומה שיוצא בסוף הוא שבזמן ההגעה לשורש אנחנו מקבלים כבר ישר את כל הקוד כתוב.

נוכל לראות דוגמא (די נאיבית) לדקדוק כזה, עם חוקים של כפל וחילוק. נראה את כללי הגזירה, ובעבורם את הכללים הסמנטיים המתאימים –

Production	Semantic Rules
S → E ;	**print (E.val)**
E → E_1 + T	**E.val := E_1.val + T.val**
E → T	**E.val := T.val**
T → T_1 * F	**T.val := T_1.val * F.val**
T → F	**T.val := F.val**
F → (E)	**F.val := E_1.val**
F → digit	**F.val := digit.lexval**

אז – יש לנו אפשרות של גזירה לפעולת חיבור בין שני מספרים, ובאמצי אנחנו מוסיפים אפשרות גם לכפל ולסוגריים. אם אנחנו מסתכלים על הכללים הסמנטיים מלמטה למעלה, אנחנו יכולים לראות בצורה די פשוטה איך כל המידע עובר רק בכיוון אחד – כל הכללים הם מסוג אב=בן.ערך וכך כל המידע מפעפע.

אם ניקח למשל את הקלט – 3*5+4 נוכל לקבל את העץ הבא –



התוצאה הסופית של S למעשה מתקבלת מחיבור שני הבנים שלו. הכפל בבן השמאלי נעשה גם הוא מקבלת המיד מהבנים והחזרת התשובה בצמצום כלפי מעלה.

לסיכום – S היא מחלקת דקדוק המשוייכת לLR, בו אנחנו עוברים מלמה למעלה, ותוך כדי בניית העץ אנחנו דואגים גם לניתוח סמנטי על הדרך שיביא לנו ברגע שנגיע לשורש העץ את התוצאה הסופית של הקלט.

דקדוקי L

המחלקה הזאת קצת פחות קפדנית מהקודמת – במחלקה זו תכונות יכולות לעבור מלמטה ולמעלה, ובנוסף אנחנו מתירים ירושות בין הצמתים השונים. אבל אנחנו מגבילים את כל התכונות הנורשות שיהיו רק כאלה שהתקבלו מהאב או מהאח משמאל. באופן כזה, המעבר על העץ יהיה באופן ודאי רק בכיוון אחד, מה שייתן לנו מצב שהוא בוודאות בלי מעגלים – אם אנחנו דואגים שתהיה לנו סריקה רק בכיוון אחד, אנחנו יכולים לדעת בוודאות שאין סיכוי שנגיע למעגל (כי מעגל בהגדרתו חוזר אחורה) כל תכונה שתגיע לקדקוד מסום בעזרת ירושה יכולה או לרדת מטה להמשך הבנים או לעבור ימינה לאד האחים, אך לעולם לא יכולה לחזור אחורה.

כמובן שהמחלקה S מוכלת בתוך L, אם זה רק מעביר מידע למעלה, הוא בוודאות עומד בחוקים של L.

יש דוגמה במצגות על דקדוק שמגדיר לנו גודל של טקסט עם גובה ועומק. הפרטים של הדוגמא לא *ממש* חשובים, ולדעתי גם אף אחד לא באמת הבין אותם אי פעם. מה שכן חשוב לזכור הוא זה – אנחנו אוספים את המידע בכיוון אחד, קצת דומה למשחק "סנייק". עוברים, לוקחים מידע, גדלים קצת וממשיכים הלאה.

איך אנחנו מחשבים את כל הפרטים העוברים ממקום למקום – על ידי חיפוש עומק (DFS). אנחנו נכנסים לבן השמאלי ביותר, מוציאים ממנו את כל המידע שאנחנו רוצים, וממשיכים האלה לעומק הבן הימני יותר. באופן הזה אנחנו יכולים לדעת בוודאות שמידע עובר רק משמאל לימין (לאחר שאנחנו מסיימים את הבן השמאלי אנחנו עוברים לבן הימני), או מלמטה למעלה – עם המידע שמועבר בסיום הריצה על הבנים ובחזרה לאבא.

קוד ביניים

את חשיבות קוד הביניים הסברנו כבר בעבר, אך נזכיר זאת שוב. במצב הראשון של בניית קומפיילר לשפה מסויימת, המפתח לוקח את השפה העילית ומגדיר עליה את כל החוקים הלקסיקליים, התחביריים והסמנטיים השונים. לאחר שהוא הגדיר את כל אלו, עליו ליצור דרך לבצע את הדילוג להבא, ולהמשיך להתממשק עם כל שפות המכונה הקיימות. אפשרות אחת לעשות דבר כזה, הוא פשוט להתחיל לכתוב את המימוש עבור כל שפת מכונה שקיימת אי שם. אם מישהו יוציא מכונה חדשה עם שפה חדשה, נתאים את השפה שלנו אליו ונמשיך, כמובן, שעלויות פיתוח כאלו יהיו מאוד גבוהות, בעיקר בהתחשב בזה שבעבור כל שינוי בשפה (שלא לדבר על כתיבת שפה נוספת) נצטרך לעבור את כל התהליך הזה מחדש. על מנת שמצב כזה לא יקרה, הוחלט שכל השפות העיליות יעברו לשפת ביניים בסיסית ביותר, שאליה ניתן להגיע מכל שפה שהיא, וממנה ניתן לצאת לכל שפת מכונה. כך שכל תהליך הפיתוח של שפה עילית חדשה אמור להגיע רק עד שלב קוד הביניים, וכל שפת מכונה אמורה להתמשק אליו.

ברגע שיש לנו קוד שהוא מוחלט וברור לכל המכונות והשפות, גם יותר קל לנו ליצר אופטימיזציות – כשכולם עובדים על אותו שדה, ניתן להרחיב את האלגוריתמים השונים ולעבוד בצורה שהיא רחבה ולא מופנית כלפי שפה אחת בלבד.

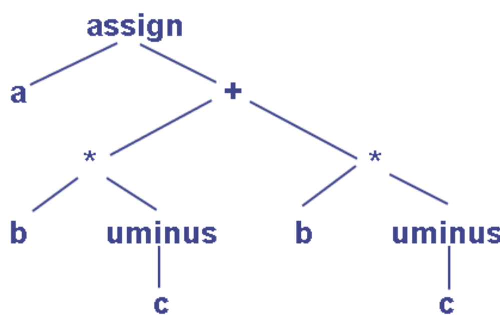
את קוד הביניים עצמו, אנחנו מחלצים מעיטור העץ עליו עבדנו עד עכשיו, וברגע שנעבור על העץ מהעלים כלפי מעלה, נוכל לקבל את הקוד השלם של התוכנית בשורש העץ. את כל המידע שאנחנו מוסיפים אנחנו אוספים והופכים בצורה פשוטה לקוד הביניים.

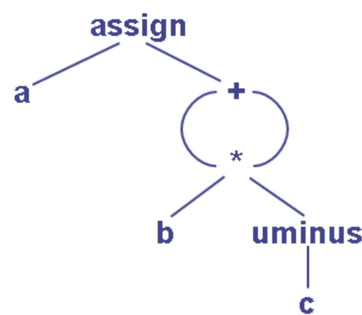
ייצוג ביניים

יש שלוש שיטות עיקריות לייצוגים של קוד הביניים –

- **עץ תחביר** – בין אם מופשט או רגיל, יכול להביא לנו ייצוג של קוד הביניים – שימו לב שאנחנו מדברים פה על ייצוג בלבד ולא על קוד הביניים עצמו. אנחנו יכולים לדבר הן על ייצוג בצורה של עץ תחביר, והן על ייצוג של גמ"ל (שאם נעבור עליו בצורה נכונה, נוכל פשו לקרוא את הסדר הנכון של הקוד). ניתן לראות את שני ייצוג הקוד האלה עבור הקלט הבא –

a := b * –c + b * –c





משמאל מופיע לנו עץ התחביר הפשוט - בשורש העץ יש לנו השמה, וכל חלק של המשוואה מחולק לצד אחר של הגרף. הגמ"ל לעומת זאת קצת שונה, יש כאן זיהוי שבעצם אנחנו עוברים על אותו חלק של הקלט (b*-c) פעמיים ומחברים ביניהם. ולכן אנחנו יוצרים פה את תת העץ השייך לקטע הקוד הזה, ומחברים אותו פעמיים לפעולת החיבור. זה נראה קצת מסורבל בעיניים, אך נוכל לראות בהמשך שברמת היעילות של הפעולה הזאת, אנחנו משיגים די הרבה.

- **ייצוג postfix** – כתיבה של המשוואה הזאת מחדש בצורה שבכל תת-חלק של המשוואה, יופיעו קודם כל שני המשתנים, ולאחריהם האופרטור שעובר עליהם. כך שלמשל הקלט שקיבלנו יהפוך להיות הדבר הבא –

a b c uminus * b c uminus * + assign

בעצם, אנחנו עובדים פה בצורה של מחסנית, כאשר בכל פעם שאנחנו פוגשים אופרטור מסוים, אנחנו מוציאים את המשתנים הרלוונטיים אליו (משתנה אחד לאופרטור אונארי, ושניים לבינארי) עושים את החישוב, ומחזירים את התוצאה למחסנית כמשתנה חדש. ננסה להדגים את זה על הקלט, כאשר לטובת הדיון נגדיר כי b=2, c=3.

a
a 2
a 2 3
a 2 3 –

עכשיו יש לנו בעצם אופרטור אונארי, אז נפעיל אותו על הכניסה האחרונה –

a 2 -3
a 2 -3 *

האופרטור הזה הוא בינארי, אז נוציא את שני המשתנים, ונחזיר למחסנית את התוצאה, ונמשיך –

a -6 2 3 –

שוב, אנחנו מכניסים את כל המשתנים עד שנגיע לאופרטור שמצריך אותנו לפעול –

a -6 -6
a -6 -6 +

מכניסים את תוצאת החיבור, וממשיכים –

a -12
a -12 assign

ומבצעים את ההשמה ל-a, וסיימנו.

- **קוד תלת מעני** – קוד שכל פקודה בו תכיל עד שלוש פעולות. בדרך כלל אנחנו נעבור עם הקוד הזה שהוא יותר קל ופשוט לנו לקריאה.

קוד תלת מעני

על מנת לראות את דרך יצירת הקוד והעבודה עליו, נוכל לראות את הדוגמה עבור כללי הגזירה הבאים –

production	semantic rule
$S \rightarrow \underline{id} := E$	S.nptr := mknode (' assign ' , mkleaf (id , id.place), E.nptr)
$E \rightarrow E_1 + E_2$	E.nptr := mknode (' + ' , E_1.nptr , E_2.nptr)
$E \rightarrow E_1 * E_2$	E.nptr := mknode (' * ' , E_1.nptr , E_2.nptr)
$E \rightarrow - E_1$	E.nptr := mkunode (' uminus ' , E_1.nptr)
$E \rightarrow (E_1)$	E.nptr := E_1.nptr
$E \rightarrow \underline{id}$	E.nptr := mkleaf (id , id.place)

את הכללים הסמנטיים אנחנו מנסים לנסח עכשיו על מנת שיהיו יותר תואמים לנו ליצירת עץ. אנחנו משתמשים בפונקציות של **mknode** או בצורה יותר מובנת make node שיוצר לנו צומת (בינארית) בעץ על פי הגדרה פשוטה של שלישיה – (father, left son, right son). כאשר בכל פעם שאנחנו מגיעים למצב בו אנחנו צריכים ליצור עלה, נשתמש ב-**mkleaf** שיכניס את שם המשתנה והמיקום שלו לתוך טבלת הסמלים. הפונקציה האחרונה שנשתמש בה היא **mkunode** שזה פשוט צומת אונארית ועוהדת בדיוק אותו דבר.

צורת הבנייה הזאת, שהפעולות משמשות בתור האב של תת העץ חוסכת לנו הרבה מקום בעץ הסופי – עד עכשיו אנחנו יצרנו את האב להיות החלק השמאלי של כלל הגזירה, ועכשיו אנחנו כבר לא משתמשים בו. רק אפשר לחשוב על כל החוקים שהיו לנו קודם שאנחנו גוזרים ארבע משתנים בשביל להגיע ל-id אחד. עכשיו כל זה יורד לנו מהעץ.

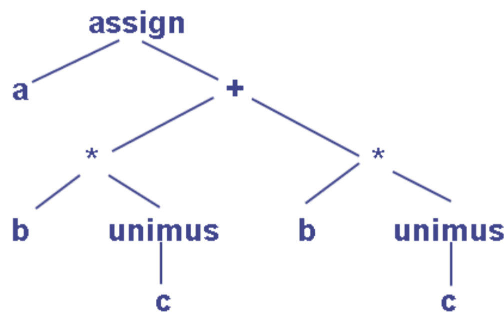
אם נרצה לעשות ייצוג של גמ"ל, נצטרך לוודא שאין לנו כפילויות בתתי העצים (ואם כן לטפל בהם כמו שכבר ראינו על ידי חיפוש של גרפים איזומורפים), ורק אחרי זה נחשב את הDAG עצמו.

אחרי שנבנה את העץ (באיזה אופן שלא יהיה) אנחנו יכולים להתחיל ליצור את הקוד התמ"ע. הצורה הכללית כמו שכבר אמרנו תהיה מהצורה הבאה:

X := Y op Z

כאשר כל אחד מהמשתנים האלה הוא לאו דווקא מקומות בזיכרון, אלא שמות של קבוע/משתנה כלשהו. אנחנו ננסה להגיע לרמה הפשוטה ביותר בשביל שיתאים היטב לקוד המכונה.

השיטה הבסיסית שאנחנו עובדים איתה, היא יצרה של משתנים זמניים ממוספרים $t_1..t_n$, והצבה של הערכים על פי שמות משתני הגזירה והטרמינלים השונים. נוכל לראות את הבנייה של הקוד מתוך שני העצים שראינו קודם, ונתחיל עם עץ התחביר המופשט –

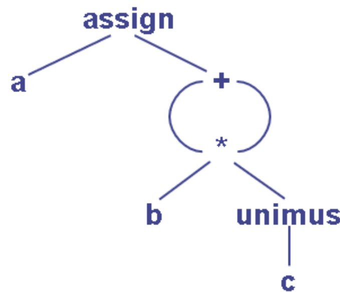


הקוד שאנחנו עובדים עליו עובר מלמטה למעלה ואוסף את הקוד והמידע על מנת שיכול להציב ב-a את תוצאת העץ. אנחנו נתחיל בלהציב ערכים ולחשב את התוצאות באופן הבא –

t_1 := -c
t_2 := b*t_1
t_3 := -c
t_4 := b*t_3

$t_5 := t_2+t_4$
$a := t_5$

לפני שנמשיך, יש כמה דברים חשובים שכדאי להעיר – קודם כל, כדאי לשים לב שאנחנו עובדים רק עם המשתנים הזמניים. כל עוד לא הגענו ממש לשלב האחרון, אנחנו לא משתמשים במשתנים אחרים, אפילו את ההצבה הפשוטה ביותר של הפיכה של c להיות שלילי צריכה להיות תחת משתנה. בנוסף, כשאנחנו עובדים בצורה הזאת, אין לנו קיצורי דרך – אנחנו צריכים לעשות פעמיים את כל הסיבוב של חישוב שני תתי העץ ורק אז לחבר את שני החלקים. נראה עכשיו בדיוק אותו דבר, רק עם מימוש עץ בצורה של DAG –



אמרנו קודם שהעץ הזה מביא לנו תוצאה שהיא יותר אופטימלית. איך זה קורה? את החישוב של b*-c אנחנו עושים רק פעם אחת למשתנה זמני, ואילו את החיבור עצמו אנחנו עושים על אותו משתנה משני צדדי האופרטור. כדאי לזכור, שבתצורה הקודמת לא היתה לנו שום אפשרות לדעת שבשני החלקים יש את אותו הקוד בדיוק, ולכן שם היינו צריכים לעבור ממש על כל הצמתים השונים. הקוד עכשיו ייראה כך –

$t_1 := -c$
$t_2 := b*t_1$
$t_3 := t_2+t_2$
$a := t_3$

חסכנו לנו כאן שתי שורות. אמנם זה לא נראה הרבה, אבל ביחס כמות שורות גדולה יותר זה כמובן יהיה משמעותי.

סוגי המשפטים בקוד ביניים

נראה עכשיו רשימה ארוכה של משפטים אפשריים שיהיו משפטים חוקיים לצורת קוד תלת מעני. יש פה כמה תתי קבוצות של משפטים שונים, וכדאי לשים לב במה מותר לנו להשתמש, שתכלס זה לא מעט –

<u>משפטי השמה</u>

```
x := y op z
x := op y
x := y
```

השמה יכולה להיות בינארית – אחרי פעולה מסויימת על שני אופרנדים, או אונארית (בדרך כלל מדובר על מינוס), וכמובן גם השמה מסוג של הצבת ערך בתוך משתנה.

<u>קפיצות</u>

```
goto L
if x relop y goto L
```

אנחנו עובדים עם שני סוגים של קפיצות (ונראה בהמשך איך משתמשים בכל אחת מהן) קפיצות בלתי מותנות – ברגע שאנחנו מגיעים לשורה הזאת אנחנו ישר חייבים לקפוץ לתווית מסויימת שהגדרנו קודם. וקפיצות מותנות – אם

יתקיים לנו תנאי x מסויים ביחס ל- y (RelOP = Relational Operator), אז אנחנו נקפוץ לתוית L. כשאנחנו בודקים האם ערך מסוים הוא אמת אז זה בדיוק הפקודה הזאת – x = true זה בדיוק x relop y.

<u>פרמטרים וקריאה לפרוצדורות</u>

```
param x
call p, n
return y
```

כשאנחנו קוראים לפונקציה אנחנו קודם כל דוחפים למחסנית את כל המידע הרלוונטי. כל חתיכת מידע כזאת נדחפת קודם כל ב-param x, ואז כשאנחנו קוראים לפונקציה עצמה עם call p,n אנחנו מבקשים את הפרוצדורה בשם p ומכריזים שעבורה אנחנו שמרנו n משתנים במחסנית וצריך להשתמש בהם. בסוף פעולת הפונקציה אנחנו מוסיפים כמובן את הערך המוחזר – y.

<u>גישה והשמה לאינדקסים</u>

```
x := y [ i ]
x [ i ] := y
```

למעשה, גישה לאינדקס מסויים בזיכרון זה פעולה קצת יותר מסובכת ממה שזה נדמה, אנחנו צריכים להגיע למקום y ואז לגשת עוד i מקומות קדימה. נראה בהמשך השלכות של השימוש באופרטור הזה.

<u>השמה של כתובות ומצביעים</u>

```
x := addr y
x := * y
* x := y
```

אין פה הרבה מה להרחיב.

עד כאן כל הרשימה הזאת, אך נשאלת השאלה, האם הרשימה הזאת מספיקה? האם יש פה יותר מידי פקודות? אנחנו יכולים לחשוב על עוד פעולות שאפשר להכניס לרשימה כמו =+. ואם נתרכז מספיק, נוכל לחשוב על אופרטורים נוספים שאולי יהיו יעילים ויעזרו לנו, כי אחרת את האופרטור =+ אנחנו נצטרך לפרק לשתי שורות נפרדות. אבל למעשה, יש לזכור שאנחנו שואפים להגיע בסוף לקוד מכונה. אם יהיה לנו הרבה אפשרויות למשפטים שהם קצת יותר "נוחים" לנו, אנחנו נצטרך לעבוד הרבה יותר קשה בשביל התרגום ברמה הבאה, שגם ככה אנחנו עלולים לתרגם חלק מהשורות כאן למספר שורות נוספות, אז לכן כדאי להמנע מזה.

איך נעבוד על יצירת הקוד בעצמו? נגדיר "תרגום מונחה דקדוק" שמשתמש במשתנים ובתכונות על מנת לצבור את הקוד. עבור כל משתנה גזירה אנחנו נגדיר תכונת code, למשל E.code, ונאסוף את כל הקוד מלמטה למעלה (זה מה שאנחנו עובדים כברירת מחדל, כמו שכבר ציינו קודם, בשביל לאסוף את הקוד לכיוון השורש). כך שאם יש לנו כלל של $E \rightarrow E_1$, ואנחנו כמובן צריכים לזכור שה"בן" של משתנה הגזירה הוא E אחר לגמרי, אז אנחנו נגדיר את הכללים הסמנטיים באופן הבא –

$E.var := E_1.var$
$E.code := E_1.code$

הטיפוס של האבא יוגדר כטיפוס של הבן, ובאותו האופן גם הקוד יעלה כלפי מעלה – במקרה הזה ללא שינויים, אך בדוגמא שנראה תיכף, אנחנו משרשרים חלקי קוד וסימנים שונים על מנת שהקוד יהפוך להיות קריא ו"אמיתי".

production	semantic rule
S → <u>id</u> := E	S.code := E.code || gen (id.var ' := ' E.var)
E → E_1 + E_2	E.var := newtemp; E.code := E_1.code || E_2.code || gen (E.var ' := ' E_1.var ' + ' E_2.var)
E → E_1 * E_2	E.var := newtemp; E.code := E_1.code || E_2.code || gen (E.var ' := ' E_1.var ' * ' E_2.var)
E → – E_1	E.var := newtemp; E.code := E_1.code || gen (E.var ' := ' ' uminus ' E_1.var)
E → (E_1)	E.var := E_1.var E.code := '(' || E_1.code || ')'
E → <u>id</u>	E.var := id.var ; E.code := ' '

קודם כל – הסימן "||" מבטא שרשור של קוד, והפונקציה gen לא כל כך רלוונטית למטרות שלנו אז נתעלם ממנה.

אנחנו יכולים לראות שבעבור חלק גדול מהמשתנים יש לנו כמה פעולות – קודם כל הגדרה של משתנה זמני (newtemp) שאנחנו יכולים לעבוד איתו, ואחר כך העבודה על הקוד. כדאי לשים לב, ששרשור הקוד לוקח קודם כל את כל מה שנאסף עד לאותו רגע ורק אז מוסיפים אליו את הקוד הרלוונטי לכלל הגזירה. זה דבר הגיוני, כי תכלס אנחנו אמנם עוברים על העץ מלמטה, אבל כל השורות שעברנו צריכות להיות הרישא של הקוד ולעשות את כל החישובים השונים, ורק כשנסיים איתם נוכל להתפנות למה שהיה לנו להוסיף תחת אותו כלל גזירה.

לצורך הדוגמא הראשונית לכתיבת קוד לכלל גזירה והצגה של תוויות אנחנו נסתכל על מימוש של כללים סמנטיים לפקודת while פשוטה. אבל לפני שנתחיל, ניתן הקדמה קצרה.

קודם כל, תנאי ה-while בעצמו מורכב משלושה חלקים עיקריים: 1. תנאי 2. פעולה 3. קפיצה לראש הקטע. כאשר הסדר של כל חלק בפני עצמו יש לו חשיבות – אם נקדים את הפעולה לתנאי, אז אנחנו לא ניצור while אלא do-while. ונראה בהמשך גם איך מממשים כל אחד מאלה.

יש עיקרון חשוב, שאנחנו נדבר עליו עוד בהמשך, אבל אנחנו צריכים לדעת בדיוק היכן מתחיל כל "קטע" קוד ואיפה הוא נגמר. השאיפה שלנו, הוא שנוכל לשים בכל אחד כזה תוית ולומר "זה ההתחלה של קטע X", "זה הסוף של קטע X". ברגע שיהיו לנו כל התויות המתאימות אנחנו נוכל לשתול בקוד את הקפיצה למקומות הנכונים. לענייננו – כאשר יש לנו קפיצה מותנית כמו ב-while, המימוש חייב להתחשב בהתחלה ובסוף, מדוע? כי כאשר נעבור על קטע הקוד ונגיע לסופו, נצטרך לעשות קפיצה בחזרה (קפיצה לא מותנית) לראש הקטע – התנאי. ואם התנאי לא מתקיים, אנחנו צריכים ישר לקפוץ לסוף הקטע (ראש הקטע הבא) ולהמשיך משם. לאחר כל ההקדמה הנ"ל אנחנו יכולם לראות את הקוד ולראות מה פעלנו –

production	semantic rule
S → <u>while</u> E <u>do</u> S_1	S.begin := newlabel ; S.after := newlabel ; S.code := gen (S.begin ' : ') || E.code || gen (' if ' E.var ' = ' ' 0 ' ' goto ' S.after) || S_1.code || gen (' goto ' S.begin) || gen (S.after ' : ')

כלל הגזירה שלנו יוצא מ-S ואומר while E do S_1. יש לזכור שלא רק ש-E הוא תת עץ שלא ידוע לנו כמה קוד הוא מכיל, גם S_1 הוא תת עץ ואף יכול לחזור ולתת את כל המשפט הזה מחדש. עכשיו נראה מה הכללי הסמנטיים שאנחנו משתמשים בהם – קודם כל אנחנו מייצרים שני תוויות חדשות – אנחנו לא מאתחלים אותם בערך מסוים, אלא רק מגדירים שיש לנו את שתי התוויות, אחת לראש הקטע S, והשניה לסוף הקטע. יש לשים לב – החלק הזה הוא לא קוד! אנחנו כאן מדברים עדיין על תכונות של S, רק בשורה השלישית אנחנו מתחילים להתעסק עם הקוד –

קודם כל, רגע לפני שאנחנו מתחילים להתעסק עם הקוד, אנחנו "שותלים" את תווית ההתחלה. עד עכשיו זה היה רק איזה פוינטר שהצביע לשום-מקום, ועכשיו יש לנו את המקום הראוי לו. לאחר מכן, אנחנו משרשרים את כל הקוד של E – למה אנחנו לא כותבים בכלל את המילה while בכל הקוד הזה? אנחנו צריכים לזכור שהפקודה הזאת היא פקודה של שפה עילית, ועכשיו אנחנו מנסים כמה שאפשר לפשט את השפה, ולכן אנחנו לא כותבים while, אלא ממששים אותו.

איך עובד המימוש? אנחנו מכניסים את כל הקוד של E, הלא הוא התנאי הרצוי, ובודקים את התוצאה הסופית שלו. כזכור, E הוא תת-עץ שעשה חישוב מסוים ורץ על הערכים שהוא קיבל, בסוף כל הריצה הזאת, יש איזה ערך שיושם ב-E.var, ואנחנו מתייחסים כרגע לערך בוליאני של 1/0, ומה שאנחנו עושים זה בדיקה האם E.var ==0, כלומר האם קיבלנו false על הקוד של E. אם אנחנו אכן קיבלנו 0, זה אומר שהתנאי לא מתקיים, ואנחנו יכולים לצאת מהבלוק של ה-while ולכן אנחנו נשרשר לזה את הקפיצה לסוף – S.begin. שימו לב, שלפי מהות התנאי אנחנו נחליט על הבדיקה – במקרה של while אנחנו נקפוץ רק ברגע שהתוצאה תהיה false, ולכן זה מה שנבדוק. אם יש לנו do-whille, אז אנחנו נבדוק שהתנאי עדיין מתקיים כלומר שהוא true ורק אם זה אכן כך, אנחנו נקפוץ בחזרה ל-S.begin.

לאחר שטיפלנו בקיום התנאי, אנחנו צריכים לשתול את הקוד שייעשה תחת אותם התנאים, ולכן אנחנו משרשרים את כל הקודם של S_1. לא באמת אכפת לנו מה הוא, וברגע שנסיים את קטע הקוד הזה, אנחנו נקפוץ ישר בחזרה לראש הקטע – אנחנו לא בודקים פה שוב את התנאי, בגלל שאנחנו מנסים כמה שאפשר לחסוך בקוד ולא לכתוב פעמיים, ורק אחרי שנכניס את הקפיצה, נוכל להכניס בחזרה את התווית לסיום קטע הקוד של S. זה ייתן לנו את האפשרות לקפוץ אליו במידה והתנאי שם למעלה לא יתקיים.

ייצוג של קוד תלת מעני בזיכרון

עד עכשיו ראינו איך כותבים את הקוד, אבל לא דיברנו איך הוא מוצג ונשמר בזיכרון עצמו. הקוד עצמו שאנחנו עובדים עליו מיוצג בצורה שכבר תיארנו של שני ערכים, אופרטור והשמה למשתנה אחר. אבל השאלה של השמירה בזיכרון, חשובה לנו בשביל שנוכל להשתמש בקוד באופן יעיל. נראה שתי שיטות עיקריות –

1. רביעיות – אמנם אנחנו מדברים פה על קוד תלת מעני, אבל גם אותו אפשר לשמור ברביעיות ולהרוויח מזה. כיצד? כל שורה נשמר בצורה הבאה – op, arg_1, arg_2, result. כאשר מלבד האופרטור עצמו, כל השאר הם מצביעים לטבלת הזיכרון למשתנים שונים, חלקם זמניים וחלקם מוגדרים בקוד, ובכל שורה אנחנו בעצם אומרים כך – תעשה את הפעולה (op) הבאה על שני המשתנים (או אחד) ותשמור לתוך המשתנה הבא (result). למשלעבור קטע הקוד שכבר פגשנו בעבר –

t_1 = - c
t_2 = b * t_1
t_3 = - c
t_4 = b * t_3
t_5 = t_2 * t_4
a = t_5

נוכל לכתוב את הטבלה הבאה -

	op	arg 1	arg 2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

המימוש הזה הוא מאוד פשוט, ועקרונית גם אין מניעה לשנות ולהחליף שורות לטובת אופטימיזציה – כמובן, ברמת הסביר, אנחנו לא מדברים לשים את שורה 5 ראשונה, אלא למשל להעלות את שורה 2 ודברים דומים כאלה.
החיסרון של השיטה הזאת, הוא באופן שאנחנו צריכים כל הזמן להוסיף משתנים זמניים וזה אוכל לנו הרבה מקום בזיכרון.

2. שלישיות – השיטה הזאת מבקשת לחסוך בהקצאות זיכרון, ומתייחסת אך ורק לפעולות שאמורות להתרחש – $op\ arg_1\ arg_2$, כאשר במקום לשמור את התוצאה, אנחנו מתיחסים למספר השורה בה נעשתה הפעולה ומפנים אליה. איך בדיוק זה נראה?

	op	arg 1	arg 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

במקום להשתמש במשתנים זמניים, אנחנו מגדירים את ה-arg בדרך כלל השני להתייחס למספר השורה הרלוונטי. כי אם אנחנו גם ככה עושים שם את החישוב, אנחנו לא צריכים באמת לשמור את התוצאה. כמובן שהשיטה הזאת חוסכת לנו בהרבה הקצאות זיכרון, אבל יש בה בעייתיות מובנית – אם אנחנו משועבדים למיספור של השורות, אנחנו כמעט ולא יכולים לעשות שינויים בקוד, אם נחליט להעלות את שורה 2 למעלה, אנחנו נאבד מהמשמעות של הקוד והכל ילך לפח וחבל. מעבר לזה, מאחר ואנחנו מוגבלים בכמות המידע שאנחנו יכולים להכניס לתוך כל שורה, יש פעולות שידרשו שתי שורות, כמו למשל השמה או גישה למערך בזיכרון –

x := y [i]

	op	arg 1	arg 2
(0)	= [ ]	y	i
(1)	assign	x	(0)

x [i] := y

	op	arg 1	arg 2
(0)	[ ] =	x	i
(1)	assign	(0)	y

למה אנחנו צריכים שתי שורות בכל פעם? כי גישה למערך בעצם הולכת למקום y בזיכרון ומשם מחשבת עוד גישה למקום ה-i מאותה נקודה, מה שכמובן אומר שאנחנו צריכים לעשות פה פעולת חישוב. ובגלל שאין לנו איפה להכניס את החישוב הזה, אנחנו צריכים לעבור שורה, ולהגדיר שם את ההשמה על פי החישוב שאנחנו עושים באותה שורה. אז יופי לנו, חסכנו משתנים, אבל אנחנו צריכים יותר שורות וחזרנו לנקודה בעייתית. יותר מזה, בדוגמא הימנית אנחנו מכניסים את החישוב של שורה 0 וממשיכים הלאה, אבל בדוגמא השמאלית, בכל פעם שנרצה להכניס משהו לתוך המערך נצטרך לחזור ולחשב את השורה בשביל לדעת לאן להכניס את הערך.

יש ייצוג שלישי, שמשתמש בשיטה הזאת ובא לפתור את הבעיה שאי אפשר להזיז שורות. השיטה הזאת אומרת שאנחנו נשמר מערך שייצג לנו את סדר המימוש של הקוד. אנחנו יודעים את מספר השורות, אז במקום להזיז את השורה עצמה, אנחנו נשמור את הרצף בו אנחנו אמורים לבצע את הקוד עצמו. זה כמובן נוח למימוש ויכול להכניס אופטימיזציות, אבל עדיין לא פתרנו את הבעיה של הפעולות הטרינאריות אותן ראינו למעלה.

הקצאת מקום בזיכרון

תכונות נוספות שחשובות לנו לעיטור העץ הוא הגודל שאנחנו צריכים להקצות לכל קטע הקוד. אנחנו מדברים עקרונית על זיכרון לוגי, אליו אנחנו רוצים להכניס את הקוד ואת המימוש שלו, אז אנחנו דואגים בדרך כלל לדעת מראש מה גודל המקום שאנחנו נשתמש בו, נקצה אותו, ואז נתחיל לעבוד על הקוד והקלט.

לצורך הדוגמה הפשוטה, אם יש לנו הגדרה של שני int, אז כל אחד מהם יתפוס ארבעה ביטים. כך שהראשון יהיה 0-3 והשני 4-7. השימוש במידע על הקצאת הזיכרון משמש הן להקצאת הראשונה כמו שציינו, וגם בגישה לכל משתנה. כמו שהבאנו בדוגמא, אנחנו יודעים שאם אנחנו רוצים לקרוא את המספר השני אנחנו צריכים לגשת למקום 4 ורק משם נוכל להתחיל לקרוא. מכאן, שעלינו לא רק לחשב את כמות הזיכרון המוקצה, אלא לשמור בכל משתנה את המיקום שלו (להלן: אופסט) ביחס לתחילת הזיכרון.

בשביל להכניס את האופסט לכל משתנה, אנחנו נגדיר ערך 0 התחלתי בכלל הגזירה הראשון, ובהגדרה של הטיפוסים בעצמם אנחנו נכניס שדה של גודל נצרך – T.width = x. כך שעכשיו נוכל לחשב לכל משתנה את הגדל המתאים לו, וכן את האופסט שיילך וייצבר לאט לאט. נראה כללים ומימוש חוקים –

production	semantic rule
P → D	**{ offset := 0 }**
D → D D	
D → T <u>id</u> ;	**{ enter (id.name, T.type, offset); offset := offset + T.width }**
T → <u>integer</u>	**{ T.type := integer ; T.width := 4 }**
T → <u>real</u>	**{ T.type := real ; T.width := 8 }**
T → T_1 [<u>num</u>]	**{ T.type := array (<u>num</u>.val, T_1.type) ; T.width := <u>num</u>.val × T_1.width }**
T → *T_1	**T.type := pointer (T_1.type) ; T.width := 4 }**

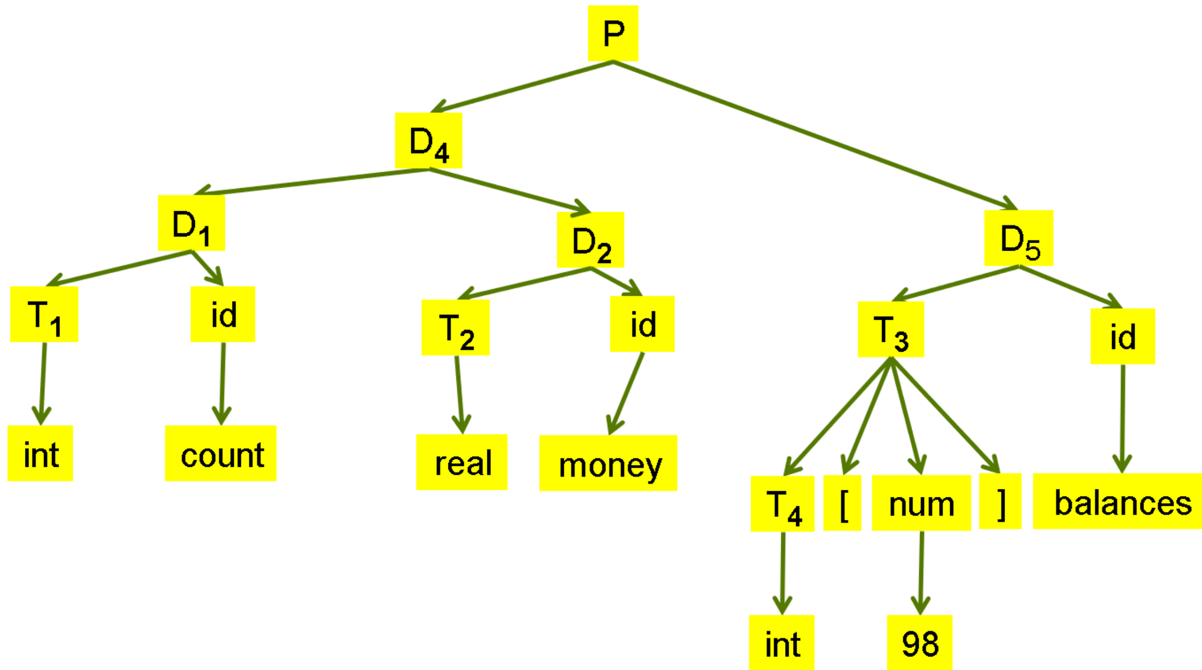
נשים לב- הכלל הראשוני מגדיר לנו Offset = 0, והגזירה של T לטיפוסים השונים תגדיר הן את הטיפוס והן את הגודל הדרוש לו. דבר זה ייתן לנו להגדיר בכלל השלישי את המיקום ביחס לאותו אופסט, שיהיה האופסט הנוכחי בתוספת הגודל של המשתנה.

דבר נוסף – הגזירה למערך בגודל מסוים לא מוגדרת מראש – קודם כל אנחנו מגדירים שהטיפוס הוא מערך מהסוג של T_1 – מידע שייאסף לנו מלמטה, וגודל המערך (כמערך) יהיה בגודל הערך של num. אחרי זה אנחנו נגדיר את הגודל האמיתי בזיכרון שלו- הכפלה של הערך של num בגודל של T_1. כמה הגודל הזה מוגדר? תלוי אם הוא ייגזר ל-integer או ל-float.

נראה עכשיו דוגמה של עץ גזירה עבור הקלט הבא:

```
int count
real money
int[98] balances
```

נגיד ויש עוד קוד, אבל אנחנו לא מתייחסים אלו בשלב זה. איך ייראה העץ?



עכשיו כשנרוץ על העץ, נוכל להכניס את כל הערכים למקומם –

- T_1 יוגדר כ-int מה שיגדיר גם את הגודל שלו בהתאם.
- השם של id יוגדר כ-count.
- D_1 יגדיר משתנה חדש עם השלישיה: שם (שמגיע מ-id.name), טיפוס (T_1.type), ואופסט (שכרגע מוגדר 0), ובנוסף יזיז את האופסט בגודל המשתנה (offset+T_1.width).
- אותו תהליך יקרה גם עם D_2.
- וכן על זה הדרך.

כל הסיפור הזה טוב ויפה, אבל רק אם אנחנו מתייחסים לעובדה שאנחנו עובדים top-down. ככה אנחנו יכולים להגדיר את הגודל של כל משתנה ולהעביר את זה הלאה לאופסט וכל הסיפור הזה. אבל מה קורה אם אנחנו עובדים בגזירה של bottom-up? כל המידע הזה לא יעבור בחזרה לאן שאנחנו צריכים. בשביל להתמודד עם זה, אנחנו מגדירים worker. כלל גזירה שייאפס לנו את האופסט, ואיתו אנחנו נוכל להתחיל לעבוד. כך שבמקום כלל של P→D, יהיה לנו P→MD, וכל מה ש-M יעשה הוא רק להיגזר ל-ε ונכניס בו פעולת דמי שמגדירה את האופסט להיות 0. עכשיו כשנעבוד מלמעלה, אנחנו קודם כל נגזור לשם, נגדיר את האופסט ואז נוכל להמשיך ולגזור.

ניתוח משמעות : דקדוקי תכונות ופתירת שמות – מצגת דוגמאות מס' 5

בתרגול הזה אנחנו נראה איך אנחנו מתמודדים עם כתיבת כל הפעולות הסמנטיות הדרושות בהינתן חוקי גזירה, ואיך אנחנו מיישמים את העבודה עליהם עבור קטע קוד, מה שייתן לנו את האפשרות לכתוב קוד ביניים מתוך ניתוח הגרף.

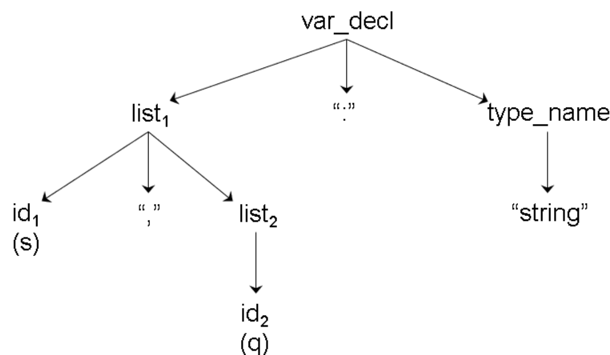
נתחיל קודם כל בקוד עליו אנחנו מדברים –

```
s, q: string;
procedure t(s: real, r: string) is
        function f(s: real) return string;
        function f(s: real, r: string) return string;
begin
        q := q * r;
         …
end;
```

יש לנו פרוצדורה (שאינה מחזירה ערכים) המכילה שתי פונקציות שונות ועושה חישוב קטן. אנחנו כרגע נדבר רק על השורה הראשונה של הכרזת שתי המשתנים, ונראה כיצד אנחנו יכולים בעצם לעבור את כל השלבים ולהוציא את הקוד. כללי השכתוב הרלוונטיים לנו לשורה הזאת הם אלו –

var_decl → list “:” type_name
list → id “,” $list_1$
list → id
type_name → “string”

אין פה משהו מסובך מידי, ואפשר ליצור את עץ הגזירה המתאים הבא –



הערה קטנה – בתרגיל הבית (ואולי במבחנים) מבקשים מאיתנו קודם כל לכתוב את הפעולות הסמנטיות לפני שניצור את העץ. לענ"ד, הגישה היותר נכונה היא ליצור קודם את העץ, ואז אפשר לראות באמת אילו תכונות עוברות לאן ובאיזה אופן ולאור זה לכתוב את הפעולות. ואני אסביר למה הכוונה. בדוגמה שלנו, אנחנו נדרשים להעביר את המידע לגבי ה-type, אל המשתנים השונים.

במבט מלמעלה, אפשר כבר להבין שהמשתנים שגוזרים לעלים כלשהם, הם אילו שיביאו לנו את התכונות הנוצרות הבסיסיות. במקרה שלנו, אם ניקח את הכלל -

type_name → “string”

אז כאן אנחנו בעצם פוגשים לראשונה את המהות של הטיפוס עליו אנחנו עובדים, ולכן נוכל להגדיר שבגזירה פה אנחנו מגדירים את הטיפוס של הצומת type_name. או באופן פורמלי יותר –

type_name.type := string

כמובן, שבמקרה יותר רציני כמו בתרגיל הבית, יהיה לנו אפשרות לגזור את ה-type_name לעוד כמה טיפוסים, וכל אחד מהם יהיה כלל גזירה אחר ופעולה סמנטית נפרדת. נעבור לכלל הבא (לפי המעבר בעץ) שנוגע במידע ש"קיים" לנו –

var_decl ⟶ list “:” type_name

אנחנו יודעים כבר מה הטיפוס של החלק הימני, וברור לנו שכל החלק השמאלי יורש ממנו את הטיפוס על מנת להעביר למשתנים שיוגדרו, ולכן פשוט נעבר את הטיפוס הלאה –

list.type := type_name.type

בשלב הבא, אנחנו מטפלים בהעברת המידע הזה וטיפול במשתנים עצמם. אנחנו רואים ש-list יכול לגזור לשתי אפשרויות דומות –

$list \longrightarrow id$ “,” $list_1$
$list \longrightarrow id$

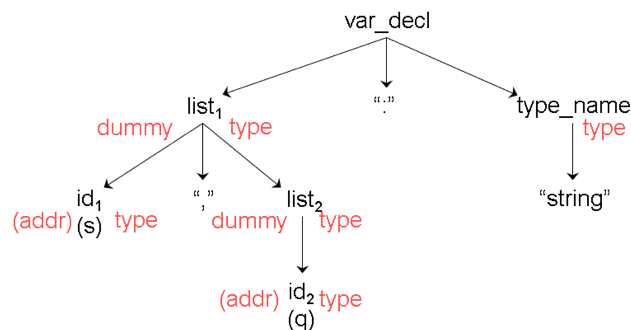
כלומר, אנחנו פותחים בכל פעם רק משתנה Id בודד, ואם יש לנו יותר ממשתנה אחד, אנחנו נמשיך לשרשר אותו לרשימה החדשה. עכשיו, ברור לנו שאת הטיפוס נצטרך להעביר לשני חלקי הרשימה על מנת שיעבור כמו שצריך, ולכן נכתוב על כל אחד מהם פעולה נפרדת של הכרזת הטיפוס, אך לא די בזה. ברגע שאנחנו נתקלים במשתנה id אנחנו נידרש לרוב להכניס אותו לטבלת הסמלים, ולכן נכניס לזה פעולת דמה נפרדת –

id.type := list.type
$list_1$.type := list.type
Sym_Table.Enter_Name(id.addr)

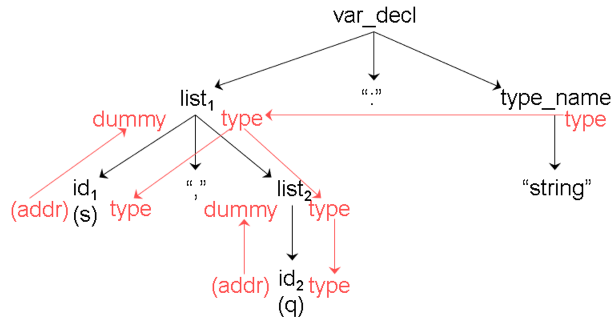
כמובן, שכללים אלו רלוונטיים במידה ויש לנו גם List המשך. אך אם אין לנו, ואנחנו גוזרים ישירות ל-id, נוכל להספק רק בשתי פעולות מתוך השלוש –

id.type := list.type
Sym_Table.Enter_Name(id.addr)

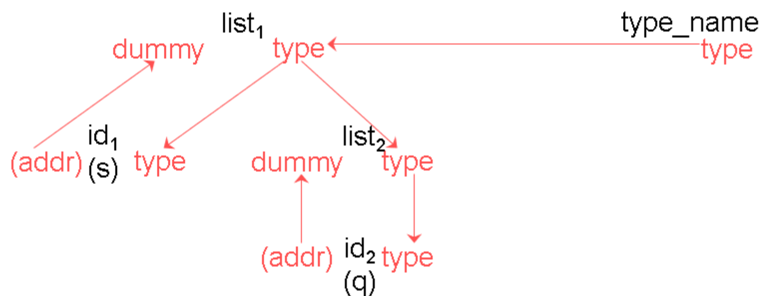
לאחר שסיימנו לקבוע את הפעולות עבור כל גזירה, נוכל לחזור לגרף ולסמן את התכונות המתאימות לכל צומת –



ועכשיו אנחנו יכולים לעקוב אחרי שטף המידע ולראות את הצורה בה אנחנו מעבירים את המידע ממקום למקום –

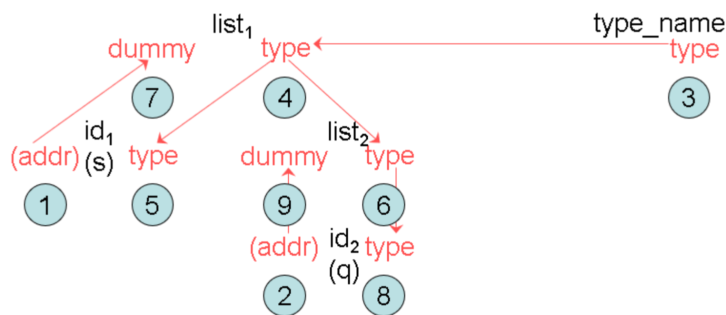


כדאי לשים לב, שפעולת הדמה של הכנסת כתובת של משתנה נחשבת תכונה נורשת של צומת ה-list שמעליה.
בשביל שיהיה לנו קצת יותר קל ו"נקי" להסתכל על הכל, נשאיר רק את התלויות השונות, ואת הגרף המכוון הנוצר בו –



אנחנו שואפים לקבל פה DAG, ובמבט ראשון אנחנו עלולים לחשוב שיש ה מעגלים, אבל זה לא נכון! המידע ש-list מוריש ל-id, הוא הטיפוס, ומה שעולה בחזרה היא פעולה נוצר אחרת, ואמנם יש פה מעגל, אך זה פשוט דרך אחרת לגמרי ויכולה להתבצע ללא כל תלות במה שקורה בצד השני.

כל שנותר לנו הוא להחליט מה הסדר שנעבוד בו. כמובן שברוב המקרים תהיה לנו יותר מאפשרות אחת, וזה בסדר. למשל בגרף לפנינו, אנחנו יכולים להתחיל להתעסק עם ה-type, ואנחנו יכולים להתחיל עם פעולות הדמה, ומה שלא נבחר, אין לזה השלכות ממשיות כרגע. אנחנו מחליטים ללכת על הסדר הבא –



אנחנו יכולים גם לכתוב את הפעולות הסמנטיות באופן שיהיה מספיק ברור בשביל לקבוע מה יכול להיות סדר פעולות, אבל כאן לא נתעסק עם זה, נתחיל בלהעלות את הכתובת של המשתנים לצמתי האב של כל אחת מהן, נטפל בשרשור הטיפוסים ממקום למקום, ולבסוף נפעיל את פעולות הדמה.

בשלב זה, אנחנו יודעים את הסדר של העבודה, ואת הקוד הנדרש בכל שלב, לכן נוכל לאסוף את הכל לכלל "תוכנית" – נעבור על כל צומת ונאסוף את הקוד הרלוונטי לפי הסדר –

type_name.type := string

list$_1$.type := type_name.type
id$_1$.type := list$_1$.type
list$_2$.type := list$_1$.type
Sym_Table.Enter_Name(id$_1$.addr)
Id$_2$.type := list$_1$.type
Sym_Table.Enter_Name(id$_2$.addr)

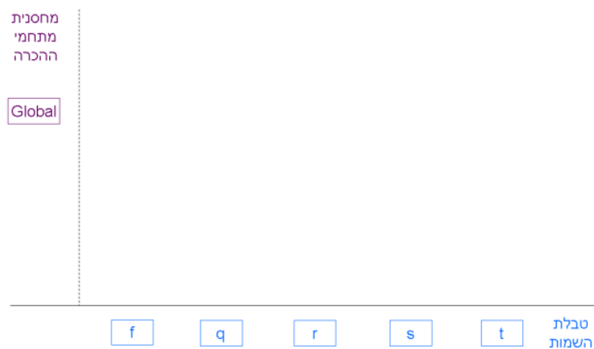
מתחמי הכרה וישויות

החלק השני של התרגול מתעסק בטבלת הסמלים. אנחנו צריכים לדעת בכל רגע נתון אילו משתנים יש לנו, מי נמצא בתוך מי ועוד מידע בסגנון הזה. בנוסף, אנחנו צריכים להתחשב במתחמי ההכרה השונים של כל פונקציה, אם פתאום פונקציה תקרא למשתנה שללא נמצא בתחום שלה, אלא ברמות מעל, האם הוא באמת יכול להגיע אליו? את כל זה נבדוק על ידי טבלת הסמלים.

בטבלה יופיעו לנו כל מיני ישויות שונות שייצגו את מופעי המשתנים בכל שלב. אנחנו מגדירים כי כל יישות מוכרת במתחם ההכרה (scope) שלה, וכן בכל המתחמים מתחתיו (המוגדרים בו). כמובן, שאנחנו מתירים להעמיס שמות. בטבלה אנחנו לא נתעסק בשאלה האם ה-x שאנחנו מתעסקים איתו הוא Int או string, אלא בכל פעם שיופיע לנו את אותו שם נכניס אותו לטבלה. העמסת שמות שכזאת נקראת הומונים.

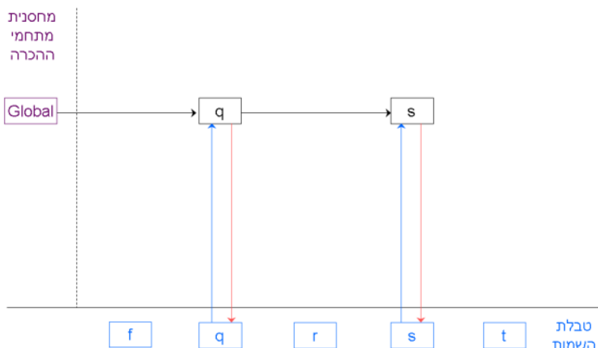
על מנת להגדיר את טבלת הסמלים בצורה יעילה ומשמעותית, אנחנו נשתמש במטריצה דלילה – כל תא במטריצה יכיל מצביע "אנכי" ומצביע "אופקי", ויקושר לתא הבא בו מופיע התוכן. שורות המטריצה יציינו מתחמי הכרה שונים – בכל פעם שנוסיף מתחם, נרד שורה ונעבוד במקום הנמוך יותר, והעמודות יהיה שמות המשתנים השונים שאנחנו יודעים שעלולים להיות פה.

עכשיו נעבור על כל קטע הקוד הקודם, ונראה כיצד כל שורה (ולפעמים אף פחות זה) אנחנו משנים את טבלת הסמלים באופן מתאים. –



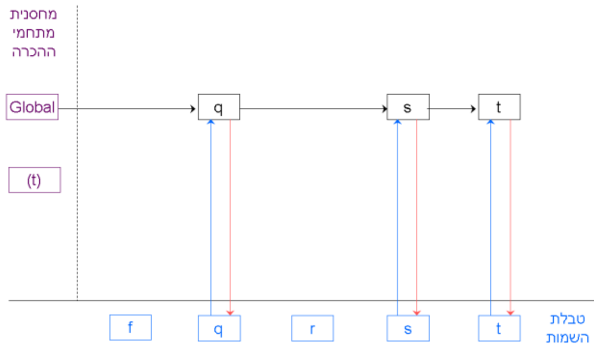

```
1. ↓s, q: string;
2.  procedure t(s: real, r: string) is
3.    function f(s: real) return string;
4.    function f(s: real, r: string) return string;
5.
6.  begin   -- body of t(real, string)
7.    q := q * r;
8.    ...
9.  end;
```

בשלב ההתחלתי אנחנו יכולים לראות שהטבלה ריקה ואין בינתיים שום הגדרה חשובה.



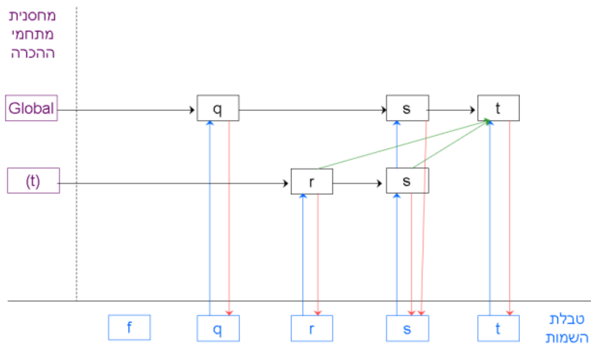

```
1.  s, q: string;↓
2.  procedure t(s: real, r: string) is
3.    function f(s: real) return string;
4.    function f(s: real, r: string) return string;
5.
6.  begin   -- body of t(real, string)
7.    q := q * r;
8.    ...
9.  end;
```

ברגע שאנחנו מסיימים את השורה הראשונה, הוגדרו לנו שני שמות בטבלה. מצד שמאל יוצא חץ היישר ממתחם ההכרה המתאים לשני המשתנים. בנוסף, לכל אחד מהמשתנים יש מצביע לטבלת השמות (אדום), וחץ כחול שמצביע לנו על ההומונים הבא, כלומר, יכול להיות שיהיו לנו באותו מתחם שני משתנים עם אותו שם, אבל על ידי החץ הכחול נדע למי מהם אנחנו מתייחסים.



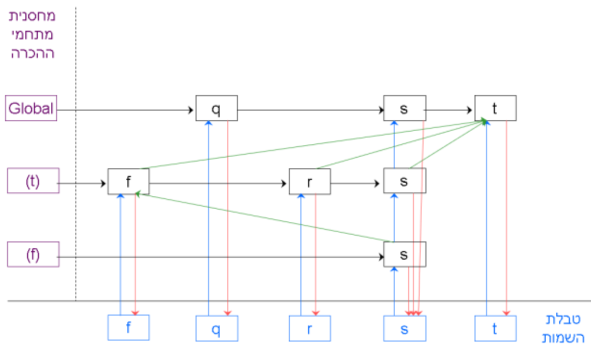

```
1.  s, q: string;
2.  procedure t(s: real, r: string) is
3.    function f(s: real) return string;
4.    function f(s: real, r: string) return string;
5.
6.  begin   -- body of t(real, string)
7.    q := q * r;
8.    ...
9.  end;
```

עכשיו אנחנו מתחילים לעבוד על הפרוצדורה הראשונה והיחידה שיש לנו t-. קודם כל אנחנו מגדירים אותה בדיוק כמו שנגדיר את שאר השמות הרלוונטיים, בנוסף נפתח לו מתחם הכרה חדש מתחת למתחם הגלובלי, ואז נגדיר את הארגומנטים שלו –



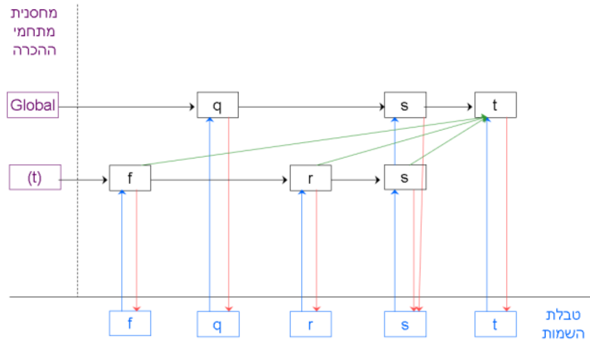

```
1.  s, q: string;
2.  procedure t(s: real, r: string) is
3.    function f(s: real) return string;
4.    function f(s: real, r: string) return string;
5.
6.  begin   -- body of t(real, string)
7.    q := q * r;
8.    ...
9.  end;
```

את הארגומנטים אנחנו מכניסים ברמה שמתחת הגלובלית, וכל אחד מהמשתנים יצביע בחץ ירוק על "מתחם ההכרה העוטף". מתי שלא מדובר על ה-scope הראשי, כל המשתנים יצביעו למתחם בו הם מוגדרים ברמה שמעליהם. דבר נוסף שכדאי להתעכב עליו – החץ הכחול של s מצביע עכשיו ל-s הנמוך יותר, מאחר ואנחנו בתוך הפרוצדורה עצמה, ואם נקרא לסתם s זה בוודאי יהיה המקומי, אך אם נרצה משתנה אחר, אנחנו נצטרך לפנות אליו בצורה מסודרת יותר. לא רלוונטי כרגע. נמשיך להגדרה של הפונקציות הפנימיות יותר –



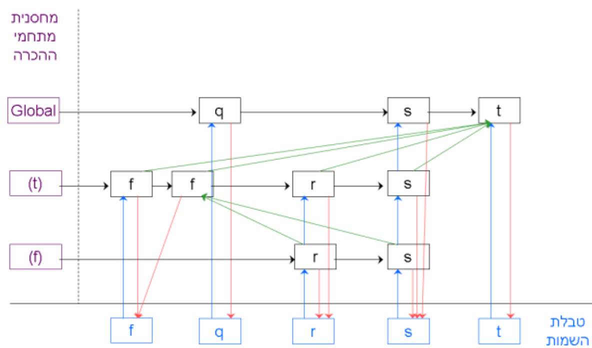

```
1.  s, q: string;
2.  procedure t(s: real, r: string) is
3.    function f(s: real) return string;
4.    function f(s: real, r: string) return string;
5.
6.  begin   -- body of t(real, string)
7.    q := q * r;
8.    ...
9.  end;
```

כאן אנחנו מגדירים עוד רמה של פונקציה פנימית – קודם כל אנחנו מכניסים את ה"שם" של f לתוך ה-scope של t, ואז פותח לו מתחם משלו עם המשתנה s שמצטרף לשרשור הנפלא של ה-sים.



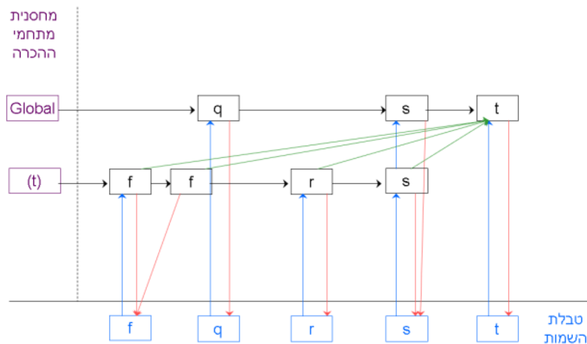

```
1.  s, q: string;
2.  procedure t(s: real, r: string) is
3.    function f(s: real) return string;
4.    function f(s: real, r: string) return string;
5.
6.  begin   -- body of t(real, string)
7.    q := q * r;
8.    …
9.  end;
```

כשאנחנו מסיימים להגדיר את f הראשונה, ואז יוצאים ממנה, אנחנו סוגרים את מתחם ההכרה שלו, אך השם שלו יישאר שם עד להודעה חדשה כי הוא מבחינתו מוגדר בתוך t בדיוק כמו כל משתנה אחר. עכשיו נמשיך לשורה הבאה בה נגדיר פונקציית f נוספת –



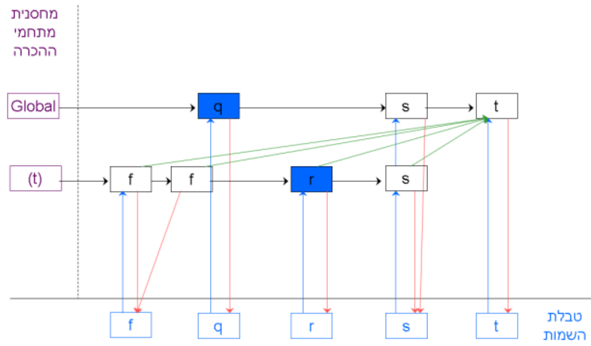

```
1.  s, q: string;
2.  procedure t(s: real, r: string) is
3.    function f(s: real) return string;
4.    function f(s: real, r: string) return string;
5.
6.  begin   -- body of t(real, string)
7.    q := q * r;
8.    …
9.  end;
```

שוב אנחנו פותחים מתחם הכרה חדש לפונקציה עליה אנחנו עובדים, ומגדירים את המשתנים הפנימיים שלה. שימו לב, שבמתחם של t הפונקציה הזאת מופיעה אחרי ה-f הראשונה עליה עבדנו בשורה הקודמת – בכל מקרה כזה שיהיו לנו שני שמות עם משמעות שונה, אנחנו נפתח אותם מחדש ונתייחס לכל אחד מהם בנפרד.




```
1.  s, q: string;
2.  procedure t(s: real, r: string) is
3.    function f(s: real) return string;
4.    function f(s: real, r: string) return string;
5.
6.  begin   -- body of t(real, string)
7.    q := q * r;
8.    …
9.  end;
```

בשלב הזה, אנחנו סיימנו להגדיר את כל מה שחשוב לטובת פעולת הפרוצדורה, וכל מה שנותר לנו הוא לנסות ולהבין בשורה 7 לאילו משתנים עלינו ללכת –




```
1.  s, q: string;
2.  procedure t(s: real, r: string) is
3.    function f(s: real) return string;
4.    function f(s: real, r: string) return string;
5.
6.  begin   -- body of t(real, string)
7.    q := q * r;?
8.    ...
9.  end;
```

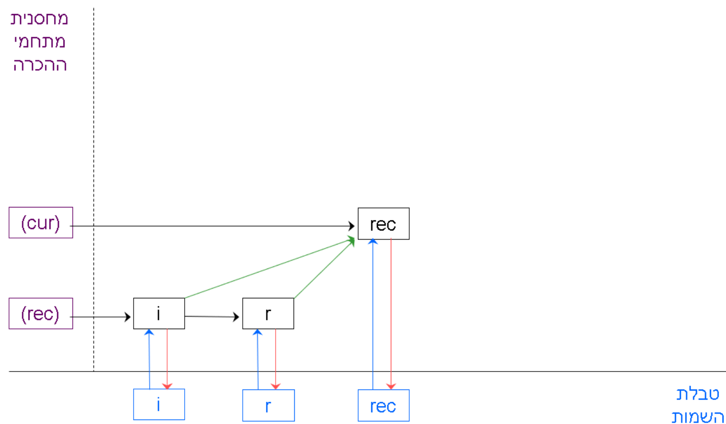
אם
עשינו הכל נכון, אז התשובה די פשוטה – ברגע שאנחנו קוראים למשתנה כלשהו, פשוט נסתכל בטבלת השמות, ונראה לאן מצביעים החיצים, והמשתנה הקרוב ביותר הוא יהיה זה אליו אנחנו מתייחסים. במקרה שלנו לא עשו את זה קשה במיוחד, אבל זה התהליך.

שמות מוסמכים

יש לנו בסוף המצגת דוגמא קטנטנה שממשיכה את טבלת הסמלים באופן טיפה שונה. איך אנחנו מתעסקים עם מחלקות (המכונות record בשפה הרלוונטית עד מאוד שכתבו לנו את התכנית). נתון לנו קטע הקוד הבא –

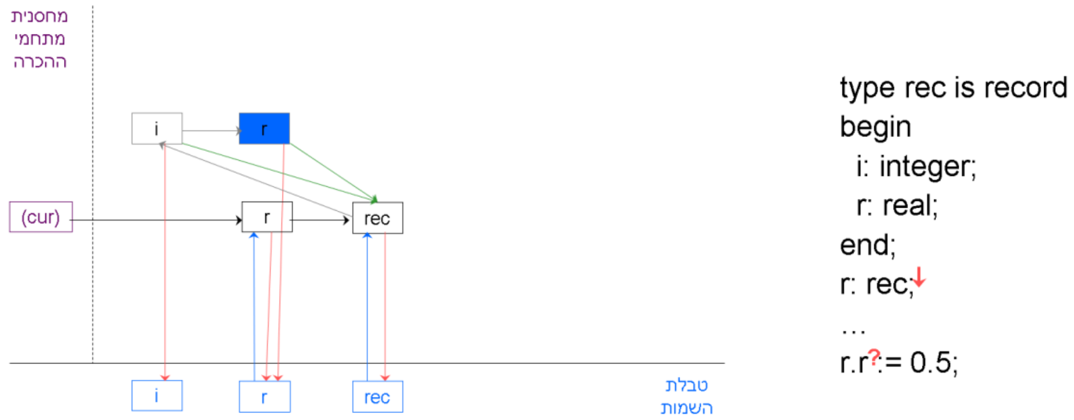
```
type rec is record
begin
        i: integer;
        r: real;
end;
r: rec;
…
r.r := 0.5;
```

אנחנו מגדירים רקורד (רשומה/מחלקה) בשם rec, ובתוכה שני פרמטרים – מספר שלם i ומספר עשרוני r. כאשר נגדיר את הרשוה הזאת, הטבלה תיראה באופן הבא -




```
type rec is record
begin
  i: integer;
  r: real;
end;
r: rec;
…
r.r := 0.5;
```

בתור התחלה, נפתח
לרשומה מתחם הכרה בפני עצמו, ושם יתייחסו המשתנים ישירות אליו, שיוגדר ב-current או ב- main שלא יהיה. ברגע שנסיים את ההגדרה של הרשומה, כמובן שהמתחם הכרה שלו ייעלם כמו שראינו קודם עם הפונקציות השונות. אז איך זה יופיע כשנגדיר מופע של הרשומה בתוך התכנית? (רמז רמז – תרגיל בית 5)



אנחנו מגדירים את הפרמטרים של הרשומה **מעל** אותו מתחם הכרה, ומחכים שיקרה משהו. בשורה האחרונה אנחנו ניגשים ל-r.r ומכניסים לשם את הערך 5.5. העניין הוא, שעד עכשיו אמרנו שאנחנו ניגשים למופע הנמוך ביותר של r ופועלים עליו, אבל ברור לנו שכאן זה לא יעבוד, כי אנחנו צריכים את הפנימי יותר שנמצא דווקא למעלה.

לכן, אנחנו אכן נגיע ל-r הראשון שהוא בוודאי רלוונטי לנו, ונחפש אחר כך מאיזה טיפוס הוא. כאשר נמצא כי הוא מטיפוס rec, אנחנו נחפש אותו ושם נבדוק את הפרמטרים המורכבים בו. כאשר נמצא שם את r, נבין שמדובר באותו אחד שאנחנו רוצים ואליו ניגש על מנת לעשות השמה.

ניתוח משמעות: פתירת העמסה - מצגת דוגמאות מס' 6

בתרגיל זה אנחנו מתמקדים בהיבט נוסף של ניתוח סמנטי – העמסת פונקציות. בהתאם למה שתרשה לנו השפה בחוקים שלה, אנחנו יכולים ליצור מספר פונקציות בעלי אותו שם אך עם פרמטרים שונים. כמובן שדבר כזה עלול ליצור לנו בעיה בהבנת המשמעות של הטיפוסים השונים אליהם אנחנו מתייחסים. אם אנחנו כותבים משפט כמו –

i = f(x)

אנחנו צריכים להיות בטוחים מה יחזור לנו מהפונקציה ומה נכנס אליה, ולפי זה ללכת.

[2]

לצורך תחילת הדיון, אנחנו מציגים מספר פונקציות בשפת Ada, שכולנו מכירים ואוהבים –

```
procedure P (x : integer; y : float;          // P1: integer * float
procedure P (x : float; y : float);           // P2: float * float
procedure P (x : boolean; z : integer);       // P3: boolean * integer
function F (x : integer) return integer;      // F1: integer → integer
function F (x : float)   return integer;      // F2: float → integer
function F (x : integer) return float;        // F3: integer → float
function F (x : float)   return boolean;      // F4: float → boolean
```

לפנינו יש 3 פרוצדורות (פונקציות שלא מחזירות ערך – void) בעלות אותו שם – P. כל פרוצדורה מקבלת ערכים שונים, שימו לב שהערכים שונים לגמרי, ואין שתיים שמקבלות אותם טיפוסים בדיוק. מהעבר השני יש לנו 4 פונקציות, המקבלות כל אחת ערך בודד, אך כאן יש לנו כפיליות – שתי פונקציות מקבלות int ושתיים float. השוני בפונקציות הדומות הוא הערך המוחזר שמשתנה בכל פעם.

לטובת הסימון בהמשך העבודה, יש לנו את החלק בהערות. אנחנו רושמים את שם הפונקציה וממספרים כל אחת, ומעבר לנקודותיים מכניסים קודם כל את הערכים המוכנסים לפונקציה, אם ש יותר מאחד מחברים אותם עם *, ובפונקציות שיש ערך מוחזר מוסיפים את הערך המוחזר אחרי חץ. כך שאם נסתכל על F_3, F_3: integer → float, נבין שאנחנו מכניסים integer, עושים פעולה כלשהי, ומחזירים float.

את הניתוח אנחנו נבצע עבור קטע הקוד הקטנטן הבא –

P(f(3.14), f(1));

שימו לב, ממבט ראשון אנחנו עלולים לחשוב שמדובר על הפרוצדורה P_1, כי יש פה מספר עשרוני ואחריו שלם. אבל אם נעשה את ההנחה הזאת זה טעות! אנחנו מכניסים את המספר הזה לתוך פונקצית f, ויש לנו 2 כאלה שיכולים לקבל כל אחד מהם, וכל גבר כזה יוציא לנו משהו אחר לגמרי. לכן, נעבור בצורה מסודרת ונראה מה אנחנו עושים.

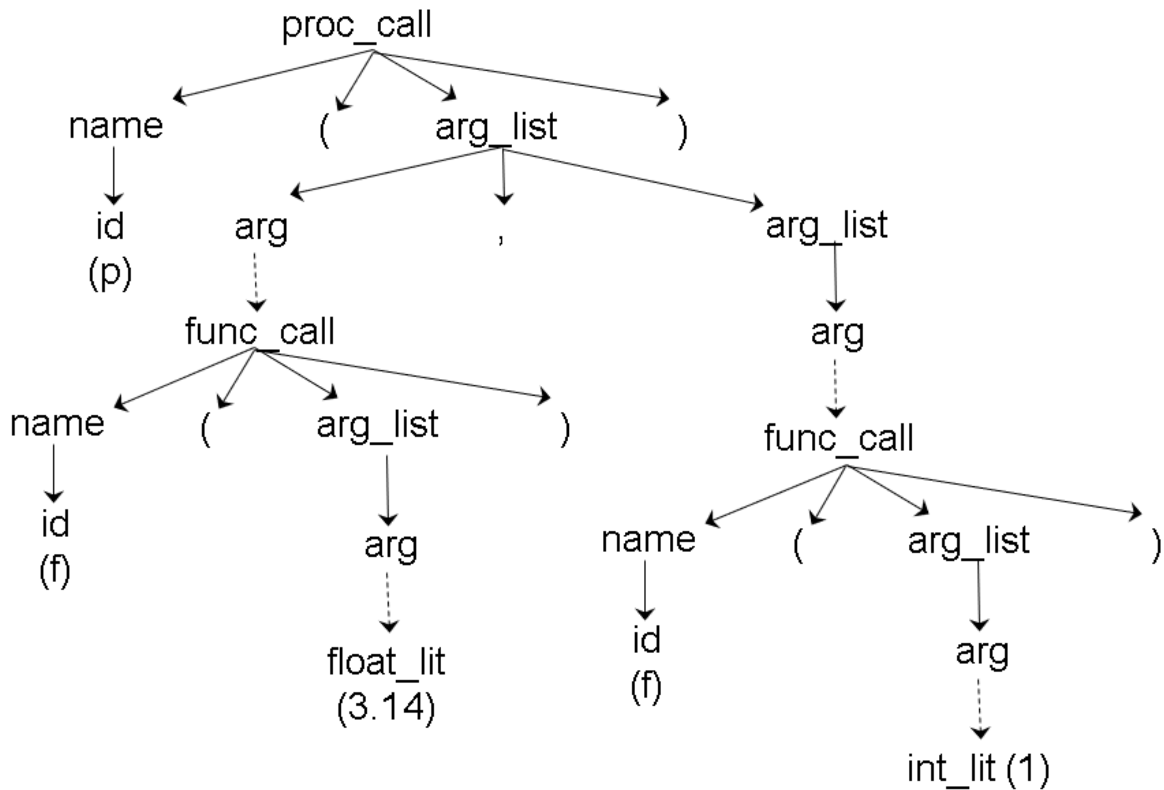
[3]

ראשית נסתכל על כללי הגזירה –

```
proc_call → name "(" arg_list ")"
arg_list → arg "," arg_list | arg
arg → expr
expr → name | func_call | int_lit | float_lit | … (others)
func_call → name "(" arg_list ")"
name → name "." id | id
```

[4]

אנחנו לא רוצים להתעכב פה על בניית העץ, אלא על הניתוח של המשמעות, ולכן ישר נסתכל על העץ המוכן –



החיצים המקווקים שמופיעים באמצע פשוט מקצרים לנו גזירה למשתנה שגוזר ישר הלאה וחוסך לנו מקום, אבל כשאנחנו נידרש לעשות דברים כאלה, אל תקצרו.

[5]

הדרך שנפתור את העץ ואת הביטוי תתחלק לשני חלקים:

- ניתוח – מעבר משורשי העץ כלפי מעלה, וקריאת כל האפשרויות הניתנות לנו תוך כדי המעבר. הרעיון הוא שנוכל להגיע לשורש העץ עם תשובה כמה שיותר חד משמעית לגבי המשמעות של הביטוי, כלומר איזה פונקציות ופרוצדורות אנחנו משתמשים בשביל להגיע לתוצאה הרצויה.
- פתירה – ברגע שנקבל את המידע על הפונקציות, ונכריע מה אנחנו עושים, אנחנו נוריש את כל התכונות הרלוונטיות בחזרה כלפי מטה. דבר נוסף שאנחנו פותרים כאן, הוא מקרה בו יש לנו יותר מאפשרות הכרעה אחת – נגיד ועשינו את כל הסיבוב הזה, והגענו למסקנה, שיש לנו שני מסלולים שנוכל ללכת בהם ולכאורה שניהם ראויים באותה מידה. במקרה כזה, אנחנו פונים לכללי הכרעה שינוסחו מראש, ויחליטו לנו במקרה של בעיות כאלה את סדר העדיפויות בו נשתמש.

לפני שנתחיל את הריצה עצמה, יש לנו שני דברים נוספים שנצטרך להבין במטרה לעשות את הפתירה: תכונות ופונקציות עזר.

[6] תכונות לפתירת העמסה –

יש לנו ארבע תכונות בהם אנחנו משתמשים לכל אורך הניתוח. אך הדבר החשוב ביותר להבין, הוא שאנחנו רצים פה על שני מושגים - **משמעות** – מה הפונקציה אומרת לנו, או בביטוי שאנחנו מכירים יותר מה הערך המוחזר, כלומר לפרוצדורה שאינה מחזירה ערך לא יהיו משמעויות. **הקשרים** – מה הביטויים השונים שאנחנו עובדים איתם, כלומר מה הפרמטרים שיכולים להכנס לתוך הפונקציה. כמובן שגם את זה לא יהיה לנו לכל הפונקציות, מאחר ויש כאלה

שלא מקבלות ערכים. לצורך העניין, ההקשרים של f זה כל סוגי הפרמטרים שיכולים להכנס לאחד מארבע המופעים השונים שלו. אחרי שהבנו את זה, וכדאי להבין את זה, נסתכל על התכונות–

- **sm (Set of Meanings)** – קבוצת המשמעויות של הביטוי כשלעצמו (ללא תלות בהקשר). לפעמים אנחנו נחפש את המשמעות לקבוצה של ביטויים. זה יכול להיות כל ה-f, וזה יכול להיות רק מספר פונקציות מתוך f שנחליט שהם מתאימות – כך או כך, מדובר על קבוצת הערכים האפשריים שיהיו לנו מהקבוצה הנתונה. המשמעויות מתייחסות בדרך כלל ל"מה קורה בפועל".
- **sc (Set of Context)** - קבוצת ההקשרים האפשריים של הביטוי (ללא תלות במשמעותו אלא בסביבתו בלבד). מדובר כמובן בדיוק על אותו רעיון של ה-sm, רק הפוך. בעבור קבוצת פונקציות/פרוצדורות נתונה, אנחנו רוצים לדעת את כל האפשרויות של הערכים שיכולים להכנס על מנת להתחיל בפעולה. ההקשרים מדברים על "מה הפוטנציאל".
- **candidates** – קבוצת הישויות המועמדות. בתחילת הניתוח, אנחנו רושמים לנו את **כל** המועמדים האפשריים. כאן מדובר כמובן, על המועמדים הנתונים לפני שאנחנו מסתכלים על ערכים נכנסים ויוצאים, אלא רק על השם. ראינו f – נכניס את כל הפונקציות f לתוך רשימת המועמדים.
- **Ctype (Context Type)** – אחרי שנסיים את הניתוח עצמו, ונחליט על איזה פונקציות מדובר, נוכל להתחיל להכניס את ערכי טיפוסי ההקשר – כלומר אילו טיפוסים אנחנו מצפים למצוא כאלו הנכנסים לפונקציה.

[7-8] פונקציות עזר

בשני השקפים האלה יש לנו רשימה של פונקציות עזר בהם אנחנו משתמשים בניתוח. תכלס, חלק גדול מהם בכלל לא מופיע בהמשך המצגת, לכן נתעכב רק על הרלוונטיים באמת (אלה שמופיעים במצגת) –

- **set_of_param_lists(S: entity_set)** – קבוצת רשימות הפרמטרים של הישויות (שגרות/פונקציות) שבקבוצה S. אנחנו שולחים קבוצה של פונקציות אפשריות, ומחזירים את קבוצת הפרמטרים, למעשה אנחנו משתמשים בזה על מנת לקבוע את קבוצת ההקשרים (sc).

- **set_of_ret_types(S: entity_set)** – קבוצת טיפוסי הערכים המוחזרים של הישויות (פונקציות) שבקבוצה S. אותו רעיון בדיוק כמו הפונקציה הקודמת, אלא כאן אנחנו מחזירים את המשמעויות האפשריות לתוך sm מסוים, על פי הפונקציות המצומצמות יותר.

- **apply_decision_rules(S: list_set)** – מנסה לצמצם את קבוצת רשימות הפרמטרים לרשימה בודדת על-פי כללי החלטה של השפה. במידה והגענו לסוף הניתוח ואין לנו הכרעה (יש לנו יותר מאפשרות אחת שתהיה נכונה), אנחנו מכניסים את כל האפשרויות, ומה שיוצא לנו זה רק הקבוצה המקיימת את כללי ההכרעה.

- **head(L: list)** – האיבר הראשון ברשימה L. בא לידי שימוש כאשר אנחנו מורישים הקשרים לבנים השונים של פונקציות בשלב הפתירה. אם יש לנו פונקציה עם רשימת פרמטרים נכנסים, אנחנו נעביר לכל צד של אחד הבנים את הפרמטרים לתוך ה-Ctype בעזרת הפונקציה הזאת.

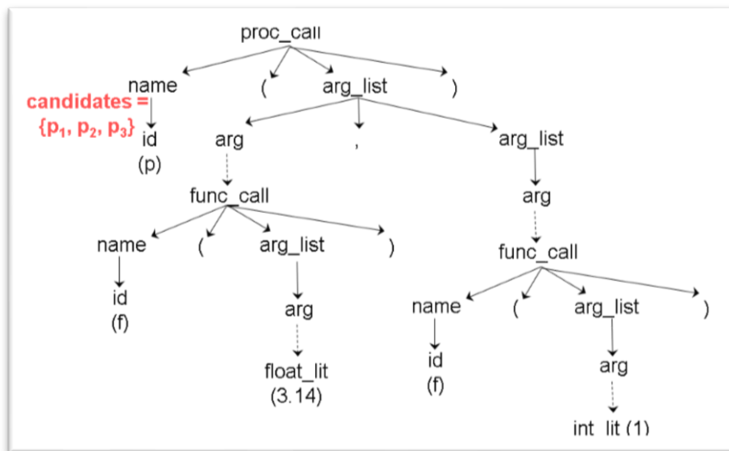
- **tail(L: list)** – הרשימה המתקבלת לאחר הסרת האיבר הראשון של L. בהמשך לפונקציה הקודמת, אנחנו עושים כאן חלוקה של "ראש וזנב", כאשר כל מה שלא יהיה האיבר הראשון נכנס לזנב. ברגע שיש לנו רשמת פרמטרים שנכנסים לתוך פונקציה/פרוצדורה, אז בדרך כלל אנחנו נגזור לצורה של arg_list_1, arg, arg_list ← arg_list, ואז arg יקבל את הראש, ו-arg_list_1 יקבל את הזנה, שבמידה ויהיו שם יותר מפרמטר אחד ימשיך את אותה פעולה עד שנסיים "לחלק" את הפרמטרים.

- **elem(S: set)** - כאשר |S| = 1, האיבר (היחיד) שמכילה הקבוצה S. אם ידוע לנו שעשינו חיתוכים והכרעות, והגענו לנקודה בה יש לנו איבר בודד בקבוצה, אנחנו יכולים פשוט לקרוא לו באופן הזה.

- **innermost_homonyms(N: NT_entry)** – מחזירה את קבוצת ההומונימים הפנימיים ביותר, כלומר התקפים, המוצבעים מן הכניסה N של טבלת השמות. כלומר, אם יש לנו קבוצה של פונקציות בעלי אותו שם, הפונקציה תבדוק את כל האפשרויות הנתונות, ותחזיר לנו את כולם. אין כאן התחשבות במשמעויות והקשרים, אלא פשוט מילוי של כל המועמדים לתכונת candidates.
- **param_type(T: type_expression)** – חלק הפרמטרים של ביטוי-הטיפוס T. לטובת ניתוח המידע, אנחנו לפעמים נבקש את הטיפוס של הפרמטרים הנכנסים לפונקציה מסוימת. כדאי לשים לב, לא מדובר פה על קבוצה, אלא על פונקציה בודדת בכל פעם.
- **ret_type(T: type_expression)** – החלק של טיפוס הערך המוחזר מתוך ביטוי-הטיפוס T. אותו דבר כמו הפונקציה הקודמת – אנחנו מחזירים עבור פונקציה מסוימת מה הטיפוס אותו היא מחזירה.

ואחרי כל זה, ניתן להתחיל בניתוח עצמו –

[9-42]



קודם כל, אנחנו יורדים לבן השמאלי ביותר, ובודקים את המועמדים שלו. מבחינת קטע הקוד, אנחנו משתמשים בפונקציה שתביא לנו את כל ההומונימים האפשריים:

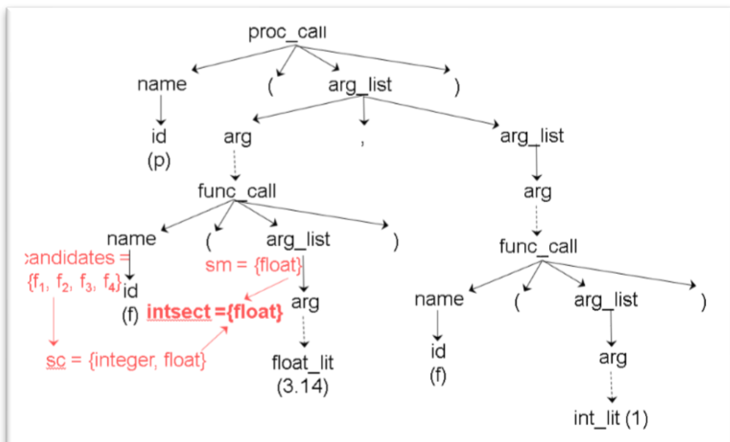
name.candidates :=innermost_homonyms(id.entry)

לאחר מכן, אנחנו ממשיכים הלאה, ועושים את אותה פעולה גם עבור ה-f שנמצא בהמשך. לאחר מכן אנחנו רואים שהערך arg מקבל לתוכו float_lit ליטרל שהוא עשרוני, מה שיכול להביא לנו כבר איזה סוג של צמצום אפשרויות. אנחנו מעדכנים את התכונות –

expr.sm := {float}
arg.sm := expr.sm
arg_list.sm := arg.sm

עכשיו יש לנו כבר משמעות כלשהי, הביטוי "מחזיר" (מוגדר) כ-float, ולכן אנחנו יודעים שגם arg יחזיר את אותו טיפוס, שיעלה בתורו גם ל-arg_list.

עכשיו אנחנו עושים כאן כמה פעולות, שיוכלו לצמצם לנו את הביטוי f. קודם כל, אנחנו צריכים למצוא את ההקשרים האפשריים לביטוי, לכן נפעיל את הפונקציה –

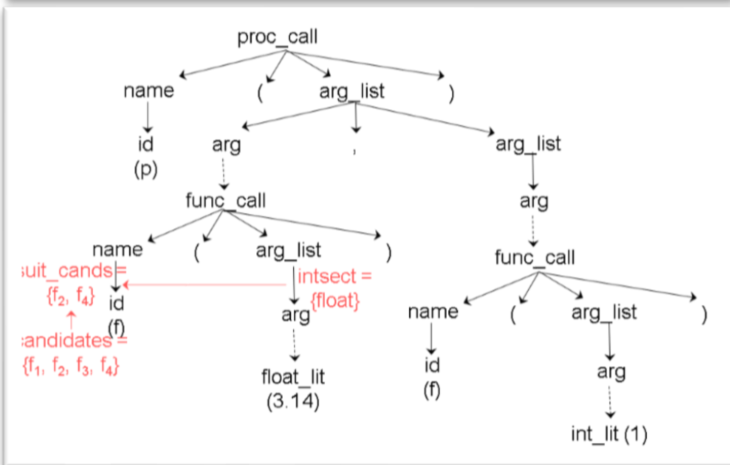


arg_list.sc :=
set_of_param_lists(name.candidates)

לאחר שקיבלנו את ההקשרים, ויש לנו את המשמעויות האפשריות, אנחנו עושים חיתוך בין שתי הקבוצות ובודקים מה יצא לנו –

arg_list.intsect := arg_list.sm ∩ arg_list.sc

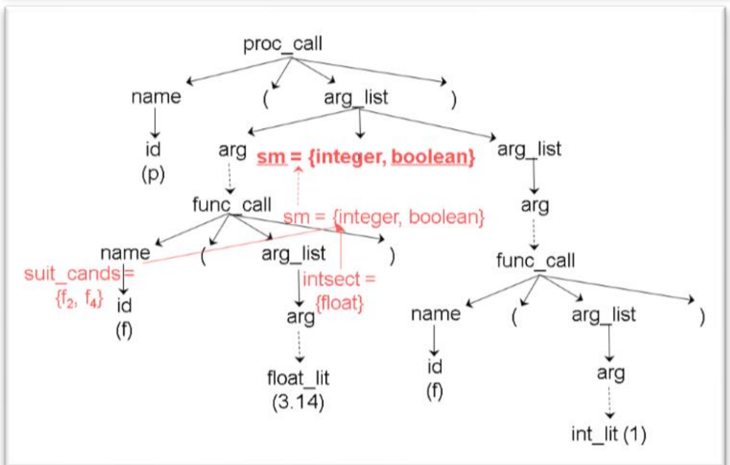
כמובן שהחיתוך בין שתי הקבוצות יותיר אותנו עם float, מה שיוכל להוביל אותנו לצמצום האפשרויות למועמדים.



name.suit_cands :=
{c ∈ name.candidates |
param_type(c.type) ⊆ arg_list.intsect}

אנחנו לוקחים את כל המועמדים האפשריים, ובודקים עבור כל אחד מהם (במין לולאת foreach), האם הפרמטרים שהוא מקבל מתאים לפרמטרים שיצאו לנו בחיתוך. כל פונקציה שעומדת בזה, תכנס לתוך התכונה suit_cands.

עכשיו אחרי שהכרענו מה הפונקציות הראויות לבוא בקהל, נוכל להכריע עבור הקריאה לפונקציה את קבוצת המשמעויות שלה –



func_call.sm :=
set_of_ret_types(name.suit_cands)
expr.sm := func_call.sm
arg.sm := expr.sm

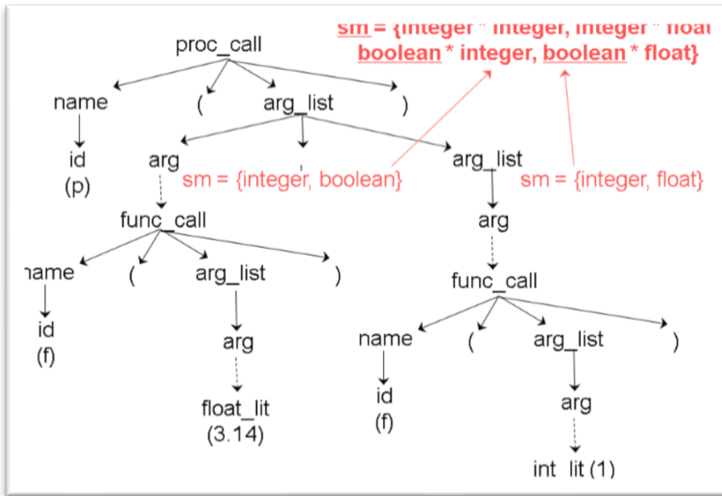
שימו לב, שאת ההשמה של התכונה sm אנחנו עושים במשתנה הראשי של הפונקציה, ולא בשם או משהו כזה. כאן אנחנו לוקחים את שתי הפונקציות f_2,f_4 ואנחנו בודקים עבורם את קבוצת המשמעויות האפשריות.

לאחר שהבאנו את המשמעויות למשתנה מסוים, אנחנו מעלים את האפשרויות מעלה בעץ, עד שנגיע למקום האחרון הרלוונטי.

סתם הערה קטנה: כדאי לזכור שאנחנו מדברים פה על קבוצות מתמטיות, כלומר אין פה כפילויות. גם אם יהיו לנו 300 פונקציות שיחזירו א אותו ערך, ב-sm אנחנו מכניסים רק מופע אחד.

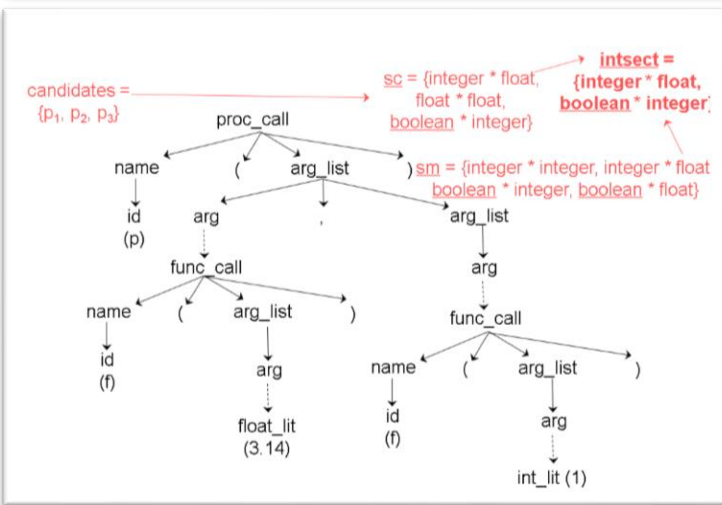
עכשיו אנחנו עושים כאן דילוג קטן בשקפים, מאחר והחלק השמאלי של העץ עושה בדיוק את אותם פעולות ואין צורך להתעכב על כל הסיפור הזה מחדש, אלא רק לראות את התוצאה הסופית.

לאחר שעלינו והכרענו את arg_list.sm, אנחנו צריכים להעלות את המידע הזה לאב המשותף שלהם. למעשה יש לנו כאן פרוצדורה מסוימת, לה יש שני פרמטרים, אך לכל אחד מהפרמטרים יש לנו שתי אפשרויות. אם היתה לנו



אפשרות אחת בלבד, הכל היה יותר נוח (ולמעשה יש לנו משהו דומה בחלק הימני שלא הראיתי – arg_list מקבל שני ארגומנטים אפשריים, ומבחינתו אין לו בעיה להעלות את שניהם ומישהו אחר יכריע). אבל כאן אנחנו צריכים לבוא לידי הכרעה. לכן, קודם כל נראה את כל האפשרויות שיש לנו – אנחנו עושים מכפלה קרטזית בין שני הצדדים האפשריים, ומכניסים בתור התחלה את כל האפשרויות של ההכפלה לתוך arg_list.sm –

arg_list.sm := arg.sm × arg_list_1.sm

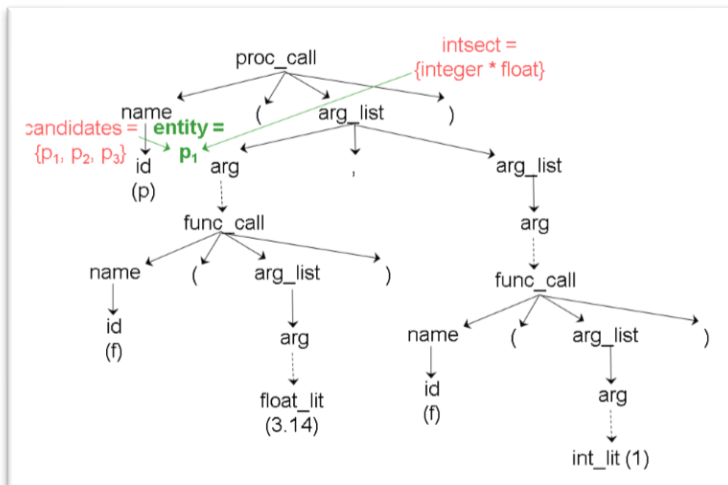


עכשיו אנחנו לוקחים את ה-sc של כל הפרוצדורות האפשריות, שזה ייתן לנו את האפשרויות של הערכים שייכנסו אליהם, ועושים חיתוך עם ה-sm שהוא מה שראינו שיכול בפועל להיות לנו בהינתן קטע הקוד –

arg_list.sc :=
set_of_param_lists(name.candidates)
arg_list.intsect :=
arg_list.sm ∩ arg_list.sc

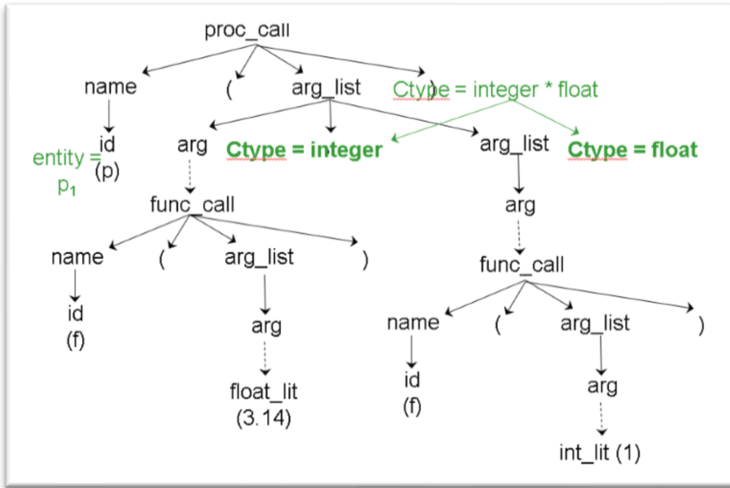
עכשיו, בעולם אופטימלי כאן היינו יכולים להכריע – יש לנו את כל האפשרויות הפוטנציאליות והאפשרויות בפועל, והשאיפה שלנו היא שזה יתאים ויוציא לנו בחיתוך רק ערך אחד. אך לצערנו, אנחנו חיים בעלמא דשיקרא והחיתוך הביא לנו שתי אפשרויות. מה עכשיו? עכשיו אנחנו פונים לכללי ההכרעה שצצו לנו משום-מקום, ורואים שכתוב שם שיש לנו העדפה לפרמטרים מספריים. כלומר, אנחנו נעדיף את האפשרות הראשונה שמכניסה שני ערכים מספריים מסוג כלשהו, על פני האפשרות השנייה שם יש לנו ערך בוליאני.

למעשה, אנחנו כבר עכשיו בשלב הפתירה, על אף שיש לנו כאן מעין שלב ביניים שמסיים את חלק הניתוח. מאחר שאנחנו מדברים על פונקציה שמקבלת מספרים, אנחנו יכולים להכריע שמבין כל האפשרויות, הישות הכי מתאימה היא P_1.



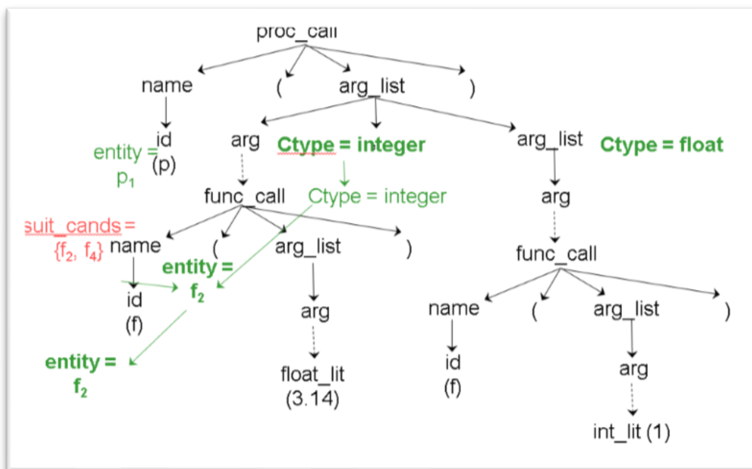
name.entity :=
c ∈ name.candidates |
c.type = elem(arg_list.intsect)

אנחנו כאן מורישים ל-name את היישות המתאימה לפי ההכרעה שהבאנו – אנחנו מציינים שמדובר במועמד הבודד שנמצא ב-candidates ומתאים לאיבר הבודד ברשימת החיתוך של atg_list (שבמקרה שלנו מדובר על איבר בודד רק לאחר שהרצנו עליו פונקציה שתביא לנו הכרעה).



עכשיו, אנחנו מורישים את שם היישות גם ל-id מצד שמאל, ומתחילים להוריש את האפשרויות השונות לחלק הימני. ראשית, אנחנו מכריעים את ה-CType – ההקשר שאנחנו החלטנו שהוא הנכון והסופי יוגדר ב-atg_list, וכל אחד מהבנים שלו יקבל בהתאמה את ה-CType המתאים לו –

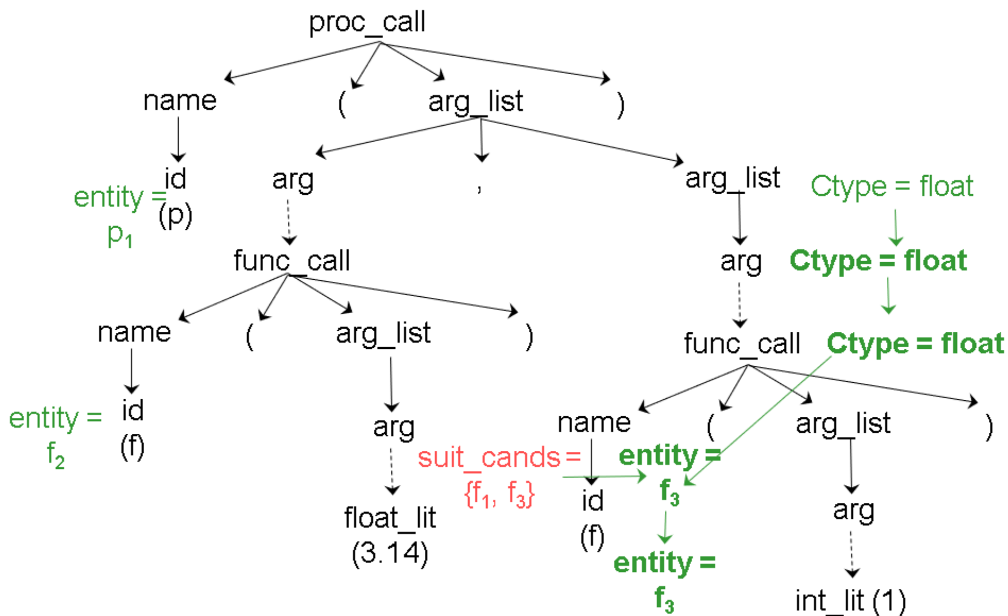
arg_list.Ctype := elem(arg_list.intsect)
arg.Ctype := head(arg_list.Ctype)
arg_list$_1$.Ctype := tail(arg_list.Ctype)



עכשיו אנחנו מעבירים את ה-Ctype למטה ב-func_call, ובעזרת המידע שכבר יש לנו אנחנו יכולים להכריע גם בצד השמאלי – אנחנו לוקחים את ה-suit_cands ובודקים אילו ערכים מוחזרים מהפונקציות האפשריות יכול להתאים ל-Ctype שקבענו -

expr.Ctype := arg.Ctype
func_call.Ctype := expr.Ctype
name.entity :=
c ∈ name.suit_cands |
ret_type(c.type) = func_call.Ctype

שוב, אנחנו לא נראה את כל הדרך לחלק הימני יותר כי הוא עובד בדיוק באותו אופן, אלא רק נראה את התוצאה הסופית –



[43-47]

אחרי כל הכיף הזה, נשאר לנו להבין את הפעולות הסמנטיות שנצרך לכל הסיפור הזה. הכללים הסמנטיים בעצם צריכים להתחשב בשני השלבים – הן הניתוח והן הפתירה. נעבור על הכללים שראינו למעלה, ונראה מה נצרך להם –

<table>
<tr><th>פעולות סמנטיות</th><th>כלל הגזירה</th></tr>
<tr><td></td><td></td></tr>
<tr><td>if name.candidates does not contain only procedures then error
arg_list.sc := set_of_param_lists(name.candidates)
arg_list.intsect := arg_list.sm $\cap$ arg_list.sc
if |arg_list.intsect| > 1 then
apply_decision_rules(arg_list.intsect)
if |arg_list.intsect| $\neq$ 1 then error
name.entity := c $\in$ name.candidates |
c.type = elem(arg_list.intsect)
arg_list.Ctype := elem(arg_list.intsect)</td><td>proc_call $\rightarrow$ name “(“ arg_list “)”</td></tr>
<tr><td colspan="2">כאן אנחנו מתייחסים רק לקריאה לפרוצדורה. בכל הפעולות שנעשה, נוכל לראות כי אנחנו קודם כל מכניסים את הניתוח, ולאחרי זה אנחנו מכניסים את הפתירה. כאן אנחנו קודם כל מעכנים את המועמדים האפשריים על פי שם הפרוצדורה. הפעולה הבאה (החיתוך) היא למעשה כבר החיבור בין הניתוח לפתירה של הגזירה, ולאחריה אנחנו בודקים אם עלינו להחיל את כללי ההכרעה, וארי זה לוודא שלא הגענו למצב בו אין לנו הכרעה מוחלטת – אם החיתוך עצמו לא הביא לנו אפס אפשרויות, או אם כללי ההכרעה גם הם לא הספיקו לנו להכריע. לאחר מכן, אנחנו מכניסים את היישות המתאימה לפי ההכרעה ומעבירים בהתאם את ה-Ctype ל-arg_list משם הוא יכול להמשיך הלאה.</td></tr>
<tr><td>arg_list.sm := arg.sm $\times$ arg_list$_1$.sm
arg.Ctype := head(arg_list.Ctype)
arg_list$_1$.Ctype := tail(arg_list.Ctype)</td><td>arg_list $\rightarrow$ arg “,” arg_list$_1$</td></tr>
<tr><td colspan="2">כאן אנחנו אוספים את כל האפשרויות למשמעויות על ידי מכפלה קרטזית.

מבחינת הפתירה – אנחנו מעבירים לערך הבודד את ראש הרשימה, ולרשימה הפנימית את זנב הרשימה ששם יטפלו בשאר ההשמות</td></tr>
<tr><td>arg_list.sm := arg.sm
arg.Ctype := arg_list.Ctype</td><td>arg_list $\rightarrow$ arg</td></tr>
<tr><td colspan="2">כאן יש רק העברת מידע פשוטה מצד לצד (אפשר לשים לב לכיוונים של הניתוח מלמטה למעלה והפתירה מלמעלה למטה)</td></tr>
<tr><td>arg.sm := expr.sm
expr.Ctype := arg.Ctype</td><td>arg $\rightarrow$ expr</td></tr>
<tr><td colspan="2">כנ"ל – העברות מידע.</td></tr>
<tr><td>expr.sm := name.sm
name.entity :=
c $\in$ name.candidates |
c.type = expr.Ctype</td><td>expr $\rightarrow$ name</td></tr>
<tr><td colspan="2">בזמן הפתירה, אנחנו יכולים להכריע לגבי היישות שנמצאת תחת ה-name, על ידי שנבדוק מי מבין המועמדים מתאים ל-Ctype הנורש.</td></tr>
<tr><td>expr.sm := func_call.sm
func_call.Ctype := expr.Ctype</td><td>expr $\rightarrow$ func_call</td></tr>
<tr><td colspan="2">העברת מידע פשוטה.</td></tr>
</table>

<table>
<tr><td>expr → int_lit</td><td>expr.sm := {integer}</td></tr>
<tr><td colspan="2">כאן אנחנו נהיה למעשה בתחתית העץ, ונוכל ליצור את המשמעות הראשונה שתעבור על פי הגזירה בעצמה ש"תכריע" לנו את הטיפוס שאנחנו עובדים עליו.</td></tr>
<tr><td>expr → float_lit</td><td>expr.sm := {float}</td></tr>
<tr><td colspan="2">כנ"ל.</td></tr>
<tr><td>name → $name_1$ “.” Id</td><td>e_1 := $name_1$.entity
name.candidates := {e | is_record(e_1) ∧
is_local_entity(id.entry, e_1) ∧
e = local_entity(id.entry, e_1)}
name.sm := {t | c ∈ name.candidates, t = c.type}
if |name.candidates| = 1 then
name.entity := elem(name.candidates)
id.entity := name.entity</td></tr>
<tr><td colspan="2">בגזירה הזאת לא השתמשנו בדוגמא, אבל אנחנו מדברים כאן על גישה לתוך תכונה של אובייקט. לא רלוונטי.</td></tr>
<tr><td>name → id</td><td>name.candidates := innermost_homonyms(id.entry)
name.sm := {t | c ∈ name.candidates, t = c.type}
if |name.candidates| = 1 etc.
id.entity := name.entity</td></tr>
<tr><td colspan="2">בשלב הניתוח, כאשר אנחנו מזהים גזירה ל-id אנחנו עדיין לא יודעים במי מדובר, ולכן אנחנו מכניסים לרשימת המועמדים של name את כל האפשרויות הניתנות מאותו id.

בשלב הפתירה – אנחנו מעבירים ל-id את ההכרעה לגבי היישות המתאימה שנבדקה ב-name.</td></tr>
<tr><td>func_call → name “(“ arg_list “)”</td><td>if name.candidates does not contain only functions then error
arg_list.sc := set_of_param_lists(name.candidates)
arg_list.intsect := arg_list.sm ∩ arg_list.sc
if |arg_list.intsect| = 0 then error
name.suit_cand :=
{c ∈ name.candidates | param_type(c.type) ⊆
arg_list.intsect}
func_call.sm := set_of_ret_types(name.suit_cands)
name.entity :=
c ∈ name.suit_cands |
ret_type(c.type) = func_call.Ctype
arg_list.Ctype := param_list(name.entity.type)</td></tr>
<tr><td colspan="2">כאן מדובר על קריאה לפונקציה (שאינה פרוצדורה) אך הפעולה שלה דומה מאוד לנעשה קודם. ע"ש.</td></tr>
</table>

טיפול בהפניות בקרה

ראינו בצורה כללית, איך אנחנו מתייחסים לקפיצות מותנות – שתילה של תויות, הכנסת תנאי הקפיצה והתייחסות למעבר להמשך הקוד. עכשיו אנחנו נראה באופן פורט איך אנחנו מכניסים את הפניות הבקרה לשלושה סוגים שנים של קפיצות, ונתייחס להבדלים הדקים בין כל אחד ואחד.

אנחנו נדבר כרגע על הקפיצות הבאות –

$S \rightarrow$ if B then S_1 |
if B then S_1 else S_2 |
while B do S_1

נשים לב – עבור המשפט הראשון – יש לנו קפיצה אחת, במידה והתנאי לא מתקיים, אנחנו מדלגים על S_1. אם התנאי מתקיים, אנחנו רצים על כל הקוד. במשפט השני – יש לנו שני קפיצות, כאשר אחת מהן היא וודאית. אם B מתקיים, אנחנו עושים את S_1 ומדלגים מעל S_2, ואם לא אז להיפך – מדלגים על S_1 ומבצעים את S_2. במשפט האחרון יש לנו גם שתי קפיצות – בסוף הקטע S_1 אנחנו קופצים לראש התנאי, וברגע שהוא לא יתקיים, אנחנו נדלג מעל S_1.

לצורך המשך הדיון, רק נאמר כי אנחנו נשתמש במושגים של נפילה (fall), המציינים כי בעבור קיום/אי קיום של תנאי אנחנו עוברים לשורה הבאה בקוד השייכת למקטע אחר. לצורך הדוגמה – אם יש לנו תנאי if רגיל, והוא מתקיים, המעבר לקטע הקוד שלו נעשה על ידי נפילה.

כמובן, שאנחנו נעדיף גם פה את החישוב העצלני, ולא נכניס את הקוד "בשלמותו". כלומר, כאשר אנחנו נחשב את B, ברגע שנראה איזו תוצאה שיכולה לקבוע לנו האם עלינו לקפוץ או לא, ישר נבצע את הקפיצה הרצויה. הדרך לעשות זאת הוא ליצור שתי "תכונות" חדשות לכל קטע קוד של B (תנאי) בשם B.true ו-B.false. את התויות האלה אנחנו נציב בתור S.next (תוית בקטע הקוד, אותו נשרשר לאחר S) או תוית S.begin שניצור לתחילת הקטע S. באופן כזה, כל תנאי ידע בוודאות לאן עליו לקפוץ ברגע שהוא מגלה תוצאה.

לפני שנמשיך, נדגים רק את השרשור של התוויות התחלה וסוף –

$P \rightarrow S$	S.next = newlabel() ; P.code = S.code || label(S.next) ;
$S \rightarrow S_1\ S_2$	S_1.next = newlabel() ; S_2.next = S.next; S.code = S_1.code || gen(S_1.next ':') || S_2.code

לפני שאנחנו מתחילים בכלל להתעסק עם הקוד, אנחנו מגדירים תוית "חדשה", אך עדיין לא מציבים בה ערך. את ערך הקוד של משתנה הגזירה אנחנו מגדירים באופן שיכניס קודם את הקוד של S, ובסופו יכניס תוית שהיא S.next שהגדרנו לפני כן. ככה אנחנו יכולים לדעת שאם עלינו לדלג מעל S נוכל לקפוץ לתוית הזאת.

במידה ויש לנו שני קטעי קוד (ונשתמש ברעיון הזה גם בשביל ההתעסקות עם התנאי), אנחנו מגדירים תוית next עבור כל אחד מהקטעים – עבור הקטע הראשון, אנחנו לא מציבים בו כלום, כי אנחנו עדיין לא יודעים איפה זה יהיה, אך עבור הקטע השני, אנחנו יודעים לומר שברגע שהוא יסתיים, יסתיים גם כל הקוד של S שגזר אליו, ולכן אנחנו כבר יכולים להגדיר שה-next של S ו-S_2 הוא אותו אחד. לאחר מכן, אנחנו יכולים להגדיר את שרשור הקוד – מכניסים את התכונה של הקוד של כל אחד מהבנים, וביניהם מכניסים את התוית S_1.next שיהווה יעד קפיצה במקרה הצורך.

ולאחר כל ההקדמה הזאת, נוכל להתעסק עם המשפטי תנאי שראינו קודם –

<u>If B then S</u>

$S \rightarrow$ if B then S1	{ B.true := newlabel() ; B.false := S.next ;

	S1.next := S.next ; S.code := B.code || gen (B.true ' : ') || S1.code }

אנחנו קודם כל מגדירים עבור B את שתי התויות המתאימות לאמת ושקר. תוית השקר ידועה לנו כבר – אנחנו אמורים לדלג מעל כל המשך הקוד, ולכן נקפוץ ישר ל-S.next. באותו אופן גם נגדיר את S_1.next לאותה תוית ל S.next, שזה בעצם "נפילה" ולהמשיך את השורה הבאה. נשים לב – B.true עדיין לא הושם.

עכשיו אנחנו מתחילים לעבוד על הקוד – מכניסים את התנאי וקטע הקוד, וביניהם את התוית עבור הקפיצה של B.true.

הערה: B.false ו- S_1.next הן תכונות נורשות – הערך המוצב בהם הוא של האבא. אך S.code היא תכונה נוצרת – ולכן גם B.true תחשב כנוצרת.

<u>If B then S_1 else S_2</u>

S → <u>if</u> B <u>then</u> S_1 <u>else</u> S_2	{ B.true := newlabel (); B.false := newlabel (); S_1.next := S.next ; S_2.next := S.next ; S.code := B.code || gen (B.true ' : ') || S_1.code || gen (' goto ' S.next) || gen (B.false ' : ') || S_2.code }

נתחיל דווקא בשורות 3-4. אנחנו יודעים שיהיה מה שיהיה, אנחנו מבצעים רק קטע קוד אחד, ולכן אנחנו יודעים להגדיר שה-next של כל אחד מקטעי הקוד של S, הוא נורש ומוגדר להיות זה של האבא.

קטע הקוד של S הוא נוצר, ותוך כדי שאנחנו יוצרים אותו אנחנו גם עושים שתילה של התוויות true ו-false של B, ולכן אנחנו מגדירים גם אותם כתכונות נורשות, על אף שאם נתייחס רק לתכונה S.code אנחנו נוכל לור שהיא תכונה נוצרת.

מעבר לזה, נסתכל על אופן יצירת הקוד – ראשית, מכניסים את הקוד של התנאי B. אם התנאי מתקיים אנחנו עוברים לקטע הקוד הראשון, ולכן נשים לפניו את התכונה B.true, אבל אנחנו לא מסתפקים בזה – ברגע שנסיים את S_1 יש לנו קפיצה בלתי מותנית, מאחר ואנחנו יודעים שאין לנו צורך להתעכב על המשך הקוד, ולכן נקפוץ ישר ל-S.next. עכשיו אנחנו עוברים לקטע S_2, לפניו אנחנו מגדירים את התוית B.false אליה נקפוץ במידה והתנאי לא יתקיים, ואז משרשרים הכל וממשיכים הלאה.

את While כבר חישבנו קודם בחלק של סוגי המשפטים בקוד ביניים.

ביטויים בוליאניים בייצוג מספרי

כשאנחנו כותבים קוד, ומופיע לנו תנאי בוליאני, אנחנו דואגים לקפיצות שלו כמו שראינו כבר קודם עם ה-while. אבל כמו שהופיע לנו כבר שם, יכול להיות שיהיה לנו תנאי מורכב – כזה שיכיל גם or ו-and. כמובן שזה לא אמור להרתיע אותנו, אנחנו נשתלט על כל קטע בנפרד, ובסוף נחשב את הכל. זה כמובן דרך שהיא קצת נאיבית לטיפול בתנאים. נראה את המימוש שלה, ואיך אנחנו יכולים לעשות את זה באופן קל יותר.

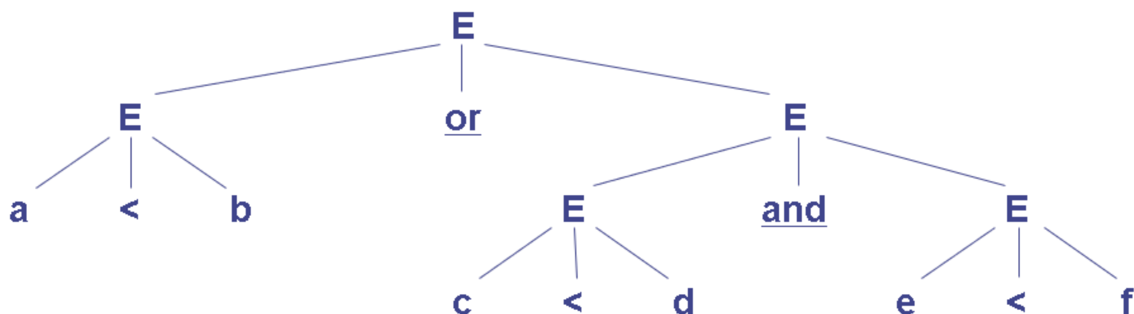
אנחנו נתייחס כרגע לקלט הבא –

(a<b) **OR** ((c<d) **AND** (e<f))

תחת כללי הגזירה והכללים הסמנטיים הבאים –

E ⟶ E_1 <u>or</u> E_2	{ E.var := newtemp() ; emit (E.var ' := ' E_1.var ' or ' E_2.var) }
E ⟶ E_1 <u>and</u> E_2	{ E.var := newtemp() ; emit (E.var ' := ' E_1.var ' and ' E_2.var) }
E ⟶ <u>not</u> E_1	{ E.var := newtemp() ; emit (E.var ' := ' ' not ' E_1.var) }
E ⟶ (E_1)	{ E.var := E_1.var }
E ⟶ <u>true</u>	{ E.var := newtemp() ; emit (E.var ' := ' ' 1 ') }
E ⟶ <u>false</u>	{ E.var := newtemp() ; emit (E.var ' := ' ' 0 ') }
E ⟶ <u>id</u>$_1$ <u>relop</u> <u>id</u>$_2$	{ E.var := newtemp() ; (נחפף את lookup) emit (' if ' <u>id</u>$_1$.var <u>relop</u>.op <u>id</u>$_2$.var ' goto ' nextstat + 2) ; emit (E.var ' := ' ' 0 ') ; emit (' goto ' nextstat + 1) ; emit (E.var ' := ' ' 1 ') }

קודם כל נתחיל בעץ הגזירה עצמו –



עד כאן אין חידוש. עכשיו נעבור חלק חלק, ונוסיף את הקוד הרלוונטי אליו, על פי כלל הגזירה של האופרטור הרלציוני. איך ניישם את החלק של האופרטור? אנחנו ממספרים את השורות, ואנחנו יודעים שכל קטע הקוד הזה אמור לקחת מספר שורות מוגדר – ולכן אנחנו נבדוק את התנאי, אם הוא מתקיים נקפוץ לשורה הבאה+2 (כדאי לזכור, שאנחנו בודקים פה עבור true), ושם נציב את הערך היוצא ב-t_1 להיות 1. אחרת, אנחנו ממשיכים לשורה הבאה, שם אנחנו מציבים 0, ואז קופצים ל-E.end כלומר לקטע הבא. אם כ, נרשום את הקוד שהתקבל לנו –

```
100.    if a<b goto 103
101.    T1 = 0
102.    goto 104
103.    T1 = 1
104.    if c<d goto 107
105.    T2 = 0
106.    goto 108
107.    T2 = 1
```

```
108.	if e<f goto 111
109.	T3 = 0
110.	goto 112
111.	T3 = 1
```

נשים לב שאם התנאי מתקיים, אנחנו רק קופצים להשמה, אבל אחר כך אנחנו ממשיכים פשוט לשורה הבאה. את הדבר הזה אנחנו נכנה בהמשך fall – נפילה לשורה הבאה.

עכשיו נשאר לנו לטפל בתנאים של התוצאות. קודם כל נחשב את ה-and, שמתייחס לשני קטעי הקוד האחרונים, ואז את ה-or בינו לבין הקטע הראשון –

```
112.	T4 = T2 and T3
113.	T5 = T1 or T4
```

האם היינו יכולים ליצור את כל הקוד הזה קצר יותר? בוודאי. נסתכל רגע על המהות של האופרטורים – and יתקיים לנו במידה ואחד מכל המשתנים יתברר כאמת – אם יש לנו רשימה של מיליון משתנים עם OR ביניהם, ברגע שיש לנו ערך אמת ראשון, התוצאה תהיה true ואין לנו מה להמשיך ולבדוק. באותו אופן, אם יש לנו שרשור של משתנים עם and ביניהם, אנחנו חייבים שכולם יהיו בעלי ערך אמת, כך שאם יש לנו false אחד, אנחנו יודעים שאין לנו מה להמשיך לבדוק.

איך זה מתקשר אלינו?

נתייחס לשלושת הקטעים בתור A,B,C. אם בקטע A יהיה לנו ערך true, אז ה-or שמקשר את כל שאר החלקים יצא בוודאות גם כן true. כלומר, אין לנו צורך להמשיך ולקרוא את הקוד! אנחנו יכולים לשנות את הקפיצה ישר לסוף הקוד ולהיות מבסוטים. כנ"ל לגבי ה-and – אם קטע B יהיה false, אנחנו לא צריכים לקרוא את C, נוכל ישר להגדיר false ולהמשיך הלאה.

לחישוב בוליאני מסוג כזה קוראים lazy evaluation (חישוב עצלני), ומכיוון שאחת התכונות החשובות למתכנתים היא עצלנות, אנחנו אוהבים את החישוב הזה. אבל עד כמה הוא תורם לנו? נסתכל עכשיו על הקוד לאחר כל האופטימיזציה העצלנית –

```
100:	if a < b goto 105
101:	if !(c < d) goto 103
102:	if e < f goto 105
103:	T := 0
104:	goto 106
105:	T := 1
```

סגרנו הכל בשש שורות – אנחנו רק צריכים לדאוג אם לקפוץ להשמה של 0 או של 1, ולהמשיך הלאה.

כמה שאלות שחשוב שנשאל (או ליתר דיוק, שאלות שמופיעות במצגת) –

האם החישוב המקוצר שקול לחישוב הרגיל?

לא. יכול להיות שאנחנו נכניס כל מיני side-effect של הדפסות או כל מיני פעולות שונות שבחישוב מקוצר לא יהיה לנו איך להכניס.

מתי אסור להשתמש בחישוב מקוצר?

כשאסור. תשובה די אידיוטית, אבל יש שפות שלא מרשות שימוש בקיצור תנאים.

מתי חייבים להשתמש בחישוב מקוצר?

מתי שהתנאים האם כאלה שמבצעים פעולות בתוך התנאי עצמו, כמו למשל

```
if ( (file=open(“c:\grades”) or (die) ) printfile(file);
```

אם הקובץ המבוקש לא פתוח, אז אנחנו הורגים את התוכנית, אבל אם כן, אנחנו לא רוצים שהוא ימשיך לקרוא את המשפט אלא ישר ידלג להדפסה של אותו קובץ. למעשה, כתיבת הקוד בצורה כזאת מתבססת על העובדה שאנחנו משתמשים בחישוב מקוצר ולכן אין לנו ברירה אלא לעמוד בציפיות. יש לציין, רוב השפות היום משתמשות בחישוב מקוצר, אז פתרו לנו את הבעיה הזאת.

הפקת קוד ביניים - מצגת דוגמאות מס' 7

התרגיל הזה מתמקד בייצוג קוד ביניים במשמעויות של הפניות בקרה, מה שראינו לפני לא הרבה זמן, ועכשיו ננסה להסתכל על זה באופן יותר מפורט בהתחשב גם בגישת ה-Lazy Evaluation. ראשית נסתכל על חוקי הגזירה איתם נעבוד –

S → if (B) S |
  if (B) S else S |
  while (B) S |
  do S while (B);
B → B || C | C
C → C && R | R
R → E relop E | N
N → ! N | F | (B)
F → A | true | false
A → A . id | A [E] | id

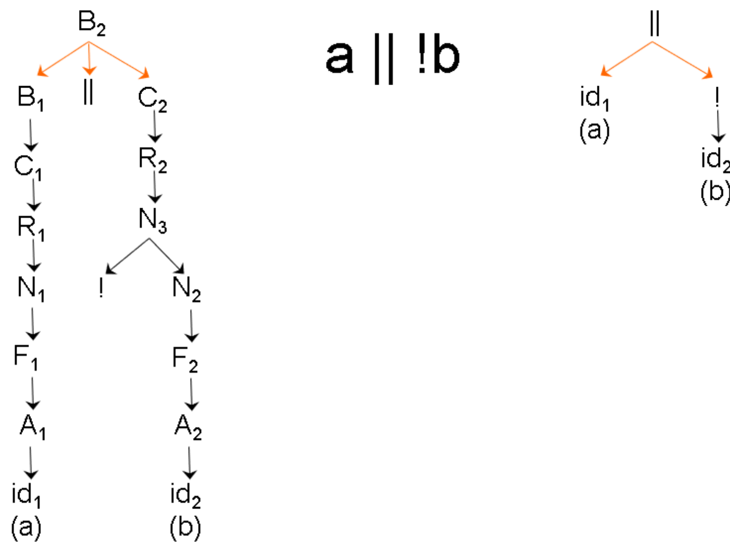
יש לנו את ארבע סוגי הבקרה הראשיים – if, if else, while, do while, כאשר אנחנו כבר יודעים שכל אח מהם אמור להיות קצת שונה מהאחרים ברמת המימוש. המשתנה B מייצג את כל המשפטים הבולייאנים שיכולים להיות לנו, כאשר בשביל להיות חד משמעיים, כל אפשרות בוליאנית מגיעה ממשתנה גזירה אחר.

על מנת לעבוד בצורה היעילה ביותר אנחנו נבנה AST – עץ תחביר מופשט. העץ הזה יחסוך לנו את כל מעברי הביניים המיותרים (אם נרצה לגזור ל-true זה סתם יעשה לנו עץ ענק), וככה יהיה לנו יותר קל לחשב בסוף את קוד הביניים. נעיף מבט על הכללים הסמנטיים הנתונים –

כללים סמנטיים	כלל שכתוב
S.nptr := mknode('if', B.nptr, S_1.nptr);	S → if (B) S_1
S.nptr := mknode('if', B.nptr, S_1.nptr, S_2.nptr);	S → if (B) S_1 else S_2
S.nptr := mknode('while', B.nptr, S_1.nptr);	S → while (B) S_1
S.nptr := mknode('do-while', S_1.nptr, B.nptr);	S → do S_1 while (B);
X.nptr := Y.nptr	X → Y s.t. X, Y ∈ V (לכל כללי היחידה דלעיל)
B.nptr := mknode('||', B1.nptr, C.nptr);	B → B1 || C
C.nptr := mknode('&&', C1.nptr, R.nptr);	C → C1 && R
R.nptr := mknode(relop.name, E.nptr, E1.nptr);	R → E relop E1

N → ! N1	N.nptr := mknode('!', N1.nptr);
N → (B)	N.nptr := B.nptr;
F → true	F.nptr := mkleaf('bool_lit', true);
F → false	F.nptr := mkleaf('bool_lit', false);
A → A_1 . id	A.nptr := mknode('.', A_1.nptr, mkleaf('id', id.entry));
A → A_1[E]	A.nptr := mknode('[]', A_1.nptr, E.nptr);
A → id	A.nptr :=mkleaf('id', id.entry);

רק בשביל להבין עד כמה זה משמעותי, מוצגת לנו דוגמה של גזירת העץ של הביטוי a || !b, ולידו יונח העץ המופשט של אותו ביטוי –



במצגת עצמה ממש אפשר לראות שלב אחרי שלב, כיצד החוקים הסמנטיים מייתרים לנו פה את כל העץ ומשאירים אותנו עם שיח קטן ויעיל, אבל אין סיבה להתעכב על זה – התוצאה הסופית היא אותו דבר בדיוק רק קצת נוח יותר.

לענייננו, אנחנו מזהים שיש פה יחס קבוע ב-AST, עבור כל אב מסוג כלשהו יחס הבנים תמיד יישאר קבוע – אם האב הוא while/if אז הבן השמאלי תמיד יהיה התנאי והבן הימני יהיה המשך הקוד. בעקבות כך אנחנו מגדירים לנו כללי עבודה שיהיו אחידים עבור כל סוגי התנאים – כשנדבר על האב נכנה אותו p, והבנים שלו ימוספרו משמאל לימין בתור $c_1..c_k$. לכל אב או בן אנחנו נייחס תכונות שונות אותם נעביר לפי הצורך, ובסוף נקבל עץ מעוטר ויפה.

יעדי קפיצות והערך fall

כשאנחנו נייצר את קוד הביניים לא יהיו לנו הפניות בקרה של שפה עילית כמו if, while והנגזרים מהם, אלא אז שתופיע לנו קפיצה (GOTO) או קפיצה מותנית (if GOTO) ואנחנו נצטרך לשחק איתם בשביל לממש את ההפניות השונות.

במידה ואנחנו נגיע לקפיצה מותנית ואנחנו לא נקפוץ בה, אנחנו נגדיר את זה פשוט כ-fall – אנחנו נופלים הלאה לשורה הבאה. למשל, אם יינתן לנו הקוד הבא – if (a < b) {<S>}, אז אנחנו מבצעים את הפקודות S רק במידה ו-a<b, אז אנחנו יכולים להבין מזה ככה – במידה והתנאי יתקיים אנחנו ממשיכים בקוד כרגיל, ובמידה ולא, אנחנו נדלג מעל הקוד. עכשיו, אנחנו צריכים לזכור שאין פה תנאי של "אחרת". או שהקוד מתקיים או שהוא לא מתקיים, ולכן אנחנו צריכים בכל מצב כזה לנתח ולהחליט מה יהיה התנאי שיעמוד לפני S. במקרה שלנו אנחנו נייצר את התנאי כך – if Not a<b goto L, כך שאם התנאי כן נכון אנחנו פשוט נמשיך הלאה לקוד.

פעולות סמנטיות להפניות בקרה

נראה עכשיו את הפעולות הסמנטיות הנוגעות לתנאים בלבד, ונרחיב אותו בהתחשב ביעדי הקפיצה על פי התוצאות השונות שלהם –

צומת אב	**פעולות סמנטיות**
if c_1 = condition c_2 = then statements (c_3 is null – no else part)	if *p*.next is undefined then *p*.next := new_label(); c_2.next := *p*.next; c_1.true := fall; c_1.false := *p*.next; *p*.code := c_1.code || c_2.code || label(*p*.next);
if c_1 = condition c_2 = then statements c_3 = else statements	if *p*.next is undefined then *p*.next := new_label(); c_2.next := *p*.next; c_3.next := *p*.next; c_1.true := fall; c_1.false := new_label(); *p*.code := c_1.code || c_2.code || gen('goto ' *p*.next) || label(c_1.false) || c_3.code || label(*p*.next);

while
c_1 = condition
c_2 = statements
	if *p*.next is undefined then *p*.next := new_label();
	p.begin := new_label();
	c_1.true := <u>fall</u>;
	c_1.false := *p*.next;;
	p.code := label(*p*.begin) || c_1.code || c_2.code ||
	gen('goto ' *p*.begin) || label(*p*.next);

do-while
c_1 = statements
c_2 = condition
	p.begin := new_label();
	c_2.true := *p*.begin;
	c_2.false := <u>fall</u>;
	p.code := label(*p*.begin) || c_1.code || c_2.code;

ננסה קצת להעלות נקודות שאפשר להבין מהכללים.

- בכל מקרה בו התנאי מופיע בתחילת ההפניית הקרה, אנחנו בודקים האם יש לנו כבר תווית שתייצג את ה-next שלו (כי יכול להיות שהאבא של p כבר חישב את זה בשבילו), במידה ואין תווית כזאת, אנחנו נייצר ונשמור לימים שחורים. כשאנחנו מגיעים ל-do while אין לנו צורך בזה, כי יש לנו לעשות את הקוד של ה-statements בכל אופן, ולכן במידה והתנאי לא יתקיים יהיה לנו פשוט fall.
- השלב הבא הוא לבדוק אילו סופי חלקים מקבילים לסוף החלק של האבא, למשל כשיש לנו תנאי ואז קוד, אז אם ניכנס לקוד ונסיים אותו נסיים במקביל את כל תת העץ הזה, ולכן אנחנו יכולים להגיד ש-c_2.next = p.next. ואם יש לנו if else אז כל חלק קוד שנעשה אותו ולא את השני, יצטרך לקפוץ בסופו ל-p.next.
- לאחר שנעשה את כל החישובים השונים, נוכל "לייצר" את הקוד על ידי שנרכיב את כל חלקי הקוד השונים, ביחד עם התויות שימוקמו כל אחד במקום הנכון שלו.

ייעול קודים בוליאניים

כאן אנחנו נחזור למה שהזכרנו קודם – Lazy Evaluating. ניתן כמה ללי אצבע שאיתם אנחנו עובדים ועוזרים לנו להכריע בצורה קלה ופשוטה את התנאים השונים –

קודם כל, נזכור כי עבור כל תנאי, אנחנו מחשבים רק קפיצה אחת, האפשרות השנייה תהיה fall באופן אוטומטי. החלק החשוב ביותר בהכרעה העצלנית, זה הידיעה מתי אפשר להפסיק לבדוק. כבר הזכרנו את זה קודם, אבל נזכיר שוב את שני החוקים העיקריים –

- אם יש לנו ביטוי or – כאשר אגף שמאל אמת, הביטוי כולו אמת – גם אם אגף ימין יהיה שקר, זה לא ישנה לנו – מספיק לנו אמת אחת.
- אם יש לנו ביטוי and – כאשר אגף שמאל שקר, כל הביטוי שקר – אנחנו צריכים שכל הביטוי יוציא לנו אמת, לכן ברגע שזה נופל, כל הביטוי נופל.
- בכל מקרה, אם יש צורך נבדוק גם את האגף השמאלי.

דבר נוסף שאנחנו מקבלים על הדרך, האגף הימני יקבל בדיוק את אותם הקפיצות של האב, אם אמת ואם שקר – מאחר ואם הגענו לאגף השמאלי, אז ההכרעה שלו תביא הכרעה לביטוי כולו, ולכן זה בדיוק אותם יעדי קפיצה (או נפילה).

עבור אגף שמאל, הערך שיכריע עבור שאר הביטוי (אמת עבור or, או שקר עבור and), יקבל לאותו הערך את ערך האב. כלומר אם יש לנו $c_1 \,||\, c_2$ אז ברגע ש-c_1 = true אנחנו נוכל להמשיך הלאה לחלק שאחרי כל התנאי הזה, ולכן נוכל להגדיר כי c_1.true = p.true וכן יהיה גם ב-and רק לעניין ה-false. בכל אופן, הערך השני תמיד יהיה fall, כי הבדיקה פשוט תצטרך לרדת ולבדוק את c_2.

דבר אחרון, ביטוי not פשוט יהפוך את היחס לאב כלומר c_1.false = p.true ולהיפך.

על מנת לראות איך אנחנו מממשים כל תנאי ומה סדר הפעולות, נצטרך ליצור תנאים סמנטיים מותאמים לצורך העבודה –

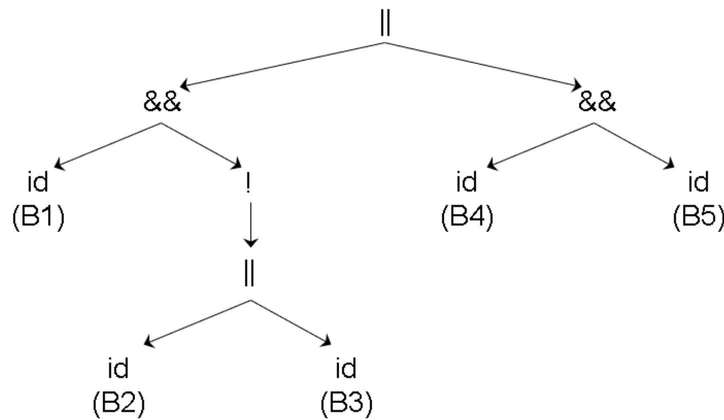
צומת אב	פעולות סמנטיות
|| c_1 = left-hand side c_2 = right-hand side	if p.true ≠ fall then c_1.true := p.true else c_1.true := new_label(); c_1.false := fall; c_2.true := p.true; c_2.false := p.false;

<table>
<tr><td></td><td>p.code := c_1.code || c_2.code;
if p.true = fall then p.code := p.code || label(c_1.true);</td></tr>
<tr><td>&&
c_1 = left-hand side
c_2 = right-hand side</td><td>c_1.true := fall;
if p.false ≠ fall then c_1.false := p.false else c_1.false := new_label();
c_2.true := p.true;
c_2.false := p.false;
p.code := c_1.code || c_2.code;
if p.false = fall then p.code := p.code || label(c_1.false);</td></tr>
<tr><td>!
c_1 = inner expression</td><td>c_1.true := p.false;
c_1.false := p.true;
p.code := c_1.code;</td></tr>
<tr><td>R
where R is a relational operator (relop)
c_1 = left-hand expr.
c_2 = right-hand expr.</td><td>if p.true ≠ fall then p.code := gen(‘if ‘ c_1.var ‘ R ’ c_2.var ‘ goto ‘p.true);
else p.code := gen(‘ifNot ‘ c_1.var ‘ R ’ c_2.var ‘ goto ‘p.false);</td></tr>
<tr><td>id</td><td>if p.true and p.false are defined then begin
  if p.true ≠ fall then p.code := gen(‘if ‘ p.name ‘ goto ‘ p.true);
  else p.code := gen(‘ifNot ‘ p.name ‘ goto ‘ p.false);
end;</td></tr>
<tr><td>.
c_1 = record
c_2 = field</td><td>p.addr := address(c_1.c_2);
p.var := new_temp();
p.code := gen(p.var ‘ := *’ p.addr);
if p.true and p.false are defined then begin
  if p.true ≠ fall then p.code := p.code || gen(‘if ‘ p.var ‘ goto ‘ p.true);
  else p.code := p.code || gen(‘ifNot ‘ p.var ‘ goto ‘ p.false);
end;</td></tr>
<tr><td>bool_lit
הערה: קוד טוב יותר יושג אם
המהדר כבר צמצם את הביטוי;
למשל:
false && B2 ➔ false</td><td>if p.true ≠ fall and p.value = ‘true’ then p.code := gen(‘goto “ p.true);
elsif p.false ≠ fall and p.value = ‘false’ then p.code := gen(‘goto “p.false);
else p.code := ‘’;</td></tr>
<tr><td>[]
c_1 = array
c_2 = index</td><td>Perform type checking for $c_1[c_2]$
Generate code for $c_1[c_2]$ (we don’t detail rules here)
Assume p.var holds $c_1[c_2]$
if p.true and p.false are defined then begin
  if p.true ≠ fall then p.code := p.code || gen(‘if ‘ p.var ‘ goto ‘ p.true);
  else p.code := p.code || gen(‘ifNot ‘ p.var ‘ goto ‘ p.false);
end;</td></tr>
</table>

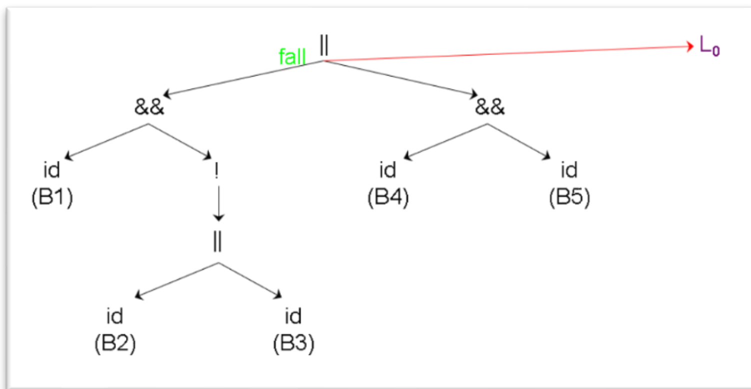
אין פה משהו חשוב שאנחנו צריכים לראות ולהתמקד בו, רק אפשר להתעמק קצת ולראות כיצד כל צומת אב דואגת לכך שלכל בן יהיה קפיצה או נפילה. על פי החוקים האלה אנחנו בעצם עוברים על העץ בהמשך. אנחנו עובדים על AST של הביטוי הבא –

(B1 && !(B2 || B3)) || (B4 && B5)

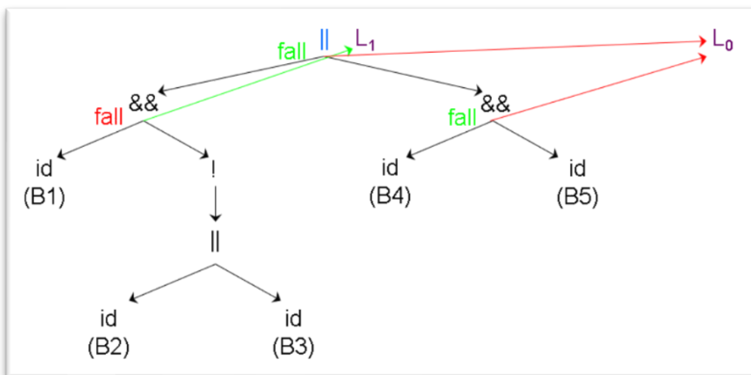
יש לנו כאן תנאי גדול שהוא שני חלקים עיקריים עם or ביניהם, כאשר החלק השמאלי מפוצל לעוד תתי חלקים. בסך הכל, העץ נראה כך –

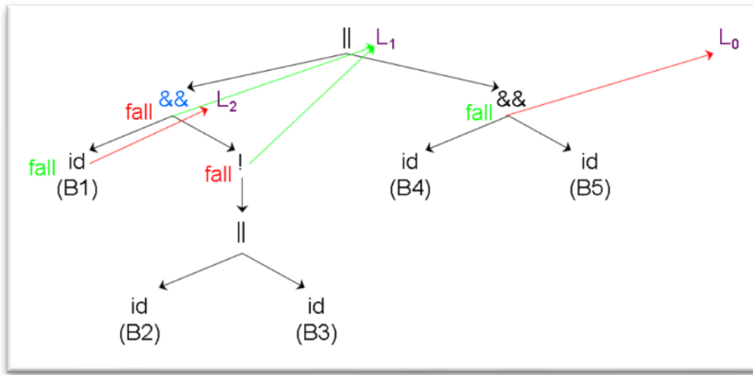


נתחיל קודם כל עם קביעה ל-or הראשי את יעדי התכונות שלו – נתחיל דווקא בערך ה-false. אם התנאי לא יתקיים, אז קטע הקוד שאחרי ה-if (או while לצורך העניין) לא יתבצע. אנחנו מגדירים תוית חדשה (מאחר ונתקלנו בצורך), ומותחים אליה קו אדום (כי false זה רע), אך אם הערך שלו מוגדר כאמת אנחנו פשוט ניפול להמשך הקודף, וכמובן שזה יהיה בצבע ירוק, כי ירוק זה טוב.

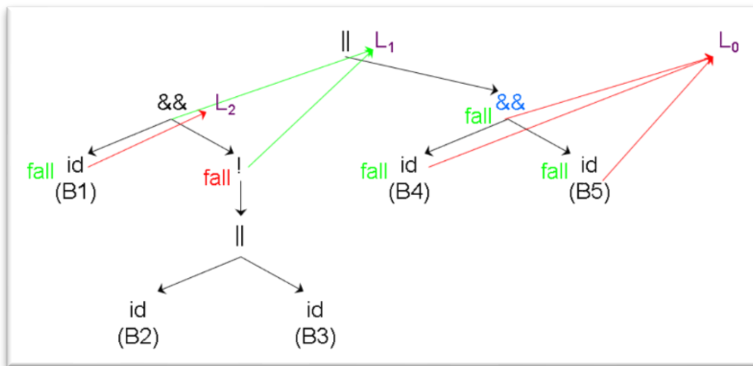


עכשיו נסתכל על הבנים של ה-or. הבן הימני יהיה בדיוק כמו האבא שלו, ולכן שקר יצביע ל-L0, ואמת תוגדר כנפילה. אך מה יהיה עם הבן השמאלי? אם c_1 יוציא לנו ערך true הוא לכאורה צריך להמשיך לערך p.true, אבל אנחנו הגדרנו את הערך הזה להיות fall שזה דבר שאנחנו לא יכולים להרשות כרגע, כי אין להם את אותה נפילה. לכן אנחנו נאלץ ליצור כאן תווית חדשה L1 ולהגדיר שבמידה ויהיה לנו c_1 = true אנחנו נעבור לנפילה של p.true, אך זה יוגדר מעכשיו תחת התוית שיצרנו.

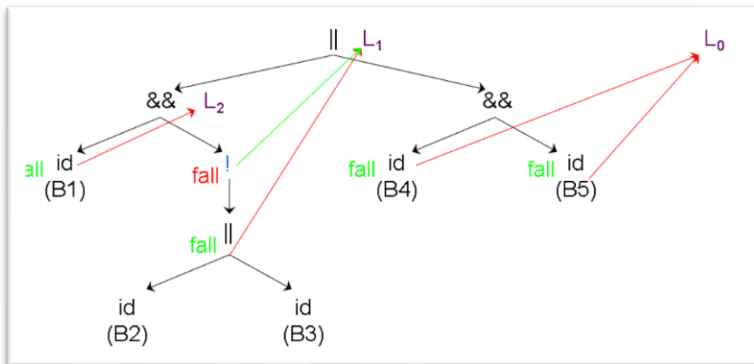




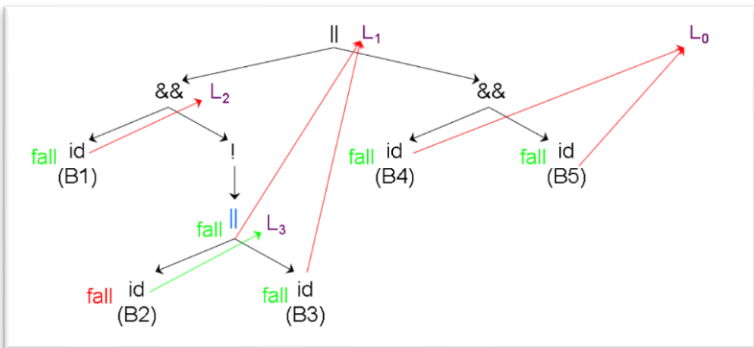
עכשיו נעבור לבנים של $p.c_1$, הבן הימני שלו יקבל את אותם ערכים של האבא. מאחר ואנחנו תחת and, אז אנחנו יודעים שאם הבן השמאלי יקבל false הוא מכשיל את כל התנאי, ולכן הוא יקפוץ פשוט לערך ה-false של האב, אבל שוב אנחנו נתקלים שם ב-fall. לכן אנחנו מגדירים תווית חדשה L2, ומגדירים אותו להיות יעד המעבר של fall. כמובן, שאנחנו מגדירים רק קפיצה אחת לצומת, ולכן ערך true יהיה פשוט נפילה.



נעבור לבן השני של האב הראשי, ואחרי זה נרד לרמה הבאה. הבן הימני בוודאי יקבל את אותם הערכים כמו האב. הבן השמאלי יבחן את המצב – כישלון שלו זה כישלון של האב && ובתור בכלל כישלון כל התנאי, לכן את ערך ה-false אנחנו מעבירים ל-L0. מה שנותן לנו את האפשרות השניה – לעשות fall ברגע שזה אמת.



עכשיו לפני שנרד לגמרי לרמה האחרונה, יש לנו Not קטן שפשוט יחליף לבן שלו את הסימנים, מה שטוב רע, מה שרע טוב.

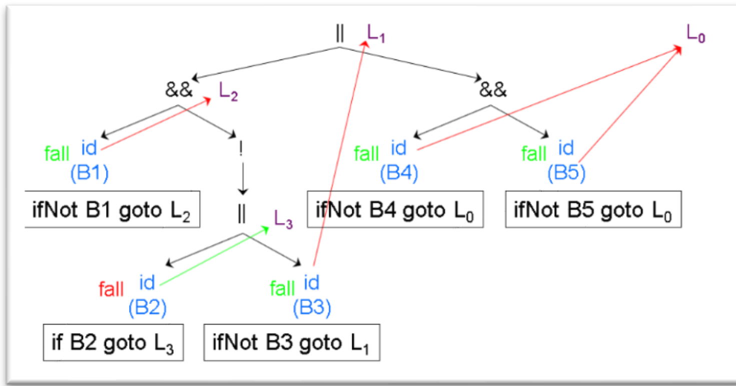


שני הבנים האחרונים יעבדו באותה דרך שראינו עד עכשיו – הבן הימני ייקח את אותם הערכים של p. הבן הימני מוגדר בתור אמת לעבור לערך P.true, אך שם יש לנו רק fall, ולכן הוא מגדיר שם תווית חדשה L3.

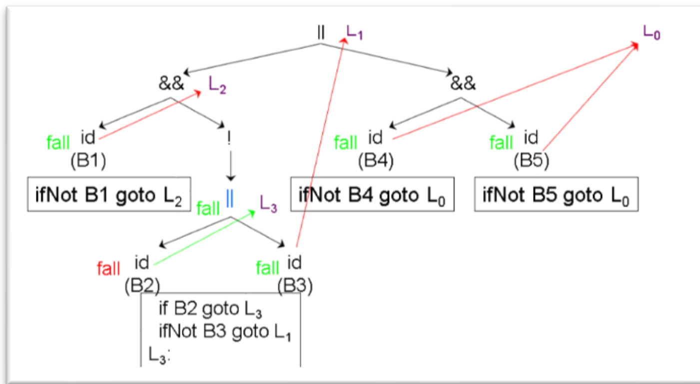
ערך ה-false לא נותר לו אלא להיות fall.

עד עכשיו חישבנו את מקומות הקפיצה של כל צומת. עכשיו אנחנו צריכים להתחיל ולחשב את הקוד עצמו.

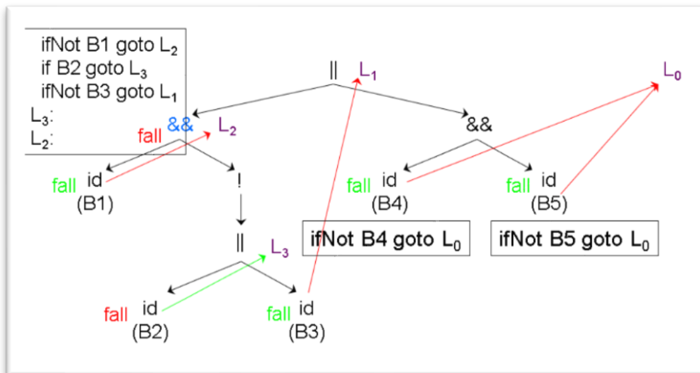
הדבר הראשון שנעשה, הוא לכתוב את תנאי הקפיצה המתאים לכל צומת. כזכור, יש לנו רק אפשרות קפיצה אחת לכל צומת, ולכן אנחנו נבחן את התנאי שגורם לנו לקפוץ ונכתוב אותו בקוד.



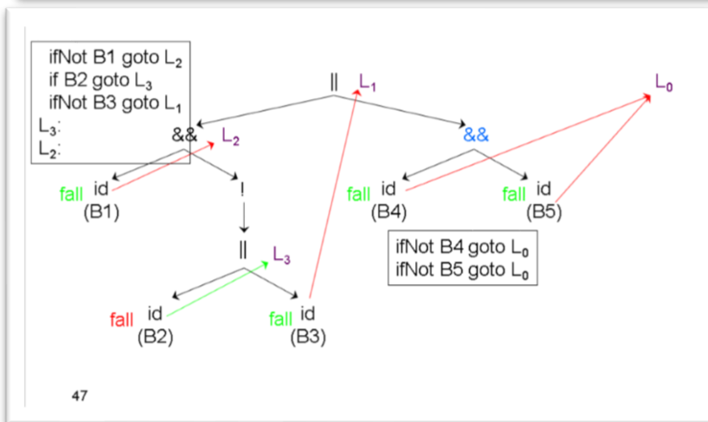
עכשיו אנחנו מתחילים לאסוף את הקוד. אנחנו עוברים בצורת post-order, כלומר נכנסים הכי עמוק שאפשר, ואז הולכים בסדר של שמאל-ימין-אב. לכן, כשנגיע ל-B2,B3 ניקח קודם כל אותם בסדר הזה, ואז נצרף את ה"קוד" של האבא שהוא פשוט התווית שאנחנו הגדרנו לו.



עכשיו אנחנו יכולים לדלג על הצומת של ה-not כי אנחנו בעצם ייצרנו את הקוד בהתחשבות בו על ידי ששינינו את יעדי הקפיצה של הבן שלו, ואנחנו עכשיו לוקחים את B1, ומצדו ימין את כל הקוד שייצרנו עד עכשיו, ואחרי זה את האב, יעד הקפיצה L3.

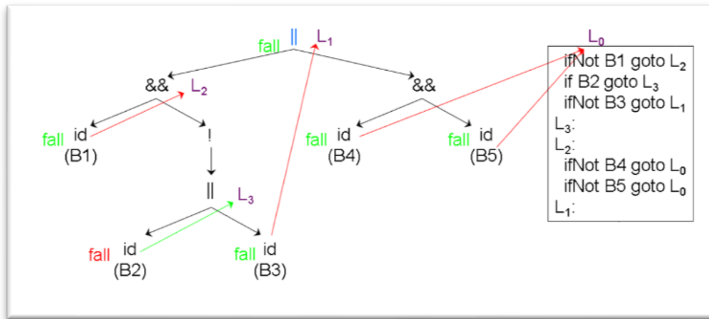


אותו דבר אנחנו עושים גם עם הבן הכללי הימני – יש לנו רק תנאי קפיצה של הבנים, אך אין תוית לאב.



עכשיו נשאר לנו לאחד את כל הקוד של שני הבנים, ביחד עם התוית L1 של האב, ולכאורה כמעט סיימנו.

למה רק כמעט? כי יש לנו את התווית L0שבכלל עוד לא נגענו בו. איך נתייחס אליה? זה כבר תלוי איפה נמצא התנאי. אם התנאי נמצא בתוך לולאת while או בתוך if הבנייה של הקוד תהיה קצת שונה.



נראה את ההבדלים –

עבור תנאי של If-else מה שנעשה הוא כזה –

```
 ifNot B1 goto L2
 if B2 goto L3
 ifNot B3 goto L1
L3:
L2:
 ifNot B4 goto L0
 ifNot B5 goto L0
L1:
 <quadruples for S1>
 goto Lnext
L0:  # else label
 <quadruples for S2>
Lnext:
```

למעשה, L2 וגם L3 מגיעים לאותו מקום, אז כנראה נוריד אחד מהם בהמשך, אבל מה שחשוב הוא שאם התנאי מתקיים, אנחנו רצים על הקוד של S ואז מדלגים ישר ל-p.next. ואם לא, אנחנו מכניסים את L0 ומתחילים לעבוד עליו.

לעומת זאת, עבור while הקוד הבסיסי יישאר אותו קוד, כי תנאי זה תנאי, אבל המימוש של הקפיצות יהיה שונה -

```
Lbegin:
 ifNot B1 goto L2
 if B2 goto L3
 ifNot B3 goto L1
L3:
L2:
 ifNot B4 goto L0
 ifNot B5 goto L0
L1:
 <quadruples for S1>
 goto Lbegin
L0:  # p.next:
```

אנחנו נקפוץ בכל פעם שנסיים את הקוד לראש הקוד, וזה יהיה לא מותנה, אך אם הקוד ייכשל, נכניס שם את L0 שיוציא אותנו מהלולאה, וייתן לנו להמשיך לבלוק הבא.

סביבת זמן ריצה

הסמנטיקה של הקוד, לא נמצאת רק בקוד עצמו אותו אנחנו חישבנו עד עכשיו. בכל שפה שלא תהיה, ישנם תוספות של קוד וחלקים סמנטיים שהקומפיילר מוסיף בעצמו בזמן הקימפול של הקוד. אחד החלקים העיקריים שהקומפיילר מוסיף זה התעסקות עם פונקציות, יש חלקים שמוסיפים לפני הפונקציה, וחלקים אחרי, וכמובן עוד יש לנו התעסקות עם פונקציות רקורסייות וכו'.

כמובן, שככל שנייעל את הטיפול בפונקציות, כך גם זמן הריצה יושפע, וזה בסופו של דבר מה שחשוב לנו – לעבור מפונקציה לפונקציה בזמן הריצה באופן המהיר והיעיל ביותר. רוב מוחלט של השפות (אם לא כל השפות) היום עובדות באופן של מעבר בין פונקציות, קבלת ערכים והחזרתם, חישוב פנימי, וכל חלק כזה חשוב לנו לראות איך נבצע אותו נכון.

שלושת הדברים העיקריים איתם נתעסק במימוש של הקומפיילר, הקשורים לזמן הריצה הם:

שמירת כתובות חזרה – הדבר החשוב לא פחות מאשר יציאה לפונקציה הוא החזרה מהפונקציה. אנחנו רוצים שנדע לחזור בדיוק לאותה נקודה בקוד בה היינו קודם, ולכן עלינו לשמור בצורה יעילה את המיקום בו היינו, ולדעת איך לחזור לשם.

ממשק בין פונקציה קוראת לנקראת – קישור פרמטרים ומוסכמות – הפונקציה הנקראת יכולה/לא יכולה לגשת לפרמטרים של הפונקציה שקראה לה (על פי עקרונות של כל שפה ושפה), מעבר לכך, אם הפונקציה הקוראת השתמשה ברגיסטרים מסוימים על מנת לשמור בהם ערכים, אנחנו צריכים לשים לב ולא לדרוס אותם, או לפחות לשמור את הערכים האלה במקום אחר שלא יידרס.

הגדרת משתנים מקומיים – איך ניגש אליהם, מי יכיר אותם תחת איזה scope ועוד.

צריך לזכור שאנחנו מתעסקים פה עם זיכרון ורטואלי – המערכת מקצה לנו לשימוש שטח מסוים אותו אנחנו מחלקים לפי השימושים שלנו בתוכנית. יש לנו את מחסנית זמן הריצה, ויש גם את הערימה (heap), איתה אנחנו מתעסקים להקצאות זיכרון דינאמיות, כך שאין לנו יכולת פשוט לשמור את כל המידע במחסנית, אלא עלינו להזהר מ- stack overflow.

Dynamic vs. Static Scope

לפני שנתחיל את הדיון, נדבר גם על ההבדלים בין scope סטטי לדינאמי. ככלל – אנחנו מדברים פה על מתחמי הכרה, ואיזה פונקציה מכירה אילו ערכים של איזו פונקציה. בדרך כלל אנחנו מתייחסים לכך, שפונקציות מגדירות פונקציות אחרות בתוכן. אך בזמן הריצה, סדר הקריאות האמיתי הוא לא כמו שכתוב בתוך הקוד הסטטי – פונקציה יכולה לקרוא לעצמה או לעשות סיבוב ולהקרא פעמיים על ידי פונקציות שונות, ובכל פעם כזאת יש לנו את כל ההגדרות של הפונקציה והמשתנים שצריכים להיטען. כאן בא ההבדל בין שני הגישות

הגישה הדינאמית – הקישור בין הפונקציות יהיה לפי סדר הקריאה. אם פונקציה נקראה פעמים, אנחנו מתייחסים בכל פעם רק לקריאה האחרונה שלה.

הגישה הסטאטית – אנחנו מתייחסים להיררכיה של הפונקציות לפי איך שהוגדרו בקוד הסטטי, ומתעלמים כמה שניתן מסדר הריצה.

בפועל, רוב השפות המודרניות מתעסקות עם מתחמי הכרה סטטיים, אבל נראה בהמשך שלא מדובר באופן סטטי לגמרי, אלא יש פה רמה של דינאמיות (ועדיין, זה מוגדר סטטי), ואם נרצה למנוע גישה, תמיד נוכל להגדיר פונקציות מסוימות כ-private או protected וכך להגן על המידע שם.

על מנת להבין קצת את המשמעות הסטטית, נוכל להסתכל על הקוד הבא –

```
main ( )
{
    int a = 0 ;
    int b = 0 ;
    {
        int b = 1 ;
        {
            int a = 2 ;
            printf ("%d  %d\n",  a,  b)
        }
        {
            int b = 3 ;
            printf ("%d  %d\n",  a,  b) ;
        }
        printf ("%d  %d\n",  a,  b) ;
    }
    printf ("%d  %d\n",  a,  b) ;
}
```

B0 B1 B2 B3

יש כאן 4 פונקציות שונות (ללא שם) התחומים כל אחד בבלוקים השונים (סימוני ה-B), כאשר בכל החלקים מוגדרים משתני Int בשם a,b. שימו לב – בכל בלוק חדש בו יש לנו משתנה עם השם הזה, אנחנו מגדירים אותו מחדש, אחרת אם בשורה השלישית של הקוד (השורה הראשונה של B_1) נעשה השמה פשוטה של b=1 אז נשנה את זה גם לרמה מעל. אם נעקוב אחרי ההשמות השונות, נוכל למיין את כל הבלוקים השונים והמשתנים לטבלה הבאה –

הגדרה	scope
a=0	B0, B1, B3
b=0	B0,
b=1	B1, B2
a=2	B2
b=3	B3

תכלס, יש פה קצת בלאגן באופן שהם סידרו את הטבלה, אבל ננסה לעקוב.

B0 הוא ה-main. הוא מגדיר את a,b שיהיו שניהם שווים 0.

B1 פשוט משנה את b=1 (כלומר, a נשאר באותו הערך), ומגדיר את שתי הפונקציות הבאות בתוכו.

B2 משנה את a=2 ומדפיס את שני הערכים, כלומר מה שיודפס לנו יהיה 2,1.

B3 משנה את b=3 אך הוא לא נוגע ב-a. מה יהיה הערך של a? אמנם שינינו את a בתוך B2 אבל מאחר והוא רק אח שלו, אנחנו לא מתייחסים לזה כאל שינוי קובע, ולכן אנחנו ממשיכים לתור למעלה בהכרה, עד שנגיע ל-B0 שם הגדרנו אותו לראשונה שיהיה שווה 0. ולכן, אם נדפיס בסוף הפונקציה נקבל 0,3.

אחרי שעברנו את כל זה, קופץ לנו שוב B1 ורוצה להדפיס את הערכים שלו, אך יש לשים לב, הערכים אליהם הוא מתייחס הם לא אלה שהוגדרו בבנים שלו, אלא רק בו וברמות מעליו, ולכן מה שיודפס בו הוא 0,1.

באופן דומה, גם ההדפסה עכשיו של B0 תביא לנו את הערכים המקוריים של המשתנים האלה, שיהיו 0,0.

בשביל להבין ממש את ההבדל, נוכל לראות את התכנית הבאה –

```
var x: int;
function foo(): int      {      return x;      }
function bar(): int      {      var x: int;
                                x = 1;
                                return foo();  }
procedure main()         {      x = 0;
                                print bar();   }
```

השאלה הנשאלת היא – מה יודפס בשורה האחרונה?

אם נממש את ההכרה הסטטית - מאחר וכל הפונקציות מוגדרות ברמה הגלובלית, שלושת הפונקציות מוגדרות כ"אחים" אחת של השניה. ולכן כאשר bar מגדיר int x חדש זה לא משנה ל-foo, כי הוא לא מכיר את מה שקורה אצל אח שלו, אלא רק ברמה שלו וכלפי מעלה, ולכן הוא מתייחס להשמה המקורית של x=0 בתוך הפונקציה main, ולכן יודפס 0.

לעומת זאת, בדינאמית – אמנם שמנו 0 בהתחלה, אבל כשנכנסנו ל-bar שינינו את הערך של x ל-1, וזה מה שמוגדר במתחם ההכרה שלו, וכש-foo יחפש x הוא ימצא את מה שהוגדר ב-bar וידפיס 1.

מימוש גישה למשתנים

Dynamic Scping

לכל משתנה שמוגדר בתוכנית אנחנו מגדירים מחסנית חדשה משלו. את המחסניות האלו אנחנו מתפעלים לכל אורך זמן הריצה, כאשר ברגע שיש לנו השמה כלשהי של משתנה בתוך scope מסוים, אנחנו נדחוף פנימה את ההשמה. וברגע שנצא מאותו scope אנחנו נוציא החוצה את ההגדרה. כך אנחנו נוכל לגשת בכל רגע נתון לאותו משתנה ולדעת מה היתה ההשמה האחרונה שהוכנסה אליו. לצורך העניין, אם נסתכל על הקוד למעלה, המחסנית של x תיראה כך –

Global::x = 0
Bar::x = 1

החסרון של מימוש כזה, הוא שאם יש לנו בקוד גישה לא חוקית לאותו משתנה, אנחנו לא נוכל לדעת את זה עד שלא נרוץ על הקוד ונראה פתאום שיש לנו שגיאה. בנוסף, אם אנחנו שומרים את זה באופן הזה אנחנו עלולים להגדיר x בתור מספר או מחרוזת במקומות שונים. אם היה לנו אפשרות לבדוק את כל זה בקימפול זה היה עלול לחסוך לנו שגיאת זמן ריצה.

רק נזכיר, אנחנו מעדיפים להמנע ככל האפשר משגיאות בזמן הריצה. תמיד נעדיף שהשגיאות שנקבל יהיו בזמן הקימפול, כך שנוכל לתקן הכל לפני שנוציא הלאה את התכנה. אם השגיאות יופיעו בזמן הריצה, זה עלול ליפול בזמן שהלקוח עובד וזה כבר פחות טוב.

Static Scoping

בזמן הקומפילציה אנחנו מגדירים כתובת לכל משתנה. ובכל גישה לאותו משתנה אנחנו יודעים בדיוק לאן ללכת. באופן זה, אם יש מתחם הכרה כלשהו שרוצה ליצור את אותו משתנה עם שם אחר, אנחנו פשוט נקצה מקום חדש ואליו נפנה את אותו משתנה.

המימוש הזה פותר לנו את שתי הבעיות שראינו בדינאמי – אם אנחנו יודעים מראש לאן כל אחד הולך, אנחנו לא נצטרך לדאוג לגישה לא מאושרת (כי נבדוק את זה קודם), וגם לא נחשוש להשמה של טיפוס לא מתאים, כי נבדוק את זה קודם, ובמידת האפשר פשוט נקצה מקום חדש.

אופן מימוש של דבר כזה, יכול להיות על ידי טבלה –

שם	כתובת שהוקצתה עבורו
x (גלובלי)	432234
x (בתוך bar)	432238

ובזמן הקומפילציה, אנחנו פשוט נשמור עבור כל משתנה את המיקום אליו הוא מתייחס.

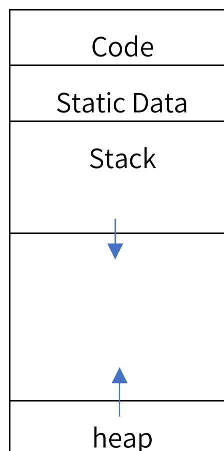
הבעיה במימוש מהסוג הזה, הוא רקורסיה. אם אנחנו משתמשים באותה פונקציה מספר פעמים, בתוך עצמה, בכל פעם אנחנו מתייחסים לאותו מקום בזיכרון, אז אנחנו בכל פעם דורסים את הזיכרון עם ערך חדש, וכך תהיה לנו בעיה מאוד רצינית כשנרצה לאחזר את המידע. ניקח לדוגמא קוד של פיבונאצ'י –

```
procedure fib(n: int)    {    var a: int;
                              var b: int;
                              if (n ≤ 1) return 0;
                              a = fib(n – 1);
                              b = fib(n – 2);
                              return a + b;        }
procedure main()         {    print fib(5);        }
```

אם נגדיר מראש מקום עבור המשתנים a ו-b, אזז בכל גישה אנחנו נדרוס את הערכים שלהם, כך שברגע שנגיע לרגע בו צריך להחזיר את הערכים, לא נסכום את התוצאות כמו שאנחנו מצפים, אלא נחזיר 0+1.

מסיבה זאת, אנחנו מחליטים לממש את התצורה הסטטית באופן שונה –

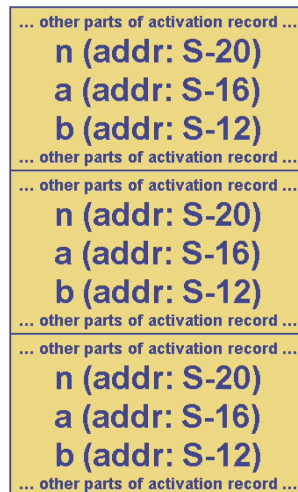
אנחנו נגדיר לכל פונקציה מתחם שייקרא Activation Record (רשומת הפעלה), ובו נשמור מידע חשוב לגבי הפונקציה. כל גישה לפונקציה, תיכנס למחסנית זמן הריצה ביחד עם ה-AR שלה, כאשר גם אם נקרא לאותה פונקציה משתי רמות שונות, ה-AR שלהם יהיה מעט שונה, מה שייתן לנו אפשרות לגשת למידע בצורה שהיא קצת יותר דינאמית מאשר האופן הסטטי הפשוט. כך שהזכרון המוקצה לתוכנית ייראה באופן הבא –



הקוד הוא החלק הסטטי ביותר – התוכנית עצמה איך שהיא כתובה. ב-static data אנחנו נשמור משתנים גלובליים, אליהם נוכל תמיד לגשת ולא משנה באיה רמה. כדאי לשים לב, כי החלקים האלו מוגדרים בזמן הקימפול ולא משתנים לאורך הריצה של התוכנית. המחסנית תגדל ותקטן בהתאם לגישות לפונקציות השונות, והערימה תגדל ותקטן על פי הקצאת הזיכרון הדינאמי.

המשתנים המקומיים, כמו למשל a,b של קריאה בפיבונאצ'י יהיו חלק מה-AR שייכנס בכל פעם לקריאה חדשה של פונקציה, וכך נוכל להיות בטוחים שאנחנו לא נדרוס אותם. על מנת לוודא שאנחנו ניגשים באופן נכון למשתנים, בזמן

הקומפילציה אנחנו נחשב את כל המשתנים הקשורים לפונקציה מסוימת ונקצה לה מקום יחסי לערך הפונקציה, כאשר הערך יוגדר ברמה של S+a. S יבטא את ראש המחסנית, ו-a יהיה ההיסט בשביל להגיע לאותו משתנה. באופן הזה, לא משנה כמה כניסות של אותה פונקציה יהיה לנו, המשתנה שאותו נחפש יהיה תמיד במרחק של חישוב קטן מראש המחסנית. נוכל לראות באיור הבא, המשתנים מוגדרים בהיסט קבוע, ובכל פעם אנחנו יכולים לגשת בדיוק לאותו מקום ביחס לראש המחסנית -



הערה: במצגת מדובר על S-a, ולא +. אבל במימוש אמיתי אם מוקצה לנו מקום בגודל 8000, אנחנו מתחילים דווק מהקצה העליון, כלומר אם יש לנו קודם כל להכניס איזה int שתופס 4 ביטים, אז הוא יוקצה בשטח של 8000-7997, ולכן כאשר נדבר על גישה מראש המחסנית אנחנו נצטרך להוסיף לתוצאה של ראש המחסנית בשביל "לחזור אחורה" על המידע.

הערה נוספת: אנחנו גם לא נשתמש בראש המחסנית (אבל זהו, אין עוד שינויים). ראש המחסנית הוא בר שזז כל הזמן במשך הריצה, ויש חשש שאם נסתמך עליו יותר מידי, אנחנו עלולים לא לשים לב לשינוי שלו וככה להגיע לתוצאה לא טובה, לכן אנחנו נשים pointer בתוך רשומת ההפעלה, ונדע להגדיר ממנו משתנים בהיסט חיובי או שלילי לפי הצורך.

רשומת הפעלה

כמו שכבר הוזכר, אנחנו שומרים את רשומת ההפעלה המכילה את המידע החשוב לכל פונקציה, וזה נכנס ויוצא עם כל קריאה וחזרה מאותה פונקציה. מה יש ברשומת ההפעלה?

returned value
actual parameters
optional control link
optional access link
saved machine status
local data
temporaries

Returned Value -

Returned Value – יכול להיות מקום שמוגדר במחסנית עמה ובו שמור הערך המוחזר, ויכול להיות מצביע לרגיסטר מסוים בו אנחנו נשמור את המידע. לפעמים גם אנחנו מחזירים מערך או מידע דינאמי אחר, והמצביע יוביל אותנו ל-heap שם נמצא אותו ערך.

Actual parameters – הערכים המוכנסים לפונקציה הנקראת מהפונקציה שקראה לה. אם הוכנסו שני ערכי Int, נדע בכל פעם שאנחנו קוראים לפונקציה להקצות שני בתים עבור השמירה שלהם.

Optional control link – יכיל מצביע לפונקציה שקראה לפונקציה הנוכחית. בנוסף, אנחנו מתייחסים אליו הרבה בתור "Frame pointer" או fp. כמו שכבר הזכרנו קודם, אנחנו קובעים את היסט הגישה למשתנים לא על פי ראש המחסנית, אלא על פי מצביע בתוך ה-AR. וזה מה שאנחנו מדברים עליו עכשיו. כשנסתכל על ה-fp אנחנו נוכל לדעת שכל הפרמטרים שהתקבלו מהפונקציה הקוראת נמצאים למעלה (היסט חיובי), ושאר המשתנים המקומיים של הפונקציה יהיו בהמשך (היסט שלילי).

Optional access link – מצביע לערך האב הסטטי של הפונקציה. כלומר, יכול להיות שיש לנו תוכנית שכתובה באופן הבא –

```
f(){
  a(){}
  b(){
    c(){}
  }
}
```

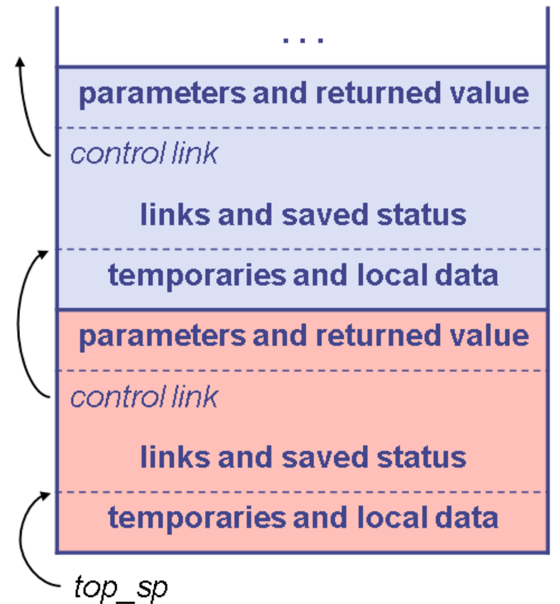
אז אם פונקציה מסוימת תקרא ל-a או ל-b לשניהם יופיע ב-Access link את f בתור האבא שלהם, וכך אנחנו יודעים שאנחנו יכולים להתיחס גם לאבא במידה ונצטרך לגשת אליו ולמשתנים המוגדרים בו.

Saved machine status – מצב המכונה (לפני הקריאה לפונקציה). חלק זה יכול להכיל כתובת לחזרה בסיום הריצה על פונקציה, אבל יכול להיות גם רגיסטר מסוים ששייך לפונקציה הקוראת, ואנחנו לא יודעים מה יש בו, אבל אנחנו רוצים לוודא שאנחנו לא נדרוס את המידע בזמן הריצה על הפונקציה, לכן אנחנו לוקחים את המידע שיש שם ושומרים בסטטוס. ברגע שנצא בחזרה מהפונקציה, לא רק שנדע לאן לחזור, אלא גם מה הערכים שהיו שם לפני שהתחלנו לעשות בלגן.

Local data – משתנים מקומיים של הפונקציה.

Temporaries – המשתנים הזמניים של חישובי הביניים בפונקציה.

על מנת לראות איך אנחנו עובדים עם הקריאות השונות, נתייחס לאיור הבא –



הקונטרול לינק של כל פונקציה מצביע אחורה לעבר הפונקציה שקראה לו, וכך זה עולה למעלה עד לרמה הראשונית ביותר. עכשיו נדבר על סדר הפעולות שהפונקציה הקוראה (הכחולה) צריכה לעשות בשביל להפעיל פונקציה חדשה (האדומה) –

1. הקורא מחשב את הפרמטרים האקטואליים – לפני שהוא קורא לפונקציה, על הקוראת לוודא שהיא שולחת לנקראת ערכים אמיתיים. כדאי לזכור לפעמים במקום מספר אנחנו שולחים תרגיל מתמטי או דברים כאלה, ולכן, לפני שאנחנו מתחילים את העבודה, אנחנו צריכים לחשב את כל מה שעובר אליה.
2. הקורא שומר את כתובת החזרה ואת ערכו הנוכחי של *top_sp*. כמובן, שהפונקציה הנקראת לא יודעת לבד את המצב הזה, ואם נפעיל אותה ואז נשאל איפה ראש המחסנית נמצא זה יהיה כבר מאוחר מידי, לכן גם את זה הפונקציה הקוראת עושה.
3. הקורא מקדם את *top_sp* - בעצם מגדיר את המיקום החדש.
4. הפרוצדורה הנקראת שומרת את הרגיסטרים ואת שאר ה- status information – בתור דאגה ווידוא של הפונקציה הנקראת שלא לדרוס ערכים של הקוראת, וכן לוודא שהיא חוזרת למקום הנכון, אנחנו ניגשים לפונקציה הקוראת ובודקים שם, היכן אנחנו עומדים. כדאי לזכור, שאנחנו יכולים פשוט לדלג בקונטרול לינק ולחפש באבא את ערכי המשתנים השונים.
5. הפרוצדורה הנקראת מאתחלת את המשתנים ומתחילה לפעול – כאן אנחנו עובדים בצורה לוקאלית של קריאת המשתנים הרלוונטיים ואז מתחילים לרוץ.

כדאי לשים לב, שבעצם רוב רשומת ההפעלה של כל פונקציה נקראת, מנוהלת בכלל על ידי הפונקציה שקראה לה, ורק חלק קטן שרלוונטי לפונקציה הנקראת מנוהל על ידיה.

סדר היציאה מהפונקציה, עובד בדיוק הפוך מזה –

1. הפרוצדורה הנקראת מאכסנת את הערך המוחזר במקום המתאים – מכניסים את התוצאה ל- returned value.
2. הפרוצדורה הנקראת משחזרת את *top_sp*, את ערכי הרגיסטרים, ואת ה- status information – בגדול, מחזירים את כל המצב של מערכת למה שהיה לפני הגישה לפונקציה הזאת.
3. למרות ש- *top_sp* עודכן, ניתן לגשת לערך המוחזר – שנמצא לכאורה אחרי ראש המחסנית.

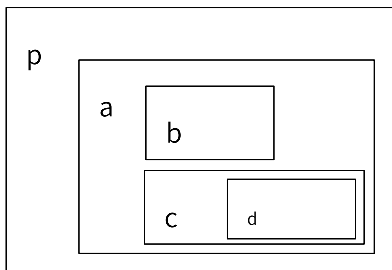
כמובן, שפה אין לנו שום סיבה להתעסק עם הפרמטרים האקטואליים וכל מה שקשור לזה, מאחר וכל המידע הזה כבר לא רלוונטי לנו.

פרוצדורות בתוך פרוצדורות

ישנן שפות כמו פסקל, בהם אנחנו יכולים להגדיר פונקציה בתוך פונקציה. הדבר הזה גורר לנו למעשה כמה רמות שונות ותת-רמות של הכרה. לצורך הדוגמא, נגדיר את התוכנית הפסקלית הבאה –

```
program p;
var x: Integer;
procedure a
          var y: Integer;
          procedure b begin…b… end;
          function c
                    var z: Integer;
                    procedure d begin…d… end;
                    begin…c…end;
          begin…a… end;
begin…p… end.
```

רק על מנת להבין את הרמות השונות נראה את זה ככה –



כל ריבוע תוחם מתח הכרה הרלוונטי אליו, כאשר a יכול לגשת ולקרוא לשני בניו, וכן b ו-c יכולים לקרוא אחד לשני.
באופן כללי יותר נגדיר זאת כך –

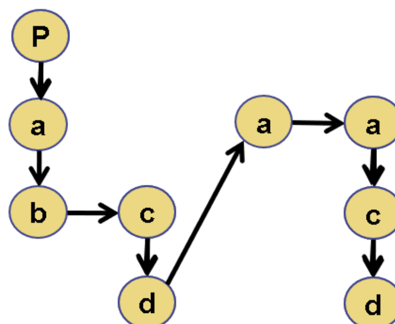
כל פרוצדורה יכולה לקרוא הן לעצמה, לילד, אח או אב קדמון כלשהו (כולל אב ישיר).

כך שאם נרצה נוכל לקבוע שסדר הקריאות הנ"ל הוא חוקי –

p→a→b→c→d→a→a→c→d

אפשר לעקוב בקלות ולראות שאין מניעה שקריאה כזאת תקרה. השאלה עכשיו היא כזאת, אנחנו נמצאים בפרוצדורה d ורוצים לגשת למשתנה y של a. איזה מהמשתנים אנחנו נקבל ביחס למופעים השונים של a?

ננסה לשרטט את הרמות השונות של ההכרה אליהם הגענו בגרף –



צריך גם כן לזכור שיש לנו עכשיו שלושה מופעים של a בתוך מחסנית זמן הריצה, ומן הסתם יכול להיות שהערכים בכל אחד מהם שונים אחד מהשני. מה שנעשה הוא כמובן לחפש את ה-y בתוך מופע a הקרוב ביותר. בשיעורי הבית בתרגיל 5, אנחנו רואים את טבלת השמות שהולכת ומתעדכנת עם כל גישה נוספת, אבל גם אם יהיו לנו שלוש מופעים של המשתנה y, אנחנו פשוט מחפשים את ההומונים הקרוב ביותר ומתייחסים אליו.

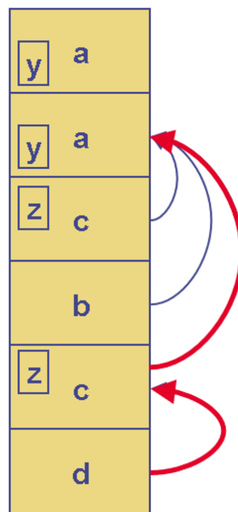
פרוצדורות בתוך פרוצדורות: שימוש ב-access link

מה שעשינו עד עכשיו התייחס לגישה למשתנים לפי סדר הקריאות של ה-control link, אבל מה קורה אם אנחנו צריכים לגשת למידע שקיים בהכרה העוטפת אבל רק באופן סטטי?

למשל, הפרוצדורה d מוגדרת אצלנו בתוך c. מה שזה אומר זה שאנחנו יכולים לגשת מתוך d ישירות למשתנים של c. אז כמו שכבר אמרנו אנחנו יכולים להכניס ב-access link את הגישה ל-c לא רק בתור הפונקציה שקראה לו, אלא גם בתור האב הסטטי שלו. אז אמנם פה זה לא יהיה משמעותי להחזיר שני מצביעים לאותו מקום, אבל יש מצבים שכן. נסתכל למשל על רשימת הקריאות הבאה –

a→a→c→b→c→d

קל לראות שאם אנחנו מתייחסים רק לקריאות ה"דינאמיות" יהיה לנו מאוד קשה לגשת למשתנים של a מתוך c השני, ואפילו הגישה מ-b לא כל כך פשוט. לכן בכל פעם שאנחנו מחשבים את ה-AR אנחנו עושים חישוב של המיקום האחרון בו נמצא האב הסטטי של הפרוצדורה ולשם נקשר את ה-access link. מה שייראה בעצם כך –



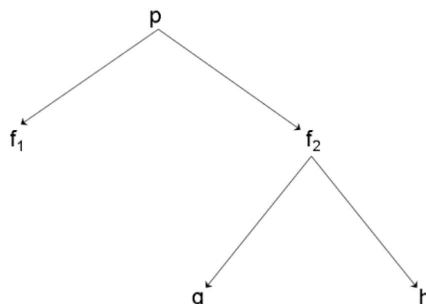
סביבת זמן-הריצה - מצגת דוגמאות מס' 8

לאור מה שלמדנו, נראה איך אנחנו פותרים ורצים על מחסנית **בזמן ריצה**. שימו לב, לא מדובר כאן בקימפול שקורה רק פעם אחת, אלא בריצה עצמה – מה מתרחש בכל פעם מחדש. לצורך הדוגמה נתון לנו קטע הקוד הבא -

```
function p() return float is
        a, b: float;
        function f(x, y: float) return float is
        begin
        …
        end;
        function f(x, y: integer) return float is
                function g(z: integer) return float is
                begin
                …
                end;
                function h(z: integer) return float is
                begin
                …
                end;
        begin
                return a * f(g(x), h(y));
        end;
begin
        a := 5.5;
        b := f(3, 5);
        return b;
end;
```

ניסיתי לסדר ולרווח את הפונקציות בצורה קצת יותר הגיונית מהפשע מלחמה שעשו לנו במצגת. למעשה, התוויות של begin,end משמשות נו כמו הסוגריים המסולסלות בשפות C שאנחנו מכירים, והן תוחמות את הפונקציה. את כל מה ששייך בתוך אותה פונקציה, הן הגדרות של פונקציות מקוננות והן פעולת הפונקציה בעצמה השתדלתי לעשות ריווח אחיד כך שנדע מה שייך למה.

עכשיו לפני שנתחיל, יש לנו שני עצים חשובים שאנחנו חייבים לצורך הריצה העץ הסטטי ועץ ההפעלה (הדינאמי) –



בעץ ההפעלה הסטטי אנחנו נרשום בצורה הפשוטה ביותר, מי מוגדר בתוך מי ברמת הקוד. מאחר ואנחנו מדברים פה על סטטיות, אנחנו מסתכלים על הקוד היבש רק ברמת הפונקציות, כל הקוד והפעולות עצמן לא רלוונטיות לנו בשלב הזה. P מגדיר לנו שתי פונקציות של f, ובתוך הפונקציה השניה יש לנו עוד שתיים אחרות – g,h.

עכשיו נסתכל על העץ הפעלה (activation tree). בחלק מהמקרים, למשל בתרגיל הבית, ישאלו אותנו לגבי הפעלה של קוד מסוים ממקום מסוים. במקרה שלנו, אנחנו פשוט מדברים על זה שהפעלנו את p, לצורך העניין מה-main. הקוד שירוץ בעצם ויפעיל לנו פונקציות שונות, זה השורה הבאה בתוך p–

```
b := f(3, 5);
```

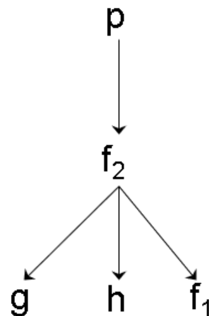
המשתנה b הוא real, מה שנותן לנו את שתי הפונקציות של f בתור מועמדים, אך מאחר ורק f_2 מקבלת מספרים שלמים, אנחנו מתייחסים אליה בתור הפונקציה הפועלת. בתוך f_2 יש לנו הגדרה של שתי פונקציות נוספות, ואת שורת הקוד –

```
return a * f(g(x), h(y));
```

לטובת המשך הפעולה, נזכור ש-a הוא משתנה שהוגדר ב-p, ואנחנו משתמשים בשתי הפונקציות הנוספות, או ליתר דיוק בערך המוחזר מהן, בתור ארגומנטים לפונקציה שנפעיל f_1. למה דווקא f_1? כי הפונקציות שלנו מחזירות real, וזה מה ש-f_1 מקבלת. מי שרוצה לעשות את העץ של תרגיל 6 שירד מהחומר[3] יגיע גם כן לאותה מסקנה.
אם כן, ננסה להבין בשורה הזאת את סדר ההפעלה – אמנם הפונקציה הראשונה שאנחנו רואים היא f, אבל כמובן שהיא תהיה האחרונה שאנחנו נפעיל, כי את הארגומנטים שלה אנחנו עדיין צריכים לחשב. לכן נפעיל קודם כל את הפונקציות האלה לפי סדר, נשמור כל ערך ברגיסטר מקומי כלשהו, ואז נוכל להפעיל את f. מבחינת קוד הביניים הוא ייראה בערך כך –

```
t1 := g(x)
t2 := h(y)
t3 := f1(t1, t2)
```

מה שיביא לנו את עץ ההפעלה הבא –

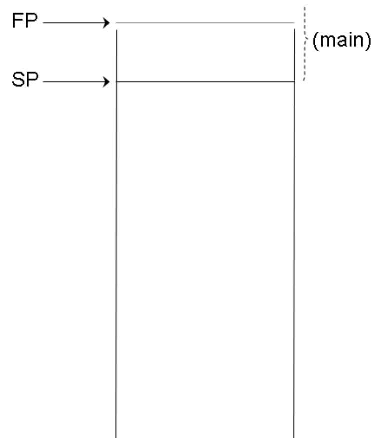


כלומר p קורא ל- f_2שהוא בתורו מפעיל לפי הסדר משמאל לימין את g,h,f_1. יכול להיות שגם הפונקציות האלה ממשיכות ונכנסות הלאה, אבל אין לנו מידע לגבי זה ולכן זה לא מעניין אותנו.

טיפ חשוב שיכול להביא לכם/ן 5 נקודות במבחן! אם ישאלו אותנו כמה רמות יש בעצים האלה, אנחנו צריכים לזכור שאנחנו סופרים כאן מ-1, ושורש העץ הוא לא 0. אמנם זה נגד כל אינסטינקט תכנותי שיש לנו, אבל אנחנו צריכים לזכור שקומה 0 קיימת בלי שנראה אותה – הלא היא רמת ה-main. אבל זה לא מופיע בעץ, ולכן יש לנו 3 רמות גם בהפעלה וגם בסטטי.

עכשיו נראה את הקוד בשפת הסף ואיך הוא משפיע על מה שקורה במחסנית. דיברנו כבר על כל עניין הפריימים והסיפור הזה, ועכשיו נראה איך זה עובד בפועל. לפני שאנחנו קוראים ל-p, אנחנו מכילים במחסנית את כל מה ש-main צריכה. זה ייראה באופן מאוד סכמטי ככה –

[3] נכון לשנת תש"ף



המצביעים של המחסנית והפריים יהיו כל אחד במקום מסוים, תלוי כמה ארגומנטים ועבודה יש לנו בתוך הפונקציה הראשית, ועכשיו אנחנו מגיעים לקריאה ל-p. את זה נעשה בצורה פחות סכמטית.

לפני שנתחיל את העבודה, אנחנו צריכים להסגר על כמה מוסכמות –

- את הארגומנטים לפונקציות הנקראות אנחנו מעבירים דרך המחסנית, ולא דרך שמירה לרגיסטרים בצד. כלומר בכל פעם שנצטרך להעביר ארגומנטים לפונקציה, נצטרך להעביר אותם בפועל ורק אז לקרוא לפונקציה.
- הערך החוזר (return value) יישמר ברגיסטר ייעודי – להלן RV.
- ה-FP (שייקרא בהמשך previous FP) מצביע על הערך של הפונקציה הקוראת. ליתר דיוק, על ה-FP של הפונקציה הזאת.
- מבחינת ניהול המחסנית, אנחנו עובדים עליה מלמטה למעלה. כלומר, אם יש לנו 1000 מקומות למחסנית, אנחנו נתחיל מערך 1000, או למעשה מ-996 כי כל אוגר לוקח 4 בתים, ונרד לכיוון ה-0. מה שיוצא מזה, הוא שאם נתייחס ל-FP הארגומנטים הנכנסים יהיו מעליו (כלומר להוסיף לו בשביל להגיע אליהם), והמשתנים הלוקאליים יהיו מתחתיו, כלומר להוריד מהמספר.

על מנת לקרוא לכל פונקציה יש מספר ערכים שחייבים להופיע באופן קבוע ולפי רצף מסודר –

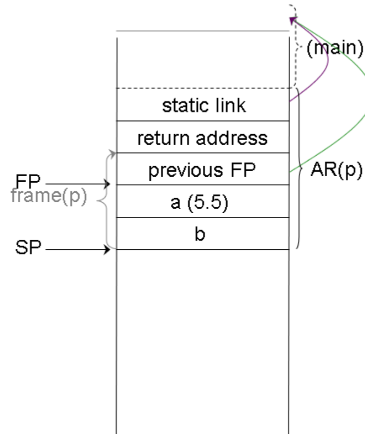
Static link – יכיל מצביע (כתובת) של האב הסטטי של אותה פונקציה, ההצבעה תהיה לתוך המחסנית כמובן (ל-FP) ולא לחלק חיצוני. טיפ לתרגיל הבית: כשנרצה להכניס את זה ברקורסיה, אז נוכל פשוט לשמור בפונקציה הנקראת את אותו הערך בדיוק של הפונקציה הקוראת, וזה יהיה 8 בתים מעל ה-FP הנוכחי.

Return address – המיקום בקוד בו עצרנו לטובת הקריאה לפונקציה הזאת, כאן אנחנו נשמור ממש את מספר השורה בקוד הסטטי.

Previous FP – או כמו שקראנו לו עד עכשיו ה-control link.

לפני שלושת אלה יהיו הארגומנטים הנכנסים (במידה ויש), ואחריהם יוכנסו, או לפחות נפנה מקום למשתנים מקומיים (במידה ויש).

לאור זאת, לאחר שנקרא ל-f המחסנית תראה כך –



בקריאה לפונקציה p אין לנו ארגומנטים נכנסים, ולכן אנחנו נתחיל ישר בשלשה, אבל יש משתנים מקומיים ואותם נכניס. עכשיו נסתכל איך אנחנו ממשים את הכל בקוד אסמבלי. לצורך זה רק נזכיר שמבחינת הפריים עצמו הוא לא שווה לרשומת ההפעלה של הפונקציה. נוכל ראות גם בתרשים למעלה, שה-FP שמסמל את תחילת הפריים נמצא רק אחרי הלינק הסטטי וכתובת החזרה, אותם ציינו שהפוקנציה הקוראת אחראית עליהם.

לכן הקוד יחולק לשלושה מקטעים:

1. Prologue – הקדמה. הכנה של השטח לכניסה לפונקציה.
2. Body – גוף הפונקציה.
3. Epilogue – סיכום. שיחרור הפונקציה והערכים ששמרנו.

להלן הקוד הרלוונטי עבור הפונקציה p -

```
      # Prologue:
1.    push    FP              # store previous value of FP
2.    mov     FP, SP          # copy current value of SP to FP
      # No saved registers
3.    sub     SP, SP, 8       # allocate memory for 'a' and 'b'
      # Body:
4.    loadc   RT1, 5.5        # temp. register
5.    store   RT1, -4(FP)     # a := 5.5
6.    pushc   5               # arg. 2 of f2
7.    pushc   3               # arg. 1 of f2
8.    push    FP              # static link for f2
9.    call    f2              # value returned by register RV
10. [ add     SP, SP, 12 ]    # free space allocated for arguments and static link;
11. [ store   RV, -8(FP) ]
      # Epilogue:
12.   mov     SP, FP          # SP now points to previous value of FP
13.   pop     FP              # restore previous value of FP
14.   ret                     # pop return address and assign it to program counter (PC)
# returned value is already located at the returned-value register
```

מספרתי את השורות רק בשביל שיהיה יותר קל לדבר על מה קורה פה. אז ככה – נתחיל בפרולוג – מה שאנחנו צריכים לזכור הוא שיש לנו כמה רגיסטרים שאנחנו עובדים עליהם באופן קבוע – RV,FP,FP ונראה בהמשך שנעבוד גם עם כמה רגיסטרים זמניים. אנחנו קודם כל שומרים את הלינק הסטטי, ומכיוון של-p אנחנו מגיעים דרך הפונקציה

הראשית, אז אנחנו פשוט דוחפים לראש המחסנית את הערך ששמור כבר ב-FP, שהוא המצביע ל-FP של ה-main. את האוגר של ה-return address אנחנו לא נוגעים כי המערכת דואגת לזה ולא אנחנו. ועכשיו אנחנו רוצים לקדם את ה-FP שיצביע לפונקציה הנוכחית, ולכן אנחנו מכניסים לרגיסטר הזה את הערך של המיקום של המחסנית באותו רגע. לאחר מכן אנחנו יודעים שיש לנו שני משתנים מקומיים לא מאותחלים בפונקציה, ולכן אנחנו רק מפנים להם מקום על ידי הורדה של ה-SP ב-8 בתים, מה שיספיק לנו לטובת המשתנים הדרושים.

עכשיו נדבר על גוף הפונקציה p, המכיל שלוש שורות –

```
a := 5.5;
b := f(3, 5);
return b;
```

קודם כל נדאג להשמה של הערך בתוך a. מאחר ואנחנו צריכים להכניס את זה למחסנית, אך אין לנו אפשרות לעשות את זה בצורה ישירה. אנחנו צריכים לשמור את הערך (הקבוע) לרגיסטר, ואז לתת מצביע במחסנית לאותו רגיסטר בו שמור הערך, כלומר מקום אחד (4 בתים) מתחת ל-FP, לכן נרשום את שתי הפקודות שבשורות 4-5.

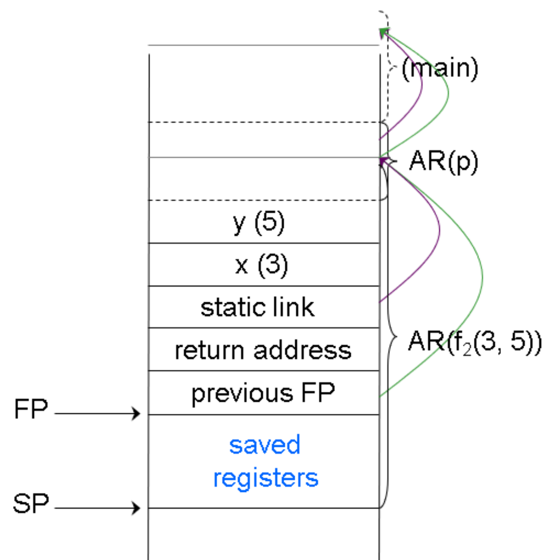
עכשיו נדאג שיהיה לנו ערך ל-b. בשביל זה אנחנו צריכים לקרוא ל-f. לטובת זה אנחנו דוחפים למחסנית את שני הקבועים שישמשו כארגומנטים ל-f (שורות 6-7), ולאחר מכן נכניס את ה-FP שיהיה הסטטיק לינק של f (שורה 8), ואז כשהכל מוכן, אנחנו יכולים לקרוא לפונקציה עצמה (שורה 9).

הפונקציה f פועלת ועושה משהו, אנחנו לא יודעים ולא אכפת לנו כל כך מה היא באמת עושה. מה שחשוב לנו הוא שבסיום העבודה שלה, יהיה לנו ערך שמור ב-RV שנוכל להשתמש בו.

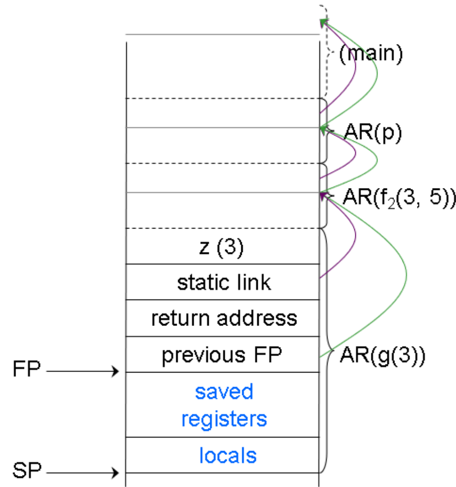
שתי השורות הבאות מצוינות במצגת בתור שורות מיותרות. למה? אנחנו שומרים את הערך המוחזר ב-RV ובמקביל מגדירים שהוא הערך של b שיוחזר גם הוא מתוך p. כלומר אותו ערך שחזר מ-f עובר ללא כל שינוי, ולכן שורה 10 שמזיזה את ה-SP ושורה 11 ששומרת את הערך המוחזר שלכאורה נמצא במחסנית בתור המשתנה b מיותרים כי אותו ערך כבר נמצא שם.

עכשיו נשאר האפילוג – שחרור הפונקציה p. קודם כל אנחנו נעביר את מצביע המחסנית להיות ב-FP כלומר, באב של p. מה קורה אם כל המידע שעכשיו נשאר אחרי המצביע? כפרת עוונות. אחרי זה אנחנו מעבירים את מה שנמצא בראש המחסנית שיעבור להיות ב-FP, כלומר מחזירים את המצב הקודם, ואז פקודת ret דואגת לשאר הדברים – החזרת ה-PC לשורה הנכונה בקוד מתוך ה-return address, ושימור הערך החוזר במידה ויש צורך.

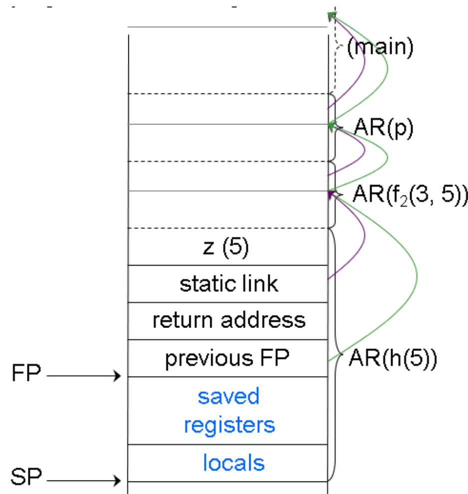
עכשיו נוכל לראות את רצף הפעולות שקורה במחסנית עצמה עם הקריאה לפונקציות השונות. קודם כל, הקריאה ל-f –



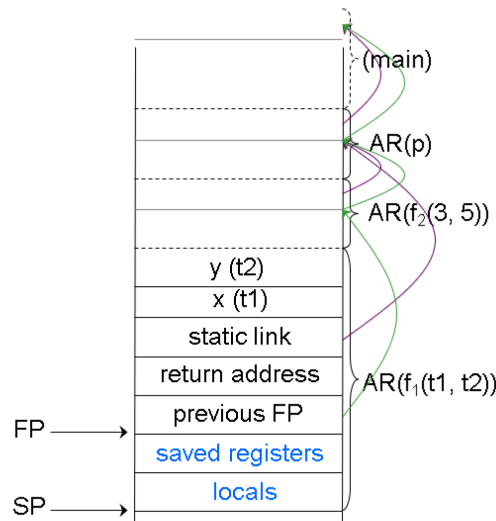
יש לנו כאן את הערכים שהכנסנו, והן הסטטיק לינק והקונטרול לינק מצביעים שניהם ל-p.



כמו שראינו קודם בעץ הקריאות, אנחנו עוברים מ-f לתוך g. כשנסיים עם g נוכל לפנות את המקום ולקרוא לפונקציה הבאה, h –



ושוב, נסיים, נפנה את המקום, ואז נקרא לפונקציה הבאה – f_1 –



כדאי לשים לב, שכאן זה המקום היחיד שהלינק הסטטי והדינאמי לא מצביעים לאותו מקום. מבחינת ה-control link אנחנו נמצאים בתוך f_1 רק בגלל ש-f_2 קראה לו, ולכן מבחינת השליטה אנחנו נצטרך לשמור קישור לחזרה לשם. אך אם נצטרך גישה למשתנים או מידע מפונקציית האב, אנחנו נצטרך להגיע ל-p, ונראה תיכף את הקוד של f_2 שם מאחר ואנחנו יודעים שמדובר על אותו אב סטטי, שהוא פשוט מעביר א המידע שלו לאותו מקום ב-f_1. נראה את הקוד של f, ואיך הוא קורא ומטפל בכל פונקציה בזמנה-

```
    # Prologue
1.  push    FP
2.  mov     FP, SP
3.  push    RS1
    # No locals
    # Body
    # Call g(x):
4.  load    RT1, 12(FP)     # load 'x'
5.  push    RT1             # arg. of 'g'
6.  push    FP              # static link, to frame of lexical parent of 'g', which is f2
7.  call    g
8.  mov     RS1, RV         # copy returned value to a saved register, RS1
9.  add     SP, SP, 8       # free space of arguments ('x' and static link)
    # Call h(y):
10. load    RT1, 16(FP)     # load 'y'
11. push    RT1             # arg. of 'h'
12. push    FP              # static link, as above
13. call    h
14. add     SP, SP, 8       # free space, as above
    # Call f1(g(x), h(y)):
15. push    RV              # value returned from h
16. push    RS1             # value returned from g
17. load    RS1, 8(FP)      # static link of f2, i.e. address of the frame of 'p'
18. push    RS1             # f1 and f2 have the same lexical parent, i.e. 'p'
19. call    f1
20. add     SP, SP, 12      # free space of 3 arguments
    # Compute a * f(g(x), h(y))
21. load    RT1, -4(RS1)    # load 'a'
22. mul     RV, RT1, RV     # RV := a * f1(g(x), h(y))
    # Epilogue
23. sub     SP, FP, 4       # SP now points to previous value of RS1
24. pop     RS1
25. pop     FP
26. ret
```

מבחינת האפילוג, אנחנו יודעים שאנחנו משתמשים באוגר שמור אחד – RS_1, ולכן אנחנו דואגים לדחוף אותו לתוך המחסנית לטובת השימוש העתידי (שורה 3), לפני זה אנחנו כמובן עושים את שתי השורות שהן האפילוג של כל פונקציה שלא תהיה אי פעם- דחיפה של FP למחסנית, ושמירה מחודשת של הערך שב-SP.

לאחר מכן, אנחנו מתחילים ישר עם הקריאה לפונקציות השונות, ובראשן g. הארגומנט המועבר ל-g הוא x שהתקבל כארגומנט בעצמו בראש f_2, ולכן בשורה 4 אנחנו טוענים לרגיסטר זמני את הערך הזה שנמצא ב-12(FP). כלומר, אנחנו מדלגים מעל ה-return address וה-static link ומגיעים לארגומנטים הנכנסים. יש לזכור שסדר הארגומנטים במחסנית הוא כזה – כל הרשימה תכנס לפי הסדר משמאל לימין, כאשר במחסנית הערך הראשון (השמאלי) יהיה

הקרוב ביותר ל-FP, כלומר (FP)12 וכל השאר יהיו מעליו בדילוגים של 4. כשנגיע לפונקציה הבאה שתרצה את y, במקום נוסיף 4 לבקשה הקודמת, מה שיהיה בעצם (FP)16. יש לזכור, שאנחנו קודם טוענים לרגיסטר ואז דוחפים למחסנית, ואין לנו דרך לעשות את זה ישירות. אחרי שהכנסנו את כל הארגומנטים הרלוונטיים, דוחפים את FP וקוראים לפונקציה (שורה 7).

כשנחזור מהפונקציה, נצטרך לטפל בערך המוחזר, לכן נעביר את הערך מהרגיסטר המוקצה לו RV, לתוך הרגיסטר השמור שהגדרנו בהתחלה RS_1, וכשנסיים את זה נוכל למחוק את כל הערכים שהכנסנו לטובת הפונקציה (RS, y) על ידי זה שפשוט נזיז את המצביע של המחסנית 8 בתים למעלה.

הקריאה לפונקציה הבאה עובדת באותו אופן, רק שאנחנו ניגשים למקום 12 מה-FP ולא 8, כמו שכבר ציינו. אבל נוכל לשים לב בחזרה מהפונקציה, שאנחנו לא מתעסקים עם RV, ולא מעבירים אותו לרגיסטר חדש. למעשה, הסיבה שעשינו את זה קודם, היה כי אנחנו יודעים שתיכף נקרא לפונקציה אחרת, מה שאומר שאנחנו נדרוס את RV, אז אנחנו דואגים לכך שהערך יהיה שמור. לעומת זאת, כאן אנחנו יכולים להשתמש ישירות מהרגיסטר.

בשביל לקרוא לפונקציה האחרונה בשרשרת f_1, אנחנו קודם כל מכניסים את הארגומנטים שלה – שני הערכים החוזרים מהפונקציות לפי הסדר הראוי (שמאלי-תחתון, ימני-עליון) מתוך הרגיסטרים בהם הם שמורים (שורות 15-16), ואז עלינו להכניס את הלינק הסטטי, שהוא בעצם אותו אחד ששמור לנו כבר, לכן אנחנו נטען את הכתובת לתוך הרגיסטר RS_1 שעכשיו יצא מהשימוש כאוגר לערך מוחזר, על ידי שימוש ב-(FP)8, ונכנסי אותו למחסנית (שורות 17-18) מה שייתן לנו את היכולת לקרוא לפונקציה f_1. לאחר שנסיים עם הפונקציה נשחרר את המקום ונוכל להתפנות לחישוב עצמו של f_2.

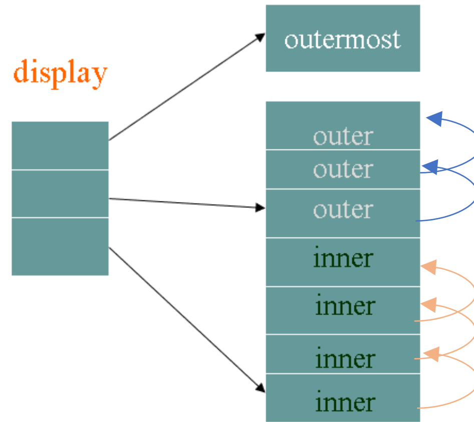
בשורה 21 אנחנו נטען לתוך רגיסטר זמני את a שנמצא בתוך p. בשביל זה אנחנו נלך ללינק הסטטי שאנחנו שמרנו לפני כמה שורות בתוך RS_1 ובו נרד 4 בתים בשביל להגיע למשתנים המקומיים שלו, ולטעון אותם לתוך הרגיסטר. עכשיו אנחנו יכולים לעשות את ההכפלה בעצמה ולשמור את התוצאה בתוך RV על מנת להחזיר הכל למעלה.

באפילוג, אנחנו פשוט נדאג לסגירת הפונקציה והחזרת הערכים לרגיסטרים בצורה הפוכה בדיוק למה שעשינו עד שנוכל לחזור מהפונקציה.

ניהול access link באמצעות מערך display

השיטה של ה-access link כמו שראינו קודם (עם control link) עלולה להיות מעט בעייתית וקשה למימוש – בכל פעם אנחנו צריכים להתחיל ולחפש מי האב וזה לא תמיד פשוט – בעיקר אם נרצה להתייחס לדורות קדמונים יותר. שיטה נוחה יותר למימוש מתייחסת ליצירה של מערך בשם display שכל איבר במערך מבטא רמת קינון אליה אנחנו יכולים להגיע.

בכל פעם שאנחנו ניכנס לרמת קינון חדשה (רק כדאי לזכור שאנחנו מדברים ברמה הסטטית, אז יכול להיות שאנחנו עולים ויורדים ברמות הקינון עם התקדמות התכנית), אנחנו נעביר את המצביע של אותה רמת קינון למופע החדש, כך שבעצם המערך תמיד יצביע על המופע האחרון של רמות הקינון ומשם נוכל ללכת ולחפש את המידע הרלוונטי. באופן כללי אפשר לראות את השרטוט הבא –

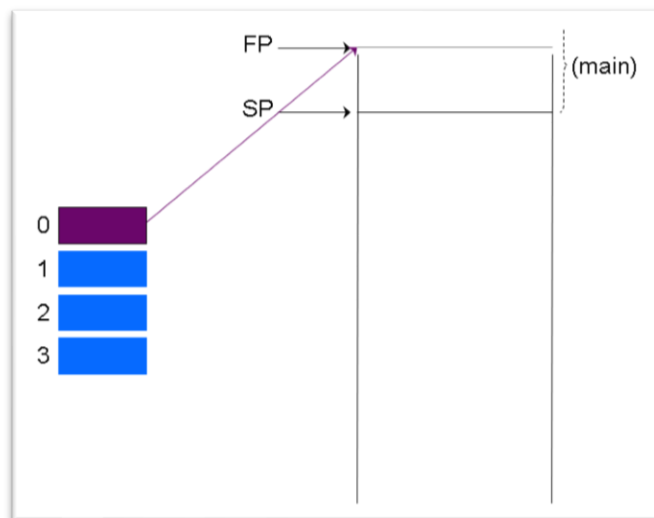


אם נרצה לחפש מידע שקשור ל-scope חיצוני יותר, אנחנו נוכל ללכת ישר למצביע של הרמה האחרונה ומשם להתחיל לחפש. בכל פעם שאנחנו נוסיף פרוצדורה חדשה הקשורה לאותה רמה, אנחנו נדאג קודם כל שהפרוצדורה הנכנסת תצביע לפרוצדורה שהיתה האחרונה עד כה, ורק אז נעביר את המצביע של המערך display לעבר הכניסה החדשה.

באופן הזה, הגישות שלנו יהיו יותר מהירות בתוך הרמות ולא נצטרך לעשות חיפושים ארוכים, אלא ישר לעבור דרך המצביע במערך.

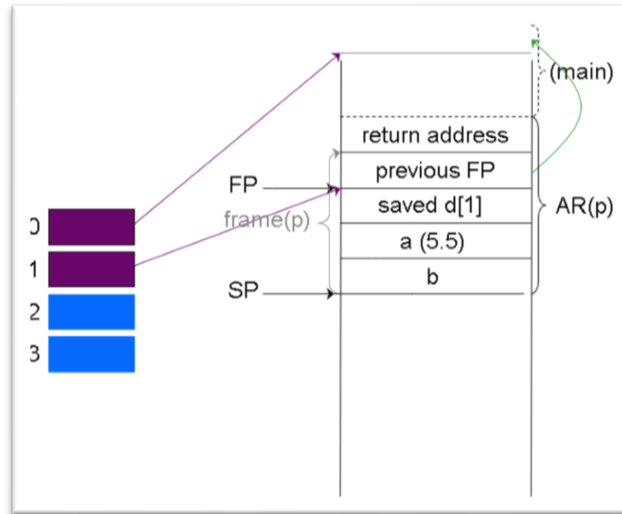
אתחול המערך יהיה כגודל הרמות הסטטיות הקיימות לנו, ובתוכו יהיו ערכי null. בכל פעם שאנחנו נבקש להצביע על רמה סטטית חדשה, אנחנו נשמור באותו AR את הערך שהיה שמור לפני כן אליו הצבענו עם המערך. במופע הראשון של כל רמה זה כמובן יהיה מצביע ל-null, אך עם התקדמות התוכנית אנחנו עלולים להגיע למצב בו יהיו לנו במחסנית שתי פונקציות מאותה רמה, ואז המערך יצביע על המופע האחרון, שהוא בתורו ישמור את הערך של המיקום של המופע מאותה רמה שהיה לפניו, וכך אם נחפש פונקציה מסוימת, נוכל לעבור בין המצביעים עד שנמצא אותו.

נסתכל על המחסנית עכשיו בהתאמה לקוד שהיה לנו ולמערך ה-display –

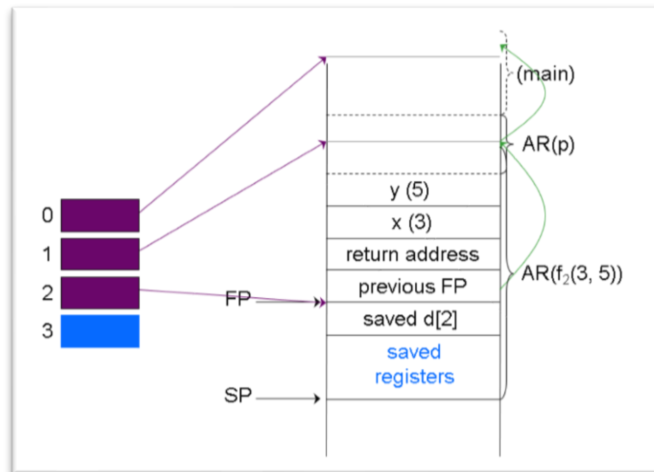


במצב הראשוני, יש לנו רק את d[0] שמצביע על המיקום של ה-main, כמובן שזה הדבר היחיד שלא ישתנה לאורך כל הריצה.

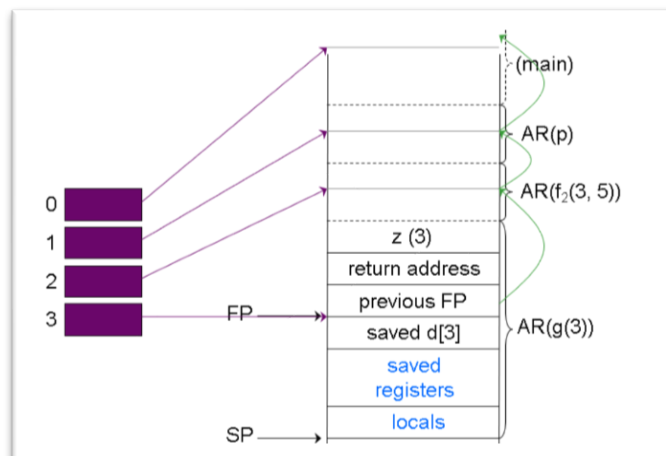
נקרא לפונקציה p ונראה את ההבדלים בגישות.



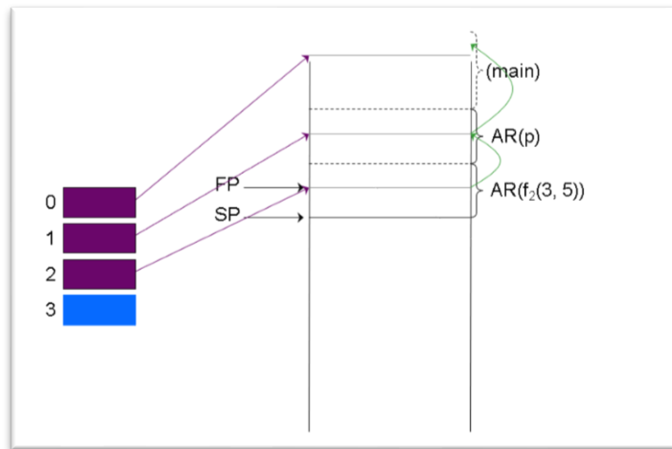
עכשיו, d[1] מצביע ל-p. כאשר במקום השדה של static link שהיה לנו בראש השלשה, אנחנו עכשיו עובדים מול שדה בשם saved d[1], וכמובן שהמספר ישתנה לפי הרמה, שישמור לנו את הערך הקודם שהיה במערך. במקרה שלנו כאמור – NULL.



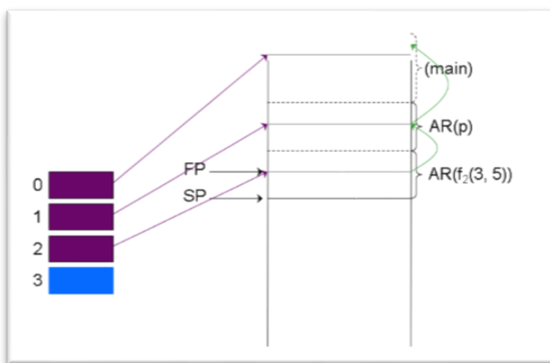
בהמשך אנחנו פותחים את הרמה הבאה, ואז מתחיל לקרוא לפונקציות f_2 שלו לפי הסדר –

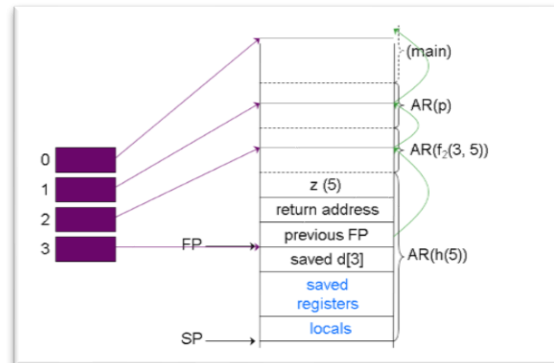


אנחנו לא מתעכבים פה מה קורה בדיוק בכל AR אלא מתבססים על זה שהוא די דומה למה שהיה קודם, מלבד השינויים שעשינו בשמירת הלינק הסטטי. בכל פעם שאנחנו מסיימים קריאה של פונקציה אנחנו משחררים את המחסנית, ובהתאמה גם את השמירה במערך –

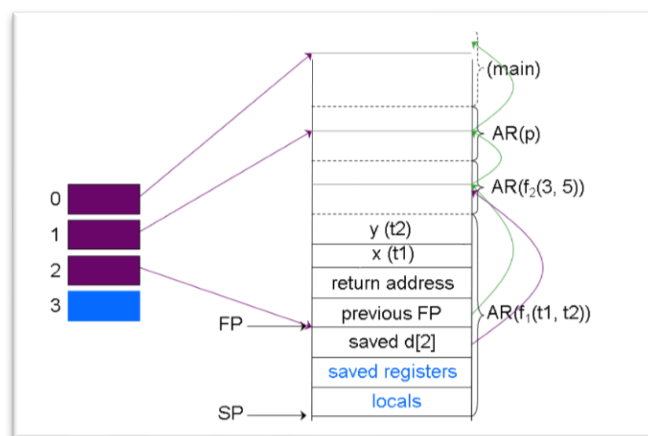


אנחנו לא רואים שום סיבה לשמור ערכים מתים במערך, ולכן פונקציה שנסגרת יוצאת גם משם. אותו סידור עובד גם לפונקציה הבאה –





מכניסים למחסנית/מערך ויש מוציאים את המידע המיותר. בשלב הבא, נכניס את f_1 שהיא כבר באותה רמה של f_2, ולכן המצב יהיה קצת שונה –



בעוד ש-d[2] יצביע אל ה-FP של f_1, הוא בעצמו יצביע כלפי מעלה אל f_2 למקרה ונחפש אותו. מכאן והלאה אנחנו משחררים הכל, ואין צורך להראות את זה, תעצמו את העיניים חזק ותדמיינו את זה קורה.

יצירת הקוד

עד עכשיו התעסקנו עם דברים שהם יותר ברמת הפרונט, גם אם דיברנו על יצירת קוד ביניים, זה היה ברמה פשוטה יחסית, עכשיו אנחנו רוצים להכנס קצת יותר מאחורה ולראות מה אנחנו יכולים לעשות. באילו דברים אנחנו רוצים לגעת עכשיו? טיפול ברגיסטרים, אופטימיזציה של קוד ברמת השמות מיותרות, קטעי קוד מתים, משתנים מיותרים ועוד.

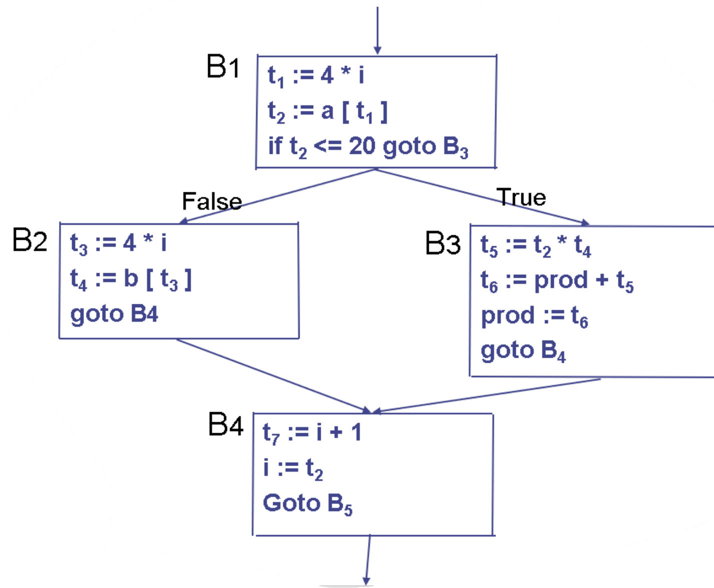
בתור התחלה נדבר על הרגיסטרים בהם אנחנו משתמשי לגישה מהירה לנתונים. ככל שאנחנו נייעל את השימוש ברגיסטרים, כך ריצת התוכנית עצמה תהיה יותר יעילה – כזכור לנו מקורסים אחרים, אם בעבור כל בקשה של מידע מהזיכרון נצטרך לגשת ולאסוף את המידע מחדש זה ייקח לנו המון זמן, לכן אנחנו דואגים לשמור מידע על רגיסטרים קרובים ופשוט לשלוף משם כשנצטרך. ברגע שכל הרגיסטרים יהיו מלאים אנחנו נצטרך למצוא איזה שיטה לפנות את המידע בצורה יעילה, כזאת שלא תגרום לנו לפנות ולהכניס מידע יותר מידי, וכל הדברים האלו והאלגוריתמים בהם משתמשים פורטו בקורס מערכות הפעלה. בגדול, אם יש לנו 2 רגיסטרים, ובאחד מהם יש ערך שאנחנו נשתמש בו רק בעוד 100 שורות, אין לנו שום צורך לשמור את המידע הזה כל כך הרבה זמן, השאיפה שלנו הוא שבזמן הקימפול, אנחנו נכל להכריע כמה שאפשר איזה מידע לשמור באיזה רגיסטר ולכמה זמן.

כשאנחנו באים לעשות אופטימיזציה כזאת, אנחנו יכולים לקחת את כל התכנית בעצמה ולהתחיל לבדוק שורה שורה ולראות את המקומות שאנחנו יכולים לייעל. הבעיה עם גישה כזאת, היא שתכניות בדרך כלל ארוכות מאוד, ואם ייתנו לנו לנסות לעשות אופטימיזציה ככה על עשרות אלפי שורות, אנחנו לא נגיע רחוק. בעיקר בהתחשב בעובדה שתמיד אפשר לייעל עוד קצת, וכמעט אף פעם לא נגיע לתוכנית מושלמת. לכן מה שאנחנו עושים הוא חלוקה של הקוד לבלוקים, והתעסקות עם כל בלוק בפני עצמו. אמנם במבט הכללי זה פחות טוב מאופטימיזציה של כל הקוד, אבל זה יותר אפשרי ופשוט.

איך נגדיר בלוק? יש לנו מספר פרמטרים, שברגע שנעבור על הקוד במספר איטרציות, אנחנו נוכל לחלק את המקטעים השונים ולהוציא את הבלוקים הבסיסיים. ככלל, אנחנו מגדירים בלוק כמקטע קוד עליו אנחנו יכולים להיות בטוחים שנרוץ מתחילתו ועד סופו ללא הפסקות או קפיצות. מה התועלת בקביעה הזאת? אם אנחנו יודעים שאנחנו בכל רגע נתון עוברים על מקטע קוד מסוים ולא יהיו לנו הפתעות, אנחנו נוכל יותר בוודאות לעשות אופטימיזציה יעילה. לצורך העניין, אם יש לנו חלק בקוד עם לולאה, ואנחנו מייעלים את הקוד שמתבצע בכל פעם, אז אולי עבדנו על חלק מאוד מסוים, אבל בזמן הריצה אולי ייעלנו לנו קטע שנעבור עליו עשרים פעמים, וכל ייעול משפיע הרבה יותר.

חשוב לזכור – לכל בלוק יש רק כניסה אחת ורק יציאה, זה נגזר ממה שאמרנו קודם על כך שאנחנו רוצים לוודא שאנחנו רצים על הקוד מתחילתו ועד סופו. לכן, ברגע שיש לנו יותר מכניסה אחת לקוד, אנחנו קצת פוגמים לנו באפשרות לייעל, כי אולי מה שאנחנו עושים מתאים לקטע הקוד הגדול, אבל רוב הכניסות יהיו מאוחר יותר ופחות יעילות. באותו אופן, יש לנו רק יציאה אחת, על מנת לוודא שלא נקפוץ מוקדם מידי ולא נגיע לחלק היעיל.

כשאנחנו נחלק את הקוד לבלוקים, אנחנו נמספר את הקוד מחדש לכל בלוק ונתייחס אליו לפי שמו $B_1..B_n$. דבר כזה יוכל ליצור לנו מעין תרשימי זרימה של הקוד בעצמו. נוכל לראות דוגמה של חלוקה לבלוקים בתרשים הבא –



כאן בעצם יש לנו קוד של if-else – יש לנו תנאי ולו שני יעדי קפיצה, שכל אחד מהם קופץ בסיומו לבלוק שממשיך את הקוד. איך אנחנו מחליטים את אופן החלוקה? יש לנו אלגוריתם שבעזרת מספר מעברים מצומצם על הקוד, יניב לנו את החלוקה הפשוטה. האלגוריתם יגדיר לנו מובילים (leader) בקוד, כאשר כל בלוק יהיה מלידר מסוים עד הלידר הבא (לא כולל):

1. השורה הראשונה בקוד תוגד "לידר".
2. כל שורה שהיא **מטרת קפיצה** בקוד (מותנית או לא מותנית) תחשב לידר – בעצם כל מטרת קפיצה היא כניסה מסוימת למקטע קוד, ועל פי מה שאמרנו זה ההגדרה לתחילת בלוק.
3. כל שורה שמופיע **אחרי פקודת קפיצה** תחשב "לידר" – זה נובע מכלל ה"יציאה אחת". ברגע שאנחנו קופצים, הגדרנו יציאה מהמקטע, ולכן השורה הבאה תהיה ראש הקטע הבא.

אם ניקח את הציור מעלינו, השורה השלישית ב-B_1 הגדירה לנו קפיצה לשורה מסוימת (לפני שאנחנו מחלקים לבלוקים, פקודות הקפיצה מופיעות כקפיצה למספר שורה, רק אחרי זה אנחנו משנים את זה). אנחנו עוברים לאותה שורה ומגדירים כבלוק (אך כרגע ללא מספור). השורה שאחרי הקפיצה המותנית ייחשב כבלוק הבא, ואם נמשיך לקרוא את B_2 ולאחרי זה גם את B_3, אנחנו נראה ששניהם מקפיצים אותנו לאותה שורה, שם נגדיר את הבלוק הבא.

Flow Graph

לאחר שאנחנו מגדירים את הבלוקים הבסיסיים אנחנו יוצרים לנו גרף זרימה. הגרף יבטא לנו את המעבר בין המקטעים כאשר צמתי הגרף יהיו הבלוקים השונים, והקשתות יוגדרו בין הבלוקים כאשר יש לנו מעבר (ישיר או קפיצה) לבלוק אחר.

לאחר שנגדיר את הגרף אנחנו נוכל להגיע למצב של לולאות, להם יש לנו שני סוגים שונים –

- לולאה פנימית – קטע קוד שחוזר על עצמו (קפיצה מותנית שחוזרת לראש הבלוק).
- לולאה חיצונית – מספר בלוקים שמהווים קבוצה קשירה היטב. כלומר, שמכל בלוק אנחנו יכולים לעשות סיבוב ולהגיע לכל בלוק אחר.

לצורך הבנת התהליך של חלוקת הקוד, נסתכל על מקטע קוד, ונעבור עליו על פי האלגוריתם.

1. i = 1
2. j = 1

3. $t_1 = 10 * 1$
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. $t_5 = i - 1$
14. $t_6 = 88 * t_5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

(רק בשביל הנוחות, אני אצבע כל מקטע בצבע אחר, וככה נראה את החלוקה)

בשלב הראשון, אנחנו לוקחים החל מהשורה הראשונה ומחליטים שיש לנו כאן בלוק גדול –

1. i = 1
2. j = 1
3. $t_1 = 10 * 1$
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. $t_5 = i - 1$
14. $t_6 = 88 * t_5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

עכשיו אנחנו מתחילים לרדת בקוד, עד שאנחנו מגיעים לשורה 9, שם יש לנו קפיצה מותנית. הקפיצה מביאה לנו שני בלוקים – החל משורה 3, לשם אנחנו קופצים, והחל משורה 10, שהיא השורה הבאה בקוד –

1. i = 1
2. j = 1
3. $t_1 = 10 * 1$
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)

10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. $t_5 = i - 1$
14. $t_6 = 88 * t_5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

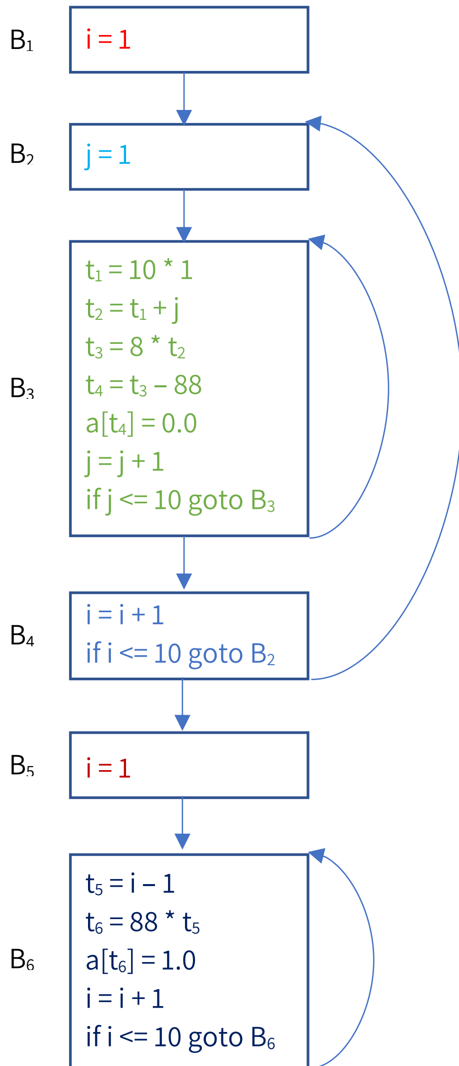
נמשיך. אנחנו מגיעים לשורה 11, שם יש לנו קפיצה לשורה 2. שוב חלוקה לשני בלוקים חדשים –

1. i = 1
2. j = 1
3. $t_1 = 10 * 1$
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. $t_5 = i - 1$
14. $t_6 = 88 * t_5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

ממשיכים עד סוף הבלוק – שם יש לנו קפיצה לשורה 13. אך הפעם יש לנו רק בלוק אחד חדש להכניס, מאחר שאין לנו שורה שעוקבת אחרי הקפיצה. אם כן, הבלוקים יחולקו על פי מה שמופיע עכשיו –

1. i = 1
2. j = 1
3. $t_1 = 10 * 1$
4. $t_2 = t_1 + j$
5. $t_3 = 8 * t_2$
6. $t_4 = t_3 - 88$
7. $a[t_4] = 0.0$
8. j = j + 1
9. if j <= 10 goto (3)
10. i = i + 1
11. if i <= 10 goto (2)
12. i = 1
13. $t_5 = i - 1$
14. $t_6 = 88 * t_5$
15. $a[t_6] = 1.0$
16. i = i + 1
17. if i <= 10 goto (13)

כל שנותר, הוא להגדיר את גרף הזרימה של הבלוק. נסמן חיצים שיעברו בין החלקים השונים, ואת יעדי הקפיצה להגדיר בתור בלוקים ולא מספרי שורות –



אופטימיזציות

עכשיו אחרי שחילקנו את הבלוקים, אנחנו יכולים לעבוד בצורה מקומית ולעשות אופטימיזציות על כל בלוק נתון בנפרד. במעין פשרה, אנחנו טוענים כי למעשה תכנית היא אוסף של פרוצדורות, ולכן הדרך היעילה באמת אינה לעשות את כל הקוד כאחדבדלל שזה יותר מידי עבודה, אך גם לא להתייחס לכל בלוק בפני עצמו כי השינויים יהיו מקומיים מידי, ולכן הדרך הנכונה ביותר היא לעשות אופטימיזציה לאוסף בלוקים שבתוך פרוצדורה.

טיפול במשתנים

הקצאת רגיסטרים למשתנים היא חלק קריטי בכל ייעול של קוד, וכבר הסברנו קודם את חשיבות העניין. עקרונית, נרצה שבעבור כל משתנה בוא אנחנו משתמשים נוכל להקצות רגיסטר לעבודה, אך לא תמיד כמות הרגיסטרים תואמת את המשתנים המופיעים בתכנית. לכן אנחנו נתייחס לכל משתנה בתור "חי" או "מת" בכל נקודה נתונה.

כאשר מופיעה לנו שורת קוד כדוגמת $x := y + z$[4], אנחנו למעשה עשינו כאן שתי פעולות – 1. הגדרנו את x (הכנסנו לתוכו ערך מסוים) 2. השתמשנו ב-y,z (עשינו שימוש בערכים שהיו מוחזקים בהם).

האמירה הזאת למעשה אומרת לנו כמה דברים – קודם כל, אנחנו לא יודעים מה x היה עד עכשיו, ולמעשה זה גם לא משנה לנו. ברגע שאנחנו עושים השמה לתוך המשתנה x, אנחנו יכולים להתעלם מכל מידע שהיה בו קודם כי אנחנו שמים בו עכשיו ערך חדש. בנוסף, אם נביט על y,z נבין ששני המשתנים האלו מאוד רלוונטים לנו, ואנחנו צריכים לוודא ששני המשתנים האלו יכילו את הערכים הנכונים ויהיו זמינים לנו כשנגיע לשורה הנוכחית.

לאור זאת נאמר, כי המשתנה אותו הגדרנו ייחשב כ"מת" (ביחס לקטע הקוד לפני), והמשתנים בהם אנחנו משתמשים הינם משתמשים "חיים" – אם הם הוגדרו בשורה 3, ואנחנו עכשיו בשורה 20, אנחנו צריכים לדאוג שהמידע הזה יישאר רציף.

על מנת לדעת מי חי ומי ומת, מי בקיצו ומי לא בקיצו, נרצה להשתמש ב-

טבלת ה-Next Use

כאמור, נרצה לדעת בכל רע נתון מה מצב המשתנים בהם אנחנו משתמשים. לכן אנחנו ניצור טבלה שכל שורה שלה תהיה מקבילה לשורת קוד, ובה נכתוב האם יש לנו שימוש בעתיד עבור משתנה נתון (כלומר אם הוא חי), ובמידה ויש שימוש, נרשום לנו גם את מספר השורה שנעשה בה שימוש.

על מנת למלא את הטבלה, אנחנו נעבור על הקוד מהשורה האחרונה כלפי מעלה, על מנת שברגע שניתקל במשנה כלשהו נוכל לדעת אם כבר ראינו אותו קודם (וזה אומר שיהיה בו שימוש כלשהו בהמשך).

אנחנו גם נבדיל בתהליך העבודה בין משתנים מקומיים לבלוק לאילו שאינם כאלה – משתנים שאינם מקומיים, כאלה שנצטרך לכאורה גם בבלוקים אחרים יהיו חיים בסוף הריצה על הבלוק, על מנת לוודא שהמידע שלהם יעבור בצורה טובה. כך שבתחילת העבודה נסמן לנו בצד את כל המשתנים הלא-מקומיים ואת כל הקבועים בהם נשתמש בתור L, ואז נתחיל את העבודה.

סדר הפעולות יהיה כדלקמן –

עבור כל שורה $x := y$ op z .i-

1. נצמיד ל- i את האינפורמציה שנאספה בטבלת הסמלים על y, x ו- z עד עתה – בשורה של אותו מספר ב-next-use, אנחנו נכניס את הערך הנוכחי של מה שאנחנו שמרנו בעבר – בפעם הראשונה, נגדיר אותם כ-L כמו שעשינו באתחול, ובשאר הפעמים נכניס את המידע ששמור לנו.
2. נקבע בטבלת הסמלים (עבור הפקודות המוקדמות יותר בבלוק): x – "לא חי"; "אין שימוש בהמשך" – כשנגיע בהמשך לאותו משתנה x במעלה הקוד, אנחנו נדע שבהמשך אנחנו דורסים את הנתונים הקיימים בו, ולכן מבחינתנו x מת מרגע השימוש האחרון בו ועד לשורה זאת.
3. נקבע בטבלת הסמלים: y, z – "חיים"; "השימוש הבא – ב- i" – אם נגיד בשורה 20 יש לנו שימוש במשתנים, אנחנו צריכים לזכור את המידע שיש בהם לפחות עד לשם, ולכן בטבלת המשתנים אנחנו נרשום שאנחנו צריכים שהוא יהיה חי לשימוש בשורה 20, או איך שאנחנו מעדיפים להגדיר – L20.

חשוב לשים לב שסדר הפעולות פה הוא קריטי – אם אנחנו נעדכן את המשתנים לפני שנשמור ב-next use את מה שהיה שם לפני המידע לא יהיה נכון. כמו כן, אם אנחנו נעשה את הפעולה השלישית לפני השניה, אנחנו עלולים שוב להכניס מידע שגוי. אם יש לנו שורה $x := X + 1$, ואנחנו קודם כל נעשה את 3 ואז את 2, אנחנו נסיים אותו עם D, למרות שאנחנו כן נצטרך שהוא יהיה חי כי הוא רק מעדכן את עצמו ולא דורס לגמרי.

---

[4] אנחנו נתייחס בהמשך גם לצורה הזאת כשנדבר על הקצאת הרגיסטרים באופן כללי ש-x יהיה המשתנה אליו עושים את ההשמה, ו-y.z הם המשתנים עליהם אנחנו עובדים.

הפקת קוד משופרת: גושים בסיסיים - מצגת דוגמאות מס' 9 חלק א'

על מנת להבין את העבודה עם טבלת ה-next use, אנחנו מדלגים כרגע למצגת התרגול, נעבור על הדוגמא המוצגת, ואז נחזור למהלך החומר הרגיל.

אנחנו מתחילים לעבוד על קטע הקוד הבא –

```
mul     T1, 4, j          # T1 := 4 * j
aelm    T2, a, T1         # T2 := a (T1)
mul     T3, 4, j          # T3 := 4 * j
aelm    T4, b, T3         # T4 := b (T3)
mul     T5, T2, T4        # T5 := T2 * T4
add     T6, prod, T5      # T6 := prod + T5
asgn    prod, T6          # prod := T6
add     T7, j, 1          # T7 := j + 1
asgn    j, T7             # j := T7
ble     j, 20, start      # If j<=20 goto start
```

ההערות מימין מייצגות את קוד הביניים, אך אנחנו עובדים למעשה על קוד המכונה – המילה הראשונה היא הפעולה המתאימה, והשלשה שאחריה מיוצגת בתור Destination, Source1, Source2. עכשיו נבנה את הטבלאות בהם נשתמש – ראשית ניצור את טבלת ה-next use שתהיה בצמוד לקוד הנתון שורה אל מול שורה, ואחרי זה ניצור את טבלאות הסמלים של המשתנים, שם נשמור את המידע הרלוונטי. בתור התחלה, אנחנו נגדיר את המשתנים הלא מקומיים ואת המספרים הקבועים כחיים, ואת המקומיים כמתים (כי אין לנו בהם צורך אחרי היציאה מהבלוק) –

	L\D	Next use
j	L	
a	L	
b	L	
prod	L	
T1	D	
T2	D	
T3	D	
T4	D	
T5	D	
T6	D	
T7	D	

		Dst	S1	S2
1	mul T1, 4, j			
2	aelm T2, a, T1			
3	mul T3, 4, j			
4	aelm T4, b, T3			
5	mul T5, T2, T4			
6	add T6, prod, T5			
7	asgn prod, T6			
8	add T7, j, 1			
9	asgn j, T7			
10	ble j, 20, start			

טבלת הקבועים		
	L\D	Next use
1	L	
4	L	
20	L	

עכשיו נתחיל לעבוד על שורה 10 – נעביר את המידע הקיים בטבלת הסמלים, ונשמור את הערכים שמשתמשים בהם כחיים (מדובר פה על קפיצה לתווית start ולכן אין פה משתנה יעד לעדכון –

	L\D	Next use
j	**L**	**10**
a	L	
b	L	
prod	L	
T1	D	
T2	D	
T3	D	
T4	D	
T5	D	
T6	D	
T7	D	

			D st	S 1	S 2
1	mul	T1, 4, j			
2	aelm	T2, a, T1			
3	mul	T3, 4, j			
4	aelm	T4, b, T3			
5	mul	T5, T2, T4			
6	add	T6, prod, T5			
7	asgn	prod, T6			
8	add	T7, j, 1			
9	asgn	j, T7			
10	**ble**	**start ,j, 20**		**L**	**L**

טבלת הקבועים

	L\D	Next use
1	L	
4	L	
20	**L**	**10**

כדאי לשים לב, שבניגוד לכל המשתנים הקיימים לנו כאן, אלו שאנחנו נוגעים בהם בפועל, אלו היחידים שלא יוגדרו רק חיים באופן כללי, אלא נכניס גם את מספר השורה שראינו אותם. נעבור כעת לשורה הבאה – השמה של T7 בתוך j – נזכור – הכנסה של המצב הקיים, עדכון היעד כ"מת", עדכון המשתנה שנעשה בו שימוש כ"חי" עם מספר השורה –

	L\D	Next use
j	**D**	
a	L	
b	L	
prod	L	
T1	D	
T2	D	
T3	D	
T4	D	
T5	D	
T6	D	
T7	**L**	**9**

			Dst	S1	S2
1	mul	T1, 4, j			
2	aelm	T2, a, T1			
3	mul	T3, 4, j			
4	aelm	T4, b, T3			
5	mul	T5, T2, T4			
6	add	T6, prod, T5			
7	asgn	prod, T6			
8	add	T7, j, 1			
9	**asgn**	**j, T7**	**L10**	**D**	
10	ble	start ,j, 20		L	L

טבלת הקבועים

	L\D	Next use
1	L	
4	L	
20	L	10

עוברים לשורה 8 – השמה של j+1 לתוך T7 – עדכון הטבלה, עדכון משתנה ההשמה, עדכון משתני השימוש –

טבלת הקבועים

	L\D	Next use
1	**L**	**8**
4	L	
20	L	10

			Dst	S1	S2
1	mul	T1, 4, j			
2	aelm	T2, a, T1			
3	mul	T3, 4, j			
4	aelm	T4, b, T3			
5	mul	T5, T2, T4			
6	add	T6, prod, T5			
7	asgn	prod, T6			
8	**add**	**T7, j, 1**	**L9**	**D**	**L**
9	asgn	j, T7	L10	D	
10	ble	start ,j, 20		L	L

	L\D	Next use
j	**L**	**8**
a	L	
b	L	
prod	L	
T1	D	
T2	D	
T3	D	
T4	D	
T5	D	
T6	D	
T7	**D**	

אם משום מה לא שמתם לב עד עכשיו, אנחנו משתמשים באותה טלה ובכל פעם מעדכנים אותה, כך שבסוף כשישאלו אותנו על משתנה מסוים האם הוא חי בשורה מסוימת, נצטרך לבדוק פשוט את המופעים שלו והאם הוא חי או מת על השורות בסביבה וכך לדעת.

נמשיך לשורה 7 -

טבלת הקבועים

	L\D	Next use
1	L	8
4	L	
20	L	10

			Dst	S1	S2
1	mul	T1, 4, j			
2	aelm	T2, a, T1			
3	mul	T3, 4, j			
4	aelm	T4, b, T3			
5	mul	T5, T2, T4			
6	add	T6, prod, T5			
7	**asgn**	**prod, T6**	**L**	**D**	
8	add	T7, j, 1	L9	D	L
9	asgn	j, T7	L10	D	
10	ble	start ,j, 20		L	L

	L\D	Next use
j	L	8
a	L	
b	L	
prod	**D**	
T1	D	
T2	D	
T3	D	
T4	D	
T5	D	
T6	**L**	**7**
T7	D	

מי שהצליח לעקוב עד כאן, יכול מכאן והלאה לרוץ די מהר – עדכון הטבלה, עדכון ההשמה, עדכון משתני השימוש. אני אתחיל לרוץ פה מהר יותר. מי שלא הבין, שיחזור אחורה לשורות הקודמות.

	L\D	Next use
j	L	8
a	L	
b	L	
prod	**L**	**6**
T1	D	
T2	D	
T3	D	
T4	D	
T5	**L**	**6**
T6	D	
T7	D	

			Dst	S1	S2
1	mul	T1, 4, j			
2	aelm	T2, a, T1			
3	mul	T3, 4, j			
4	aelm	T4, b, T3			
5	mul	T5, T2, T4			
6	**add**	**T6, prod, T5**	**L7**	**D**	**D**
7	asgn	prod, T6	L	D	
8	add	T7, j, 1	L9	D	L
9	asgn	j, T7	L10	D	
10	ble	start ,j, 20		L	L

טבלת הקבועים		
	L\D	Next use
1	L	8
4	L	
20	L	10

נמשיך לשורה 5 –

	L\D	Next use
j	L	8
a	L	
b	L	
prod	L	6
T1	D	
T2	**L**	**5**
T3	D	
T4	**L**	**5**
T5	**D**	
T6	D	
T7	D	

			Dst	S1	S2
1	mul	T1, 4, j			
2	aelm	T2, a, T1			
3	mul	T3, 4, j			
4	aelm	T4, b, T3			
5	**mul**	**T5, T2, T4**	**L6**	**D**	**D**
6	add	T6, prod, T5	L7	D	D
7	asgn	prod, T6	L	D	
8	add	T7, j, 1	L9	D	L
9	asgn	j, T7	L10	D	
10	ble	start ,j, 20		L	L

טבלת הקבועים		
	L\D	Next use
1	L	8
4	L	
20	L	10

נמשיך –

טבלת הקבועים		
	L\D	Next use
1	L	8
4	L	
20	L	10

			Dst	S1	S2
1	mul	T1, 4, j			
2	aelm	T2, a, T1			
3	mul	T3, 4, j			
4	**aelm**	**T4, b, T3**	**L5**	**L**	**D**
5	mul	T5, T2, T4	L6	D	D
6	add	T6, prod, T5	L7	D	D
7	asgn	prod, T6	L	D	
8	add	T7, j, 1	L9	D	L
9	asgn	j, T7	L10	D	
10	ble	start ,j, 20		L	L

	L\D	Next use
j	L	8
a	L	
b	**L**	**4**
prod	L	6
T1	D	
T2	L	5
T3	**L**	**4**
T4	**D**	**5**
T5	D	
T6	D	
T7	D	

הלאה. אין בכל אלו מה להתעכב, נמשיך –

טבלת הקבועים		
	L\D	Next use
1	L	8
4	**L**	**3**
20	L	10

			Dst	S1	S2
1	mul	T1, 4, j			
2	aelm	T2, a, T1			
3	**mul**	**T3, 4, j**	**L4**	**L**	**L8**
4	aelm	T4, b, T3	L5	L	D
5	mul	T5, T2, T4	L6	D	D
6	add	T6, prod, T5	L7	D	D
7	asgn	prod, T6	L	D	
8	add	T7, j, 1	L9	D	L
9	asgn	j, T7	L10	D	
10	ble	start ,j, 20		L	L

	L\D	Next use
j	**L**	**3**
a	L	
b	L	4
prod	L	6
T1	D	
T2	L	5
T3	**D**	
T4	D	5
T5	D	
T6	D	
T7	D	

שימו לב, שאמנם j כבר הוגדר לנו כ"חי" עד שורה 8, אנחנו חוזרים ומעדכנים אותו כ-L3 מאחר ואנחנו מתעסקים פה לא בשאלה כמה רחוק הוא יגיע, אלא מהו השימוש הבא של אותו משתנה, לכן אנחנו בכל פעם שנראה מופע שלו ישר נרים לו את החיות מחדש.

שורה 2-

	L\D	Next use
j	L	3
a	**L**	**2**
b	L	4
prod	L	6
T1	**L**	**2**
T2	**D**	
T3	D	
T4	D	5
T5	D	
T6	D	
T7	D	

			Dst	S1	S2
1	mul	T1, 4, j			
2	**aelm**	**T2, a, T1**	**L5**	**L**	**D**
3	mul	T3, 4, j	L4	L	L8
4	aelm	T4, b, T3	L5	L	D
5	mul	T5, T2, T4	L6	D	D
6	add	T6, prod, T5	L7	D	D
7	asgn	prod, T6	L	D	
8	add	T7, j, 1	L9	D	L
9	asgn	j, T7	L10	D	
10	ble	start ,j, 20		L	L

טבלת הקבועים		
	L\D	Next use
1	L	8
4	L	3
20	L	10

כאן אפשר לראות משהו מעניין קצת. T1 מעביר מידע שיש בו ל-T2 ומשלב זה הוא מת עד סוף קטע הקוד. כלומר אם נחשוב על אופטימיזציה כלשהי ניקח בחשבון ש-T1 חי בדיוק לשורה אחת ואולי כדאי לחשוב פעמיים על מה אנחנו מקצים לו. נעבור לשורה הראשונה –

	L\D	Next use
j	**L**	**1**
a	L	2
b	L	4
prod	L	6
T1	**D**	
T2	D	
T3	D	
T4	D	5
T5	D	
T6	D	
T7	D	

			Dst	S1	S2
1	**mul**	**T1, 4, j**	**L2**	**L3**	**L3**
2	aelm	T2, a, T1	L5	L	D
3	mul	T3, 4, j	L4	L	L8
4	aelm	T4, b, T3	L5	L	D
5	mul	T5, T2, T4	L6	D	D
6	add	T6, prod, T5	L7	D	D
7	asgn	prod, T6	L	D	
8	add	T7, j, 1	L9	D	L
9	asgn	j, T7	L10	D	
10	ble	start ,j, 20		L	L

טבלת הקבועים		
	L\D	Next use
1	L	8
4	**L**	**1**
20	L	10

עכשיו יש לנו את כל הטבלה, ואנחנו יכולים להסיק לגבי כל משתנה בכל רגע נתון האם הוא חי או מת. אם נרצה למשל לדעת מה עם T3 בשורה 7, נוכל לעלות משורה ה7, ולמצוא את פעם הבאה שהמשתנה הזה יופיע, במקרה שלנו – שורה 3, ושם אנחנו רואים שהוא מוגדר כ-D, ולכן החל משורה 3 המשתנה הזה מת ולא יקום יותר. מה עכשיו?

טרנספורמציות פשוטות בתוך בלוקים

כאן אנחנו חוזרים לאופטימיזציות עליהם התחלנו לדון. ישנם מספר אופטימיזציות פשוטות שאם נעשה אותם (ובהתחשב בידע שיש לנו עכשיו בnext use זה יהיה יותר קל) נוכל להרוויח לא מעט ייעול בקוד.

- **סילוק ביטויים משותפים** – אם יש לנו קוד שחוזר על עצמו באותם חישובים, אך את תוצאת החישוב כבר שמרנו איפשהו, הרבה יותר נכון לעשות השמה ולא חישוב מחודש. במה דברים אמורים?

```
a    := b + c
b    := a – d
c    := b + c
d    := a – d
```

אם ניקח את הקוד הזה, נוכל לראות שאותו חישוב חוזר על עצמו פעמיים, כך שנוכל את השורה האחרונה ולכתוב במקומה השמה מתאימה –

```
a    := b + c
b    := a – d
c    := b + c
d    := b
```

כך חסכנו לנו באופן די פשוט.

- **ביטול קוד מת** – השמה למשתנה שלא חי לאחר מכן. אין צורך להכניס ערכים למשתנה כלשהו אם אין לו שימוש בהמשך. במקרה כזה פשוט נמחק את השורה.
- **החלפת סדר בין פקודות שאינן תלויות אחת בשניה** – במידה וזה אפשרי, אנחנו לעיתים נצמצם מרחק בין שורות מסוימות על מנת לחסוך זמן בו המשתנה תופס מקום ברגיסטר ולא להשאיר סתם משתנה חי עד סוף התוכנית בלי שום צורך אמיתי.
- **פישוטים אלגבריים** – אם יש לנו השמה של ערכים אלגבריים שלא באמת תורמים לאנושות, כמו הכפלה של איבר ב-1, או הוספה של 0, וכל מיני מרעין בישין דומים אנחנו פשוט יכולים להוריד את השורה.
- **שימוש בפקודות זולות יותר** – אם יש לנו איבר בחזקת 2, המערכת תפעל יותר מהר בחישוב של הכפלת האיבר בעצמו, ולא בשימוש בפונקצית חזקה. אותו דבר גם עם הכפלה בשניים וחיבור. כמו כן, משחק עם הביטים (הזזות ימינה/שמאלה) יכולות להחליף תרגילים מתמטיים של הכפלת איבר ב-2^i.

תהליך יצירת קוד

אנחנו עוברים שורה אחר שורה ובודקים את המשתנים והרגיסטרים הרלוונטיים אליהם. אנחנו שואפים שאם אנחנו מחפשים מידע של משתנה, הוא כבר יופיע ברגיסטר. אם הוא לא שם, אנחנו צריכים לדאוג ולהביא אותו לתוך הרגיסטרים שמוקצים לנו לעבודה. כמובן שאנחנו מניחים שיש לנו רגיסטרים שמוקצים לנו, ואנחנו לא מדברים על כאלה שהמערכת משתמשת בהם כמו המצביע של המחסנית ודומיו.

מעקב אחרי משתנים ורגיסטרים

לכל רגיסטר שמוקצה לנו לשימוש אנחנו נרשום לנו "מתאר" Descriptor שיגיד לנו מה קורה באותו רגיסטר. כשאנחנו מתחילים לרוץ, כמובן שהרגיסטרים ריקים, כי אנחנו לא נתחיל להכניס ערכים בלי לדעת בוודאות שאנחנו נשתמש במה שנכניס, כי להוציא ולהכניס עולה יותר מלהכניס.

במהלך ייצור הקוד, רגיסטר יכול להכיל יותר ממשתנה אחד. הכיצד? אם רגיסטר מסוים יחזיק את הערך של x ולפתע תופיע לנו פקודה של y:=x אין לנו צורך להקצות רגיסטר חדש בשביל y, אלא אנחנו יכולים פשוט להגיד לרגיסטר של x שאם מישהו שואל, אז הוא גם y.

במקביל, עבור כל משתנה אנחנו נשמור את הרגיסטר בו הערך שלו שמור.

סיכומו של דבר – היחס של המשתנים והרגיסטרים הוא יחיד לרבים – רגיסטר אחד יכול להיות ליותר ממשתנה אחד, אך לא להיפך.

Code Generation

כאן אנחנו מדברים על יצירת קוד בשפת מכונה. אנחנו לוקחים את שפת הביניים בקוד התלת מעני ומבצעים כך

- עבור כל שורה x:= y op z בקוד –
  - להפעיל את הפונקציה getreg(x:= y op z) שתקבע לנו מה המיקום של הרגיסטר עבור כל משתנה.
  - אם המשתנה y לא מושם בתוך הרגיסטר שקבענו, אז יש לטעון אותו – LD R_y, Y.
  - כנ"ל עם z.
  - הכנסת השורה op R_x, R_y, R_z לקוד.
  - עדכון הדסקריפטורים של כל משתנה. דברים חשובים לציון – המשתנה אליו עשינו את ההשמה (x) נמצא בתוך רגיסטר, אנחנו צריכים לוודא שמבחינת x הוא באמת המקום היחיד בו הוא מופיע, ומבחינת הרגיסטר, אין בו ערכים נוספים מלבד x. כלומר שאם היה לנו קודם רגיסטר שהחזיק את x,y אנחנו צריכים לחפש ל-y מקום חדש.

דברים שכדאי לזכור – אם יש לנו השמה (x:=y) אנחנו פשוט מעדכנים את הדסקריפטורים – את הכתובות אנחנו מעבירים שיהיו לאותו הרגיסטר, ואת הרגיסטר עצמו לזה שהוא מכיל מידע לשני משתנים.

בסיום העבודה על בלוק, אנחנו שומרים את כל המשתנים שלא עודכנו עדיין מהרגיסטרים למקומם האמיתי, על מנת לוודא שבבלוק הבא יהיו לנו את הנתונים עדכניים.

הפונקציה getreg

באלגוריתם של ייצור הקוד, אמרנו שאנחנו שולחים את כל המשתנים הרלוונטיים לפונקציית getReg. תפקיד הפונקציה הוא לבדוק אילו רגיסטרים אנחנו יכולים להקצות לצורך שימוש בשורה הקרובה.

סדר הפעולות הוא קודם כל לטפל ב-y,z שהם האופרנדים המחושבים, ורק אחרי זה להתעסק עם המשתנה המושם x. האלגוריתם עובד בצורה של קבלת החלטות שמתייחסת בכל פעם מהאפשרות הטובה ביותר להקצאת רגיסטר ועד הפחות טובה, בסדר הבא –

1. <u>אם y ברגיסטר R, קבע R_y=R.</u> – המצב הכי אופטימלי הוא שלמשתנה y יש כבר איזה רגיסטר שהקצינו בעבר ואנחנו עדיין לא זרקנו אותו, במקרה כזה פשוט נמשיך לעבוד על אותו רגיסטר.

2. <u>**אחרת**, אם יש רגיסטר פנוי R, קבע R_y=R.</u> – מקרה קצת פחות טוב, ל-y עדיין אין רגיסטר, אבל יש לנו רגיסטר פנוי ללא שום משתנה (בדרך כלל זה יקרה רק שלבים הראשונים של הפעולה). כמובן שאם אפשר פשוט להקצות משהו ללא שום עלות והשלכות זה מה שנעדיף.

3. <u>**אחרת**, אם יש רגיסטר המכיל רק את x, וגם x אינו אופרנד (z במקרה זה) אז קבע R_y=R.</u> – על פי מה שראינו מקודם, מבחינתנו המשתנה x ברגע הזה הוא מת, אנחנו גם ככה הולכים לשים בו ערך אחר. לכן, אם נמצא רגיסטר שמוקצה לו אנחנו יכולים להשתמש בו בתור הקצה לצורך החישוב. כדאי לשים לב, יכול להיות שהפקודה שאנחנו עובדים עליה היא x:= y + x, ובמקרה כזה אנחנו לא יכולים להשתמש ברגיסטר של x כי

הערך שלו עדיין "חי" ואנחנו צריכים אותו. לאחר שנדרוס את הערך של x, מה נעשה עם x בעצמו? נראה בהמשך.

4. **אחרת**, אם יש רגיסטר R שהמשתנה שבו v אינו חי, קבע $R_y=R$. – אנחנו קודם כל מנסים את הדברים שיביאו לנו הכי פחות השלכות, ולכן אם אין ברירה, אנחנו נתחיל להוציא משתנים שאינם חלק מהמשוואה. בעדיפות ראשונה אנחנו ניקח קודם כל משתנים שהם מתים (נראה בתרגיל המשך, שאנחנו בדרך כלל נעבור כל פעם ונבדוק לפי הסדר, אבל זה סתם כמוסכמה כרגע ולא חייב להיות מה שבאמת נעשה).
5. **אחרת** אם יש רגיסטר R שהמשתנה שבו v קיים גם במקום אחר, קבע $R_y=R$. – לא ברור לי עדיין איך הגענו למצב הזה, אבל משתנה מופיע על שני רגיסטרים שונים, אחד מהם מיותר ואפשר להשתמש בו.
6. **אחרת**, בחר רגיסטר R כלשהו שמייצג מספר מינימלי של משתנים {v1, v2, …, vm}. לכל משתנה שמיוצג רק ע"י R העתק את ערכו לזיכרון. קבע $R_y=R$. – במצב הכי גרוע אנחנו צריכים להוציא רגיסטר שיש בו מידע שעלול להיות חי. במקרה כזה, אנחנו נבחר את הרגיסטר שיכיל כמה שפחות משתנים (כי יכול להיות שרגיסטר יכיל 3 משתנים שונים, ואנחנו לא נוציא את כולם, אם יש לנו אפשרות למשהו טוב יותר), נעדכן את המידע שלהם במקור ונקצה את הרגיסטר.
7. **בכל מקרה** עדכן את ה-descriptor של R, v, (או v_i אם רלוונטי) בהתאם. – לא משנה מה נחליט, אנחנו חייבים לעדכן את הדסקריפטור של הרגיסטר ושל המשתנה על השינויים שאנחנו עושים.

את האלגוריתם הזה מריצים על y ועל z, ועכשיו אנחנו עוברים ל-x. ההבדל בין שני האלגוריתמים הוא בעיקר בסעיפים הראשונים ואחרי זה אנחנו ממשיכים באותו אופן –

1. אם x ברגיסטר R המכיל רק את x, קבע $R_x=R$. (זה בסדר גם אם y או z זהים ל-x). – זה בדיוק אותו דבר גם כן כמו קודם, אם הצלחנו לא להעיף את המשתנה כשהקצינו רגיסטרים למשתנים הקודמים, אז נשתמש כמובן באותו אחד.
2. אחרת, אם לא משתמשים ב-y (או ב-z) לאחר הפקודה הנוכחית, והרגיסטר שנקבע לו מייצג אותו באופן יחיד, אז ניתן לקבוע את הרגיסטר של x להיות זהה לרגיסטר של y (או z). – כאן ההיגיון קצת הפוך ממה שראינו קודם. אם קודם אמרנו ש-x גם ככה מת מבחינתנו, אז כאן אנחנו אומרים מהכיוון ההפוך – גם ככה הערכים האלה רלוונטים רק לשורה הזאת (בהנחה שלא משתמשים בהם אחר כך), ולכן אנחנו יכולים להקצות אותו לאותו מקום.

מכאן והלאה זה בדיוק אותו דבר.

יש לנו דוגמא נאיבית יחסית לפעולה עבור הקלט - d := (a – b) + (a – c) + (a – c)

אנחנו יוצרים טבלה עם ארבע עמודות-

1. **Statements -** חלק הפקודה איתו אנחנו מתעסקים. אנחנו לא עושים כרגע את כל העץ, אלא רק מחשבים כל חלק סוגריים בנפרד, ואז את החישוב הסופי.
2. **code generated** – הקוד בשפת האסמבלי שאנחנו נרכיב לאט לאט. בכל טעינה של מידע לרגיסטר אנחנו נשתמש בפקודת Load, ואחרי זה נשתמש בפעולה עצמה שאנחנו עושים.
3. **register descriptor** – בסוף כל כתיבת קוד, אנחנו נכתוב איזה מידע שינינו ברגיסטורים.
4. **address descriptor** – בכל שלב יופיעו לנו הכתובות של **כל** המשתנים. אם המשתנה נמצא גם ברגיסטר, אז מלבד a in a אנחנו נכתוב גם את המיקום של המשתנה ברגיסטר a in R1.

נראה איך עובד המימוש -

<table>
<tr><th>statements</th><th>code generated</th><th>register descriptor</th><th>address descriptor</th></tr>
<tr><td></td><td></td><td>registers empty</td><td></td></tr>
<tr><td colspan="4">בתחילת העבודה, אין שום הקצאה לזיכרון ולרגיסטרים.</td></tr>
<tr><td>t := a – b</td><td>LD R1, a
LD R2, b
SUB R2, R1, R2</td><td>R1 contains a
R2 contains t</td><td>a in R1,
a in a,
b in b,
c in c,
d in d,
t in R2</td></tr>
<tr><td colspan="4">אנחנו מתחילים עם האופרנדים מימין – מאחר וכמובן שאין להם הקצאה, אנחנו טוענים את שניהם לתוך שני הרגיסטרים הראשונים האפשריים. מבחינת t אנחנו יכולים לראות שאין שימוש בהמשך למשתנה b ולכן אנחנו יכולים להשתמש ברגיסטר שלו לטובת השמת החישוב, ולכן בפקודה הסופית השמת התוצאה תהיה בערך של b\z.

לאחר שסיימנו את זה, אנחנו יכולים לראות שבתוך הרגיסטר R1 יהיה לנו את a, ובתוך R2 יהיה לנו את t, ולכן בתיאורי הכתובות אנחנו נעדכן גם את המיקומים שלה ברגיסטרים.</td></tr>
<tr><td>u := a – c</td><td>LD R3, c
SUB R1, R1, R3</td><td>R1 contains u
R2 contains t
R3 contains c</td><td>a in a,
b in b,
c in c,
c in R3,
d in d,
t in R2,
u in R1</td></tr>
<tr><td colspan="4">אנחנו עושים פה קצת קיצורי דרך, ומזהים שהפעולה a-c חוזרת על עצמה פעמיים, ואנחנו בוודאי לא רוצים לחשב זאת שוב, אם אנחנו יכולים פשוט לשמור את התוצאה. עכשיו, a כבר קיים לנו על רגיסטר, אבל c עוד לא קיים. למזלינו, יש לנו עדיין רגיסטר פנוי שאפשר להשתמש בו – R3 לכן אנחנו טוענים אליו את c. מבחינת u(x) אליו אנחנו עושים השמה, אנחנו יכולים לראות לפי הכלל השני באלגוריתם getReg שאנחנו יכולים להשתמש ברגיסטר של a כי אין בו שימוש בהמשך (שוב, אנחנו משתמשים בתוצאה ולא מחשבים שוב), ולכן את הפעולה הזאת אנחנו נחשב לתוך R1.

לסיכום – השינוי ברגיסטרים יהיה רק הטעינה של c לתוך R3, ובכתובת – a נוריד את הרגיסטר שכבר לא מוקצה לו, ונכניס את המשתנה המקומי u במקומו.</td></tr>
<tr><td>v := t + u</td><td>ADD R3, R2, R1</td><td>R1 contains u
R2 contains t
R3 contains v</td><td>a in a,
b in b,
c in c,
c in R3,
d in d,
t in R2,
u in R1,
v in R3</td></tr>
<tr><td colspan="4">את t,u כבר יש לנו על רגיסטרים, אז מה שנשאר לנו הוא לשמור את התוצאה רק ברגיסטר חדש, ולכן אנחנו מעיפים את c שהוא כבר מת מבחינתנו. (לדעתי, צריך להוריד מרשימת הכתובת את c in R3 כי הוא כבר לא רלוונטי)</td></tr>
<tr><td>d := v + u</td><td>ADD R1, R1, R3
ST d, R1</td><td>R1 contains d
R2 contains t
R3 contains v</td><td>a in a,
b in b,
c in c,
d in d,</td></tr>
</table>

c in R3, d in d, t in R2, **d in R1,** v in R3			
גם u וגם v נמצאים ברגיסטרים ולכן איתם אין לנו בעיה. את התוצאה של d אנחנו נשמור על מה שהולך למות הכי מהר. תכלס כולם הולכים למות כי סיימנו פה את הבלוק, ולכן אנחנו מקצים את R1 לתוצאה. לאחר החישוב כל מה שנשאר לנו הוא לעשות שמירה של כל זה לתוך המשתנה הגלובלי d וסיימנו.			

בשביל לראות את הפעולה השלמה מההתחלה ועד הסוף, נחזור למצגת התרגול –

הפקת קוד משופרת: גושים בסיסיים - מצגת דוגמאות מס' 9 חלק ב'

נמשיך כאן במקום בו עצרנו במצגת. מילאנו את כל טבלת ה-next use, ואנחנו יודעים להגיד מה קורה בכל רגע נתון. עכשיו אנחנו צריכים להעביר את הקוד לשפת האסמבלי בהתחשב ברגיסטרים שיוקצו לנו.

אנחנו נמשיך לעבוד פה בהמשך לקוד שראינו עד עכשיו, ונצטרך קודם לעשות שינוי קטן בטבלאות –

	L\D	Next use	In Mem.	Reg #
j			m	
a			m	
b			m	
prod			m	
T1				
T2				
T3				
T4				
T5				
T6				
T7				

טבלת הקבועים

	L\D	Next use	Reg #
1	L		
4	L		
20	L		

Reg #	descriptor
R1	
R2	
R3	
R4	

R	R_i
R_y	
R_z	
R_x	

אז ככה – אנחנו מוסיפים לטבלת הסמלים שתי עמודות, האחת מסמנת האם המידע מעודכן בזיכרון, במידה ולא, אנחנו נשנה צבע. והעמודה השניה אומרת לנו באיזה רגיסטר אנחנו שומרים את המשתנה. לטבלת הקבועים אנחנו כמובן לא צריכים להכניס את העמודה של in Mem כי זה קבוע, הוא לא משתנה.

מעבר לזה אנחנו מוסיפים טבלת דסקריפטור לרגיסטרים שיגיד לנו איזה משתנים שמורים עליו, ועוד טבלת עזר קטנה שבה נכתוב כל פעם לאיזה רגיסטר אנחנו מקצים לטובת ה-x,y,x של הפעולה.

עכשיו אנחנו ניקח כל פעם שורה אחת, נקצה רגיסטרים ונחשב את הקוד לשפת הסף. בהצלחה לנו. כמובן שבמצג עושים ממש כל שלב בנפרד, אבל אני אשתדל לתמצת את זה כי אין לנו כח ועצבים לעבור ממש כל שלב בנפרד.

	L\D	Next use	In Mem.	Reg #
j	**L**	**3**	m	**2**
a			m	
b			m	
prod			m	
T1	**L**	**2**		**3**
T2				
T3				
T4				
T5				
T6				
T7				

1 mul T1 (L2), 4 (L3), j (L3)
loadc R1, 4 load R2, j mul R3, R1, R2

טבלת הקבועים			
	L\D	Next use	Reg #
1			
4	**L**	**3**	**1**
20			

Reg #	descriptor
R1	**4**
R2	**j**
R3	**T1**
R4	

R	R_i
R_y	$\mathbf{R_1}$
R_z	$\mathbf{R_2}$
R_x	$\mathbf{R_3}$

התחלנו.

אנחנו לוקחים את השורה הראשונה של הקוד (מי שלא זוכר מוזמן לקפוץ למעלה, ולראות את הקוד). אנחנו מוסיפים לכל משתנה בסוגריים את ערך הזומבי שלו (L/D בלע"ז) אותו חישבנו בכל התהליך הארוך קודם, ואנחנו יכולים לעדכן בעמודות ה-next use את הערך של כל אחד. למה אנחנו צריכים את זה? כי כזכור, כחלק מהאלגוריתם, אנחנו בודקים האם יש ערכים מושמים שהם מתים ואנחנו יכולים לפנות, כי אנחנו לא אוהבים שהמתים מהלכים בינינו, זה רק גורם לפאניקה ואפוקליפסה.

עכשיו נסתכל על השמת הרגיסטרים. נתחיל עם R_y שהוא הקבוע 4. מאחר וכל הרגיסטרים פנויים מבחינתנו, החלק הזה יהיה לנו יחסית קל – פשוט נטען את הקבוע עם פקודת loadc (load constant) לתוך R1. אחרי זה נטען את j לתוך R2, ומאחר שכל המשתנים הנתונים לנו חיים לפחות עוד שורה אחת, אנחנו נדאג לעשות השמה של התוצאה לתוך רגיסטר חדש. ולכן המשתנה המקומי T1 יוקצה להיות ב-R3.

נמשיך –

	L\D	Next use	In Mem.	Reg #
j	L	3	m	2
a	**L**		m	
b			m	
prod			m	
T1	**D**			
T2	**L**	**5**		**3**
T3				
T4				
T5				
T6				
T7				

2 aelm T2 (L5), a (L), T1 (D)
load R4, a aelm R3, R4, R3

טבלת הקבועים

	L\D	Next use	Reg #
1			
4	L	3	1
20			

Reg #	descriptor
R1	4
R2	j
R3	**T2**
R4	**a**

R	R_i
R_y	**R4**
R_z	**R3**
R_x	**R3**

קודם כל נעדכן את הערכים בטבלת ה-next use. זה רלוונטי כי המשתנה T1 היה חי רק עד שורה זאת, ומבחינתנו זה מידע שהוא חשוב להמשך.

עכשיו נקצה רגיסטרים, אנחנו מתחילים עם R_y שהוא a ואין לו עדיין הקצאה, אבל יש לנו רגיסטר שעדין פנוי ולכן נשתמש בו לאכסן את a. לאחר מכן נבדוק עבור R_z – T1, אבל הוא כבר שמור לנו בזיכרון, אז אין לנו צורך להקצות חדש. כל שנותר לנו הוא לבדוק מקום ל-(R_x) T2. כמובן שהוא עדיין לא נמצא בשום מקום, ולכן נחפש האם אחד האופרנדים שעושים את החישוב הולכים למות. אנחנו רואים ש-T1 מסומן לנו ב-D (ולכן חשוב לעדכן את זה לפני ההקצאה), אנחנו מוודאים הריגה, ומעדכנים שהתוצאה תיכנס לרגיסטר הנ"ל.

	L\D	Next use	In Mem.	Reg #
j	**L**	**8**	m	2
a	L		m	
b			m	
prod			m	
T1	D			
T2	L	5		3
T3	**L**	**4**		
T4				
T5				
T6				

3 mul T3 (L4), 4 (L), j (L8)
mul R1, R1, R2

טבלת הקבועים

	L\D	Next use	Reg #
1			
4	**L**		
20			

Reg #	descriptor
R1	**T3**
R2	j
R3	T2
R4	a

R	R_i
R_y	**R1**
R_z	**R2**
R_x	**R1**

T7				

כאן זה מתחיל להיות מעניין – 4 וגם j שניהם נמצאים בזיכרון כבר וטעונים לרגיסטרים. יש לשים לב שעד עכשיו שניהם היו מוגדרים להיות חיים עד שורה זאת, וגם עם הגיענו לשורה הזאת אנחנו מעדכנים אותם שיהיו חיים להמשך. למה זה חשוב? כי עכשיו אנחנו מחפשים מקום ל-R_x שלא נמצא בזיכרון – T3. הבעיה היא כזאת – אין מקום ברגיסטרים, וגם כל מי שנמצא שם חי, אז את מי נוציא? יש לנו שיקולים נוספים להוצאה שאנחנו נשתמש בהם – אם אין ברירה נעדיף להוציא קבוע על פני הוצאה של משתנה, מאחר וטעינת קבוע עולה לנו פחות מאשר טעינת מידע מהזיכרון. ולכן אנחנו נחליט שמידע ההכפלה ישמר לאותו רגיסטר בו היה הקבוע.

	L\D	Next use	In Mem.	Reg #
j	L	8	m	2
a	L		m	
b	**L**		m	**4**
prod			m	
T1	D			
T2	L	5		3
T3	**D**			
T4	**L**	**5**		
T5				
T6				
T7				

4 aelm T4 (L5), b (L), T3 (D)

```
load    R4, b
aelm    R1, R4, R1
```

טבלת הקבועים

	L\D	Next use	Reg #
1			
4	L		
20			

Reg #	descriptor
R1	T3
R2	j
R3	T2
R4	**b**

R	R_i
R_y	**R4**
R_z	**R1**
R_x	**R1**

כאן אנחנו רוצים לגשת למשתנה בתוך מערך (לא שזה רלוונטי לנו), ואנחנו מחפשים מקום להניח בו את b. עכשיו יש לנו בעיה – קודם כל הכל מלא. מעבר לזה, כל המשתנים ברגיסטרים חיים (מלבד T3 אבל אנחנו צריכים אותו בשורה הנוכחית), ולכן אנחנו צריכים להחליט למי אנחנו דופקים כדור בראש. אמנם זה לא פשוט, אבל אם נתעמק נוכל לראות כי בעוד כל הערכים החיים ברגיסטרים חיים, יש לנו את a שהוא חי "סתם", אין לנו שימוש בו באחת השורות בהמשך, וגם לא עשינו בו שימוש חוץ מקריאה, ולכן אם אנחנו נוציא אותו מהרגיסטר הוא ישפיע לנו הכי פחות, ולכן הוחלט להוריד אותו.

	L\D	Next use	In Mem.	Reg #
j	L	8	m	2
a	L		m	
b	L		m	4
prod			m	
T1	D			
T2	**D**			**1**
T3	D			
T4	**D**			
T5	**L**	**6**		**3**
T6				
T7				

5 mul T5 (L6), T2 (D), T4 (D)

```
mul     R3, R3, R1
```

טבלת הקבועים

	L\D	Next use	Reg #
1			
4	L		
20			

Reg #	descriptor
R1	**T4**
R2	j
R3	**T5**
R4	b

R	R_i
R_y	**R3**
R_z	**R1**
R_x	**R3**

אז ככה – אנחנו מעדכנים את המידע ומתחילים לחפש מקום. T2 כבר נמצא ב-R3 ואנחנו צריכים למצוא מקום ל-T4. אנחנו רואים שיש לנו את R1 בו קיים הערך של T3 שהוא כבר מת, ולכן נכניס אותו כאן. אחרי זה נחפש מקום ל-Rx – הוא קודם כל יבדוק האם ברגיסטרים של y,z יש ערכים מתים, ומאחר ויש לנו ב-R_y ערך מת אנחנו נשתמש בו.

	L\D	Next use	In Mem.	Reg #
j	L	8	m	2
a	L		m	
b	L		m	4
prod	**D**		m	
T1	D			
T2	D			
T3	D			
T4	D			
T5	**D**			3
T6	**L**	**7**		**1**
T7				

6 add T6 (L7), prod (D), T5 (D)

```
load    R1, prod
add     R1, R1, R3
```

טבלת הקבועים

	L\D	Next use	Reg #
1			
4	L		
20			

Reg #	descriptor
R1	**T6**
R2	j
R3	T5
R4	b

R	R_i
R_y	**R1**
R_z	**R3**
R_x	**R1**

עכשיו זה כבר אמור להיות לנו יותר פשוט מה שקורה – prod לא נמצא בכלל ברגיסטרים, לכן אנחנו נחפש מקום לטעון אותו. ב-R1 היה לנו את T4 שמסומן לנו כ"מת". לכן אנחנו יכולים להקצות ש-R_y = R1. T5 כבר נמצא ברגיסטר,

ואנחנו עכשיו צריכים למצוא מקום רק לתוצאה – T6. מאחר ואין מקום, אנחנו בודקים האם אחד מהאופרנדים הימניים הולך למות, וכשאנחנו רואים ש-prod מת בעוד רגע, אנחנו מגדירים שהתוצאה תדרוס את הערך של prod ואנחנו יכולים להשתמש בו.

	L\D	Next use	In Mem.	Reg #
j	L	8	m	2
a	L		m	
b	L		m	4
prod	**L**		m	**1**
T1	D			
T2	D			
T3	D			
T4	D			
T5	D			3
T6	**D**			1
T7				

7 asgn prod (L), T6 (D)

טבלת הקבועים			
	L\D	Next use	Reg #
1			
4	L		
20			

Reg #	descriptor
R1	T6, **prod**
R2	j
R3	T5
R4	b

R	R_i
R_y	**R1**
R_z	
R_x	

הזכרנו את עניין ההשמה קודם, וכעת נראה איך זה בא לידי ביטוי. אנחנו אומרים ש-prod יהיה שווה לערך של T6. עכשיו, אם זה אותו ערך, אין סיבה לשמור את זה ברגיסטרים שונים. ולכן כל מה שנעשה, הוא לעדכן את הרגיסטר R1 שיחזיק גם את prod, ולשנות את המיקום שלו בהתאם. עכשיו אנחנו צריכים לשים לב, שהמידע בזיכרון שקיים לנו לגבי prod כבר לא רלוונטי – יהיה מה שיהיה הערך באותו מקום, הערך האמיתי של prod הוא T6. מסיבה זאת אנחנו משנים את הסימון ב-in mem לכך שיש לעדכן אותו.

	L\D	Next use	In Mem.	Reg #
j	**D**		m	
a	L		m	
b	L		m	4
prod	L		m	1
T1	D			
T2	D			
T3	D			
T4	D			
T5	D			
T6	D			1
T7	**L**	**9**		**2**

8 add T7 (L9), j (D), 1 (L)

```
loadc   R3, 1
add     R2, R2, R3
```

טבלת הקבועים

	L\D	Next use	Reg #
1	**L**		**3**
4	L		
20			

Reg #	descriptor
R1	T6, prod
R2	**T7**
R3	**1**
R4	b

R	R_i
R_y	**R2**
R_z	**R3**
R_x	**R2**

כאן יש לנו דרך רבת תהפוכות. מעדכנים את טבלת הסמלים, ומתחילים לבדוק הקצאות. J נמצא לנו כבר ב-T2 ואנחנו מחפשים מקום עבור הקבוע 1. אנחנו מוצאים את T5 ב-R3 שהוא גם ככה מת, ולכן אנחנו מוציאים אותו ושמים בו את 1. עכשיו אנחנו מחפשים מקום עבור R_x, ורואים שיש לנו את j שהוא מת, ולכן אנחנו יכולים להשתמש ברגיסטר שלו לטובת התוצאה. רואים את הסוף –

	L\D	Next use	In Mem.	Reg #
j	**L**	**10**	m	**2**
a	L		m	
b	L		m	4
prod	L		m	1
T1	D			
T2	D			
T3	D			
T4	D			
T5	D			
T6	D			1
T7	**D**			**2**

9 asgn j (L10), T7 (D)

טבלת הקבועים

	L\D	Next use	Reg #
1	L		3
4	L		
20			

Reg #	descriptor
R1	T6, prod
R2	T7, **j**
R3	1
R4	b

R	R_i
R_y	**R2**
R_z	
R_x	

שוב, פעולת השמה. פשוט מעדכנים את הדסקריפטורים.

	L\D	Next use	In Mem.	Reg #
j	L	10	m	2
a	L		m	
b	L		m	4
prod	L		m	1
T1	D			
T2	D			
T3	D			
T4	D			
T5	D			
T6	D			1
T7	D			2

```
store    R1, prod
store    R2, j
```

טבלת הקבועים

	L\D	Next use	Reg #
1	L		3
4	L		
20			

Reg #	descriptor
R1	T6, prod
R2	T7, j
R3	1
R4	b

R	R_i
R_y	**R2**
R_z	
R_x	

רגע, מה קורה פה? אין קוד. הפקודה האחרונה בבלוק היא פקודת קפיצה (שזה די הגיוני ביחס לחוקי הבלוקים), ולכן לפני שנקפוץ, אנחנו צריכים לבדוק האם כל המידע שלנו מעודכן בזיכרון. אנחנו רואים שיש לנו את j,prod שעשינו להם השמות, אבל לא עדכנו את הערכים שלהם בזיכרון, לכן אנחנו מעדכנים ורק אז ממשיכים הלאה.

	L\D	Next use	In Mem.	Reg #
j	**L**		m	2
a	L		m	
b	L		m	4
prod	L		m	1
T1	D			
T2	D			
T3	D			
T4	D			
T5	D			
T6	D			1
T7	D			2

10 ble j (L), 20 (L), start

```
loadc    R3, 20
ble      R2, R3, start
```

טבלת הקבועים

	L\D	Next use	Reg #
1	L		
4	L		
20	**L**		**3**

Reg #	descriptor
R1	T6, prod
R2	T7, j
R3	**20**
R4	b

R	R_i
R_y	**R2**
R_z	
R_x	

אנחנו צריכים לבדוק האם j<=20. אנחנו טוענים את הקבוע 20 לתוך רגיסטר. אנחנו רואים שב-R3 יש לנו קבוע, והכי קל זה להוריד קבועים, בעיקר בהתחשב שהכל חי. j כבר נמצא ברגיסטר, לכן כל מה שיש לנו לעשות הוא לבדוק את שני הרגיסטרים R2,R3 ולקפוץ אם יש צורך.

עכשיו נוכל לראות את הקוד שייצרנו –

```
# T1 := 4 * j
loadc   R1, 4
load    R2, j
mul     R3, R1, R2
# T2 := a[T1]
load    R4, a
aelm    R3, R4, R3
# T3 := 4 * i
mul     R1, R1, R2
# T4 := b[T3]
load    R4, b
aelm    R1, R4, R1
# T5 := T2 * T4
mul     R3, R3, R1
# T6 := prod + T5
load    R1, prod
add     R1, R1, R3
# prod := T6
# T7 := j + 1
loadc   R3, 1
add     R2, R2, R3
# j := T7
# Storing towards end of block
store   R1, prod
store   R2, j
# if j <= 20 goto start
loadc   R3, 20
ble     R2, R3, start
```

שכוייח.

הקצאה והשמת רגיסטרים

כאשר אנחנו מגיעים לשלב הסופי של הקצאת הרגיסטרים, עומדות בפנינו שתי החלטות עיקריות – האחת שאלת הקצאת הרגיסטרים (Register Allocation), והשניה שאלת השמת הרגיסטרים (Register Assignment).

הקצאת רגיסטרים – אנחנו צריכים להחליט קודם כל איך אנחנו מחליקים את הרגיסטרים שיש לנו. או במילים אחרות, אילו ערכים יישמרו באילו רגיסטרים. ידוע לנו שהגישה לרגיסטרים היא מהירה, ולכן כל החלטה שנקבל בעניין הזה צריכה לבוא מתוך כך שנתחשב בתוצאה הטובה ביותר.

נסתכל על שתי גישות קיצון – גישה גלובלית – כל משתנה יכול להיות בכל רגיסטר. אנחנו צריכים משתנה? נשים אותו איפה שיש לנו מקום ברגיסטר ונמשיך הלאה. הבעיה עם הגישה הזאת, היא שיש לנו מידע חשוב יותר וחשוב פחות. אם אנחנו צריכים משתנה לשורה מסוימת ובודדת, ואנחנו מחליטים להוציא את אחד המשתנים מהרגיסטר, אך המשתנה ההוא היה כזה שרצנו עליו כל התכנית. אנחנו נצטרך לעשות שמירת משתנה, ואז טעינה וחוזר חלילה בהמשך. עכשיו זה לא כזה נורא שמדובר על פעם בודדת, אבל אם יש לנו תוכנית שלמה שאנחנו כל הזמן שומרים וטוענים אנחנו נפגום בעבודה שלה. הקצאת קבוצות – אנחנו מחלקים את המשתנים השונים לקבוצות על פי חשיבות או פרמטרים שונים, ומגדירים קבוצת רגיסטרים שתשמש עבור כל קבוצה. מצד אחד, פתרנו את העניין שיש לנו רגיסטרים למשתנים חשובים, אבל הקיבעון הזה עלול לבוא בעוכרינו כשנצטרך רגיסטרים למשתנים זמניים ולא נקבל.

הגישה שמשתמשים בדרך כלל, היא מעין גישת ביניים. אנחנו נגדיר רגיסטרים קבועים רק למספר בודד של משתנים בעלי עדיפות גבוהה מאוד, כמו למשל ה-stack pointer, הוא חשוב מכדי להתעסק איתו. את שאר המשתנים אנחנו נמקם בצורה גלובלית.

הקצאת רגיסטרים גלובלית

כאמור, אנחנו רוצים לחסוך כמה שניתן בפעולות כמו store-iload. בשביל זה אנחנו צריכים כמה שאפשר לנסות עוד לפני שנתחיל לעבוד למצוא את האופטימיזציה להקצאת הרגיסטרים. מחשבה פשוטה בעניין, הוא הסתכלות על משתנים והבנה של הולכים להשתמש בכל משתנה. למשל, אם יש לנו משתנה בתוך לולאה הוא יהיה יחסית חשוב, כי ככל הנראה נשתמש בו יותר מפעם אחת. משתנים שכאלה יקבלו רגיסטרים עצמאיים ויעבדו באופן בלתי תלוי. אחרי זה ננסה להסתכל על שאר המשתנים שרצים בבלוקים. ננסה להעריך עלות של כל משתנה על ידי ספירה של כמה פעמים הוא מופיע בבלוקים השונים, ונכנס לזה שני פרמטרים חשובים –

- על כל מופע של משתנה, נעניק לו נקודה אחת.
- על כל מופע שנמצא בשני בלוקים ברציפות נביא לו 2 נקודות.

הניקוד של הבלוקים העוקבים מגיע בנוסף לנקודות שהוא מקבל על כל מופע ובעצם נועד "לפרגן" למשתנה, כי על ידי זה שאנחנו אומרים שהמשתנה הזה שווה יותר, אנחנו בעצם נותנים לו יותר כי הוא חוסך לנו שמירה וטעינה בין הבלוקים.

לאחר שנסיים להעריך את כל המשתנים ניתן עדיפות להקצאה על פי הניקוד של כל משתנה.

כמובן שגם השיטה הזאת לא מושלמת, מאחר ואם יש לנו משתנה שנמצא בין הבלוקים לסירוגין, כלומר הוא לא נמצא אף פעם ברצף, אבל הוא נמצא כל בלוק שני, הוא יקבל פחות נקודות ובהתאמה גם פחות הקצאות, למרות שהוא אולי שווה יותר ממשתנה אחר שנמצא בשני בלוקים רצופים, ואז יש לו דילוג. אנסה להראות את העניין בצורה מופשטת, נגיד שיש לנו 10 בלוקים, ואת מופעי המשתנים x,y הבאים –

x		x		x		x		x	
y	y							y	y

בספירה פשוטה, x יקבל רק 5 נקודות. לעומת זאת, ל-y יש ארבעה מופעים, כאשר הוא מופיע בצמדים, כלומר יש לו 8 נקודות. זאת על אף, שיש לו חלל עצום בו הוא בכלל לא שמיש, לעומת x ש"פעיל" לכל אורך העבודה. כלומר, יש פה הפסד די גדול, וחבל, אך זה המצב.

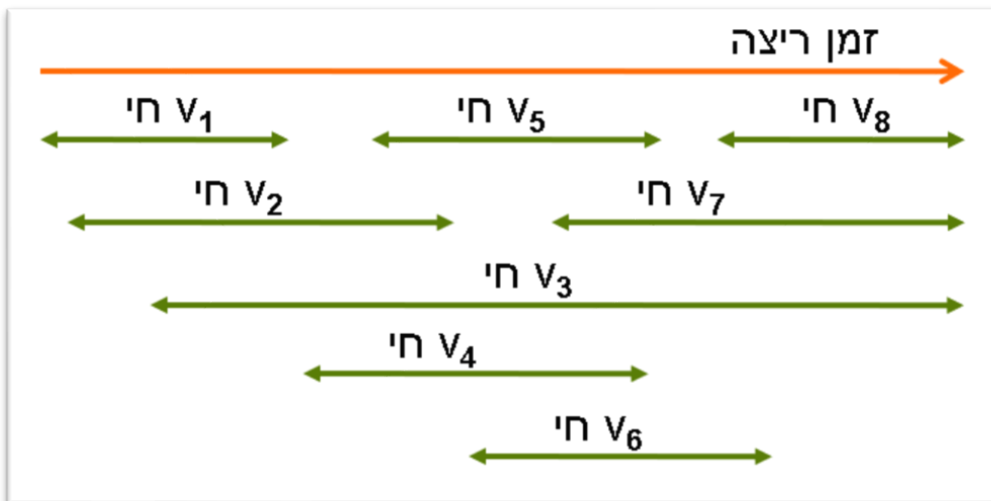
צביעת גרפים

נסתכל על דרך נוספת לבחירת הקצאת רגיסטרים – צביעת גרפים.

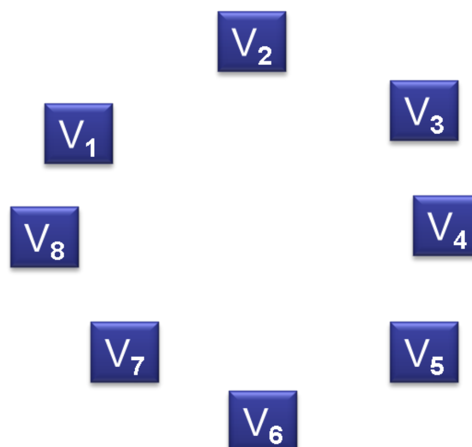
נתחיל קודם מהסיבה בגללה הגענו לזה – ברגע שנגיע למצב שיש לנו רגיסטרים מלאים, אנחנו נצטרך "לשפוך" (spilling) מידע החוצה מהרגיסטר. בכל התקלות כזו ה-spilling יעלה לנו לא מעט ברמת שמירת המשתנים וטעינתם. לכן אנחנו מנסים למצוא איזה דרך שתפוסת הרגיסטרים תהיה אופטימלית – כלומר, ככל שההחלטה שלנו לשמירת המידע תהיה יותר נכונה, כך יהיה לנו פחות spilling.

בשביל לפתור את הבעיה הזאת, אנחנו מנסים לגשת לבעיית צביעת הגרפים - נתון לנו גרף, ו-k מסוים. אנחנו רוצים לדעת האם בעבור k נתון אנחנו יכולים לצבוע את הקשתות באופן כזה, שבכל קדקוד יהיה רק מופע אחד מכל צבע. כלומר, אם יש לנו k=3 וקדקוד שיש לו 4 קשתות, אנחנו בבעיה.

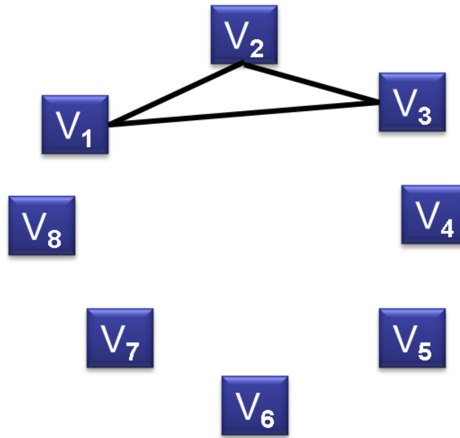
בשביל להתחיל את הפיתרון, ננסה קודם כל ליצור לנו איזה ציר זמן שיראה לנו מי עומד איפה –



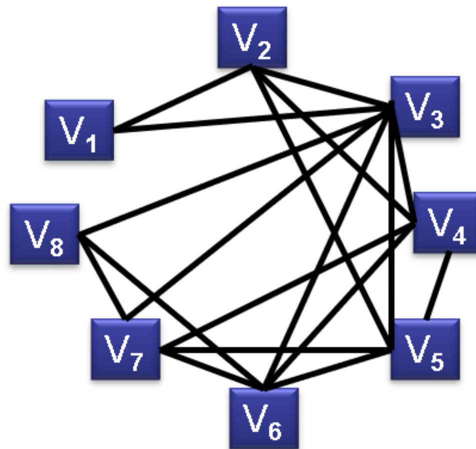
עכשיו, כמובן שאם יש לנו משתנה כמו v_3 שחי על טווח זמן ארוך זה לא אומר שהוא באמת פעיל לכל אורך הזמן, אלא שהוא לפחות פעיל בהתחלה ובסוף שלו. אנחנו רוצים לקחת את המידע הזה ולנסות ליצור ממנו גרף. איך נעשה את זה? קודם כל נגדיר שכל משתנה יהיה קדקוד –



בשלב הבא, אנחנו נמתח את הקשתות בין כל הקודקודים שחיים על אותו רצף זמן. כלומר, אם נסתכל על ציר הזמן למעלה, נראה ש-v_1,v_2,v_3 חופפים על אותו מסגרת זמנים, גם אם באופן חלקי. ולכן, אנחנו נמתח קשתות (דו כיווניות כמובן) בין שלושתם –

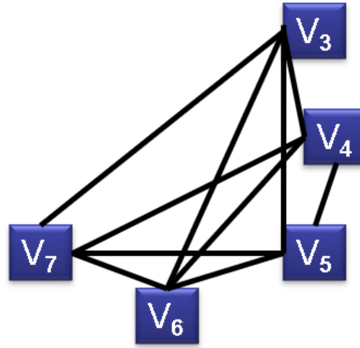


ככה נעשה לכל הקדקודים השונים, ולבסוף נקבל את הגרף הבא –



ננסה רגע להבין מה אנחנו מנסים ליצור פה. אם ניקח שוב את שלושת הקדקודים הראשונים עליהם עבדנו, נוכל להבין כי מאחר ומסגרת הזמן שלהם אחידה, עלול להיווצר מצב בו אנחנו נרצה ששלושתם יהיו בו זמנית בתוך רגיסטרים. ובהשאלה לבעיית הצבעים, אם יש לנו כחול, צהוב ואדום, זה יספיק לנו לצביעת הקשתות שלהם, בדיוק כמו שהרגיסטרים R1, R2, R3 יספיקו כדי להכיל את המשתנים.

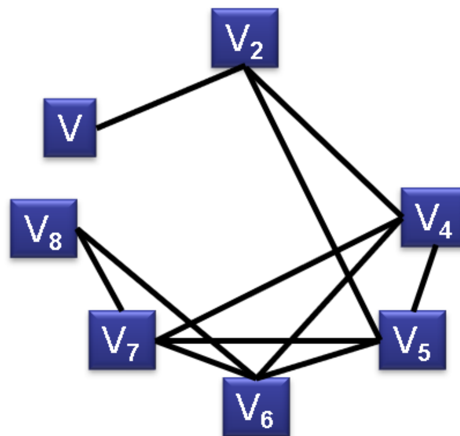
איך נעבוד בבחירת ההחלטה? בהינתן לנו k מסוים, קודם כל נוריד את כל הצמתים שהדרגה שלה (הקשתות היוצאות) קטנות מאותו k. כל הצמתים האלו שאנחנו נוריד (ונחזיר בהמשך) הם כאלה שאין לנו בעיה איתם, ואנחנו צריכים להתמודד רק עם האחרים. לצורך הדוגמה, נגדיר k=5 ונוריד את כל הצמתים בעלי דרגה פחותה מזו –



כמו שאפשר להבין, לא פתרנו את הבעיה. המצב האופטימלי ביותר, הא אם היינו עושים את השלב הזה ומקבלים גרף ריק. אך מאחר וזה לא המצב, אנחנו צריכים להתחיל ולחשוב מה אנחנו עושים הלאה.

השלב הבא באלגוריתם הוא להעיף (לגמרי) את אחד מהצמתים. יש לזכור שאנחנו מדברים פה על היוריסטיקה ולא אלגוריתם מושלם. מה ההבדל? היוריסטיקה מביאה לנו "אלגוריתם" שעובדה הרבה פעמים אך הוא לא אופטימלי באופן מוחלט. את זה אנחנו יכולים לראות בכלל ההחלטה של איזה צומת להוציא – אין כלל כזה. פשוט להוציא.

אנחנו נוציא את V_3, נחזיר את הצמתים שהורדנו לבדיקה ונראה מ הקיבלנו –



הגרף כבר הרבה יותר מרווח. אך האם זה מספיק? נבדוק – "נעלים" את הצמתים שדרגתם קטנה מ-k.

לא, זו לא טעות. הגרף ריק. הצלחנו לעשות זאת.

מה זה אומר לגבי v_3? שברגע שנגיע אליו לא תהיה לנו ברירה ונצטרך לטעון אותו לרגיסטר – כלומר, ליצור איזה spill. כמובן כאן בא לידי ביטוי הכישלון בגישה ההיוריסטית. מאחר ואין לנו כלל הכרעה, אין לנו שום ערבות שאנחנו מורידים משתנה נכון. יכול להיות שנכון יותר לחפש איזה גישה חמדנית כמו להוציא משתנה שיש לו כמה שפחות פגישות עם משתנים אחרים או אולי דווקא אחד שיש לו יותר. צריך לבדוק מה באמת יהיה יותר יעיל, אבל הבעיה היא שזה תלוי גם באופן כתיבת הקוד וכמה משתמשים באמת בכל משתנה, ולכן זה נשאר בגדר בעיה שפשוט נעשה מה שנראה לנו סביר, ונקווה לטוב.

אופטימיזציות

החלק המסיים של הקורס (יאייי!) הוא קצת ואין בו הרבה בשר, מלבד הרבה דברים שכבר נאמרו.

מה זה אופטימיזציות?

אופטימזציות זה הדרך שלנו לעשות כל מיני מניפולציות קטנות בקוד, על מנת לייעל את זמן העבודה. כמובן, שאנחנו מדברים על שינויים בתוכנית, הכוונה אינה לשנות את הסמנטיקה של התוכנית, אלא רק את דרך הפעולה שלה.

אנחנו לא מדברים על לעשות אופטימיזציה מושלמת לקוד, כלומר אנחנו יכולים להגיע לתכנית שהיא אופטימלית מכל הבחינות, אבל לייצר דבר כזה ייקח לנו הרבה מאוד זמן וכסף, אבל למי שיש, שיהיה לו בכיף.

דבר נוסף שיש להדגיש – אנחנו לא מייעלים את הקוד עצמו – אם האלגוריתם הנכתב הוא כתוב בצורה לא יעילה, יש רק מעט שאנחנו נוכל לעשות בשביל לשפר אותו. למשל, הדוגמה הידועה ביותר היא שאלת ++i,i++. ידוע לנו (ולמי שלא שידע) שכתיבה של i++ הרבה יותר יעילה מאשר ++i. כאשר אנחנו כותבים בצורה השניה, אנחנו בעצם מכריחים את המערכת לשמור את i הזה במקום נפרד, לעשות את החישוב ולשמור בחזרה. כאשר אם נכתוב רק i++ אנחנו יכולים לעשות את החישב הזה בצורה מקומית. השינוי הזה כל כך יעיל, שיש שפות שפשוט אוטומטית ממירות פקודה אחת לשניה בשביל לחסוך עבודה. מצד שני, אם כתבת אלגוריתם של $O(2^n)$ לא בטוח שזה באמת מה שיציל את התכנית.

אנחנו מדברים בדרך כלל על ארבע סוגים של אופטימיזציות עליהם אנחנו עובדים :

- זמן ריצה – אופטימיזציה הנפוצה ביותר. בסופו של דבר, אנחנו כותבים תוכנית להרצה, ואם אנחנו נכתוב קוד שלא רץ טוב, או שיש לנו אפשרות לייעל את הריצה שלו, חבל שלא נעשה את זה.
- גודל הקוד – למשל, אם יש לנו קוד מת אין סיבה שנתייחס אליו ונעבוד על האופטימיזציה שלו, אלא מחיקתו זוהי אופטימיזציתו.
- צריכת זיכרון – פחות נפוץ היום עם ירידת מחירי הזיכרון ויכולות הקיבולת, אבל חיסכון זה תמיד טוב.
- צריכת חשמל – בעיקר בחומרה שהיא קטנה ואמבדד, חשוב לנו להשתמש בכמה שפחות חשמל. לנו כרגע זה לא כל כך רלוונטי.

ממה נובע חוסר יעילות בקוד?

יש מספר סיבות בהם ניגע, לא משהו שהכי חשוב למבחן, אבל לחיים שהם המבחן האמיתי וכו'...

יתירות בתוכנית המקור – אם מתכנת כותב בצורה "יפה" יותר על אף שהוא יודע שיש עבודה נוספת על הקוד. למשל, 2*x יתורגם תמיד ל x+x. אך לנו כמתכתנים יותר קל לקרוא את x*2 ושיקפוץ הקומפיילר. מצד שני, יכול להיות גם שהמתכנת הוא לא המקש הכי רגיש במקלדת, והוא פשוט בכלי לדעת כותב תוכנית לא יעילה, גם זה קורה (כמובן לא אצל בוגרי המכון).

יתירות מכתיבה בשפה גבוהה – דומה קצת למה שהזכרנו מקודם – אם למשל אנחנו רוצים לגשת למערך, אנחנו נכתוב בדרך כלל a[i], למרות שהקומפיילר יתרגם את זה לחישוב של a+4*i. אם זה גם יהיה בתוך לולאה, אנחנו נצטרך כל הזמן לחזור ולחשב את התוצאה הזאת.

יתירות כתוצאה מתרגום – התרגום האוטומטי לא תמיד באמת מועיל. אין דוגמא.

אופטימיזציה של זמן ריצה

בשביל לייעל את זמן הריצה יש מספר דרכים שאנחנו יכולים לעשות. בניגוד למה שאנחנו עלולים לחשוב, לא מדובר תמיד דווקא בקיצור הקוד, לפעמים קוד ארוך הוא דווקא יעיל יותר.

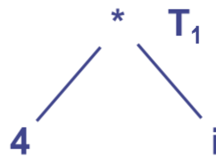
אנחנו נתמקד בשיטה אחת ספציפית לייעול קוד – בניית DAG, או גמ"ל למי שמעדיף.

יש לנו במצגת דוגמא קטנה ונאיבית, אך זו שאלה שחוזרת בהרבה מבחנים וכדאי לדעת את זה.

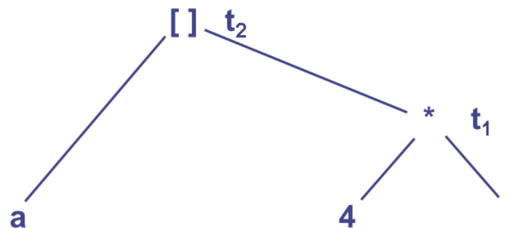
נתון לנו הקוד הבא –

```
(1):
t1 := 4 * i
t2 := a [ t1 ]
t3 := 4 * i
t4 := b [ t3 ]
t5 := t2 * t4
t6 := prod + t5
prod := t6
t7 := i + 1
i := t7
if i <= 20 goto (1)
```

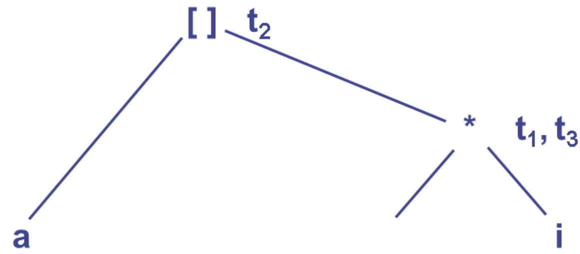
המטרה שלנו היא למצוא איזה דרך לייעל את הקוד. אם סתם ככה נסתכל נוכל לראות שיש לנו פה חישובם שחוזרים על עצמם, והשמות מיותרות, אך בשביל לדעת בצורה ברורה מה באמת ייעל נעבור שורה שורה, ונכניס את הפעולות והמשתנים לתוך הגרף. כאשר נקרא שורה, נכניס את האופרטור שיהיה אב של תת-עץ, ואת שהאופרנדים/משתנים נכניס בתור הבנים. את המשתנה אליו אנחנו משימים את התוצאה נכתוב כ"תכונה" ליד צומת האופרטור. למשל נסתכל על השורה הראשונה, אנחנו ניצור עבורה את הגמל הבא –



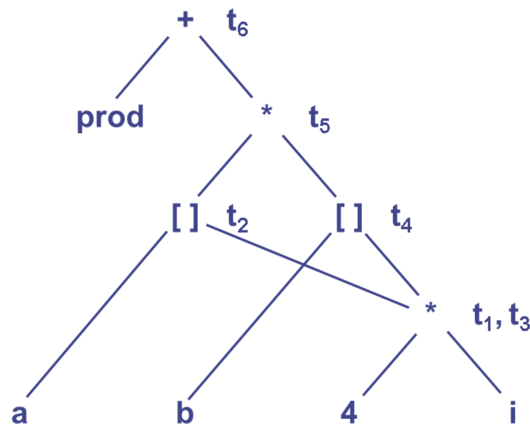
כשנרצה לעשות את השורה הבאה (השמה למערך), ואנחנו צריכים להשתמש ב-t_1 שאותו חישבנו זה עתה, נוסיף לעץ את הדבר הבא –



בשורה הבאה, אנחנו מזהים שיש לנו כבר חישוב שעשינו לפני לא הרבה זמן – 4*i. אין צורך לעשות את זה שוב. מה שנעשה הוא לצרף את המשתנה למעלה בצומת –

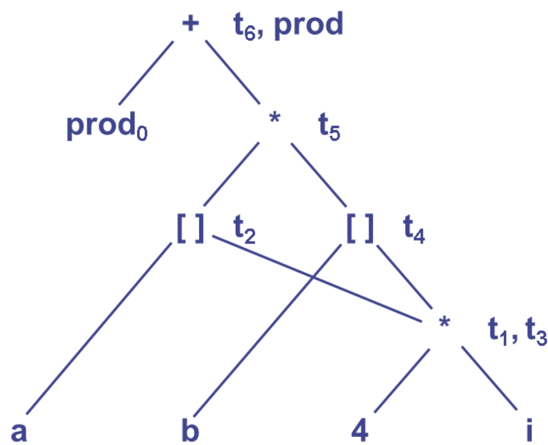


הרעיון פה די מובן ורץ כמעט עד הסוף. אנחנו ממשיכים עד השורה של t_6 := prod+t_5, ואיתו הדג הבא –

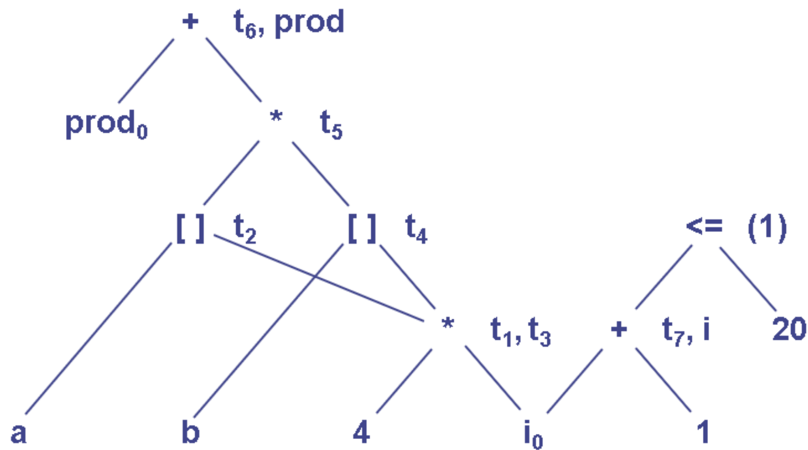


השורה הבאה, t_6 =: prod מציקה לנו מאחר והרגע הגדרנו את t_6, וגם את זה עשינו בעזרת prod, אז איך אנחנו מטפלים בזה?

קודם כל, ברור לנו שעכשיו אנחנו צריכים לשמור את prod ביחד עם t_6, אבל מה נעשה עם זה שהוא כבר כתוב מתחתיו? נסמן את הערך הקודם כ"מת" בעזרת 0 ליד שם המשתנה הישן –



עכשיו אפשר להמשיך ולבנות את העץ עד לסיומו –



כל שנותר לנו עכשיו הוא רק לייעל את הקוד בהתאם למה שעשינו.

למעשה, יש לנו שלושה מקרים שונים שאנחנו צריכים לטפל בהם, כל אחד מאופיין על ידי זה שיש לנו שני משתנים בצמתים. בעצם, אם ניקח את המשפט הבסיסי של קוד תלת מעני, אז אם יש לנו x := y + z, אם יש שתי אים שהם צמודים אחד לשני, זה אומר שמדור בדיוק על אותה שורה. נעתיק עכשיו את הקוד, נכניס בו את השינויים, ונראה בדיוק איך ומה אנחנו עושים –

(1):
t_1 := 4 * i
t_2 := a [t_1]
~~t_3 := 4 * i~~
t_3 := t_1
t_4 := b [t_3]
t_5 := t_2 * t_4
~~t_6 := prod + t_5~~
~~prod := t_6~~
prod := prod + t_5
~~t_7 := i + 1~~
~~i := t_7~~
i := i + 1
if i <= 20 goto (1)

נתחיל מלמעלה – קודם כל נוריד חישובים מיותרים. אם יש לנו כבר את הערך שצריך להיות בתוך t_3 במשתנה אחר, אז פשוט נעביר אותו (נזכור שאחרת אנחנו עלולים להתעסק עם ניהול רגיסטרים מיותר).

השלב הבא – אנחנו מחשבים לתוך t_6, כאשר השימוש היחיד שלנו במשתנה הזמני הזה הוא להחזיר את הערך לתוך Prod שעזר לו לחשב בהתחלה. כמובן שזה מיותר, ואנחנו פשוט יכולים לעשות את החישוב עצמו לתוך אותו משתנה, אנחנו חוסכים לנו כאן בוודאות רגיסטר כאשר אנחנו משתמשים רק בשני משתנים ולא בשלושה.

אותו דבר קורה גם לקראת הסוף – במקום לכתוב i++, מה שאומר שאנחנו צריכים לעשות חישוב לתוך משתנה זמני, ואז להעביר את התוצאה בחזרה ל-i, אנחנו פשוט מחשבים במקום.

לסיכום – הדברים אותם אנחנו יכולים לזהות על ידי בניית DAG –

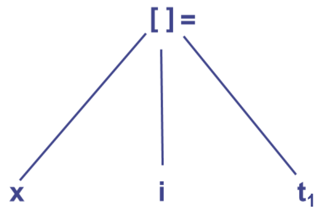
- ביטויים משותפים – כפל השמות וכדו'.
- זיהוי המשתנים בהם נעשה שימוש בקוד.

- זיהוי dead-code. חישוב ערכים שאינם בעלי שימוש בהמשך – אם אנחנו בונים תת עץ למשתנה כלשהו, אבל אף אחד לא נוגע באותו משתנה, הוא בוודאות מיותר.
- זיהוי תלות בין פקודות – על מנת לאפשר הזזת קוד לטובת ייעול, אנחנו יכולים לבחון את הdag ולראות האם יש חלקים שאינם דווקא תלויים באחרים, אך הזזת שורות הקוד שלהם יכולה לחסוך לנו (למשל, כך שלא יהיו לנו משתנים שחיים מעבר לרצוי).
- הימנעות מפקודות העתקה מיותרת – העתקת משתנים שלא רלוונטית יכולה להיחסך באופן הזה, וכך למשל אם אנחנו עובדים על שתי משתנים שיש להם את אותו ערך, אנחנו יכולים לעבוד רק על אחד מהם.

דבר אחרון שיש להעיר כי הוא חוזר לפעמים במבחנים. ראינו כיצד אנחנו עושים השמה למשתנה מתוך מערך, אך מה עושים כשצריך לעשות השמה לתוך המערך עצמו במיקום מסוים?

x[i] := 5

עלינו לזכור, כי בעצם מדובר על השמה רגילה, רק שיש לנו לעשות את החישוב של המיקום במערך. ולכן תת העץ שייבנה עבור יהיה כזה –



עד כאן.

בהצלחה!