עבודת גמר תכנון ותכנות מערכות - התמחות הגנת סייבר

**פיתוח מהדרים (קומפיילרים)**



שם התלמיד: דוד פלפלר

תעודת זהות: 215348004

שם בית הספר: תיכון מיתרים ברנקו וייס

שם המורה: רן ויינשטיין

**תוכן עניינים**

**מבוא**

**תיאור תכולת הספר**

מסמך זה נועד לתעד את פרויקט הגמר שלי, מטרתו היא לספק מידע על המוצר המפותח וכיצד להשתמש בו.

**תיאור כללי**

כפרויקט גמר בחרתי ללמוד ולפתח קומפיילר לשפת תכנות משלי, שיקח קוד משפה זו ויהפוך אותו לשפת אסמבלי של המעבד 8086. פרויקט זה יאפשר למתכנתים לכתוב קוד בשפה מצומצמת ופשוטה, בעברית ובאנגלית. כלי כזה יוכל לשמש למתחילים, וגם לילדים שאינם בקיעים באנגלית ולכן מוצאים קשיים בלמידת תכנות רגילה.

הסיבה לבחירתי בפרויקט זה היא העניין והסקרנות שהייתה לי לתהליך ההידור. בסופו של דבר, קוד שאנו כותבים בכל שפה, כולל אסמבלי, הוא הינו טקסט בלבד, ורק אנו יכולים להבין אותו. לכן התהליך בו אנו מתרגמים קוד זה לאפסים ואחדות תפס את תשומת ליבי. סיבה נוספת לבחירתי בפרויקט הייתה לראות אם זה בכלל אפשרי לתלמיד כמוני ליצור מערכת שכזו. לא הכרתי בקרבתי אנשים שעשו פרויקטים כאלה וככה יכולתי לאתגר את עצמי.

מכיוון שקוד המערכת שלי פתוח לכולם ומתועד היטב, שימוש נוסף לכלי יהיה ללמידת תהליך ההידור. במערכת קיימות פונקציות שמדפיסות שלבים מסוימים במהלך התהליך, ובנוסף לכך היא שומרת את קובץ האסמבלי שיצרה ללא צורך בהנדסה לאחור.

**פונקציונליות המערכת**

במערכת שמורות המילים הבאות:

print, if, else, while, int, string, return, main

או הגרסה העברית:

תדפיס, אם, אחרת, בזמןש, מספר, מחרוזת, תחזיר, ראשי.

מילים אלה משמשות לביצוע פעולות או הצהרות על משתנים מטיפוסים שונים.

במערכת ניתן ליצור פונקציות, לקרוא לפונקציות, להחזיר ערך יעודי מפונקציה, להצהיר על משתנים, לבצע פעולות אריתמטיות על טיפוס מספרי, להדפיס מחרוזות או משתנים מסוגים שונים ולתעד.

משמעות המילים השמורות:

|  |  |  |
| --- | --- | --- |
| דוגמה | משמעות ותיאור | מילה שמורה |
| print(“Hello, “, “world!”)  print(myNameString, “ is awesome!”)  print(“My favorite number is “, 18) | מקבלת מחרוזות או משתנים מסוג מחרוזת ומספר בתוך סוגריים, מופרדים בפסיק, ומדפיסה אותם לקונסולה. | print - תדפיס |
| if (x == 42) {  }  if (972 >= myNumber) {  }  if (myNumber != 1984) {  } | מקבל תנאי בוליאני מוקף בסוגריים, ונכנס לבלוק הקוד אחריו אך ורק אם התנאי נכון | if - אם |
| if (10 > 10) {  }  else {  } | נכנס לבלוק הקוד המופיע אחרי המילה, אך ורק אם התנאי בתוך הif הקודם למילה, שלילי | אחרת – else |
| while (x > 10) {  print(x);  x = x+1;  } | כל עוד תנאי שנמצא בתוך הסוגריים מתקיים, הבלוק העוקב ירוץ, בסופו הוא יחזור להתחלה ויבדוק את התנאי פעם נוספת | while – בזמןש |
| int x = 10;  int y = 30;  if (x < y) {  print(“x is smaller than y and it’s value is: ”, x);  } | הצהרה שתבוא לפני שם משתנה לבחירת המשתמש, ההצהרה תיצור משתנה בגודל 4 בתים בזיכרון, ותקשר אותו לשם הנבחר. ארבעת הבתים שומרים מספר. אם אחרי שם המשתמש מגיע = ולאחר מספר, המספר יושם למקום המתאים בזכרון | int – מספר |
| int func() {  return 4 \* 10 / 3 + (500 - 300);  }  Int func(int x) {  return x + 3030;  } | מחזיר ערך מפונקציה שמגיע לאחר המילה השמורה. | return - תחזיר |
| Int main() {  Return 0;  } | מסמן פונקציה ראשית, הקוד ייכנס אל התוכן שלה בתחילת הקוד. פונקציה זו חייבת להתקיים בכל קוד | main – ראשי |
| string message = “Hello, world!”; | טיפוס מחרוזת. שבעצם זה טיפוס מסוג const char\*, מחרוזת קבועה שלא ניתנת לשינוי. | string - מחרוזת |

\*תיעוד מגיע לאחר ‘~’. לדוגמה: int x = 10; ~ Declaring a new variable!

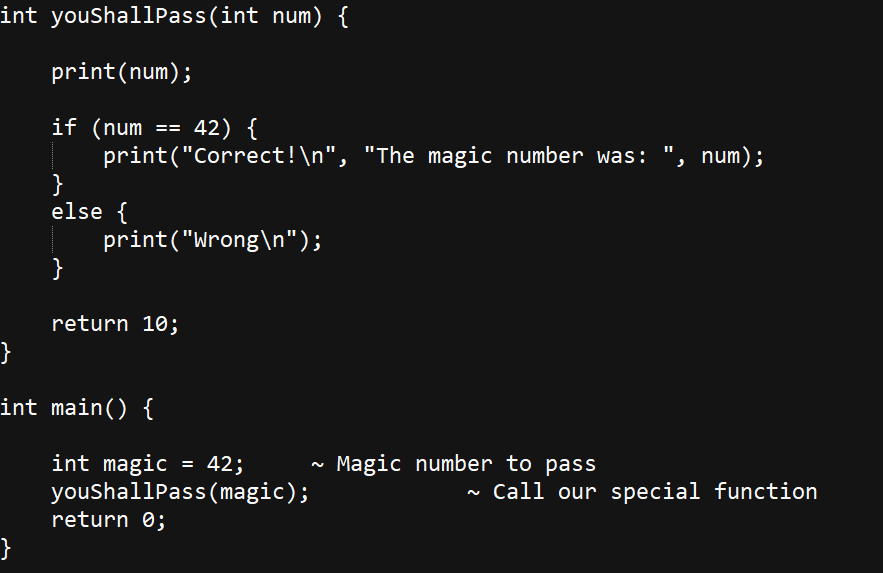
הקומפיילר יודע להתעלם מהבא אחרי סימן התיעוד ועד לשורה חדשה.

**אז איך זה עובד?**

אמנם המערכת שלי מצומצמת מאוד, אך התהליך שמעביר את הקוד הנתון לאסמבלי, במקרה שלי, MASM, הינו תהליך מורכב.

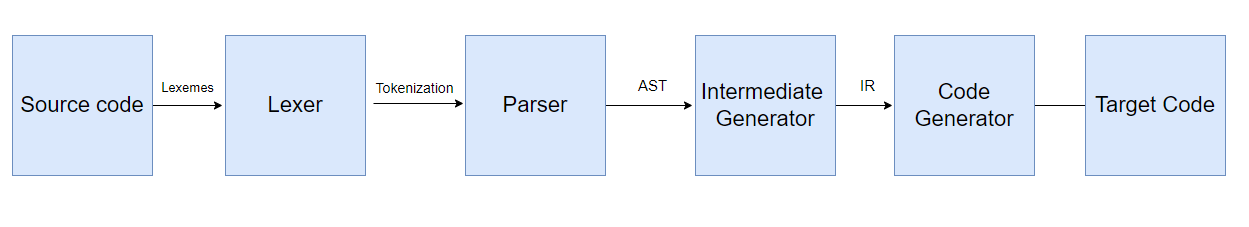
חשוב לציין שבהסברי על המערכת אני מתאר אך ורק כיצד הקומפיילר שלי עובד, ולא מדובר כאן על השיטה היחידה בה כל קומפיילר פועל. קיימות שיטות רבות לכל שלב ובחירתי בשיטות שאציג נעה מפשטות לעניין.

תחילה הקומפיילר מקבל טקסט זר, לדוגמה:



**מערכות ושלבי הקומפילציה**

הקומפיילר שבניתי מורכב ממספר שלבים שעובדים בנפרד או במקביל, ומספקים מידע אחד לשני, כעת אני אעבור על כל שלב ואסביר בפשטות מה המערכת עושה.



**לקסר**

המערכת הראשונה בתהליך הקומפילציה נקראת הלקסר (Lexer), ובה הקומפיילר לוקח את הטקסט הכתוב ומחבר אותו לרצף מבנים שנקראים תוקן, או איסמון (Token).

תוקן מכיל מידע על צירוף של לקסימס (Lexemes) שהן אותיות\צ'ארים בקובץ. שדות התוקן הם הערך הליטרלי שלו, והקטגוריה אליו שייך הערך.

לדוגמה, עבור קטע הקוד הבא: *int x = 10;* יווצרו מספר תוקנים שונים:

* המילה*int* תיצור תוקן, סוגו יהיה *Identifier* ומשמעותו תהיה ‘int’
* האות *x* תיצור תוקן משלה, סוגו יהיה *Identifier* ומשמעותו ‘x’
* האופרטור = יצור תוקן מסוג *Equals*ומשמעותו ‘=’
* *10* יצור תוקן מסוג *Number* ומשמעותו ‘10’
* לבסוף, *;* יצור תוקן מסוג *Semicolon* ומשמעותו ‘;’

כמה הערות:

* ניתן לראות שלתוקנים מסוימים, כמו  *Equals* או *Semicolon,* לא באמת צריך לייחס גם משמעות וגם סוג, משום שהינם מצביעים על אותו הדבר, אך אין זה מזיק, וככה אפשר ליצור מערכת עקבית יותר
* בעוד ש- *int* ו- *x* מהווים ערכים מסוגים שונים מבחינת היגיון, בכל זאת סוגם זהה בתור תוקנים, זאת משום שהפרדת המילים השמורות ושמות של משתנים ופונקציות נעשת מאוחר יותר. כרגע ניתן להתייחס אליהם כאל זהים, כלומר *Identifier* הוא סוג שמתאר כל שם שלא נמצא בתוך גרשיים (שמות בגרשיים הם מהסוג *String*)
* כמובן ש*-x* מורכב מלקסים אחד, אבל הוא גם יכול היה להיות מורכב מכמה
* כפי שניתן לראות, הלקסר מדלג על רווחים והם חסרי משמעות בקוד, סוג של, ברור שיש להם משמעות כאשר אנחנו עוברים על שם כלשהו, אנחנו נרצה לעצור ברגע שנגיע לרווח או תו מיוחד, אחרת יווצר מצב כמו *intx*

כעת, ממערך גדול של תוים בודדים, נוצר מערך גדול פחות של תוקנים. בפועל, הלקסר לא צריך ליצור מערך של תוקנים, אלא הוא מעביר בכל פעם שמתבקש את התוקן הבא, כך עד שהוא מגיע לתוקן שמסמל את סוף הקובץ והוא מסיים את עבודתו.

להלן מבנה הנתונים תוקן:

typedef struct TOKEN\_STRUCT {

char\* value;

enum {

TOKEN\_ID,

TOKEN\_NUMBER,

TOKEN\_EQUALS,

TOKEN\_DEQUAL,

TOKEN\_COMMA,

TOKEN\_STRING,

TOKEN\_SEMI,

TOKEN\_LPAREN,

TOKEN\_RPAREN,

TOKEN\_LESS,

TOKEN\_MORE,

TOKEN\_ELESS,

TOKEN\_EMORE,

TOKEN\_NEQUAL,

TOKEN\_NOT,

TOKEN\_MUL,

TOKEN\_ADD,

TOKEN\_DIV,

TOKEN\_SUB,

TOKEN\_LBRACE,

TOKEN\_RBRACE,

TOKEN\_FUNC\_END,

TOKEN\_NOOP,

TOKEN\_EOF,

} type;

} token\_T;

נחמד, אז מי בעצם מבקש מהלקסר להביא לו את התוקן הבא?

**פרסר**

כפי שציינתי, הפרסר ולהקסר עובדים במקביל, אך שאר השלבים אכן עובדים בנפרד, אחד אחרי השני.

תפקיד הפרסר (Parser) הוא לקבל בכל פעם תוקן אחד, וליצור בעזרתו עץ מופשט תחבירי (Abstract Syntax Tree) שנועד לייצג את הקוד במבנה של עץ. עץ כזה יכול להכיל מגוון חוליות מסוגים שונים.

**עץ מופשט חבירי**

להלן מבנה העץ:

typedef struct AST\_STRUCT {

enum {

AST\_PROGRAM = 24, // Continue enumeration from tokens (tokens.c)

AST\_FUNCTION,

AST\_DEF\_AMOUNT,

AST\_COMPOUND,

AST\_INT,

AST\_STRING,

AST\_ADD,

AST\_SUB,

AST\_MUL,

AST\_DIV,

AST\_ASSIGNMENT,

AST\_VARIABLE\_DEC,

AST\_VARIABLE,

AST\_FUNC\_CALL,

AST\_PARAM,

AST\_IF,

AST\_IFZ,

AST\_GOTO,

AST\_LABEL,

AST\_LOOP\_LABEL,

AST\_WHILE,

AST\_COMPARE,

AST\_PRINT,

AST\_RETURN,

} type;

// Compound

struct AST\_STRUCT\*\* children;

// Branch

struct AST\_STRUCT\* condition;

struct AST\_STRUCT\* if\_body;

struct AST\_STRUCT\* else\_body;

int type\_c; // Type of condition (>, ==, <=)

// Variable declaration

dtype var\_type;

// Functions

struct AST\_STRUCT\*\* function\_list;

struct AST\_STRUCT\*\* function\_def\_args;

struct AST\_STRUCT\* function\_body;

size\_t functions\_amount;

// Name used for different types of nodes, (e.g a variable node would have it's node->name be the identifier)

char\* name;

// Same for size

size\_t size;

// Int node

char\* int\_value;

// Function call

struct AST\_STRUCT\*\* arguments;

struct AST\_STRUCT\* leftChild;

struct AST\_STRUCT\* value;

struct AST\_STRUCT\* rightChild;

} AST;

קצת מבולגן, אז ככה:

נתחיל מ*type-*: כפי שניתן לראות זהו Enum בעל אפשרויות מגוונות. בקצרה, שאר ערכי העץ תלויים בערכו של *type* . נניח והשדה שווה ל- *AST\_FUNCTION,* כעת הקומפיילר ידע להתייחס רק לשדות הנוספים האלו: *function\_def\_args, function\_body וname-*

שימו לב ש *- function\_def\_args* הוא מסוג מצביע למערך של עצים אבסטרקטים תחביריים, משמעות השדה הוא להכריז על הכמות הלא ידועה מראש של פרמטרי הפונקציה. לדוגמה, עבור הפונקציה

*int foo(int x, string y)*

גודל השדה יהיה 2, והוא יכלול את שני תתי העץ כאשר ה- *type* שלהם שווה ל- *AST\_VARIABLE\_DEC*, והשדות הנוספים שלהם הם *var\_type, name ו- .value*

עבור *function\_body* שדה ה- *type* יהיה *AST\_COMPOUND* שזו חוליה שמובילה לעץ חדש שיכול להכיל מספר ילדים *children* לא ידוע. בעצם זוהי חוליה שמביאה לתת עץ המורכב ממערך *Statement* ים או הצהרות, אני אגע במשמעות של  *Statement*בהמשך.

כמובן שכל מה שהסברתי בחלק הזה אינו מפורט והוא מדבר רק על חלק ספציפי וקטן משאר העץ. אני אביא עוד כמה דוגמאות אבל אפשר להבין בפירוט טוב יותר עם קריאת הקוד עצמו.

החוליה הראשית ושורש העץ היא מסוג *AST\_PROGRAM* והיא מכילה חוליות מסוגי פונקציות, שמכילות חוליות מסוגי פרמטרים וגוף הפונקציה (גוף הפונקציה הינו *AST\_COMPOUND*), שמכיל מספר חוליות שמציינות הצהרות שונות, או עוד *AST\_COMPOUNDים*, ובהם עוד הצהרות.

*AST\_COMPOUND* היא בעצם חוליה שמציינת פתיחת סוגריים מסולסלים עד לסגירתם ({}), שם מצויות הצהרות אך אפשרי גם שחוליה זו לא תכיל שום דבר, כאשר אין קוד בין הסוגריים המסולסלים.

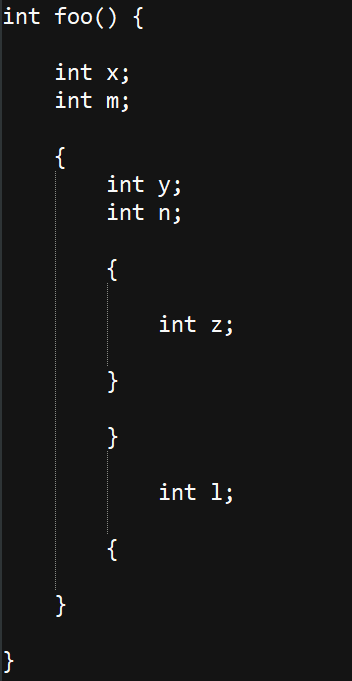
**הסבר קצר על *Symbol Table***

חשוב לציין שהפרסר, והשלבים הבאים בקומפיילר, מתקשרים עם מבנה שנקרא *Symbol Table* .

מבנה זה נועד למפות שמות של משתנים לטווחי הכרה שונים, או במילים אחרות .*Scopes* מבנה זה הוא עץ של מפות *Hash table tree*.

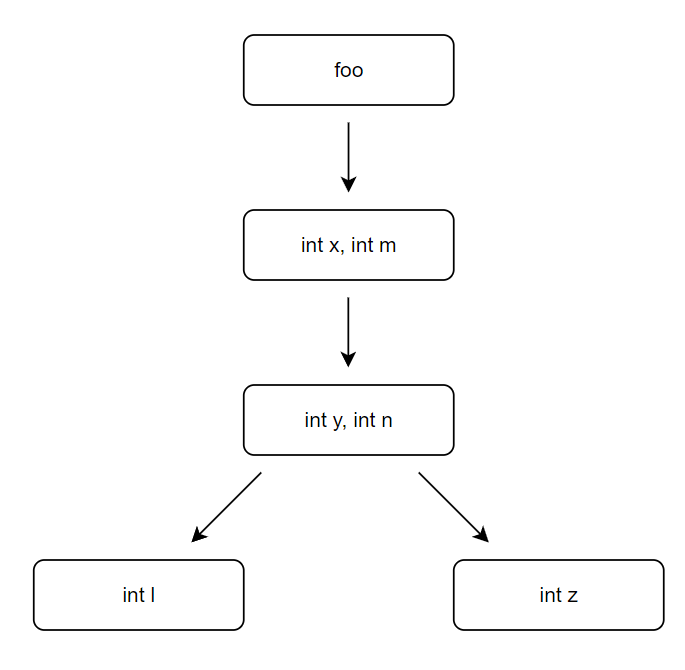
לדוגמה:

עבור קטע הקוד הבא



ידוע כי בטווח הפנימי ביותר הראשון, הקומפיילר מכיר את  *x, m, y, n, z* אך בטווח הקרוב החיצוני לו, המשתנה *z* כבר לא קיים, וכך הלאה. לכל *Scope* יש מפה שכוללת את כל המשתנים הנמצאים בה, ובנוסף לכך מכילה ילדים שהם הטווחים שהיא כוללת בתוכה, ושם נמצאים המשתנים שלהם בהתאם. כאשר מתכנת בוחר לכתוב פעולה מסוימת על משתנה, אנו נרצה לבדוק אם המשתנה קיים במפה המתאימה לטווח הנוכחי בו אנחנו נמצאים, אם לא, נעלה לאב המפה ונחפש שם, עד שנגיע לשורש. במידה ולא מצאנו: נזרוק שגיאת *Syntax* ונכתוב למשתמש שחסרה לו הכרזה על משתנה נתון. כך גם נעשה לפונקציות ומשתנים גלובלים, שנמצאים כולם בשורש העץ.

עבור התמונה הנ"ל, יווצר ה- *Symbol Table* הבא



**ירידה רקורסיבית**

ירידה רקורסיבית או *Recursive Descent* היא שיטה אחת בה הפרסר יוצר עץ מופשט תחבירי.

תחילה, קבעתי לשפה דקדוק מסוים, שאחריו יעקוב הפרסר שלנו בזמן ריצתו.

*program: function\*n, eof\**

*function: identifier, identifier, lparen, (identifier, identifier)\*n, rparen, block*

*block: lbrace, statement, rbrace*

*statement: block*

*|| if, lparen, expression, rparen, statement, else, statement if-else statement*

*|| while, lparen, expression, rparen, statement while statement*

*|| return, expression, semicolon return statement*

*|| identifier, identifier, equals, expression, semicolon assignment*

*|| expression, semicolon expression statement*

*expression: identifier, lparen, (identifier, comma)\*n, rparen function call*

*|| - || + term, - || + term add/subtract*

*term: || factor, / || \* factor multiplication/division*

*factor: identifier variable*

*|| number constant literal*

*|| lparen, expression, rparen parenthetical expressions*

*|| string string literal*

*\*eof = end of file*

הבהרות:

* n הוא מספר טבעי, בכל מקום בו הוא מופיע הכוונה היא שמה שמוכפל בו יכול להופיע כמות n פעמים
* הסימן ’||’ כוונתו ’או’, כלומר מה שנמצא אחריו הוא אחד מהאפשרויות להגדרת המושג
* צבע כתום הוא הגדרה חדשה
* צבע כחול הוא שימוש בהגדרה שעוד לו ניתנה
* צבע ירוק הוא שלא חייב שההגדרה תהיה קיימת
* צבע צהוב אומר הגדרה רקורסיבית
* צבע שחור אומר סוגי תוקנים

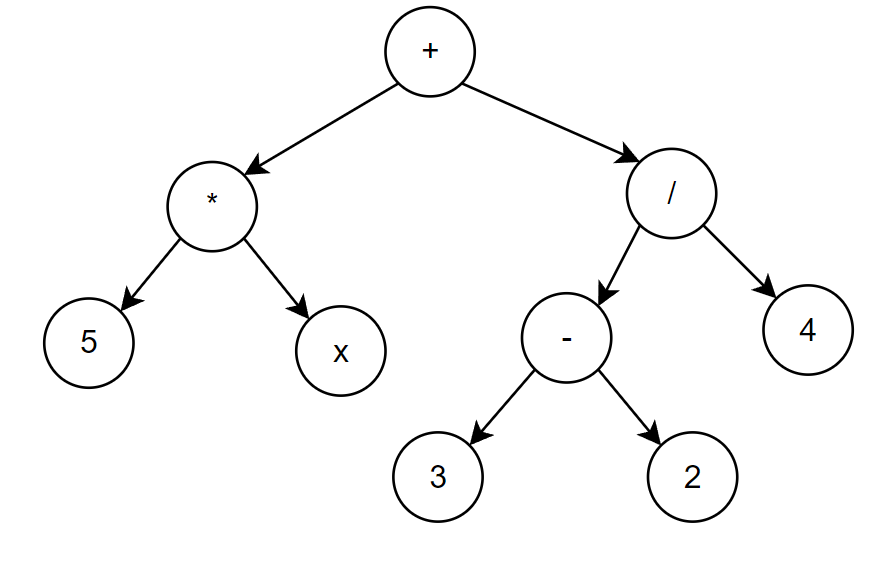
אז זהו בעצם חלק מהדקדוק של השפה שבניתי. בתחילת העבודה, הפרסר יחפש פונקציות, כלומר הוא יתחיל מ*- Program* וכך יתקדם לכל הגדרה. הדרך הכי ברורה בה ניתן להבין איך *Recursive Descent* עובד זה בעזרת פעולות חשבוניות וסדר פעולות חשבון. בוא ניקח את הפעולה הבאה: 5 \* x + (3 – 2) / 4

הפרסר יתחיל לקרוא את השורה הזאת בפונקציה המטפלת ב-*Expression* . *Expression* זה בעצםחיבור או חיסור בין *Term* שהוא בעצם כפל או חילוק בין *Factor ,* ו-*Factor* הוא מספר ליטרלי או *Expression* בסוגריים. לכן, תחילה הפונקציה תקרא לפונקציה של *Term* שתקרא לפונקציה של  *Factor* . הפונקציה הזותמצא את המספר 5 ותחזיר אותו ל- *Term* כתוקן של מספר. לאחר מכן *Term* ימצא את סימן הכפל ויהפוך אותו לשורש של עץ שילדיו הם 5 ו x ( *Term* יקרא ל- *Factor* פעם נוספת בשביל הילד השני).שורש העץ הזה יחזור לפונקציה שקראה לו *Expression* ויהפוך לילד של עץ ששורשו הוא סימן החיבור המופיע באמצע הביטוי. לאחר מכן*Expression* יקרא פעם נוספת ל-*Term*, הפעם לצד השני, שימצא את ה-*Factor* של (2 – 3(

וה-*Factor* של 4 ויסמן אותם כילדים של השורש חילוק.

\*הסיבה ש(2 – 3( הוא *Factor* היא משום ש-*Factor* יכול להיות *Expression* בתוך סוגריים.

לבסוף, יווצר העץ הבא:



בשיטה זו, הפרסר יסיים את פעולתו ויחזיר שורש לעץ שמכיל את כל התוכנית בצורת הדקדוק שקבענו לו.

בשיטה זו קיים גם *Predictive Parser* שהוא פרסר שלא צריך לבצע *Backtracking* \*. ניתן להשתמש בפרסר כזה רק לדקדוקים שעבור מספר טבעי *k* יוכל הפרסר לקבוע את צורת הייצור של  *k*  התוקנים הבאים, ולכן, אסור שהדקדוק יהיה דו-משמעי. לדקדוק כזה קוראים *Unambiguous Grammar*. הכוונה בדקדוק דו-משמעי היא שלשורת קוד אחת יכולה להיות אינטרפטציה לכמה עצים שונים.

*Backtracking\** הינו סוג אלגוריתמים בהם מוצאים מספר אפשרויות לפתרון בעיה וזורקים אפשרויות שהאלגוריתם מוצא כבלתי אפשריים לפתרון הבעיה.

כיום, קיימות מספר תוכנות שתפקידן הוא ליצור פרסר. אלו מעין שפות שמקבלות דקדוק אשר כתוב בסינטאקס וחוקים מסוימים, ויודעות ליצור ממנו פרסר העונה על הגדרותיו של אותו הדקדוק. תוכנה נפוצה כזו נקראת *Yacc* (*Yet Another Compiler-Compiler)*, בה השתמש *Bjarne Stroustrup*, מפתח שפת C++, למרות ש"נכשל בשל תחביר C". אמנם הגרסה הראשונה של C++ אכן נכתבה בעזרת *Yacc* .

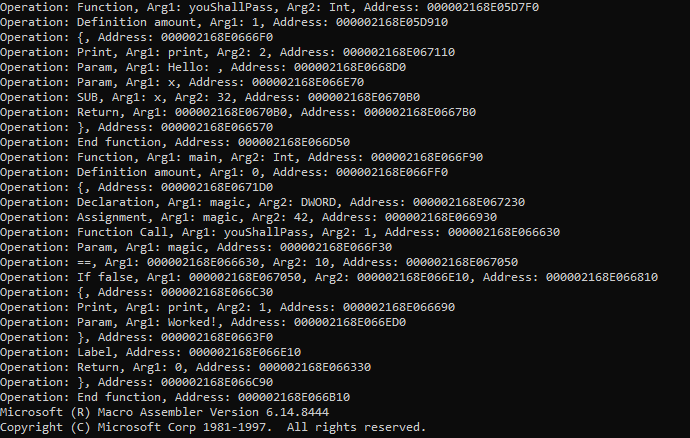
בפרויקט שלי החלטתי ליצור פרסר משל עצמי משום שחלק מאהבתי ל-*Low Level* היא יצירה והבנת כל תהליך המרכיב את מטרה מסוימת.

**ייצוג ביניים) (*Intermediate Representation***

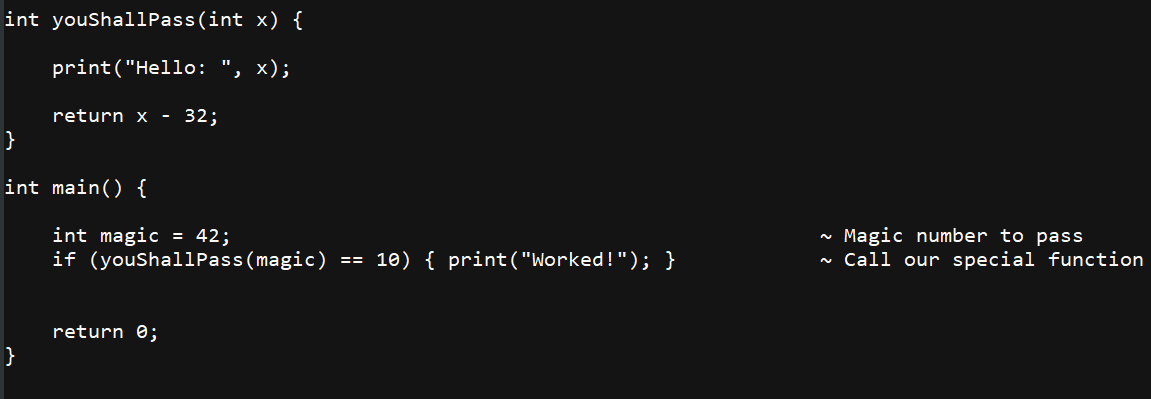
לאחר שהפרסר מסיים את עבודתו וחוזר לנוח, הוא שולח את שורש העץ התחבירי שיצר לשלב הבא, שהוא ייצוג הביניים. בשלב זה, הקומפיילר עובר על העץ בשיטת *Depth first* משמאל לימין (כדי לשמור על סדר פעולות), ויוצר קוד. הקוד שהקומפיילר יוצר בשלב זה הוא אינו ברמת אסמבלי עדיין, אלא ייצוג הנמצא בין השפה המקורית לבין אסמבלי. לכן, ייצוג הביניים היא הינה שפה משל עצמה, אז למה בכלל ליצור אותה ולא לעבור ישירות לאסמבלי, אתם בטח שואלים. למעשה, ניתן להעביר את העץ התחבירי לאסמבלי ישירות, ויש שפות שאכן עושות ככה, ולמעשה ייצוג הביניים של שפות אלו הוא העץ עצמו. לייצוג הביניים ניתן ליחס מבני נתונים שונים, וכל קומפיילר או אינטרפטר בוחר לייצג אותו בצורה שונה. לדוגמה, בפייתון ייצוג הביניים הוא גרף שמאפשר לניתוח זרימה וסידור מחדש לפני ביצוע הפעולות. אז למה עדיין אני ורבים אחרים מתעקשים ליצור ייצוג מופרד כשלב משלו? הסיבה לכך היא שעל ייצוג זה ניתן לבצע שלבים של אופטימיזציה וסידור מחדש, וכך קוד האסמבלי שלבסוף יתקבל יהיה יעיל יותר ממה שכתב המתכנת. כפי שציינתי, לייצוג ביניים ישנם מספר צורות ומבנים, אני בחרתי להשתמש בנפוץ מכל והוא *Three Address Codes*או בשביל נוחות הכתיבה, *TAC*. השם ניתן לייצוג הביניים הזה משום שבכל פקודה שהוא יוצר, יכולים להיות עד ל-3 אופרנדים.

**Three Address Codes**

להלן, דוגמה:



והנה הקוד שממנו נוצר הייצוג:



כל שורת *Tac* מכילה 4 שדות:

*Operation*: הפעולה של השורה על האופרנדים

*Arg1*: האופרנד הראשון

*Arg2*: האופרנד השני

*Address*: כתובת הפעולה

בשימוש ב-*Tac* ישנם שני סוגים עיקריים של יצוג, *Quadruples* ו-*Triples*. ההבדל היחיד ביניהם הוא שדה

*Address* שקיים רק ב-*Triples*.

**Quadruples**

המימוש היותר נפוץ הוא דווקא של  *Quadruples*, בו קיים שדה של  *Temporary*. שדה זה שומר משתנים

שהקומפיילר יוצר, ולא המתכנת. משתנים אלה מחזיקים מידע של פעולות המרכיבות פעולה ארוכה יותר של

המתכנת. לדוגמה, עבור שורת הקוד:

*x = 3 + 4 \* 5 / 2;*

הייצוג יראה כך:

t1 = 4 \* 5 {Operation = Multiplication, Arg1 = 4, Arg2 = 5, Temp=t1}

t2 = t1 / 2 {Operation = Division, Arg1 = t1, Arg2 = 2, Temp = t2}

t3 = t2 + 3 {Operation = Addition, Arg1 = t2, Arg2 = 3, Temp = t3}

x = t3 {Operation = Assignment, Arg1 = x, Arg2 = t3}

בתוך הסוגריים המסולסלים כתבתי את אותו הדבר רק בהפרדה של כל שדה. תחילה, הקומפיילר יכפיל את 4 ו-5 וישמור את התוצאה ב-*t1* שאותו הוא יחלק ב-2 וישמור תוצאה זו ב-*t2*. לאחר מכן יחבר הקומפיילר את *t2* ו-3 ולבסוף ישים את התוצאה ב-*x*. פקודות אלה דומות מאוד לאסמבלי אבל עדיין שונות ממנו. נראה זאת גם אחר כך, אבל כדוגמה, במעבד 8086 פעולת כפל מקבלת רק אופרנד אחד בשונה מייצוג זה, ומכפילה אותו ברגיסטר *EAX*.

**Triples**

אני בחרתי להשתמש ב-*Triples*, ששומר שדה בשם *Address*. למעשה ההבדל היחידי הוא שבמקום שיהיו לי

משתנים זמניים, אני משתמש במיקום בזיכרון של אופרציות מסוימות כמשתנה שלי, וכך אני חוסך מקום.

מהדוגמה שהבאתי למעלה, בה רואים את ייצוג הביניים שלי מודפס ב-*Cmd*, ניתן לראות איך אני משתמש

במיקום בזיכרון של פעולה כ-*Argument* בפעולה אחרת.

לדוגמה, כאן:



אני מחסר מ-*x* את 32 ומחזיר את התוצאה, שהיא המקום בזיכרון של השורה הראשונה.

**שימוש בייצוג הביניים לאורך כל התוכנית**

כמובן שעד עכשיו הראתי רק כמה דברים שאני עושה עם ייצוג הביניים, כמו פעולות חשבוניות. אך בעצם לכל

פעולה יש דרך אחרת לקרוא את ייצוג הביניים. לדוגמה, בקריאה לפונקציה, יכולים להופיע מספר לא ידוע של

פרמטרים. לכן, כך יראה הייצוג:



בעת הקריאה לפונקציה, *Arg1* יהיה שווה לשם הפונקציה, ו-*Arg2* יהיה כמות הפרמטרים המועברים

לפונקציה, במקרה שלנו 3. לאחר מכן יופיעו 3 פעולות נוספות, לכל פרמטר, בו יש רק *Arg1* שמציין את ערך

הפרמטר. פרטרים אלה נדחפים בסדר הפוך מהקריאה, כלומר עבור הייצוג הנ"ל, קריאת הפונקציה תראה

כך:

*youShallPass(magic, 10, 30);*

הסיבה שפרמטרי הפונקציה מיוצגים בסדר הפוך היא שכך הם גם נדחפים למחסנית באסמבלי. כלומר, הערך

הראשון שנדחוף למחסנית יהיה האחרון שנוציא שזה הפרמטר האחרון שנרצה.

כמובן שיש עוד המון דוגמאות לאיך בחרתי ליצג כל פעולה, אבל אין סיבה לפרט על כולן ובנוסף לכך, קל להבין איך כל דבר עובד מהצורה בה הוא כתוב. בקומפיילר שלי יש פונקציה בשם *traversal\_print\_instructions* שמקבלת רשימה של *Tac*יםומדפיסה אותם, כמו מה שהראתי קודם לכן. מה שחשוב לדעת הוא שתלוי בשדה ה-*Operation*, הקומפיילר ידע את משמעותו של כל *Argument* וכך הוא ידע ליצור קוד אסמבלי מתאים.

דבר נוסף- משורות הקוד בתמונות למעלה, ניתן לראות שלפעמים *Argument* מסוים מציג מחרוזת ולפעמים כתובת או פוינטר. ידוע שב-*C* כדי להדפיס מחרוזת או להדפיס פוינטר צריך לעשות פורמט שונה. לכן, *Argument* הוא מסוג *void\**. אך בשל זאת אין דרך לדעת למה *Argument* מצביע בכל רגע נתון, הרי אין דרך לבדוק זאת. בשביל פתרון הבעיה השתמשתי במבנה הנתונים *Tagged union* שנועד לייצג משתנה שיכול לתפוס סוגים קבועים אך שונים.

*Argument* יכול להיות מהסוגים הבאים:

* *TAC\**: מצביע לפקודת ייצוג ביניים אחרת, כלומר ה-*Address* שלה
* TEMP*\**: גם הוא מצביע לפקודת ייצוג ביניים אחרת
* *CHAR\**: מצביע לשם מסוים

ההבדל היחידי בין *TAC\** ל-*TEMP\** היא ש-*Argument*ים מסוג *TAC\** נוצרים בזמן תהליך ייצוג הביניים ו-*TEMP\** נוצרים בשלב יצירת הקוד המגיע בהמשך.

**אופטימיזציות**

תחום האופטימיזציה הוא תחום אין סופי שמיחד קומפיילר אחד מקומפיילר אחר. בעצם מה שעושה קומפיילר טוב יותר או פחות טוב, הוא בדיוק החלק הזה. ניתנת השאלה של מהירות נגד יעילות. נניח והקומפיילר טוב (עובד תמיד ולא מאבד מידע), אנו רוצים שיעבוד כמה שפחות זמן בהתליך הקימפול אך יממש את האופטימיזציה הטובה ביותר לקוד המתכנת. כאן נשאלת שאלה של צורך, ולכל אחד מהצרכים עונים קומפיילרים שונים. לדוגמה, הקומפיילר המפורסם, Gnu Compiler Collection או Gcc, הוא קומפיילר שיודע לעשות אופטימיזציה מדהימה שהופכת קוד מתכנת ליעיל בהרבה בעזרת שיטות מתמטיות מסובכות ורבות, אך החיסרון כאן הוא איטיות. כלומר, תהליך הקימפול יהיה איטי יחסית, אך כל הרצת הקובץ הבינארי המתקבל תהיה מהירה מזאת של קומפיילר אחר ללא אופטימיזציה מסובכת.

משום שאיני בן אלמוות, לא השקעתי עשרות שנים בפיתוח אופטימיזציות כמו שעשו המפתחים של Gcc, אך כן ממשתי כמה וכמה אופטימזציות משלי.

ניקח לדוגמה את האופטימיזציה הפשוטה ביותר: לאחר שמגיעים להצהרת החזרה מפונקציה, אני נותן לקומפיילר לדלג על שורות הקוד עד לסוגריים מסולסלים סוגרים (‘}’). זאת משום שקוד לא יקרא לעולם אחרי הסימן הזה. עוד דוגמאות לאופטימיזציה פשוטה מאוד הן כפל או חילוק ב-1 שלא עושה כלום ולכן נמחק מייצוג הביניים (כך גם לחיבור וחיסור באפס). דוגמה לאופטמיזציה פשוטה נוספת אך קצת שונה היא פעולה לפעולה שבסופו של דבר תביא ערך 0, ניתן להשתמש באסמבלי ב-*XOR* במקום הפעולה עצמה. כאשר משתמשים בשער לוגי זה על רגיסטר עם עצמו, אנו מאפסים את ערכיו.

אופטימיזציה נוספת היא הדרך בה מתורגמים תנאים לייצוג הביניים. במקום להשתמש בפעולה המתאימה של אסמבלי לכל תנאי, אנו משתמים בפעולה ההפוכה לה. כך אנו נדרשים לקפוץ לפחות מקומות בזיכרון.

לדוגמה:

*if (x == 100) { print(“Hello”);  
else {print (“Bye bye!”);*

אנו נרצה לבדוק אם המשתנה  *x* שונה מ-100. אם כן, נקפוץ למקום בזיכרון בו מתחיל ה-*Statement* הראשון (והיחיד במקרה הזה) של *else* . אחרת, לא נקפוץ לשום מקום ונמשיך לבצע את ה-*Statement*ים של ה*-if.*במקרה ו-*x* אכן שווה ל100 נרצה גם לקפוץ בסוף כל הפעולות למקום בזכרון שנמצא אחרי פעולות ה-­*else.* כך יצרנו רק שני לייבלים שונים ששומרים מקומות בזיכרון הקוד סגמנט.

ישנו מקרה בו קוד בייצוג הביינים יקפוץ ללייבל, שהוא מקום בזיכרון הקוד, והלייבל הזה יקפוץ למקום אחר. כלומר אנו נגיע מנקודה A ל-C דרך B כך:

A: goto B

B: goto C

הדוגמה האחרונה לאופטימיזציה שעשיתי היא למחוק את הקישור הזה כך ש-A יקפוץ ישירות ל-C.

A: goto C

כשנתקלתי בתופעה זו חשבתי שהיא ייחודית לדרך בה אני ממשתי את הקוד ולהפתעתי שמחתי לגלות שזהו מקרה ידוע שקורה בפיתוח קומפיילרים.

**יצירת קוד Code Generation**

הגענו לחלק האחרון! החדשות המצערות הן שזה החלק הקשה\מסובך ביותר למימוש (אחרי אופטימיזציה כמו של *GCC*). יש לנו רשימה שלמה של *TAC*ים ולמעשה יש לנו את כל קוד ייצוג הביניים מיועל. עכשיו נותר לנו לעבור עליו ולהפוך אותו לאסמבלי בעל סינטאקס MASM.

**MASM**

MASM או  *Macro Assembler* הוא אסמבלר למעבד אינטל x86 שיצרה חברת מיקרוסופט. אודה שלא ידעתי כלום על MASM לפניתכנון ויצירת הקומפיילר, אבל בדיעבד בחירתי הייתה נבונה. הסיבה לכך היא שב-MASM ישנם מגוון Macroים שימושיים לכתיבת קוד אסמבלי קל יותר. לדוגמה, במקום לזכור מיקומים בסטאק של כל משתנה לוקאלי, נוכל להשתמש בשמות שמצביעים למקומות בסטאק פריים וכך להקל על העבודה. בנוסף לכך, נוכל להדפיס ערכים של רגיסטרים בעזרת פרוצדורות מובנות וכך להדפיס משתנים מסוג *int* בקלות.

**הקצאת רגיסטרים Register Allocation**

תאורטית, יצירת קוד אסמבלי מייצוג הביניים אינו כל-כך קשה, אפשר ליצור משתנים ולשנות אותם בידיוק כמו שעשה המתכנת המקורי. לדוגמה, אם המתכנת שינה ערך של משתנה, נוכל פשוט לשים את הערך החדש במשתנה ולהמשיך הלאה. כך נוכל לסיים את העבודה במהרה ולהוציא את הקומפיילר לשוק החופשי. אבל אם אפשר לעשות משהו חכם ויעיל יותר – מי אנחנו שנסרב? בעצם שיטה זו אפשרית אבל היא טיפשה, מאוד. הסיבה לכך היא שפעולות שנוגעות בזיכרון התוכנית לוקחות זמן רב. אז מה עושים? למזלנו כשנברא המחשב נבראו הרגיסטרים. רגיסטרים נועדו לעבד מידע, והם עושים זאת מהר. אם נשתמש אך ורק ברגיסטרים נוכל ליעל בהרבה את הקוד שלנו, אבל אי אפשר להשתמש אך ורק ברגיסטרים, לפעמים נצטרך לשמור מידע בזכרון. בכל מקרה, נצטרך לנצל כמה שיותר את הרגיסטרים ולהפחית בשימוש בזכרון. תהליך זה נקרא הקצאת רגיסטרים.

דוגמה מעניינת היא הפעולה *x = y;*

אופציה גרועה תהיה הפעלת הוראת הזזה של הערך של *y* לתוך *x* , להלן:

*MOV EAX, [y]*

*MOV [x], EAX*

אנו משתמשים כאן פעמיים בגישה לזכרון, לא טוב.

במקום זה, נוכל לא לעשות כלום. כלומר, בקוד אסמבלי לא נעשה כלום, בפועל נזכור שבכל שימוש ב-*x* נחליפו ברגיסטר שמחזיק את *y*. או במילים אחרות, נזכור איזה רגיסטר מחזיק את *y* ונעדכנו כך שיחזיק גם את *x*. חסחנו כך בדי הרבה זמן. אבל יש עוד מקרים בהם נצטרך לבחור בצורה חכמה באיזה רגיסטר להשתמש ולצורך מה.

אבל לפני שנראה כיצד אנו עושים זאת, אפרט על שני הטבלאות שנשתמש בהן.

**Register Descriptor**

לכל רגיסטר ב-x86, או לפחות לכל רגיסטר שנרצה להשתמש, ניצור רשימה השומרת את מצבו, כלומר נשמור איזה ערכים או משתנים הוא מחזיק בכל רגע נתון. לא יותר מידי מסובך אבל זה יעזור לנו ליעל את קוד האסמבלי בהרבה.

**Address Descriptor**

נשתמש ב-*Symbol Table* שלנו מהפרקים הקודמים כדי לסדר רשימה לכל משתנה או *Symbol Table Entry* שמסמנת באיזה רגיסטרים נמצא כל משתנה בטווח ההכרה הנוכחי של התוכנית. במילים אחרות, לכל משתנה ב-*Symbol Table* שלנו תהיה רשימה של רגיסטרים או שמות שמחזיקים את הערך הנוכחי שלו. למה שמות? הסיבה לכך היא שבתחילת התוכנית כל משתנה יופיע בתוך הרשימה של עצמו. הכוונה היא שבהתחלה משתנה מחזיק את הערך האמיתי שלו – אך ברגע שמבצעים פעולת השמה על המשתנה לדוגמה, נצטרך למחוק את שמו מהרשימה ולהחליפו ברגיסטר שיחזיק את ערכו החדש.

להלן דוגמה לשני הרשימות בפעולה

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| AX | BX | CX |  | a | b | c | d | t | u | v |
|  |  |  |  | a | b | c | d |  |  |  |

t = b – a

MOV AX, [a]  
 MOV BX, [b]  
 SUB BX, AX

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| a | t |  |  | a, AX | b | c | d | BX |  |  |

u = a - c

MOV CX, [c]  
 SUB AX, CX

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| u | t | c |  | a | b | c, CX | d | BX | AX |  |

v = t + u

ADD AX, BX

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| v | t | c |  | a | b | c, CX | d | BX |  | AX |

a = d

MOV CX, [d]

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| v | t | a, d |  | CX | b | c | d, CX | BX |  | AX |

d = v + t  
 MOV [a], CX  
 MOV CX, AX  
 ADD CX, BX

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| v | t | d |  | a | b | c | CX | BX |  | AX |

* *t, u* ו-*v* מתחילים ללא שמם ברשומת *Address Descriptor* . זאת משום שבדוגמה שהבאתי הם לוקאלים לבלוקשל קטע הקוד. הם לא ישמרו לאחר יציאתו ולכן עדיין אין בהם ערך התחלתי שמגיע מלפני.
* הרגיסטרים בדוגמה הם של 16 ביטים, למרות שבמציאות כנראה שנפעל עם 32\64
* דוגמה זאת היא רק אופציה אחת לקוד האסמבלי. כלומר, יכולות להיות דוגמאות לא פחות יעילות שמיצרות קוד אסמבלי אחר

בדוגמה עצמה ניתן לראות כיצד אנו מגדירים ומחליפים ערכים בין רגיסטרים לצורך יעילות. דבר שלא ציינתי הוא שבסוף בלוק הקוד הזה נצטרך לשמור את ערכי המשתנים שקיימים אחריו (*Live on exit*). נעשה זאת בעזרת *MOV*:

*MOV [d], CX*

בדוגמה הנ"ל ניתן לראות כיצד אנו חוסכים שימוש בזכרון כל עוד אפשר. אבל איך יודעים מתי ניתן לשנות, להחליף או להקצות רגיסטר למשתנה וכדומה? או בעצם, איך נוכל למצוא רגיסטר פנוי עם הכמות הכל-כך קטנה שיש לנו במעבדים מסוג *CISC* ?

נעקוב אחר השלבים להקצאת רגיסטר ל-*y* עבור הפעולה הבאה:

*x = y + z;*

1. אם *y* נמצא ברגיסטר, נבחר את הרגיסטר שמכיל אותו ולא נכתוב שום קוד אסמבלי.
2. אם *y* לא נמצא ברגיסטר, נחפש רגיסטר ריק שאליו נטען את *y* .
3. אם אין רגיסטרים ריקים, נצטרך בכל זאת למצוא דרך להשתמש ברגיסטר מסוים מבלי לאבד אף מידע.

ניתן דוגמה לכך עבור רגיסטר  *R*. נניח והמשתנה  *u* נמצא כרגע בתוך ה-*Register Descriptor* של  *R*.   
כדי למצוא אם ניתן להשתמש ב-*R*, נבדוק שהערך של-*u* לא באמת חשוב לנו, או שיש מקום אחר בו נוכל להשיג אותו. האפשרויות הן:

* אם ה-*Address Descriptor* של *u* אומר ש-*u* נמצא במקום אחר מלבד *R*, נוכל להשתמש ב-*R*
* אם *u* הוא ה-*x* מהפעולה, ו-*x* לא שווה לאחד מהאופרנדים האחרים בפעולה (בדוגמה הזאת *z* ), אז נוכל להשתמש ב-*R*. הסיבה לכך היא ש-*x* יאבד את ערכו הישן לאחר הפעולה ואין סיבה לשמור אותו ברגיסטר. אבל במידה ו-*x* היה גם האופרנד *z*, לא היינו יכולים להשתמש ב-*R*, זאת משום שערכו כן היה חשוב לנו לחישוב הפעולה ולא תהיה אפשרות להתעלם ממנו.
* אם  *u* לא שווה ל-*x*, נבדוק בכל זאת אם הוא נמצא בשימוש לאחר הפעולה הנוכחית ועד לסוף בלוק הקוד, אם לא, נוכל להשתמש ב-*R*. חשוב לזכור שבמקרה ו-*u* קיים גם לאחר בלוק הקוד נצטרך לבדוק אם ערכו ישתנה בבלוק.
* אם שום דבר לא עבד, נהיה חייבים להשתמש בזיכרון. נשתמש בפעולה *MOV [u], R* כדי לשמור את הערך האחרון של *u* חזרה אליו. קוראים לפעולה הזאת *Spill* .