פתח דבר

החלק הראשון בסקר ספרות זה עוסק בקשיים של מתחילים בלימוד תכנות ומושגי יסוד במדעי המחשב. המחקרים המרכזיים בנושא נערכו בשנות השמונים, תקופה בה נערכו מחקרים דומים בתחומי דעת אחרים כגון הוראת מתמטיקה ופיסיקה. הממצאים יצרו מוטיבציה לפיתוח פתרונות וכלים לטיפול בקשיים ולהפחתתם - שיטות הוראה, איתור שפות תכנות מסחריות המתאימות לשמש כשפה ראשונה להוראת תכנות, פיתוח שפות תכנות ייעודיות ללימוד מושגי יסוד בתכנות, פיתוח מערכות הוראה נבונות (ITS) לאיתור שגיאות וניתוב התלמיד לתרגול מתאים, ופיתוח סביבות ממוחשבות המיועדות להמחשת תהליכי חישוב (כדוגמת ויזואליזציה).

במאמר המתייחס לאוספת מאמרים העוסקים בתכנות המתחיל ((Soloway & Spohrer (eds. ,1989) מעלים Gray ועמיתיו (1993) את השאלה הבאה: "האם ספר שיצא לאור בשנת 1989, ומכיל מאמרים שפרסמו לראשונה במחצית שנות השמונים, או מוקדם יותר, אקטואלי לקריאה ב- 1993?". תשובת החוקרים על שאלה זו היא "כן ודאי".
שאלה דומה עולה על אחת כמה וכמה בתחילת שנות האלפיים. האם קשיי מתחילים שאותרו בשנות השמונים רלוונטיים גם לתקופתנו? ובפרט, האם תלמידינו בארץ, המתחילים את צעדיהם הראשונים בתכנות ובלימוד יסודות מדעי המחשב, צפויים לחוות קשיים הדומים לאלו שזוהו ותוארו בעבר?
ובכן, תשובתנו לשאלה זו גם היא חיובית בהחלט. גם מחקרים שבוצעו לאחרונה מצביעים על תופעות דומות. למשל, פז (1995), ו- Haberman ו- Ben-David Kolikant (2001) מצאו שגיאות מסורתיות אצל תלמידי תיכון בישראל בשלב התחלתי של לימוד יסודות מדעי המחשב.

מטרת סקר ספרות זה להציג בפני ציבור המורים למדעי המחשב סקירה מתומצתת של מחקרים שמטרתם היתה לאתר, לסווג ולהסביר קשיי מתחילים, העושים את צעדיהם הראשונים בלימוד יסודות מדעי המחשב, תוך התנסות בתכנות בסיסי.

- הפרק הראשון הוא פרק מבוא המציג סקירה מתומצתת של קשיים, שגיאות ותפיסות של תלמידים מתחילים הלומדים קורס מבוא בתכנות.
- הפרק השני מציג תקצירים של שישה מאמרים נבחרים המתייחסים להיבטים שונים של קשיי תלמידים מתחילים. הפרק מפרט שגיאות שכיחות ותפיסות מוטעות, מציג הסברים אפשריים לקשיים, ואת המלצות החוקרים לגבי דרכי טיפול אפשריות למיגור השגיאות ולמניעתן.
- מצורפת גם רשימת מקורות עשירה ומומלצת לקריאה.

מבוא

מטרת המבוא להציג בפני המורים סקירה מתומצתת של תוצאות מחקרים שנערכו במטרה לאתר, לסווג ולהסביר קשיים ושגיאות תלמידים הלומדים יסודות מדעי המחשב, ומתנסים בתכנות בסיסי.

מיהו מתחיל, ובמה הוא נבדל מהמומחה?

דיון בקשיים של מתחילים צריך לעסוק בראש וראשונה בשאלה מיהו מתחיל? האם הכוונה לאדם אשר זה עתה החל בלימוד הנושא, ומעולם לא נחשף אליו קודם לכן? או אולי הכוונה למי שערך היכרות עם הנושא בטרם החל ללמוד אותו במסגרת לימודים מסודרת? האם תלמיד נחשב מתחיל רק בהקשר ללימוד קורס מבוא לתכנות? האם תלמיד המתחיל בלימוד פרק חדש במסגרת נושא כלשהו (בסיסי או מתקדם כאחד), נחשב כמתחיל בהקשר ללימוד פרק זה? ומה בדבר אדם המתחיל בלימוד של פרדיגמת תכנות נוספת? האם גם הוא נקרא מתחיל בהקשר של לימוד פרדיגמה זו, למרות שיש לו רקע בפרדיגמה אחרת?

ואחרון חביב - עד מתי יקרא הלומד מתחיל, ומתי ישנה סטטוס ויקרא מתקדם או מומחה? שינוי הסטטוס נעשה בעקבות תהליך הבשלה, במהלכו צובר התלמיד ניסיון, ידע ויכולות של פתרון בעיות, ומשנה את גישתו ואמונותיו ביחס לתוצרים שהוא מפתח (תכניות מחשב), ולאופן הפקתם. החוקרים מתייחסים להבדלים בין מתחילים ומומחים בהקשרים שונים. יש חוקרים המתארים קשיים ושגיאות של מתחילים במונחים של "במה הם נבדלים מן המומחים?". הבדלים אלו נמדדים בשונות של תוצרים- במה ועד כמה התכנית של המתחיל שונה מתכנית של מומחה המהווה פתרון לבעיה נתונה (Joni & Soloway, 1986); בשכיחות הופעת שגיאות (Spohrer & Soloway, 1986); באמונות כלפי מאפייני תכנית "טובה" (Joni & Soloway, 1986; Fleury, 1993); ובגישות ושיטות של פתרון בעיות (Linn, 1985; Linn & Clancy,1992; Spohrer, Soloway & Pope, 1986).

בסקר ספרות זה מוצגים מחקרים המתייחסים לקשיי מתחילים אשר למדו במסגרת פורמלית (אוניברסיטה, מכללה או תיכון) קורס מבוא בתכנות המקביל לקורס הנלמד אצלנו - יסודות מדעי המחשב.

מדוע קשה להתחיל ללמוד תכנות?

מחקרים שנערכו בהקשר של לימוד תכנות על ידי מתחילים, מתייחסים לשלשה היבטים: בעיות וקשיים שנצפו אצל מתחילים הכרוכים בתכנות, ניתוח שגיאות ותפיסות חלופיות, ניתוח שיטות וגישות של מתחילים לפתרון בעיות. כמו כן פורסמו תיאוריות המנסות להסביר מדוע לימוד תכנות כרוך בקשיים.

לימוד תכנות הוא משימה מורכבת הנערכת בשני צירים: ציר ההפשטה, המתייחס למושגים ועקרונות של מדעי המחשב ותכנות, וציר היישום, המתייחס לשפת תכנות ולסביבת העבודה, המשמשים ליישום העקרונות הנלמדים. המעבר בין שני הצירים נעשה לסירוגין. מושג או רעיון חדש מוצג תחילה בכיתה, ומיושם מיד לאחר מכן בפעילות מעבדה, ושוב דנים בו ברמת הכיתה, וחוזר חלילה. בהתאם, תלמיד מתחיל צריך להתמודד עם הבנה ותפקוד בשני המישורים בו זמנית. שיטה המבוססת על עקרון זה, המכונה שיטת ה"רוכסן" מומלצת להוראת תכנית הלימודים החדשה במדעי המחשב בארץ (Gal-Ezer, Yehudai, Harel & Beeri, 1995, Gal-Ezer & Harel 1999).

הניסיון מראה כי תלמידים מתחילים מתקשים בהבנת מושגים ועקרונות בסיסיים הקשורים בתכנות, ובו בזמן עליהם להתמודד עם הכרת שפת תכנות וסביבת העבודה בה הם מיישמים את העקרונות הנלמדים על ידי כתיבה והרצה של תכניות במחשב. לעתים, גם קשה להם להבחין בין עקרונות שאינם תלויים בשפה תכנות זו או אחרת, לבין היבטים טכניים הקשורים לסביבת העבודה.

Du Boulay (1989) מתייחס לחמש רמות קושי של להתחיל לתכנת:

(1) **התמצאות** (orientation): לתלמידים מתחילים יש קושי להבין מה זה תכנות, ו"למה זה טוב?". הם מתקשים לשלוט בגישות שונות בתכנות, ובשפות תכנות שונות. במצב זה, מיטשטשת ההבחנה בין רמות שימוש שונות בחבילות תכנה מוכנות לבין תכנות.

(2) **הבנת המכונה הוירטואלית** (notional machine): לתלמידים מתחילים יש קושי להבין איך עובד המחשב, כיצד לתקשר עם המחשב, איך לגרום לו לבצע את מה שאנו מתכוונים וכדומה.

(3) **הבנת תחביר ומשמעות של שפת תכנות** (notation): לתלמידים מתחילים יש קושי להבין את המשמעות של מבנים בשפה, ולהשתמש בהם בצורה נכונה בכתיבת תכנית.

(4) **התמחות** (transition from novice to expert): לתלמידים מתחילים קשה לעשות את המעבר לרמת מומחים, תוך רכישת כלים ותבניות לפתרון בעיות, המאפיינים את המומחים.

(5) **תפקוד בסביבת העבודה** (pragmatics): התמודדות עם היבטים טכניים של סביבת העבודה כגון, מערכת הפעלה, עריכת קבצים, קומפילציה, מעקב אחר ביצוע תכנית, מוסיפה ממד קושי נוסף.

Oliver (1993) מתייחס אף הוא לקשיי מתחילים ומבחין בין שלוש רמות של ידע בתכנות (programming knowledge): (א) הכרת מבני יסוד המשמשים לפיתוח אלגוריתמים, (ב) פיתוח אלגוריתמים עבור משימות מוגדרות, (ג) פירוק בעיה לתת משימות. לטענתו, שליטה בכל שלב מחייבת שליטה בשלבים קודמים, אך שליטה בשלבים קודמים אינה מבטיחה בהכרח שליטה בשלב הבא. הוא מצא במחקרו כי מעטים בלבד בקרב תלמידים מתחילים גילו שליטה ברמה השלישית. יתרה מזאת, מרביתם של מתחילים שגילו שליטה ברמה הבסיסית של הכרת מבני היסוד, התקשו ליישם ידע זה בפיתוח אלגוריתמים לפתרון בעיות.

קשיי תלמידים באים לידי ביטוי גם בתגובותיהם ורמת ההתמודדות שלהם עם אי-הצלחות בכתיבת תכניות. למשל, Du Boulay (1989) מצא שמתחילים מגיבים בהגזמה על הודעות שגיאה מינוריות, עד כדי מחיקת תכנית שלמה או כיבוי המחשב והפעלתו מחדש. בדומה, Perkins ועמיתיו (1989) מצאו שהתנסות שלילית, חוויה של כשלון, או התקלות בבעיה קשה לפתרון, עלולה לגרום לתלמידים מתחילים לוותר ולהפסיק להתמודד. בהתאם, הם מכנים תלמידים המתנהגים כך בשם stoppers. Perkins ועמיתיו מצאו שטיפול בבעיה זו מצליח כאשר מזמנים לתלמידים חוויות של הצלחה בהתמודדות עם פתרון בעיות.

שגיאות ותפיסות מוטעות – מהן ומה מקורן?

בספרות אנו פוגשים צורות התייחסות שונות לשגיאות ולתפיסות מוטעות של מושגים בתכנות וביסודות מדעי המחשב. בסעיף זה נציג סקירה מתומצתת של סוגי שגיאות ותפיסות של תלמידים, ובפרק 2 נציג תקצירים של מאמרים נבחרים המתייחסים להיבטים שונים של קשיי מתחילים, נתאר בהרחבה רבה יותר שגיאות שכיחות שהם עושים, ונרחיב את הדיון בגורמים לשגיאות, ובהשלכותיהן על תהליכי ההוראה והלמידה.

ראשית, נבחין בין שגיאה לתפיסה מוטעית. בדרך כלל שגיאה מתוארת במונחים של "מה עשה התלמיד לא נכון?", בעוד שתפיסה מוטעית מתוארת במונחים של "מדוע עשה התלמיד שגיאה מסוימת?". דהיינו, איזו תפיסה הביאה את התלמיד לביצוע השגיאה.

Scherz, Goldberg & Fund (1990) מבחינים בין שלוש רמות של שגיאות: (1) **טעות טכנית** (bug) הבאה לידי ביטוי בכשל טכני הנובע מחוסר שליטה בסביבת העבודה. (2) **שגיאה** (mistake) הנובעת משימוש לא נכון במבנה בשפה, או פיתוח שגוי היוצר תכנית שאינה פותרת נכון את הבעיה הנתונה. (3) **תפיסה מוטעית** (misconception) אשר לא בהכרח באה לידי ביטוי בכתיבת תכנית שגויה במונחים של התנהגות קלט/פלט, אולם בדרך כלל ניתן לזהותה על ידי התבוננות בתכנית. למשל, הימנעות משימוש חוזר במשתנים, היכולה להעיד על תפיסה שבמשתנה מותר להשתמש פעם אחת בלבד.

Du Boulay (1986) מתייחס לשלוש קטגוריות של תפיסות מוטעות: (1) **שימוש לא נכון באנלוגיות**. למשל, התייחסות אל משתנה כקופסה שניתן להכניס לתוכה הרבה ערכים. (2) **הכללת יתר** המתבטאת בהשלכה לא נכונה ממבנה אחד למשנהו, הן בהיבט התחבירי, והן בהיבט של משמעות המבנים. (3) **ניהול כושל של אינטראקציות** בין תת משימות.

Rath ו- Brown (1995) מבחינים בין שגיאות אופרטיביות (operational errors) לבין תפיסות המונחות ביסוד הבנת תכנות (underlying conceptions). הם מתייחסים לשני טיפוסים של שגיאות אופרטיביות: **שגיאות תחביר** הקשורות לכתיבה שלא לפי כללי התחביר של שפת התכנות, **ושגיאות אלגוריתמיות** הבאות לידי ביטוי בפיתוח אלגוריתמים לא נכונים, שאינם מספקים את הפתרון הדרוש לבעיה נתונה. באשר לתפיסות השגויות המונחות ביסוד הבנת תכנות, הם מבחינים בין: (1) **ייחוס של הבנת שפה טבעית למחשב**: תלמידים מצפים שהמחשב ייחס משמעות למבנה/משתנה בהתאם למשמעות הכינוי שלו בשפה טבעית. (2) **ייחוס חשיבה עצמאית למחשב**: יכולת הבנה, הסקת מסקנות, יכולות בלתי תלויות של המחשב לבצע מעבר למה ש"נאמר לו" בצורה מפורשת. (3) **תפיסות חלופיות או בלתי שלמות של המחשב**: למשל תפיסה אלגברית של תכנות באה לידי ביטוי בפרוש לא נכון של מבנים בשפת התכנות (למשל, התייחסות להוראת השמה כאל יחס השוויון), או בתפיסה לא נכונה של תהליכי חישוב. תלמידים המסגלים תפיסה חלופית מתחום ידע אחר, מבצעים למעשה העברה (transfer) לא מבוקרת מתחום מוכר להם לתחום התכנות. תפיסות לא שלמות אינן גורמות בהכרח לכתיבת תכנית שגויה, אך לפי מבנה התכנית משתמע שיש לקות מסוימת בתפיסתו של התלמיד. דוגמה לתופעה המעידה על תפיסה לא שלמה מתוארת על ידי Joni ו- Soloway (1986): תלמידים כתבו קטע קוד למימוש מסננת קלט בו נעשה שימוש במשתנה עזר לקליטת נתון חדש. תחילה מתבצע קלט למשתנה עזר, ורק אחר כך מתבצעת השמה של תוכן משתנה העזר למשתנה ה"אמיתי". כתיבה זו מבטאת את התפיסה שאסור לבצע קלט ישירות למשתנה יותר מפעם אחת, אך אפשר לבצע אליו השמה חוזרת ונשנית.

Pea (1986) מתאר שלוש שגיאות של מתחילים אשר הנן בלתי תלויות בשפת תכנות:

(1) **מקביליות** - ביצוע מקבילי ובסדר כלשהו, במקום ביצוע סדרתי של משפטי התכנית.

(2) **כוונה** - ייחוס כוונות למחשב ופרשנות משלו למה שהמתכנת כתב, לאו דווקא תואמת לכוונת המתכנת.

(3) **אגוצנטריות** - ציפייה מהמחשב להבין ולהסיק כדוגמת בן אנוש, את כוונת המתכנת, ומעבר למה שכתוב מפורשות. הוא תולה את השגיאות בתפיסה מוטעית אותה הוא מכנה "סופרבאג" (superbug) - ייחוס יכולות למחשב מעבר לאלו שיש לו במציאות. יתרה מזאת, תלמידים מצפים מהמחשב להבין את הרעיון עליו הם חושבים, גם אם לא ניסחו אותו באופן מדויק, או אם ניסחו אותו באופן שגוי במקצת בתכנית (Kurland & Pea, 1989).

גם Puntnam, Sleeman, Baxter & Kuspa (1986) מצאו שתלמידים מייחסים למחשב **יכולות חשיבה** (של אדם ממוצע). הם מצאו שמתחילים מייחסים למחשב אופן ביצוע של פעולות בהתאם למשמעות בשפה טבעית של שמות משתנים אליהם הפעולות מתייחסות. לדוגמה, לדעת תלמידים, בהינתן סדרת קלט, קליטת נתונים אינה מתבצעת בהתאם לסדר הופעתם בסדרה, אלא הערך הנבחר לשמש כקלט תורן נקבע על פי משמעות שם המשתנה אליו מתבצע הקלט. למשל, המשפט Read (Smallest) מתפרש על ידי המחשב כקלט של הערך הקטן ביותר מתוך סדרת הקלט (ולא של הנתון התורן בסדרת הקלט לתוך משתנה ש"במקרה" נקרא Smallest), או למשל, המשפט Read (First) מתפרש כקליטת ערכו של הנתון הראשון ברשימת הקלט לתוך המשתנה First גם אם נתון זה אינו קלט תורן.

בדומה לחוקרים אחרים, Spohrer ו- Soloway (1986), מצאו שתלמידים מתחילים מייחסים למחשב יכולות של "מפרש אנושי". כאשר לתלמידים יש מחשבה מסוימת לגבי משמעות של מבנה בשפה, גם אם המחשבה מוטעית, הם מצפים שהמחשב יבצע את התכנית בהתאם לפרשנות שהם מייחסים.

מה בסקר ספרות זה?

בסקר ספרות זה השתדלתי להאיר את קשיי המתחילים מזוויות שונות. כל אחד מאתנו, כמורה, נתקל ביום-יום במגוון שגיאות שעושים תלמידים, אולם לא תמיד אנו מתעניינים בגורמים לשגיאות אלו. בהתאם, נמנעתי מלפרט שגיאות ספציפיות ובחרתי להתמקד יותר בהיבטים שונים של שיטות לאיתור שגיאות, מיון שגיאות, הסברים למקור השגיאות וגורמים להיווצרותן, והתייחסות לתפיסות ואמונות של תלמידים. כמו כן התקצירים מציגים דיונים על השלכות המחקרים על תהליך ההוראה והלמידה, ומוצגות הצעות החוקרים להפחתת השגיאות, ולשיפור הלמידה.

תקצירי המאמרים הנבחרים

הפרק מציג תקצירים של ששה מאמרים הדנים בהיבטים שונים של קשיי מתחילים הלומדים תכנות. המאמרים שנבחרו אמורים לייצג היבטים שונים של קשיי מתחילים, החל מהבנת מושגי יסוד, כגון מושג המשתנה, קשיים הנובעים מהבנת מבנים בשפת תכנות, וכאלו שאינם נובעים מהבנתם, וקשיים שאינם תלויים בשפת תכנות זו או אחרת. כמו כן מוצגות אמונות של מתחילים לגבי מאפייני תכנית טובה.

התקצירים כתובים כך, שניתן לקרוא כל אחד מהם באופן בלתי תלוי באחרים. בהתאם, קיימת לעתים חזרה מסוימת על רעיונות המשותפים לחוקרים השונים.

אודות המאמרים

- המאמר הראשון (Samurcay, 1989) עוסק בתפיסת מושג המשתנה, שהוא המושג הבסיסי ביותר בהקשר של תכנות, ואין להעלות על הדעת סקר ספרות שאינו נוגע והשימוש בו בפתרון בעיות על ידי תלמידים מתחילים.
- המאמר השני (Spohrer & Soloway, 1986) מציג שגיאות שכיחות של מתחילים שלמדו קורס מבוא לתכנות בסביבת Pascal. המחברים הפריכו טענה רווחת, שמרבית השגיאות נובעות מתפיסות מוטעות לגבי מבנים בשפת תכנות ומציעים מקורות קושי אחרים, הקשורים לתפיסת יכולות המחשב.
- המאמר השלישי (Pea, 1986) מציג שגיאות אופייניות של מתחילים, שלא זו בלבד שאינן נובעות מאי הבנת המשמעות של מבנים בשפת תכנות, אלא שהן נמצאו כבלתי תלויות בשפת תכנות זו או אחרת.
- המאמר הרביעי (Perkins & Martin, 1986) מתייחס לקשיי מתחילים הנובעים מידע לקוי ומחוסר אסטרטגיות כלליות בפתרון בעיות. על פי מחקרם, שאלות הכוון ורמזים יכולים לסייע לתלמיד לפתור בעיות.
- המאמר החמישי (Joni & Soloway, 1986) מציג אמונות של מתחילים לגבי "מהי תכנית טובה", ומציג קשיים הנוצרים בעקבות כתיבת תכניות בלתי קריאות. המאמר מציג כללי התנסחות: חוקי דו-שיח המשמשים הן לכתיבת תכניות והן לקריאתן ולהבנתן.
- המאמר השישי (Fleury, 1993) מציג אף הוא אמונות של מתחילים ביחס למאפייני תכנית טובה, אך מההיבט של תכנית כמערכת תכנה. דהיינו, התכנית כתוצר בעל אורך חיים מתמשך, שיש לנפותו משגיאות, לתחזקו, ולשנותו.

מושג המשתנה בתכנות: משמעותו והשימוש בו בפתרון בעיות על ידי מתכנתים מתחילים

ר. סמורסיי

R. SAMURCAY (1989). The concept of variable in programming: its meaning and use in problem-solving by novice programmers, In *Studying the Novice Programmer*, E., Soloway, J.C., Spohrer (ed.), Lawrence Erlbaum Associates, Publishers, New Jersey, 161-178.

אודות המאמר

המאמר מתייחס לקשיים קוגניטיביים בלימוד מושג המשתנה על ידי מתחילים. המאמר מציג ממצאי מחקר שנערך במטרה לאפיין כיצד מתחילים משתמשים במשתנים לצורך פתרון בעיות. המסקנה העיקרית של המחקר היא שעל מנת לפתח רמת תכנות טובה, הלומד צריך לבנות מודל סטטי ככל האפשר של תכנית. במודל זה, כל משתנה צריך לקבל ייצוג במונחים של "תכונה יחסית נשמרת" המאפיינת אותו ומייחדת אותו בהשוואה למשתנים אחרים בתכנית. זאת בניגוד למודל דינמי ביצועי של תכנית, בו המשתנים מיוצגים במונחים של סדרת ערכים המתקבלים כתוצאה מחישוב המתבצע במהלך ביצוע התכנית.

<u>רשימת מושגים</u>

- איתחול – Initialzation
- בדיקה – Test
- העברה – Transfer
- ידע תכנותי – Programming knowledge
- מונה – Counter
- מערכת ייצוג ועיבוד – RPS-Representation and Processing System
- משתנה חיצוני – External variable
- משתנה פנימי – Internal variable
- עדכון – Update
- פתרון בעיות – Problem solving
- צובר – Accumulator
- תבניות – Plans
- תכונה יחסית נשמרת – Invariant inter-relations

לימוד תכנות

המחבר רואה את תהליך לימוד התכנות כבנוי משלושה צירים מרכזיים: ידע תכנותי, פעילויות והזדמנויות למידה, וקשיים קוגניטיביים של הלומד הנוצרים בתהליך הלמידה.

המחבר טוען שיש לזהות מרכיבים ספציפיים של הידע הנדרש כדי לתכנת, ואת המאפיינים של אירועי הלמידה המאפשרים לתלמיד המתחיל לרכוש את הידע הדרוש. כמו כן יש לאתר את הקשיים הקוגניטיביים של מתחילים הנוצרים באירועי הלמידה, ולבנות כלים דידקטיים מתאימים לטיפול בקשיים אלו.

מטרת פעילות התכנות אינה רק לספק פתרון לבעיה כתוצר סופי, אלא גם להציג בצורה מפורשת ומובנית את תהליך פיתוח הפתרון. תכנות הוא תחום מורכב המשלב ידע מדעי וניסיון מעשי. לימוד תכנות אינו יכול לבוא לידי ביטוי בלימוד של תחביר או שיטה בלבד, כי אם צריך להיות כרוך ברכישת מושגי תכנות ספציפיים, תוך העזרות בכלי טכנולוגי מתווך.

הנחת היסוד היא שהידע התכנותי צריך להיתמך על ידי מערכת ייצוג ועיבוד (RPS- Representation and Processing System). הלומד אמור לבנות לעצמו תוך כדי התנסות, ייצוג מנטלי RPS של הידע התכנותי מאורגן במבנה היררכי של תבניות (plans) המשמשות אותו לפתרון בעיות.

לכל מושג יש פעולות (פרוצדורות) מוכרות המזוהות איתו. הייצוג ההתחלתי שיוצרים מתכנתים מתחילים מאופיין על ידי העברה (transfer) מתחום הבעיה אל תחום התכנות. ההעברה היא הן של פעולות מוכרות מתחום הבעיה, והן של אופן הביצוע המנטלי של פעולות אלו בתחום הבעיה. פעולות אלו לא בהכרח תואמות לאופן הביצוע של המחשב, ולמיומנות של המתכנת. לכן, כשלב מקדים להוראת תכנות, חשוב מאד לזהות תחילה את מושגי התכנות, לאפיין את הפעולות המזוהות עם כל מושג, לאפיין את הבעיות התיאורטיות והמעשיות שהמושג משמש לפתרונן, לערוך ניתוח מהותי של המושג, לתאר התנהגות נצפית שלו, ואת המצבים בהם ההתנהגות באה לידי ביטוי.

על סמך ניתוח זה ניתן לאפיין את ההתנסות המתאימה למתחילים שתוביל אותם ליצירת המודל המנטלי המתאים של הידע התכנותי, ולבנות פעילויות למידה מתאימות.

דגשים בהוראת מושג המשתנה

מנקודת מבט תכנותית משתנה מוגדר ככתובת בזיכרון המחשב, אך הגדרה זו אינה מספיקה כדי להציג לתלמיד את המשמעות הפונקציונלית (תפקודית) של המושג. על התלמידים להבין כי כאשר ערך המשתנה מתעדכן, אין הדבר משנה את משמעות הקשר הפונקציונלי שלו עם אלמנטים אחרים בתכנית. לדוגמה, משתנה המשמש כצובר, משמש בתפקיד זה במהלך כל הביצוע של התכנית, בד בבד עם שינויים שחלים בערכו. לכן מומלץ להדגיש בפני התלמידים מהם ההבדלים בין סוגי המשתנים מההיבט הפונקציונלי: צובר, מונה, משתנה פנימי, משתנה חיצוני, משתנה בעל פונקציונליות מרובה (הממלא יותר מתפקיד אחד), מונה מפורש, מונה בלתי מפורש. כמו כן מומלץ להדגיש את ההבדל בין סוגי פעולות על משתנים: איתחול, עדכון, בקרה, ואת ההבדל בין ארבע צורות של פעולות השמה: השמה של קבוע, השמה של תוצאת חישוב של ביטוי אריתמטי, שכפול (השמת ערך של משתנה אחד בתוך משתנה אחר), וצבירה.

הייצוג ההתחלתי של מושג המשתנה שיוצרים מתכנתים מתחילים מאופיין על ידי העברה מתחום המתמטיקה. המודל המתמטי של משתנה ויחס השוויון מהווים עבור התלמיד המתחיל מודל התחלתי, אך לא מספק עבור מודל המשתנה בתכנות, בתוספת פעולות עליו (במיוחד קשה לתפוס את הפעולות המתבצעות על משתנה בלולאה). המשתנה במתמטיקה הוא סטטי ומייצג שייכות לקבוצת ערכים, או כנעלם במשוואה, בעוד שבתכנות יש לו משמעות שונה.

קיימים שני סוגי משתנים בעלי משמעות פונקציונלית שונה עבור התלמיד: **משתנים חיצוניים** המשמשים את המשתמש להזין קלט, **משתנים פנימיים** אשר הערכים שלהם נקבעים ומבוקרים על ידי המתכנת.

המחבר מבחין בין שלוש פעולות על משתנים בלולאות, הנתפסות ברמות קושי שונות על ידי מתחילים: **עדכון**, **בדיקה**, **ואיתחול**. פעולת איתחול למשל, קשה יותר לתפיסה מפעולת עדכון, ומפעולת בדיקה. רמת הקושי של הפעולות נקבעת גם לפי טיפוס המשתנה עליו הן מופעלות. למשל, עדכון מונה קל יותר מעדכון צובר (כי מונה כרוך בהוספת קבוע, לעומת צובר שכרוך בהוספת ערך משתנה).

המחבר ניסח כללים שמומלץ להציגם לתלמידים בהקשר של משימות איתחול ובקרה:

כללים הקשורים לאיתחול: כל משתנה אשר על ערכו מתבצעת פעולה בגוף הלולאה חייב להיות מאותחל. אופן האיתחול של משתנה, באם על ידי קלט, או על ידי השמה, נקבע על ידי הדרך בה הוא מעודכן. הערך ההתחלתי של המשתנה נקבע על ידי סיפור הבעיה, והן על ידי זיהוי הטרנספורמציות המתבצעות במשתנה.

כללים קשורים לבקרה: לולאה היא הליך סופי. הבקרה על קיום הלולאה מתבצעת באמצעות משתנה בקרה. בהתאם, יש לזהות את משתנה הבקרה, ולקבוע את ערכו ההתחלתי.

תיאור המחקר

מטרת המחקר היתה לאפיין את משמעות הידע התכנותי בהקשר של מושג המשתנה, ואת אופיים של אירועי למידה משמעותיות המיועדים לאפשר ללומד לרכוש את הידע הדרוש. בנוסף, מנסה המחבר לתאר ולנתח קשיים קוגניטיביים המתעוררים אצל תלמידים מתחילים כתוצאה מהשתתפות באירועי למידה שמטרתם רכישת הידע התכנותי בהקשר של מושג המשתנה.

אוכלוסית המחקר

במחקר השתתפו 26 סטודנטים בקולג' שלמדו קורס תכנות בפסקל (שעה בשבוע). הקבוצה הייתה הטרוגנית וכללה שלוש רמות של סטודנטים (חלשה, ממוצעת, וחזקה).

המחקר נערך בתום 15 שעות לימוד, לאחר לימוד של: משתנים, פעולות על משתנים, ולולאות.

השיטה

התלמידים התבקשו להשלים שורות קוד חסרות בתכנית המהווה פתרון לבעיה נתונה. המשימה של השלמת התכנית היתה כרוכה בהבנת התכנית. תצפיות לא פורמליות גילו שתלמידים מנסים "להריץ" בראשם את התכנית על מנת להבינה, ולזהות מה היא עושה. בהתאם, על מנת לגרום לתלמידים להתמקד יותר ביחסים בין המשתנים, ולא בביצוע התכנית (כדי לזהות את הבעיה אותה היא פותרת), ניתן בשאלון נוסח הבעיה אותה פותרת התכנית.

לתלמידים ניתנו שלושה סוגי משימות: איתחול, בקרה, ועידכון. המטרה הייתה לבדוק אם התלמידים הפנימו כללים מקובלים הקשורים לפעולות אלו.

(א) **משימות איתחול**

השאלון כלל שלוש משימות איתחול. התלמידים היו צריכים לזהות את המשתנים ואת ערכם ההתחלתי. המשימות היו:

(1) משימה הכרוכה בהשלמת פעולות איתחול של שלושה משתנים מטיפוס שונה - שני צוברים ומונה. ההשערה הייתה שאיתחול של מונה קל יותר לתלמידים מאיתחול של צובר.

(2) משימה הכרוכה בהשלמת פעולות איתחול של משתנה אחד בעל פונקציונליות כפולה (כמשתנה פנימי המשמש כמונה, וכמשתנה חיצוני שערכו מוצג כפלט). במקרה זה, כדי להחליט על אופן האיתחול התלמיד אמור להתייחס בו זמנית לשני פרטים: אופן העדכון של המשתנה בזמן ביצוע הלולאה, וערכו המיוחד כמשתנה בקרה של הלולאה.
(3) משימה המשלבת פעולות איתחול ופעולות עדכון.

(ב) **משימות בקרה**

השאלון כלל שתי משימות בקרה:
(1) שימוש במשתנה בקרה בעל פונקציונליות כפולה: כמשתנה חיצוני אליו מתבצע הקלט, וכמשתנה פנימי המשמש כמונה יורד. המושג של "מונה יורד" הוצג לראשונה לתלמידים במבחן זה.
(2) שימוש במשתנה בקרה – מונה "רגיל" (גדל ב-1). כדי לפתור בעיה זו, התלמיד יכול היה להשתמש במודל אנלוגי של פתרון נתון לבעיה דומה.

(ג) **משימות עדכון**

השאלון כלל שתי משימות עדכון, אותן יכול התלמיד לבצע תוך שימוש במודל אנלוגי של פתרון נתון של בעיה דומה, אך מתחום שונה- אריתמטיקה. לשם כך היה עליו לזהות את הקשר בין שתי מערכות ייצוג שונות.

ממצאים

ניתוח עבודות התלמידים העלה את הממצאים הבאים:

אתחול צוברים ומונים: תלמידים ביצעו איתחול מיותר של משתנים. איתחול צוברים היה קשה לתלמידים יותר מאיתחול מונים. הממצאים מעידים על תפיסות מוטעות בהקשר של איתחול: תלמידים סבורים ש"יש לומר למחשב מהם הערכים של המשתנים, כיוון שהוא אינו יכול להבין את שמות המשתנים". בהתאם, תלמידים סבורים שיש לאתחל צובר אך ורק על ידי פעולת קלט (גם אם לא מתחייב מנוסח הבעיה), ונמנעים מאיתחול על ידי השמה.

תפיסה מוטעית לגבי איתחול מונים היא ש"אין צורך לאתחל מונה (counter) כי ברור שמניה מתחילה מ- 0".

משמעות משפט קלט: תפיסה מוטעית אחרת מתייחסת למשמעות משפט קלט. תלמידים מבצעים איתחול על ידי קלט מלווה בהשמה. הם סבורים כי משמעות משפט קלט היא "לקבל ערך מהמסך", אולם ערך זה לא "יכנס" אוטומטית למשתנה, אלא אם כן הדבר יאמר במפורש, דהיינו, אם יהיה משפט השמה מתאים.

שגיאות הקשורות לעבודה עם משתני בקרה: תלמידים השתמשו בערך לא נכון של משתנה הבקרה בתנאי קיום הלולאה. לדוגמה, במקום לרשום תנאי לוגי מתאים, יש שרשמו משפט כגון: "החזרה תיפסק כאשר התוצאה הדרושה תתקבל" (במקרה הטוב מציינים תבנית התוצאה, כגון X*Y). אחרים רשמו כי "לכל repeat יש until מתאים", ללא ציון תנאי הבקרה על הלולאה.

קשיים הנובעים מהעברה מתחום המתמטיקה:

(1) **התייחסות לשמות משתנים:** מסתבר שהתייחסות של תלמיד למשתנה ואופן העבודה עמו עלולים להיות מושפעים ממשמעות שמו. למשל, משתנים ששמותיהם לקוחים מתחום המתמטיקה יוצרים אצל התלמיד התייחסות מיוחדת, לעומת משתנים עם שמות מסוג אחר. X נחשב בעיני תלמידים כמשתנה "מקודש" שמקובל להשתמש בו בניסוח תבניות פסוק ותבניות מספר במתמטיקה. אי לכך קשה לתלמידים לקבל שימוש בשם משתנה אחר כגון Number לייצוג ביטוי אריתמטי בתכנית.

(2) **התייחסות לסימן ההשמה:** תלמידים נוטים להתייחס לסימן השמה כאל סימן השוויון (כיחס הלקוח מתחום המתמטיקה).

(3) **קשיים בייצוג נוסחאות מתמטיות במונחי לולאה:** תלמידים גילו קושי במימוש נוסחאות מתמטיות באמצעות לולאות, ובדומה קושי בהמרת נוסח מילולי המתייחס לחישוב מתמטי לקטע תכנית מתאים.

בנוסף התגלתה בעיה כללית: תלמידים נוטים להעתיק קטעים שלמים מתכנית אחרת, הפותרת בעיה דומה, בלי להתאימם לבעיה הנוכחית.

סיכום ומסקנות

מושג המשתנה נתפס בהתייחס לסוג המידע שהוא מייצג, וכן בהתאם לאופי הפעולה המתבצעת עליו. הצירוף בין השניים קובע את מידת הקושי של התלמיד לעבוד עם המשתנה, ולמעשה את רמת המיומנות של העבודה אתו.

הממצאים מראים כי פעולת האיתחול קשה יותר מפעולות העדכון והבקרה. פעולות העדכון והבקרה נמצאות באותה דרגת קושי. הקושי באיתחול נובע מכך שיש לבצע הנחה על מצב התחלתי (ערך התחלתי של משתנה) על סמך מצב סופי ידוע והפעולות החוזרות ונישנות (לולאה).

נמצא כי תלמידים מתחילים מעדיפים איתחול על ידי קלט מועדף מאשר איתחול באמצעות השמה. הסבר אפשרי להעדפה זו - במשפט קלט, המשתמש הוא זה שקובע את הערך ההתחלתי ומזין אותו למחשב. זה יותר מוחשי ממשפט השמה.

תלמידים מתחילים פועלים על פי הנחות, שהן בגדר חוק בל יעבור. למשל, הנחה המשקפת תפיסה מוטעית, שכל תכנית חייבת להכיל לפחות משפט קלט אחד. או למשל, לתלמידים יש הנחה מוטעית אודות מונה: "הוא חייב להיות מאותחל ל- 0, ומוגדל ב- 1". כאשר משתנה מונה אינו תואם הנחה זו, עלול להתעורר קושי.

קל יותר למתחילים לעבוד עם משתני מונה מאשר עם משתנים בעלי פונקציונליות אחרת. פעולה על משתנים צוברים המהווים משתנים פנימיים, נמצאה קשה, ללא תלות במשימה.

לתלמידים קל יותר לעבוד עם משתנים עבורם יש להם ייצוג בתחומי דעת אחרים. למשל, יש להם סכמה מקובלת אודות מונה.

איתחול משתנים היא פעולה קשה לתלמידים. קשה להם להניח מראש הנחה על מצב התחלתי של מערכת, והדבר דורש פעילויות קוגניטיביות כלליות יותר מעבר לתכנות. מבחינת ההיבט התכנותי, פעולת אתחול מתבצעת בדרך כלל ברמת כתיבת הקוד, ואין מתייחסים אליה ברמת פתרון של "נייר ועפרון". פער זה יוצר קושי. לעומת זאת, אתחול באמצעות משפט קלט קל יותר לתלמידים, ומועדף יותר.

קשיים המתייחסים לפעולות עדכון ובקרה בבעיות המצריכות לולאות, הן בעלות דרגת קושי דומה.

תלמידים מעבירים את הייצוג (representation) ואת מערכת העיבוד (processing system) הנוצרות בתחום הבעיה, אל תחום התכנות. העברה זו עלולה לגרום לקשיים ולשגיאות. לדוגמה, העברה מתפיסה אלגברית של מושג המשתנה ושל אופרטור השוויון.

לסיכום, המחבר טוען כי תפיסת מתחילים את התכנית מבוססת על "הרצה מנטלית" של התכנית אשר תואמת למודל הדינמי של פעולת המחשב. המסקנה העיקרית של המחקר היא שעל מנת לפתח רמת תכנות טובה, הלומד צריך לבנות מודל סטטי ככל האפשר של תכנית. במודל זה, המשתנים צריכים לקבל ייצוג במונחים של "תכונה יחסית נשמרת" המאפיינת אותם ומייחדת אותם לעומת משתנים אחרים בתכנית. זאת בניגוד למודל דינמי ביצועי של תכנית, בו המשתנים מיוצגים במונחים של סדרת ערכים המתקבלים כתוצאה מחישוב המתבצע במהלך ביצוע התכנית.

ניתוח שגיאות שכיחות של מתכנתים מתחילים

ג. ספורר ו- א. סולווי

J.G. Spohrer and E. Soloway (1986). Analyzing the high frequency bugs, In *Empirical Studies of Programmers*, E., Soloway, Y., Iyengar (ed.), Albex Publishing Corporation, Norwood, New Jersy, 230-251.

אודות המאמר

המאמר מתאר מחקר העוסק בניתוח שגיאות של מתכנתים מתחילים. המחברים מאששים את הטענה הרווחת כי קיים מספר מצומצם של שגיאות אופייניות המופיעות בשכיחות גבוהה אצל רוב המתחילים הלומדים תכנות. לעומת זאת, הם מפריכים טענה רווחת אחרת כי מרבית השגיאות נעוצות בהכרח בתפיסות מוטעות אודות משמעות של מבנים בשפת מחשב.

המאמר דן בהשלכות הממצאים על הוראת תכנות.

<u>רשימת מושגים</u>

- בעיית המפרש האנושי – Human Interpreter Problem
- בעיית חוסר עקביות של טיפוסי נתונים – Data-Type Inconsistency Problem
- בעיית סדר מקרי – Coincidental Ordering Problem
- בעיית ערך קצה – Boundary Problem
- בעיית שפה טבעית – Natural Language Problem
- בעיית תלות בין תת-משימות – Plan-Dependency Problem
- מבנה בשפה – Language construct
- תבניות – Plans and goals

המחקר

מטרת המחקר הייתה לאשר או להפריך את הטענות הבאות, הרווחות לגבי שגיאות של תלמידים המתחילים ללמוד תכנות:

- קיים מספר מצומצם של שגיאות אופייניות שכיחות אצל תלמידים מתחילים.
- רוב השגיאות נובעות מתפיסות מוטעות שיש לתלמידים לגבי מבנה כלשהו בשפה.

מטרה נוספת הייתה לנסות להסביר מדוע שגיאות מסוימות מתרחשות.

השיטה

החוקרים פתחו שיטה לאיתור שגיאות שכיחות של מתחילים, הבוחנת האם קושי מסוים נובע מחוסר מיומנויות של פתרון בעיות, או בגלל שליטה בלתי מספיקה בשפת התכנות. במחקר השתתפו סטודנטים שלמדו באוניברסיטה קורס "מבוא לתכנות בסביבת Pascal", המיועד למסלול הומניסטי. התלמידים נתבקשו במהלך הקורס לכתוב תכניות מחשב לפתרון בעיות נתונות. נבדקו כ- 50 תכניות שנכתבו כפתרון לכל אחת משלוש בעיות שנבחרו על ידי החוקרים לצורך המחקר. הבעיה הראשונה הייתה משימה ברמה התחלתית בה התלמידים היו צריכים להשתמש לראשונה בתנאי מקונן. לפתרון השאלה השנייה היה עליהם להשתמש לראשונה בלולאה, ובבעיה השלישית – להשתמש לראשונה בפרוצדורות. בנוסף, בכל שאלה היה עליהם להשתמש באופן חדש במבנים קודמים המוכרים להם.

איסוף הנתונים נערך באמצעות הקלטת הגרסה הראשונה של תכנית נטולת שגיאות תחביריות שעברה קומפילציה. הנחת החוקרים הייתה, שכל עוד יש בתכנית שגיאות תחביריות, התלמיד עסוק בתיקון שגיאות אלו, ורק כשהתכנית נכונה תחבירית, מתחילה בדיקת ביצוע התכנית במונחי התנהגות קלט/פלט (האם התכנית נכונה לוגית). הם רצו לנתח גרסה ראשונית של תכנית שעברו קומפילציה, לפני שנערכו בהן שינויים עקב בדיקות לוגיות.

המחברים מסתמכים על מחקרים קודמים אשר תומכים בהשערה כי הידע בו משתמשים מתכנתים לפיתוח ולניתוח תכניות בנוי מאוסף תבניות. כל תבנית מורכבת ממטרה (goal) ומקוד-תכנית/אלגוריתם (plan) המשמש להשגת המטרה. המטרה מתארת את "מה צריך לבצע" כדי לפתור את הבעיה , ואילו קוד-תכנית/אלגוריתם מתאר "כיצד" תושג המטרה.

בהתאם, במחקר המתואר במאמר זה, ניתוח התכניות וזיהוי השגיאות נעשה על ידי שימוש בתבניות (plans and goals). **השגיאות מתוארות במונחים של הבדלים בין תבנית המקובלת כנכונה להשגת המטרה, לבין תכנית נבדקת, שנכתבה להשגת אותה מטרה.** תכניות שגויות הן אלו שלא תשגנה את המטרה.

המחברים מבחינים בין טיפוסי שגיאות (bug types) ומופעי שגיאות (bug tokens). טיפוס שגיאות הוא שם כולל לקבוצת מופעי שגיאות הזהים במהותם.

במחקר התגלה שלא כל השגיאות נוצרות באופן שווה. קיימות שגיאות המופיעות שוב ושוב אצל מתחילים, בעוד שאחרות מופיעות לעתים רחוקות בלבד. ובמילים אחרות, יש טיפוסי שגיאות בעלי מספר מופעי שגיאות גדול במיוחד, בעוד שקיימים טיפוסי שגיאות שמספר מופעיהן קטן, ואפילו זניח.

מקור השגיאות (הסברים מתקבלים על הדעת)

ברוב ספרי הלימוד, כל פרק מציג מבנה חדש בשפת התכנות. המורים מלמדים את החומר במקביל לספר, תוך דגש על מבני השפה, ומראים לתלמידים כיצד להשתמש בהם לכתיבת תכנית.

בגלל הדגש על לימוד מבני השפה, יכול להיראות על פניו ששגיאות התלמידים נובעות מאי הבנת משמעות של מבנים מסוימים. סברה נוספת רווחת היא ששגיאות מתרחשות בשכיחות גבוהה במצבים חדשים, או כשנעשה שימוש במבנה או במשפט חדש בשפה.

מחברי המאמר טוענים שלמעשה אין בספרות המחקרית תמיכה ממשית להשערות אלו, כיוון שהמחקרים התומכים כביכול בהשערות אלו, נערכו במועדים קצרי טווח של הוראה, ולא במסגרת "טבעית" וממושכת במהלך של לימוד קורס מתחילתו ועד סופו. לעומת מחקרים אלו, המחקר המתואר במאמר זה נערך לאורך סמסטר שלם, במסגרת כמעט טבעית.

על מנת לבדוק את תקפות ההשערה הרווחת, נערך ניתוח יסודי של תכניות שפותחו על ידי תלמידים עבור שלושת הבעיות, וזוהו 11 טיפוסי שגיאות שכיחות. אחת בלבד אובחנה באופן ודאי כנובעת מאי הבנה של משמעות מבנה בשפה, ועוד שלוש שגיאות יתכן שנובעות מכך. לגבי שבע שגיאות אחרות לא נמצאו תימוכין להשערה, וניתנו הסברים אחרים.

הבעיות עליהן מעידות השגיאות שזוהו מפורטות להלן.

בעיות הקשורות לתפיסת המשמעות של מבנים/משפטים בשפת תכנות

(1) בעיית חוסר עקביות של טיפוסי נתונים (Data-Type Inconsistency Problem)**:**

כאשר משמעות משפט בשפה תלויה בטיפוסי נתונים של משתנים המופיעים בו, עלולות להתעורר בעיות. לדוגמא, השימוש במשפט קלט readln שונה כאשר יש לקלוט סדרת מספרים, סדרת תווים, או סדרה משולבת של מספרים או תווים.

למשל, בהינתן סדרת קלט המכילה שילוב של מספרים ותווים המופרדים ביניהם בתווי רווח (spaces), הרי שעבור דוגמת קלט: + 45 10 23 a משפט קלט נכון מתייחס לתו רווח המפריד בין מספר לתו:

readln(Id, Whitespace1, Prob_type, Num1, Num2, Whitespace2, Sign)

בעוד שמשפט קלט שגוי מתעלם מתווי הרווח: readln(Id, Prob_type, Num1, Num2, Sign)

שגיאה זו מעידה על תפיסה מוטעית של משמעות משפט קלט המתייחס לקליטת תווים משולבת בסדרת נתונים מטיפוס אחר.

(2) בעיית שפה טבעית (Natural Language Problem)**:**

במקרים בהם משפטי השפה כוללים מילים שמורות הלקוחות מהשפה הטבעית, התלמידים עלולים להבין לא נכון את משמעותם.

(3) בעיית המפרש האנושי (Human Interpreter Problem)**:**

יש מקרים בהם תלמידים מצפים שמשפט שהם כתבו יתפרש בדרך מסוימת, ויתבצע בהתאם. הם מניחים שהמחשב יפרש את המשפט באופן זהה להם. למעשה הם משייכים למבנים/משפטים בשפה את המשמעות הרצויה בעיניהם.

בעיות שאינן קשורות לתפיסת המשמעות של מבנים/משפטים בשפת תכנות

(1) בעיית ערך קצה (Boundary Problem)

תלמידים אינם כוללים ערכים מסוימים בתחום הערכים החוקיים לייצוג נתונים, כדוגמת המספר אפס. שגיאה מסוג זה נקראת off-by-one bug. השגיאה באה לידי ביטוי בניסוח תנאי באמצעות אופרטור יחס (כגון <, =, <=). ההסבר שהשגיאה נובעת מאי הבנת אופן השימוש באופרטור אינו מתקבל על הדעת כאשר באותה תכנית נעשה בו שימוש נכון. במקרה זה, יש לשייך את השגיאה לגורם אחר, מהותי יותר.

מסתבר שלתלמידים יש קושי לזהות גבולות של תחום ערכים אותם יכול משתנה לקבל. קשה להם להבחין בין ערכים אפשריים, ערכים טריוויאליים, וערכים בלתי מתקבלים על הדעת. כמו כן קשה להם לקבוע האם הטיפול בערך קצה צריך להתבצע בנפרד, או בתחום/י הערכים עבורם הוא משמש כערך קצה.

תלמידים מבצעים לעתים הקבלה בין קבועים שונים בתכנית המשמשים לזיהוי ערכי קצה, גם אם הדבר אינו נכון בהקשר הבעיה. למשל, התייחסות אחידה לשני קבועים למרות שאחד מהם שייך לתחום הערכים המותרים, ואילו השני מהווה גבול, אך אינו שייך לתחום.

(2) בעיית תלות בין תת-משימות (Plan-Dependency Problem):

תלמידים מתקשים בזיהוי תלויות בין תת-משימות, ובהתאם אינם מממשים קשרים אלו. לדוגמא, תלות בין משימת פלט למשימת חישוב המחשבת את הערך האמור להיות מוצג כפלט. יש בעיות בהן מחושב ומוצג כפלט רק ערך המחושב בהתקיים תנאי מסוים. במקרה והתנאי אינו מתקיים, החישוב אינו מתבצע, ובהתאם אין מה להציג כפלט. במילים אחרות, תת המשימה של פלט תוצאת החישוב תלויה בתת המשימה של חישוב ערך התוצאה. שגיאה אופיינית מתבטאת בכך שלמרות שקטע הקוד העוסק בחישוב ערך התוצאה נוסח כהלכה תוך שימוש במבנה בקרה מתאים, ערך המשתנה המייצג את תוצאת החישוב מודפס בכל מקרה, בין אם חושב, ובין אם לא חושב בגלל אי-קיום התנאי.

מקור אפשרי לשגיאה הוא אי הבנת משמעות מבנה הבקרה if-then-else (כל המשפטים המופיעים לאחר ה- else, שייכים לבלוק המשויך ל- else ומתבצעים בהתאם). אולם זה אינו ההסבר היחיד המתקבל על הדעת. לעתים מספיק להתבונן בקוד כדי להבחין שזה אינו גורם השגיאה. אופן ארגון הקוד מעיד על ראיית התלמיד את סדר ביצועו. למשל, שימוש בהזחה או תיחום קטעי קוד ב- begin-end. כאשר תלמיד מארגן את הקוד באופן נכון, אפשר להניח שהשגיאה אינה נובעת מאי הבנת משמעות המבנה. דוגמה לכך ניתן לראות בקטעי הקוד הבאים:

קטע קוד שגוי	**קטע קוד נכון**
קלוט נתון	*קלוט נתון*
אם הנתון מקיים תנאי אזי	*אם הנתון מקיים תנאי אזי*
בצע חישוב	*בצע חישוב*
הצג את תוצאת החישוב	*הצג את תוצאת החישוב*

הסבר אפשרי לשגיאה שאינו קשור לאי הבנת משמעות מבנים בשפת התכנות הוא: תפיסה מוטעית של "קיטוע פלט" (output fragmentation) - הפלט חייב תמיד להתבצע, ללא תלות בחוקיות הפלט. מעבר לזה, יש לתלמידים תפיסה מוטעית אחרת: תבנית של ארגון הקוד - "הפלט תמיד צריך להיות בסוף". תלמידים אלו כותבים את קטע הקוד המטפל בפלט בסוף התכנית. הנה לדוגמה ציטוט מדברי תלמיד: "תחילה אני צריך לקבל את הקלט, אחר כך אני בודק אם הוא אינו חוקי אני מדפיס הודעה, אחרת אני מבצע את החישוב. בסוף, אחרי שהחישוב התבצע, הפלט מתבצע".

(3) בעיית שלילה ושלמות (Negation and Whole-Part Problem):

תלמידים מתחילים סבורים ששלילת **דבר טוב בשלמותו**, יוצרת **דבר רע בשלמותו**, ולהפך. בבדיקת תנאי תלמידים משתמשים בקשר OR במקום בקשר AND. לדוגמה, ניסוח שגוי של תנאי לבדיקה האם ערך משתנה X שונה מהמספרים 1,2,3 : If ((X<>1) OR (X<>2) OR (X<>3)) then...

הסבר אפשרי לשגיאה זו: תלמידים סבורים שכדי לנסח תנאי מורכב, באמצעות מרכיבים שכל אחד מהם מייצג דבר טוב בשלמותו, או דבר רע בשלמותו, יש לחבר את המרכיבים באמצעות OR. בדוגמה זו, תלמידים ניסו להשתמש בשלילה כדי להגדיר דבר טוב בשלמותו על ידי שלילת **דבר רע בחלקו**: לשלול שוויון של X לערכים האסורים 1,2,3 על ידי שימוש ב- '<>'. למשל, שלילת הביטוי X=1 על ידי: X<>1 . אולם כאשר ערך הביטוי X<>1 הוא אמת, אין זה מעיד על דבר טוב בשלמותו, שכן X יכול להיות שווה ל- 2, או ל- 3.

(4) בעיית ציפיות ופרשנות (Expectations and Interpretation Problem):

תלמידים עלולים להשתמש בנוסחאות לא נכונות לחישוב ערכים, שאינן בהתאם למוגדר בבעיה, כי יש להם ציפייה שונה, או פרשנות שונה לגבי האופן בו הערך אמור להיות מחושב (שגיאה זו אינה מהותית, אך בעלת שכיחות גבוהה לפי המחבר, ב.ה.).

(5) בעיית כפילות ספרה בסוף מספר (Duplicate Tail-Digit Problem):

תלמידים המשתמשים בקבועים מספריים בעלי הופעה של ספרות זהות בסוף המספר, משמיטים לעתים את הספרה "המיותרת" בסוף המספר. הגורם לכך יכול להיות טעות בחישוב ערכו של הקבוע, או בהקלדה שגויה.

(6) בעיית השפעת ידע דומה (Related Knowledge Inference Problem):

תלמידים מושפעים לעתים מאנלוגיה. למשל, מספר השניות בשעה הוא 3600, אך מספר המעלות במעגל הוא 360, והתלמיד בבואו לחשב את מספר השניות חושב על שעון, מבצע את ההשלכה הלא נכונה וכופל את מספר השעות ב- 360. אולם אין ספק שהגורם לשגיאה זו אינו אי הבנת משמעות של מבנה בשפת המחשב.

(7) בעיית סדר מקרי (Coincidental Ordering Problem):

תלמידים עשויים לבנות ביטוי אריתמטי אשר סדר כתיבתו זהה לסדר החישוב המצופה של מרכיביו, ובתוך כך הם עלולים לא לזהות חלקים בביטוי שיש לתת להם סדר עדיפות שונה. הם אינם משלבים סוגריים לעקיפת סדר החישוב הרצוי כיוון שהם סבורים שהמחשב אכן יבצע את החישוב לפי סדר הכתיבה. לדוגמה, חישוב מס הכנסה לפי הכלל "על 10000 ש"ח הראשונים של הברוטו משלמים 30% מס, ואילו על יתר הסכום משלמים 32%" תלמידים רשמו את הביטוי הבא:

Tax := 10000*0.30+Income-10000*0.32

הביטוי משקף את סדר החישוב כפי שמבצע התלמיד בראשו: "תכפיל את 10000 ב- 0.32 ותוסיף לתוצאה את ההפרש בין ההכנסה הכוללת Income ל- 10000 מוכפל ב- 0.32%". התלמיד לא רשם סוגריים לאלץ את החיסור Income-10000 להתבצע לפני פעולת הכפל, ומניח שהמחשב יבצע את החישוב כמצופה.

סיכום

שגיאות של מתחילים אינן מופיעות במידה דומה. יש שגיאות שהופעתן נדירה, ויש שגיאות ששכיחותן גבוהה במיוחד. ממצאי המחקר סותרים את הסברה שמקור מרבית השגיאות הוא בתפיסה מוטעית של משמעות במבנים/משפטים בשפת התכנות. המחברים גילו 10 בעיות אשר באות לידי ביטוי בשגיאות שכיחות של מתכנתים מתחילים, שרק שלוש מהן קשורות להבנת משמעות של מבנים או משפטים בשפת תכנות. יתר הבעיות נובעות מתפיסת מושגים וקשרים שאינם קשורים בהכרח ישירות לתכנות. המחברים מדגישים שלממצא זה יש השלכות להוראה: מורים המודעים לשגיאות שכיחות, יכולים לשפר באופן משמעותי את ההוראה ולטפל במיגורן. הבהרת המשמעות של המבנים/משפטי השפה אין די בה להפחתת השגיאות, שכן הממצאים מראים שמרבית השגיאות אינן קשורות בהכרח להבנה שגויה של משמעות זו. על המורים לגוון את ההוראה ולשלב בתהליך הלימוד דוגמאות של פתרונות בנויים היטב, במקביל לפתרונות שגויים. לדעת מחברי המאמר, הצגת שגיאות אפשריות בפני תלמידים עשויה לעזור להם לשים לב, ולהימנע מלעשותן.

שגיאות מושגיות בלתי תלויות בשפת תכנות של מתכנתים מתחילים

ר. ד. פי

R.D. Pea (1986). Language-independent conceptual “bugs” in novice programming, Journal of Educational Computing Research, Vol. 2:1, 25-36.

אודות המאמר

המאמר מתאר מחקר העוסק בניתוח שגיאות של מתכנתים מתחילים. המחבר מציג שלושה טיפוסי שגיאות אופייניות, שאינן תלויות בשפת מחשב, המופיעות בשכיחות גבוהה אצל רוב המתחילים הלומדים תכנות. טיפוסי השגיאות הם: מקביליות, התכוונות, ואגוצנטריות. מקור טיפוסי שגיאות אלו הוא בייחוס יכולות חשיבה והסקה למחשב.

<u>רשימת מושגים</u>

- אגוצנטריות – Egocentrism
- התכוונות – Intentionality
- מקביליות – Parallelism

המחקר

Pea הניח כי מתכנתים מתחילים עובדים באופן אינטואיטיבי, ומתנהגים כאילו כתיבת תכנית והרצתה, דומה לשיחה עם בן אנוש, ללא תלות בשפת תכנות זו או אחרת.

הנחה זו הביאה אותו לביצוע מחקר שמטרתו אבחון שגיאות ותפיסות מוטעות הקשורות לקושי כללי הקיים בעצם הלימוד של מתן הוראות ביצוע למחשב, ואינן תלויות בפקודות מסוימות בשפת תכנות.

טיפוסי שגיאות

Pea מבחין בין שלושה טיפוסי שגיאות שאינן תלויות בשפת תכנות: (1) **מקביליות** (parallelism) - ביצוע מקבילי ובסדר כלשהו, במקום ביצוע סדרתי של משפטי התכנית. (2) **התכוונות** (intentionality) - ייחוס כוונות למחשב ופרשנות משלו למה שהמתכנת כתב, לאו דווקא תואמת לכוונת המתכנת. (3) **אגוצנטריות** (egocentrism) - ציפייה מהמחשב להבין ולהסיק כדוגמת בן אנוש, את כוונת המתכנת, ומעבר למה שכתוב מפורשות. הוא תולה את השגיאות בתפיסה מוטעית אותה הוא מכנה "סופרבאג" (superbug) - ייחוס יכולות למחשב מעבר לאלו שיש לו במציאות.

מקביליות

תלמידים סבורים שמשפטי תכנית שונים עשויים להתבצע בו זמנית על ידי המחשב במהלך ביצוע תכנית. לדעתם, המחשב יכול להתייחס למשפטי תכנית בכל שלב של ביצוע התכנית, ללא קשר לסדר הופעתם בתכנית ולעובדה שזמן ביצועם עבר. Pea מתייחס לשני סוגי של משפטים: מבני בקרה ומשפטי השמה.

מבני בקרה: תלמידים סבורים שקטעי קוד, שביצועם מותנה בקיום תנאי מסוים, מתבצעים כל אימת שהתנאי מתקיים, גם אם זמן הביצוע של משפט התנאי כבר עבר. למשל, כאשר בזמן ביצוע משפט תנאי, התנאי אינו מתקיים, ובהתאם, קטע קוד התלוי בקיומו אינו מתבצע, אין הדבר מונע ביצועו בהמשך, אם נוצר מצב בו התנאי כן מתקיים. מחבר המאמר גילה ששגיאות מסוג זה עלולות לבוא לידי ביטוי בתכניות בהן יש משפטי תנאי מחוץ ללולאות.

לדוגמא, תלמידים שנשאלו מה הפלט של קטע התכנית הבא, ענו שהמילה Hello תוצג כפלט כאשר המשתנה Size יקבל את הערך 10.

```
if Size = 10 then
        Write('Hello');
:
:
for Size = 1 to 10 do
        write('Size');
```

הם הסבירו שהמחשב מחכה לכך שהמשתנה Size יקבל את הערך 10, על מנת שיוכל להדפיס Hello. לדעתם, כל משפטי התכנית "פעילים" בזמן ביצוע התכנית, ועשויים להתבצע בו זמנית בהתאם למצב. כדברי תלמיד: "נראה כי המחשב מסתכל על כל התכנית בבת אחת מכיוון שהוא כל כך מהיר".

המחבר מציג גורם אפשרי לטיפוס שגיאה זה, הנעוץ בהתבטאות בשפה טבעית, ובמשמעות משפטי תנאי שאינם תלויים בזמן. למשל, המשפט שאדם א' אומר לאדם ב': "אם אתה רוצה ללכת לחנות, אני אסיע אותך" מבטא נכונות של אדם א' להסיע את אדם ב' לחנות, כל אימת שאדם ב' יביע רצון לבקר בחנות (כעת או בעתיד), ולאו דווקא תכף ומיד. תלמידים העושים השלכה של משמעות משפטי תנאי בשפה טבעית אל משפטי תנאי בשפת תכנות עלולים לשגות באופן המתואר לעיל. דוגמא נוספת להשלכת משמעות משפה טבעית קשורה למשפט While. בשפה טבעית משמעות משפט כדוגמת: " כל עוד בכביש שני מסלולים, המשך צפונה" היא בדיקה בלתי פוסקת של קיום שני מסלולים בכביש, ועם שינוי התנאי, הנהג צריך לקבל החלטה על כיוון המשך הנסיעה. משמעות זו של "כל עוד" בשפה טבעית, אינה תואמת את משמעות משפט While בשפת תכנות, המחייבת בדיקת מצב משתני הבקרה רק כל אימת שגוף הלולאה צריך להתחיל להתבצע (לפני וסמוך ל"כניסה" ללולאה - ב.ה.). תלמידים המאמצים את משמעות "כל עוד" בשפה טבעית, סבורים שהבקרה על ביצוע לולאת While נעשית באופן רציף ללא הפסקה, בשאיפה לזהות מתי התנאי לביצוע הלולאה חדל מלהתקיים. מובן שהתייחסות זו עלולה לגרום לשגיאות.

משפטי השמה: שגיאות הנובעות מייחוס מקביליות במחשב התגלו אצל תלמידים שעסקו בניתוח תכניות בהן אין משפטי בקרה, אך יש יותר ממשפט השמה אחד המתייחס לאותו משתנה. תלמידים סבורים שמה שקורה בשלב מאוחר של ביצוע התכנית משפיע על מה שקורה מוקדם יותר. לדוגמא, תלמידים רבים סבורים שקטע הקוד הבא, שמטרתו לחשב ולהציג כפלט שטח של מלבן, כתוב כהלכה, למרות שסדר המשפטים בו שגוי:

```
Area = Height * Width;
read(Height);
read(Width);
write(Area);
```

התלמידים סבורים שהמחשב ימתין עד לקבלת ערכי צלעות המלבן, ורק לאחר מכן יחשב את שטחו. לדעת המחבר גם שגיאה זו נובעת מהשלכה של התנסחות בשפה טבעית: הבנת המשמעות של טקסט הכתוב בשפה טבעית אינה מחייבת בדרך כלל, קריאתו על פי סדר עריכתו.

התכוונות

שגיאות "התכוונות" הן אלו הנובעות על ידי ייחוס כוונות או ראיית הנולד לתכנית מחשב, במובן של ציפייה שהתכנית תתבצע באופן שהוא מעבר לאינפורמציה הכתובה בה באופן מפורש. השגיאות באות לידי ביטוי במשימות של ניתוח של תכניות נתונות מראש, בהן התלמיד מתבקש לציין מה יהיה פלט התכנית, או לתאר את מה שהתכנית מבצעת. לדוגמא, תלמידים מצפים שתכנית תבצע פעולות המתייחסות לערכי קצה של משתנה, כאשר בפועל היא לא אמורה לבצע דבר כאשר משתנה הבקרה מקבל ערך קצה. נימוקי תלמידים הם: "המחשב רוצה לבצע", "המחשב יבצע כי זאת המשימה שלו, ורק אחר כך יפסיק".

אגוצנטריות

שגיאות מסוג זה הן ההיפוך של שגיאות מסוג "התכוונות". השגיאות באות לידי ביטוי בפיתוח וכתיבת תכניות האמורות לבצע משימה מסוימת. תלמידים מניחים שהתכנית תבצע מעבר למה שהם כתבו בה באופן מפורש. הם מצפים שהמחשב "יודע" למה הם התכוונו, גם אם לא רשמו במפורש את כל משפטי התכנית הנחוצים לביצוע המשימה. הם מניחים שהמחשב "ישלים" לבד את החסר, כיוון שהוא יכול על פי מה שכתוב להסיק מה חסר. למשל, תלמיד ששוכח לרשום ';' בסוף משפט, מצפה שהמחשב יבין וישלים זאת בעצמו. או למשל, אם לא נעשה איתחול של משתנים, המחשב אמור לתת ערכים משלו (על פי ברירת מחדל - ב.ה.). דוגמא אחרת של שגיאות מטיפוס "אגוצנטריות" היא שימוש שגוי באותו משתנה לתפקידים שונים. תלמידים מצפים שהמחשב ידע להבחין בין תפקידי המשתנה ובהתאם יציב בהם ערכים שונים.

כדוגמא המחבר מציג קטע של תכנית שגויה העושה שימוש במשתנה X כמקום אחסון לערך המתקבל כקלט, וכמשתנה צובר:

```
repeat
read(X);
X = X + X;
until X + X > 100;
```

מסקנות

לטענת המחבר, שלושת טיפוסי השגיאות נובעות ממה שהוא מכנה "superbug" המעיד על הנחה סמויה שקיים "מוח נסתר" של שפת המחשב, שיש לו יכולות היסק ופירוש. מוח זה יודע את אשר התרחש או עתיד להתרחש בתכנית, מעבר למה שכתוב בה במפורש. הוא יכול להבין את כוונות המתכנת, גם אם כוונות אלו לא נוסחו מפורשות בתכנית.

לדעת המחבר, תלמידים אינם מניחים באופן מפורש שהמחשב "חכם". המחבר סבור שמה שקורה בפועל, הוא שהם מנסים לעשות העברה של צורת התנסחות המקובלת בדו-שיח בין בני אנוש, לדו-שיח בינם לבין המחשב, ועל כן מטבע הדברים מייחסים לו תכונות של ניהול שיחה של בן אנוש המסוגל לפרש משפטים עמומים ולוקים בחסר.

השלכות להוראה: המורים צריכים להיות מודעים לשגיאות תלמידים הנובעות משימוש באנלוגיה של דו-שיח בן בני-אנוש, לכתיבת תכניות מחשב. יש לפתח פעילויות למידה וכלי אבחון לאיתור שגיאות. דגש מיוחד צריך להיות על בעיות ניתוח ועל פעילות של ניפוי שגיאות. יש להמחיש לתלמידים באמצעות מודלים מתאימים, את תהליכי הבקרה והביצוע של תכניות מחשב, בשפת התכנות בה הם מתכנתים.

ידע שברירי ושימוש חסר באסטרטגיות של מתכנתים מתחילים

ד.נ. פרקינס ו- פ.מרטין

D.N. Perkins and F. Martin (1986). Fragile knowledge and neglected strategies in novice programmers, In *Empirical Studies of Programmers*, E., Soloway, Y., Iyengar (eds.), Albex Publishing Corporation, Norwood, New Jersey, 213-229.

אודות המאמר

המאמר מתייחס לקשיים של מתחילים בתכנות הנובעים מידע לקוי ומחוסר באסטרטגיות כלליות של פתרון בעיות. המאמר מציג ממצאי מחקר שנערך במטרה לאפיין את מקורות הקושי של מתחילים בתהליכי פתרון בעיות. הממצאים מעידים על כך שלמתקשים יש ידע "שברירי" - ידע חלקי, לא נגיש, או מנוצל לרוב בצורה לא נכונה. הממצאים מראים שהכוונה נקודתית מבוססת אסטרטגיות של פתרון בעיות עשויה בדרך כלל לעזור למתקשים בפתרון בעיות.

<u>רשימת מושגים</u>

- ידע חסר – Missing knowledge
- ידע מגובב – Conglomerated knowledge
- ידע מקובע – Inert knowledge
- ידע לא מתאים – Misplaced knowledge
- ידע שברירי – Fragile knowledge

הנחות יסוד

הגישה הטבעית להגדרת קשיים של תלמידים היא לברר מה התלמידים יודעים ומה הם אינם יודעים. לדוגמה, האם הם מכירים תחביר ומשמעות של משפטים בשפת התכנות? האם הם יודעים לזהות מתי להשתמש במשפט זה או אחר? האם יש להם אסטרטגיות של פתרון בעיות המאפשרות ניתוח בעיה? האם הם בנו מאגר של תבניות תכנות (programming plans) המשמש אותם בכתיבת תכנית, והאם הם יודעים לזהות את התבניות המתאימות לפתרון בעיה נתונה?

יתכן ולמתחילים יש יכולות קוגניטיביות כלליות של פתרון בעיות, אך השליטה שלהם בשפת התכנות דלה מדי, ואינה מאפשרת להם להשתמש כראוי באסטרטגיות של פתרון בעיות. לחילופין, יתכן והמתכנת המתחיל רכש ידע בסיסי בשפה, אך חסרות לו יכולות קוגניטיביות כלליות הדרושות לשימוש מושכל בידע זה.

המחברים גורסים כי אין לדבר על ידע מושלם או על חוסר ידע מוחלט. הם גם מתייחסים לנגישות לידע ולאופן השימוש בו לפתרון בעיות. התלמיד יכול להיות בעל דרגות שונות של ידע - חלקי אך לא מספיק, לא נגיש, או לא מנוצל נכון לפתרון בעיות. ידע כזה מכונה על ידי המחברים **כידע שברירי** (fragile knowledge).

תיאור המחקר

מטרת המחקר היתה לחקור את מוקדי הקושי של מתכנתים מתחילים. החוקרים פתחו שיטה הבוחנת אם קושי מסוים נובע מחוסר מיומנויות של פתרון בעיות, או בגלל שליטה בלתי מספיקה במבנים ומשפטים ספציפיים בשפת התכנות.

אוכלוסית המחקר

20 תלמידי תיכון (כיתות י'-י"ב) שלמדו קורס שנתי "מבוא לתכנות בסביבת BASIC". המחקר נערך במהלך הסמסטר השני לאחר שהתלמידים למדו לולאות FOR-NEXT.

השיטה

כל תלמיד עבד בנוכחות חוקר, והתבקש לבחור שלוש בעיות מתוך 8 בעיות המדורגות בקושי עולה, ולפתור אותן בזו אחר זו במשך 45 דקות. כל הבעיות התייחסו להדפסת צורות הבנויות מעימוד מסוים של כוכביות (עמודת או שורת כוכביות בגודל נתון, מלבן מלא או חלול של כוכביות וכדומה). החוקר צפה בתלמיד ומדי פעם שאל שאלות שמטרתן לעקוב אחרי דרך החשיבה של התלמיד. כאשר התלמיד נתקל בקושי אותו לא הצליח לפתור, החוקר התערב בתהליך הפתרון והציג לתלמיד תחילה **שאלות הכוון** (prompts) מהסגנון של "רגע חושבים". שאלות אלו היו כלליות כדוגמת "מה הדבר הראשון שאתה צריך לומר למחשב לעשות?", "כיצד תתאר את הבעיה לעצמך?", או שאלות ממוקדות יותר המתייחסות לשפת התכנות. אם מספר שאלות הכוון לא עזרו לתלמיד, החוקר נתן לו **רמז** (אחד או יותר). הרמזים, בניגוד לשאלות הכוון, התייחסו באופן יותר ממוקד יותר לפתרון הבעיה הנתונה, וכיוונו את התלמיד לפתרון. למשל "הבעיה מחייבת לבצע משהו מספר פעמים, איזה משפט בשפה יכול להתאים?". במידה ומספר רמזים לא קידמו את התלמיד, החוקר סיפק **תשובה נקודתית** לקושי שנתקל בו התלמיד והתלמיד התקדם לשלב הבא של פתרון הבעיה הנוכחית.

ההתערבות ההדרגתית של החוקר בתהליך פתרון בעיה על ידי התלמיד, החל משאלות ההכוון, הרמזים והתשובות הנקודתית נועדה אמנם לעזור לתלמיד, אך מטרתה העיקרית הייתה לסייע לחוקר לזהות ולאפיין את מוקד הקושי של התלמיד.

ממצאים

צורות שונות של ידע

תגובות שגויות של התלמידים לשאלות ההכוון ולרמזים מוינו לארבע קטגוריות: **השמטה** - כשלון באיתור המרכיב הדרוש, או תחליף למרכיב זה. לדוגמה, השמטת משפט שלם, השמטת מרכיב במשפט, או אי שימוש באמצעי לעיצוב הפלט. **העתקה** - הזזת מרכיב מסוים של המשפט ממקומו למקום אחר בתכנית. **סידור לקוי** - סדר לא נכון של משפטי התכנית, מרכיב ממוקם במקום לא נכון. לדוגמה, במקום קינון לולאות, רישום הלולאות זו אחר זו. **שגיאות אחרות** - תגובות שגויות שלא ניתן לשייכן לשלושת הקטגוריות לעיל.

באשר לאיפיון הידע של התלמיד, הוגדרו ארבע קטגוריות: **ידע חסר** (missing knowledge), **ידע מקובע** (inert knowledge), **ידע לא מתאים** (misplaced knowledge), **וידע מגובב** (conglomerated knowledge).

ידע חסר

עדות לידע חסר אצל התלמיד קיימת כאשר שאלות הכוון ורמזים לא עזרו לתלמיד להתמודד עם הקושי. בנוסף, התלמיד אף לא גילה הבנה של התשובה שסיפק החוקר.

ידע מקובע

זהו ידע הקיים אצל התלמיד, אך במצב מסוים, הידע אינו נגיש והתלמיד לא מצליח להשתמש בו לפתרון הבעיה. הידע אינו נגיש בהקשר של בעיה נתונה. למעשה, התלמיד אינו מצליח לעשות העברה של הידע מסוג בעיות אחד למשנהו. יתכן וקל לו להשתמש בידע הספציפי לפתרון בעיות מסוימות (למשל בעקבות דוגמה שראה בכיתה) אך קשה לו לזהות שבבעיות אחרות, או בהקשר אחר גם ניתן להשתמש בידע זה. עדות לידע מקובע אצל התלמיד קיימת כאשר התלמיד לא השתמש במרכיב הדרוש, או ביצע שגיאה אחרת (שאינה העתקה, או סידור לקוי), אך הצליח לתקן בעקבות שאלות הכוון או רמזים. לחילופין, גם אם לא הצליח התלמיד לפתור את הקושי בעקבות שאלות הכוון או רמזים, אולם לאחר שהחוקר סיפק את התשובה הצליח להבין את התשובה ואת התאמתה לפתרון הבעיה, הרי שניתן להסיק שמדובר בידע הקיים אצל התלמיד, שלא היה נגיש בסיטואציה המסוימת. עדות לידע מקובע קיימת לא רק בשאלות פיתוח, בהן התלמיד מתבקש לכתוב תכנית הפותרת בעיה נתונה, כי אם גם כאשר תלמיד צריך, אך לא מצליח, להעריך פתרון חלופי לבעיה.

כיצד מתארים המחברים את הסיבה לקיבעון של ידע? אצל מומחים קיימת רשת ענפה של אסוציאציות המקשרת הוראות בשפה עם תבניות תכנות. כאשר מתכנת מומחה אינו מצליח להגיע לתבנית מתאימה דרך מסלול מסוים ברשת האסוציאציות, הוא לרוב מצליח לשלוף את התבנית הרצויה בדרך חלופית. לעומת זאת, אצל המתכנת המתחיל, הרשת האסוציאטיבית בהכרח דלילה. כאשר ניסיון ראשון לשלוף את התבנית מתאימה נכשל, אין ניסיון נוסף לשליפה בדרך אחרת.

ידע לא מתאים

ידע לא מתאים בא לידי ביטוי בכך שמשתמשים בכלים המתאימים למשימה מסוימת, בהקשר אחר שאינו מתאים. דוגמה לכך היא שתלמידים מפעילים שיטה אותה למדו כדי לפתור בעיה מורכבת, גם כדי לפתור בעיה פשוטה בהרבה, אשר ניתנת לפתרון בדרך פשוטה יותר. ידע לא מתאים עלול לגרום לקיבוע של ידע: ידע רלוונטי נשאר לא זמין כיוון שידע לא מתאים נבחר במקומו לפתרון הבעיה.

עדות לידע לא מתאים אצל התלמיד קיימת כאשר התלמיד בוחר כלי לא מתאים, משנה את בחירתו הלוך ושוב ללא הצלחה, ולמעשה "הולך לאיבוד". במקרה זה, שאלות הכוון ורמזים אינם מועילים.

מדוע תלמידים משתמשים בידע לא מתאים? אחת הסיבות היא הכללת יתר, או חוסר הבחנה בין מקרים שונים. בהתאם, התלמידים סבורים שפתרון המתאים לבעיה מסוימת, בהכרח יתאים גם לבעיה אחרת, גם אם אין לכך הצדקה מלאה. סיבה נוספת, היא שיש לתלמידים הנחה מוטעית שחובה עליהם להשתמש בדבר האחרון שלמדו כדי לפתור בעיה. בהתאם, הם משתמשים בדבר האחרון שנלמד בין אם מתאים או לא מתאים.

ידע מגובב

ידע כזה בא לידי ביטוי בכך שתלמיד יוצר קוד על ידי דחיסה של אלמנטים שונים בצורה חריגה מבחינה תחבירית או סמנטית, בעת ניסיון לנסח קטע קוד דרוש. הקוד הנוצר אמנם מבטא, באופן לא מדויק, את הכוונה של הכותב, אך אינו נכון מבינה תחבירית או לוגית.

המחברים טוענים שאין הבחנה ברורה בין ידע מגובב וידע לא מתאים. כלומר, יש מקרים שקשה להסיק אם קוד שגוי משקף ידע לא מתאים או ידע מגובב. יחד עם זאת יש מקרים בהם ההבחנה ברורה. למשל, קוד משקף ידע לא מתאים, אם שורות קוד אינן נמצאות במקום המתאים, אך יחד עם זאת אין ערבוב של קטעי קוד שונים. לעומת זאת, ידע מגובב אינו משקף גם ידע לא מתאים, אם אין עדות לכך שקטע קוד לא מתאים משובץ בפתרון.

מה הסיבה לקיומו של ידע מגובב? תלמידים מתחילים מנסים להציג פתרון לבעיה, גם אם אינם יודעים את הפתרון הנכון. הם מנסים בכל זאת לכתוב "משהו". חלקם אינם מייחסים חשיבות למגבלות הפורמליות של תחביר השפה, ומייחסים למחשב יכולת הבנה של קוד הנכתב בנוסח הקרוב לשפה טבעית. חלקם אינם מנתחים את הבעיה, ואינם מבחינים בינה לבין בעיות אחרות, וכתוצאה מכך גורמים לשימוש בידע לא מתאים, העלול להוביל לשימוש בידע מגובב. מתחילים מתקשים במעקב אחר קוד הנוצר על ידי ידע מגובב, ולכן אינם מתקנים אותו.

שימוש חסר באסטרטגיות של פתרון בעיות

ממצאי המחקר הראו, כי בשליש מהמקרים, שאלות הכוון ורמזים עזרו לתלמידים לשלוף את הידע הדרוש לפתרון הבעיה, למרות שבתחילה הם התקשו לשלוף אותו ללא עזרה. ממצאים אלו מעידים על כך, שידע לבדו אינו מספיק, וכדי להשתמש בו במצב המתאים, יש לנקוט באסטרטגיות מתאימות של פתרון בעיות. משמעות הממצאים היא בכך שתלמידים יכולים לעזור לעצמם בתהליך פתרון בעיות על ידי הצגת שאלות הכוון לעצמם. בהתאם, מומלץ להנחות את התלמידים כי תהליך של פיתוח תכנית כולל בקרה עצמית המבוססת על שאלות הכוון. מידת התרומה של שאלות הכוון תלויה באופן ארגון הידע שיש להפעיל ובמידת הנגישות אליו. שאלות ההכוון קשורות להיבטים שונים של פתרון בעיות - הגדרת מטרות, יצירת פתרונות, הערכת פתרונות, עריכת שינויים.

אסטרטגיה חשובה בפיתוח תכנית היא בדיקה חוזרת ונשנית האם התוכנית הנבנית תענה בסופו של דבר על המטרות שהוגדרו. יישום אפשרי של אסטרטגיה זו הוא ניבוי של "מה מבצע קטע הקוד שכתבתי" ומעקב מסודר של מה שאכן מתבצע. מעקב מסודר של ביצוע התכנית משמש גם לניפוי שגיאות, ולניתוח התנהגות של תכנית - "מדוע התכנית אינה מתנהגת בהתאם לציפיות?", "מדוע מתקבל פלט מסוים?", וכדומה.

סיכום

ידע שברירי מחריף כאשר קיים חסך באסטרטגיות בסיסיות של פתרון בעיות. הסבר אפשרי לקשיי מתכנתים מתחילים, אינו יכול להתמקד בידע לוקה בחסר לעומת חסך באסטרטגיות. השילוב ביניהם הוא הקובע את יכולת פתרון הבעיות של התלמיד. כל הרחבה של ידע חייבת להיות משולבת בהרחבת הרפרטואר של אסטרטגיות של פתרון בעיות.

הסיבות להיווצרות ידע שברירי הן מגוונות: רשת דלה של אסוציאציות; חוסר הבחנה בין משפטים ומבנים בשפת תכנות; קישור של מבנים בשפת תכנות וקטעי קוד להקשרים ספציפיים, ללא יכולת הכללה (ובהתאם, ללא יכולת התאמתם לבעיות דומות נוספות); התייחסות לשפת מחשב כדומה לשפה טבעית (ייחוס יכולת הבנה של נוסח לא פורמלי של משפט); וחוסר בקרה עצמית באמצעות שאלות הכוון.

המחברים ממליצים לנצל את הממצאים לשיפור דרכי ההוראה. הוראת תכנות צריכה להיות מכוונת לצמצום תופעת הידע השברירי.

העובדה ששאלות הכוון ורמזים מסייעים לתלמיד להתגבר על קושי, משמעותה שלתלמיד לא חסר ידע, ועליו לשפר את הנגישות לידע. העובדה שתלמידים מגלים ידע לא מתאים או מגובב, מעידה על כך שיש להם יכולת בסיסית לפתור את הבעיה, והם מנסים להתמודד ולהציע פתרון, גם אם לא מושלם.

יש להדגיש את המשמעות הפונקציונלית של משפטים ומבנים בשפה, ואת כלליותם, במטרה לצמצם את התופעה של ידע מקובע. יש להביא לתודעת התלמידים את המשמעות המדויקת של משפטי השפה, במטרה לצמצם את התופעה של ידע לא מתאים או מגובב.

יש לעודד תלמידים לחשיבת חקר מבוקרת - מותר להניח פתרון, אך יש לבצע מעקב מסודר של הפתרון במטרה להעריך האם הוא עונה על דרישות הבעיה. ניסוי וטעייה מבוקרים מהווים חוויה לימודית חשובה, העשויה לתרום להבנת השפה על ידי יצירת אפשרויות ובדיקתן ביחס לידע הקיים כבר אצל התלמיד.

יש לעודד את התלמידים להשתמש באסטרטגיות בסיסיות של פתרון בעיות, וכי תהליך של פיתוח תכנית צריך להיערך תוך בקרה עצמית מבוססת שאלות הכוון, ותוך בדיקה חוזרת ונשנית של ביצוע התכנית.

אבל התכנית שלי רצה!

חוקי דו-שיח עבור מתכנתים מתחילים

ס.א. ג'וני ו- א. סולווי

S.A. Joni and E. Soloway (1986). But my program runs! Discourse rules for novice programmers. *Journal of Educational Computing Research*, Vol 2(1), 95-126.

אודות המאמר

המאמר מתאר שיטה להערכת תכניות "עובדות" (בעלות התנהגות נכונה של קלט/פלט לכל קלט ממרחב הבעיה), אך בנויות בצורה גרועה. לתלמידים מתחילים יש אמונות לגבי "מהי תכנית טובה". בעיניהם תכנית טובה נמדדת אך ורק במונחי התנהגות קלט/פלט. בהתאם לכך, הם לרוב מפתחים תכניות שאינן בנויות היטב, ולוקות בקריאותן. חשוב שלמורים יהיו כלים להסביר לתלמידים מדוע תכניות מסוימות טובות מאחרות, לא רק על סמך שיקולי יעילות, אלא בעיקר משיקולי קריאות. המאמר מציג כללי התנסחות: חוקי דו-שיח המשמשים הן לכתיבת תכניות, והן לקריאתן ופירושן.

<u>רשימת מושגים</u>

- איתחול מרובה – Multiple initializations
- התניה לא מתאימה – Inappropriate conditionalization
- חוקי דו-שיח בתכנות – Rules of programming discourse
- יעילות – Efficiency
- קריאות תכנית – Program readability
- תבניות תכנות – Programming plans

מבוא

תלמידים מתחילים מפתחים לרוב תכניות "עובדות" שאינן בנויות היטב. הם אינם מבינים מדוע המורה אינו מעריך את תכניתם כטובה מספיק, שהרי היא מבצעת את המצופה ממנה (במונחי התנהגות קלט/פלט). המטרה שלנו כמורים היא לתת לתלמיד המתחיל כלים לשיפוט של תוכניות מהיבטים נוספים מלבד נכונות.

לקויות של תכניות שאינן בנויות היטב

המחברים מציינים מספר לקויות של תכניות שאינן בנויות היטב: קוד לא נכון, שימוש במשתנים מיותרים, שימוש לא מתאים במשתנים גלובליים ו/או לוקליים, ריבוי אתחול של משתנים, מבנים מגושמים, ריכוז לא מתאים של תת משימות, ניהול בקרה לא נכון.

להלן דוגמה לשימוש במשתנה מיותר: שימוש ביותר ממשתנה אחד לייצוג אותה ישות/תכונה. הדוגמה המובאת במאמר היא קטע קוד המממש את הרעיון של מסננת קלט. הקוד נכון, אך יש בו שימוש במשתנה מיותר:

```
הצג כפלט ('אנא הקש את המשכורת שלך')
    קלוט ערך למשתנה Salary
    כל עוד  Salary < 0 בצע
        הצג כפלט ('קלט שגוי, אנא הקש שנית את המשכורת שלך')
        קלוט ערך למשתנה NewSalary
        Salary  <-- NewSalary
```

משתנה אחד (Salary), המייצג את התכונה שיש לקלוט את ערכה, משמש לקלט ראשוני. כל עוד הקלט אינו תקין, מתבצע קלט למשתנה עזר (NewSalary), וערכו מושם במשתנה Salary. כמובן שמשתנה העזר מיותר, וכן מתבצעת פעולת השמה מיותרת.

הערכת איכות של תכנית

חשוב שמתחילים ירכשו בהדרגה יכולת הערכת תכניות תוך התייחסות למספר מאפיינים כגון: נכונות, ידידותיות למשתמש, יעילות, מידת קלות לניפוי שגיאות, הרחבה ותחזוקה. השאיפה היא שהיכולת להעריך תכניות באופן זה, תגרום גם לפיתוח תכניות איכותיות.

מדוע יעילות אינה יכולה לשמש מדד עבור מתחילים?

יעילות אינה יכולה לשמש מדד מרכזי בהערכת תכניות על ידי מתחילים. הסיבה: יעילות אינה מושג משמעותי עבור מתחילים הכותבים תכניות קצרות ומצומצמות. טענת המחבר היא כי רק מי שהתנסה בפיתוח תכניות גדולות יכול להבין את משמעות מושג היעילות וחשיבותו כמדד להערכת תכנית (לא נכון בהכרח - ב.ה.).

התמקדות ביעילות בשלב מוקדם מדי עלולה ליצור אצל מתחילים את הרושם הלא נכון שהתכניות הטובות ביותר הן אלו שהן היעילות ביותר. המתחילים אינם מודעים ל- trade-off בין מאפיינים שונים של תכנית, הצגת היעילות כמדד מרכזי עלולה להביא בסופו של דבר לפיתוח תכניות יעילות אך לא קריאות, שאינן ניתנות לניפוי שגיאות בנוחות מספקת, וקשות לשינוי ולתחזוקה.

מדוע צריך להגדיר חוקי דו-שיח עבור מתחילים?

המחברים רואים כמטרה מרכזית להגדיר חוקי דו-שיח שיאפשרו למתחילים דיון במאפייני תכנית. לדעתם חשוב להשריש קריאה, ניתוח והערכת תכניות, כחלק מפעילות ההוראה/לימדה. התלמיד צריך לבצע הערכה של תכניות שהוא כתב, של תכניות שכתבו עמיתים ושל תכניות שנכתבו על ידי המורה.

המחברים מציינים כי שיפוט של תכניות מבוסס על ידע של מומחים, ובהתאם, ההתנסחות המקובלת לגבי איכות של תכניות היא במונחים בהם משתמשים מומחים. שימוש בטרמינולוגיה זו בדו-שיח עם מתחילים יוצר קשיי תקשורת, ואינו משיג את מטרתו. לפיכך, כדי לדון עם מתחילים על מאפייני תכנית, יש לנסח חוקי דו-שיח מיוחדים.

קריאות של תכנית

תכנית קריאה (readable) היא תכנית הכתובה בהתאם לחוקי דו-שיח של מתכנתים. תכנית יכולה להיות נכונה אך לא קריאה בגלל שאינה כתובה בהתאם לכללים המקובלים.

מומחים בונים תכנית תוך שימוש בתבניות (קטעי קוד המשמשים לפתרון בעיות אופייניות כגון צבירה, חיפוש איבר בסדרה וכו'). הם כותבים בהתאם לכללי התנסחות מקובלים, כגון מתן שמות משמעותיים למשתנים, ושילוב משמעותי של תבניות. מחקרים מעידים על כך שמתכנתים מצפים מעמיתיהם לכתוב בהתאם לכללים מקובלים.

תכנית קריאה היא תכנית בעלת מבנה ברור שאינה מכילה אלמנטים מפתיעים לקורא היוצרים אצלו הפרת ציפיות, כפי שמתואר בדוגמה הבאה.

דוגמה לחוק המתייחס לבחירת שמות משתנים: ***שם המשתנה צריך לייצג את תפקידו בתכנית***.

על פי חוק זה, לא מתקבל על הדעת לכנות בשם Max משתנה האמור להחזיק בסופו של דבר את הערך הקטן ביותר בסדרת מספרים. אדם הקורא תכנית המוצאת ערך מינימלי בסדרה, מצפה ששם המשתנה המחזיק את הערך המינימלי יעיד על תפקידו (למשל Min). כאשר הקורא פוגש שם משתנה הנוגד את ציפיותיו, נוצרת אצלו בעיה, ועליו "לשקם" ולשחזר את התמונה שיצר לעצמו בהתייחס לתכנית אותה הוא קורא.

אמירות תכנות וחוקי דו-שיח

המחברים ניסחו אמירות כלליות (בגדר המלצות) וכללים ספציפיים לכל אמירה בהקשר של כתיבת תכניות. מטרת האמירות והכללים להמליץ על מה רצוי לעשות ומה ממה רצוי להימנע בכתיבת תכניות. האמירות והכללים נוסחו בעקבות צורות התנסחות לא ברורות של מתחילים, הפוגמות בקריאות של תכנית.

האמירה הראשונה מתייחסת לאופן בחירת משתנים לייצוג נתונים.

<u>אמירה 1:</u>

חשוב בזהירות מהם התפקידים שיהיו לנתוני הבעיה בתכנית שלך. לכל תפקיד התאם משתנה אחד. תן למשתנה שם משמעותי המעיד על תפקידו.

פחות מדי משתנים: אמירה זו ממליצה להתאים שם אחד לכל תפקיד של מידע, אולם אינה מתייחסת למקרה שמשתנה משמש לשני תפקידים. משתנה כזה מכונה משתנה בעל תפקיד כפול (double-duty variable). מסתבר ששימוש במשתנים כאלו עלול ליצור בעיה בקריאות תכנית. בהתאם, על בסיס האמירה, נוסח הכלל הבא:

<u>כלל:</u> המנע משימוש במשתנים בעלי תפקיד כפול. בחר משתנים שישמשו לתפקיד אחד בלבד, ותן למשתנה שם ברור לציון תפקידו.

יותר מדי משתנים: לעתים תלמידים עושים שימוש במספר משתנים עבור אותו תפקיד (כמו בדוגמת האלגוריתם לסינון קלט שהוצג לעיל). משתנים כאלו מכונים משתנים בעלי תפקיד חלקי (half-duty variable). שימוש במשתנים כאלו פוגע בקריאות תכנית. הכלל שמיועד למנוע בחירת משתנים בעלי תפקיד חלקי מנוסח להלן:

<u>כלל:</u> המנע משימוש במספר משתנים לתפקיד בכל מודול של התכנית. בחר משתנה יחיד לכל תפקיד, ותן לו שם ברור לציון תפקידו.

אמירה 2 מתייחסת לכתיבת תכנית שתאפשר מענה ברור לשאלות הבאות: "מה הקוד הזה עושה כאן?", "כיצד הוא תורם להשגת מטרת התכנית?". קוד עודף מטעה את הקורא, ופוגם בהבנת התכנית. קוד מגושם ולא ברור גורם להטעיית הקורא, ופוגם אף הוא בהבנת התכנית. קורא שנתקל בקוד בעל מאפיינים אלו, עלול לצפות לתבניות שמספקות מטרות שונות מאילו להן התכוון כותב התכנית.

<u>אמירה 2:</u>

1. **וודא שכל קטעי הקוד משמשים לביצוע המשימה הנדרשת.**
2. **השתמש במבנים בשפה המציגים באופן ברור את מה שהתכנית מבצעת.**

בעיית אתחול: תלמידים נוטים להיכשל באתחול משתנים: או שהם לא מאתחלים את המשתנה, או שמאתחלים אותו יותר מפעם אחת- פעם אחת בתחילת התכנית, ופעם נוספת לפני השימוש בו. יש תלמידים הסבורים שכל משתנה יש לאתחל על ידי קבוע, ורק לאחר מכן אפשר לקלוט אליו נתון, או להשים בו ערך של תוצאת חישוב. תלמידים המאתחלים אוטומטית כל משתנה ל- 0, עלולים לגרום לקורא לצפות שהמשתנה משמש בתפקיד של מונה או צובר, וכאשר הקורא נתקל במופע הבא של המשתנה ומתברר לו שהמשתנה אינו מונה או צובר, נוצר קונפליקט. במטרה למנוע בעיות מסוג זה נוסח הכלל הבא:

<u>כלל:</u> הקפד לאתחל כל משתנה פעם אחת בלבד.

שימוש נכון במבני שפה: תלמידים אינם מקפידים על שימוש נכון במבני השפה לכתיבת תכניות. לעתים הם בוחרים במבנה לא מתאים למימוש רעיון מסוים, ולעתים למרות שבחרו במבנה מתאים אינם משתמשים בו כהלכה. תופעות אלו אינן גורמות בהכרח להתנהגות קלט/פלט לא נכונה, אך ללא ספק פוגמות בקריאות התכנית ומקשות על הבנתה. תלמידים משתמשים במבנה של ביצוע חוזר מותנה (לולאת WHILE) כאשר המשפטים בגוף הלולאה צריכים להתבצע פעם אחת בלבד. ברור שבמקרה זה המבנה המתאים יותר הוא IF.

חשוב מאד להרגיל את התלמידים לבחור את המבנים המתאימים ביותר בהקשר נתון, זאת על מנת לאפשר לקורא התכנית ליצור את ההתרשמות הנכונה לגבי מה שהתכנית מבצעת. בהתאם נוסח הכלל הבא:

<u>כלל:</u> השתמש במבנה IF על מנת לדלג על שורות קוד (ביצוע חד פעמי מותנה), ובמבנה WHILE רק כאשר יש צורך בביצוע חוזר מותנה.

העברת פרמטרים: תלמידים אינם מקפידים על הצהרת פרמטרים כפרמטר ערך (call by value) או כפרמטר משתנה (call by reference) בהתאם לשיקולים מתאימים. התנהגות זו יוצרת בעיה בקריאת תכניות על ידי אחרים. אם משתנה של פונקציה/פרוצדורה הוגדר כפרמטר משתנה שלא לצורך, קורא התכנית מפתח ציפיות לא נכונות על תפקיד המשתנה ועל מטרות הפונקציה/הפרוצדורה בה הוא מוגדר. ערכי משתנים המוגדרים כפרמטר משתנה שלא לצורך אינם מוגנים כמובן בפני שינויים, דבר העלול ליצור בעיות בעת ניסיון לערוך שינויים בתכנית.

<u>כלל:</u> אל תגדיר משתני פונקציה/פרוצדורה כפרמטר משתנה שלא לצורך. עשה זאת רק אם המשתנה אמור להיות מאותחל או מעודכן בתוך הפונקציה/פרוצדורה, וכן באם אתה מעוניין שהערך החדש יוחזר (יוכר) ב"סביבה" שקראה לפונקציה/פרוצדורה.

בכל מקרה אחר, אל תגדיר את המשתנה כפרמטר משתנה.

התניה לא מתאימה: תלמידים מתחילים משתמשים במבנה בקרה IF וכוללים בתוך שני הבלוקים שלו משפטים זהים. ביצוע משפטים אלו אינו תלוי בקיום התנאי, ולפיכך מן הראוי שיהיו מחוץ למבנה הבקרה. הדבר מקשה על קורא התכנית להבחין בין משפטים שביצועם מותנה בקיום התנאי, לבין כאלו שלא. כמו כן הקורא עלול להתקשות בהבנת מבנה הבקרה - מה הוא נועד לבדוק ומה מטרת הבלוקים בכל הסתעפות.

<u>כלל:</u> אל תשבץ בתוך מבנה IF קטעי קוד שביצועם אינו מותנה. שבץ אותם סמוך למבנה - לפניו או אחריו.

אמירה 3 מתייחסת לכתיבת תכנית בה קיים מיזוג מטרות.

<u>אמירה 3:</u>

הפחת במידת האפשר את מיזוג המטרות בתכנית שלך. השתדל להתאים לכל מטרה קטע קוד נפרד.

מיזוג מטרות: תלמידים מתחילים נוטים לכתוב קטע קוד המשלב משפטים שנועדו להשגת מספר מטרות. ובמילים אחרות, מספר מטרות של התכנית מזוהות על ידי אותו קטע קוד. מיזוג מטרות והפקת קוד משולב להשגתן אינו שגוי במהותו, ואפילו מחויב כאשר רוצים לבצע אופטימיזציה של תכנית. אולם, מחקרים מראים שמתחילים הכותבים תכניות בסגנון זה, מועדים יותר לכתוב בשגיאות. בהתאם, מומלץ שמתחילים יפרידו את הקוד, ויתאימו קטע קוד לכל מטרה. קטע קוד הממזג מטרות עלול להיות בעייתי עבור קורא התכנית. ראשית, קשה לזהות את המטרות השונות ואת קטעי הקוד השייכים לכל מטרה. שנית, הקורא עלול לזהות משפטים מסוימים כנמצאים לא במקום המתאים בתכנית (על פי כללי הדו-שיח), בו בשעה שמיקומם מתחייב מהמיזוג. אם וכאשר יחליט הקורא "לתקן" את התכנית, ויעביר משפטים אלו למקום אחר בתכנית, הדבר עלול לפגוע בנכונות התכנית.

בעייתיות מיוחדת נוצרת כאשר משלבים בדיקת תקינות של ערך משתנה עם תת משימה אחרת שאינה תלויה בערך זה. הקורא רגיל שבדיקת תקינות ערך משתנה מתבצעת בהקשר של קליטת נתונים, ולכן אינו מצפה שבדיקת תקינות ערכי משתנה תופיע ללא קשר לקליטת נתונים. שילוב מסוג זה יוצר אצל הקורא קושי. בהתאם נוסח הכלל הבא:

<u>כלל:</u> אל תשלב בדיקת תקינות קלט עם אף תת-משימה אחרת.

סיכום

לא די ללמד תלמידים מתחילים כיצד לכתוב תכניות בעלות התנהגות קלט/פלט נכונה לכל קלט אפשרי של מרחב הבעיה. בנוסף, יש ללמדם לבקר את עבודתם ולבדוק האם התכנית נכתבה במבנה נכון, והאם היא קריאה ומובנת.

כללי דו-שיח לתכנות נכון נקבעו עבור מתכנתים מתחילים. הכללים הם מסוג "עשה" ו"אל תעשה" בהקשר של כתיבת תכנית, והם משמשים גם כקווים מנחים לקריאת תכניות.

אמונות תלמידים לגבי תכנות בפסקל

א.א. פלורי

A.E. Fleury (1993). Student beliefs about Pascal programming, *Journal of Educational Computing Research*, Vol 9(3), 335-371.

אודות המאמר

המאמר מתאר מחקר שנערך במטרה לזהות אמונות של מתחילים ביחס למאפייני תכנית המשפיעים על תהליך פיתוח ותחזוקה של תכניות. המאמר עורך השוואה בין אמונות מתחילים ומומחים ודן בהבדלי הגישה שלהם כלפי היבטים שונים של פיתוח מערכות תכנה.

<u>רשימת מושגים</u>

- ניפוי שגיאות ובדיקה – Debugging and testing
- קריאות – Readability
- שינוי והרחבה – Modification and extension
- תחזוקה – Maintenance
- תכנון – Design

הנחות יסוד

הגישה המוצגת במאמר היא הגישה הקונסטרוקטיביסטית הגורסת שידע הנוצר אצל התלמיד אינו מועבר לו ישירות, כי אם נוצר בהדרגה לאורך זמן כתוצאה מפעילויות ואירועי למידה שחווה התלמיד.

תפיסות מוטעות של תלמידים משקפות מלבד המגרעות בידע התכני שלהם, גם מגרעות באמונתם לגבי התוכן הנלמד.

אמונות של תלמידים משפיעות על הלמידה שלהם. על כן מורים צריכים להיות מודעים לאמונות התלמידים ובפרט לזהות מתי אמונות אלו מפריעות לתהליך הלמידה, ולנקוט בפעילות הוראתית מתאימה.

ניתן לשפר את ההוראה על ידי הקשבה לתלמידים ועל ידי פירוש אופן פעולתם תוך מבט מהפרספקטיבה שלהם.

תיאור המחקר

כאשר מלמדים תכנות בהקשר של פיתוח תכנה בהיקף רחב, לא מספיק להכשיר את התלמידים לכתוב ולבדוק תכניות קטנות. מטרת על של הוראת תכנות היא לתת ללומד כלים להערכת מאפייני תכנית מחשב אשר מאפשרים פיתוח תכנה וטיפול ארוך טווח בה בצורה הטובה ביותר. בהתאם, מטרת המחקר היא לבדוק את האמונות של תלמידים לגבי תהליך מתמשך של פיתוח תכנית, ובאופן ספציפי יותר, מה לפי דעתם הם מאפייני תכנית מחשב אשר: (א) משפרים את קריאות התכנית, (ב) מפשטים את תהליך ניפוי השגיאות ובדיקת התכנית (ג) מסייעים ומקלים על תחזוקה, שינוי והרחבה של תכניות.

אוכלוסית המחקר

במחקר השתתפו 23 סטודנטים לתואר ראשון הלומדים קורס מבוא בפסקל, ו- 4 מומחים בעלי ניסיון מוכח בתכנות ב"עולם האמיתי".

השיטה

הסטודנטים בקורס למדו תכנות בהתאם לגישה הקונסטרוטיביסטית, תוך כדי ביצוע שוטף של משימות תכנות ופתרון בעיות. לקראת סיום הקורס, במהלך מספר שבועות, התלמידים רואיינו והתבקשו להעריך גרסאות שונות (בעלות מאפיינים שונים) של תכניות הפותרות אותה בעיה.

ממצאים

כללית נמצאו הבדלים ברורים בין אמונות מומחים לאלו של מתחילים. המומחים גילו גישה יותר הוליסטית ויותר מופשטת, בעוד שהמתחילים התמקדו יותר בדברים קונקרטיים ונקודתיים בהתייחסם למאפייני תכנית נתונה.

המאמר מציג אמונות בהתייחס לקריאות תכנית, ניפוי שגיאות ובדיקת תכנית, תחזוקה, שינוי והרחבה של תכנית, ותכנון של תכניות איכותיות.

קריאות של תכנית

מומחים סבורים ששימוש בכלים לטיפול במורכבות של תכנית משפר את רמת הקריאות שלה. למשל, שימוש במבני נתונים לייצוג נתונים כגון מערכים של רשומות, שימוש מושכל בפרוצדורות עם פרמטרים למניעת שכפול של קוד, ארגון היררכי של זימון פרוצדורות.

מתחילים לעומת זאת, סבורים ששימוש במערכים וברשומות מקשה על קריאות של תכנית. הם מעדיפים שכפול של פרוצדורות (פרוצדורה נפרדת לכל ישות בבעיה), וסבורים שארגון היררכי של זימון פרוצדורות מבלבל ומקשה על קריאות תכניות.

ניפוי שגיאות ובדיקת תכנית

מומחים סבורים שניפוי שגיאות צריך להיערך תוך ניסיון להבנת התכנית כמכלול. בהתאם, הם מעדיפים תכניות שקל להבין את מה שהן מבצעות. הם מודעים לחשיבות בדיקת התכנית ומתייחסים במשנה זהירות לניפוי שגיאות ברמה נקודתית. ברור להם שניפוי חלקי בלבד של השגיאות אינו מספיק. לדעתם מודולריות וקריאות נוחה של התכנית עשויים לעזור לניפוי שגיאות ולבדיקתן. שימוש מושכל בפרוצדורות לכתיבת תכנית עוזר אף הוא.

מתחילים לעומת זאת, רואים את ניפוי השגיאות כתהליך המורכב מסדרת צעדים שתכליתם איתור מיקום השגיאות ותיקונן. הם מייחסים חשיבות יתר לאיתור מיקום השגיאות בתכנית. בהתאם, הם מעדיפים תכניות הבנויות כך שקל בהן לאתר את מיקום השגיאות. חלקם מניחים קיום של שגיאה אחת בלבד ועם איתורה ותיקונה מפסיקים את תהליך ניפוי השגיאות. יש להם נטייה לבדוק את התכנית על דוגמת קלט אחת ויחידה. לדעתם תוכניות מודולריות, ותוכניות שניתן לבצע עליהן מעקב פשוט, נוחות יותר לניפוי שגיאות ולבדיקה.

תחזוקה, שינוי והרחבה

מומחים רואים את התחזוקה, השינוי ואפשרויות ההרחבה כתהליך טבעי במחזור החיים של תכנה. זאת כיוון שהתנסו בכך בעבודתם ב"עולם האמיתי" של פיתוח תכנה. לדעתם, קריאות נוחה של תכנה הכרחית לביצוע פעולות אלו, שכן הן נעשות בדרך כלל על ידי אנשים שלא היו מעורבים בפיתוח התכנה. הם מייחסים חשיבות רבה לאופן ארגון המידע בתכנית, ולהימנעות משימוש לא נכון בהגדרת פרוצדורות (למשל, יש להימנע משכפול פרוצדורות, שכן במקרה שיש לערוך שינוי בהן, יש להקפיד לשנות בכולן).

מתחילים לעומת זאת, רואים את התחזוקה, השינוי וההרחבה של תכנה כתרגיל אקדמי. לא ברור להם מתי במציאות יש לערוך פעולות אלו ומי אמור לערוך אותן. הם מעדיפים עריכה של שינויים קלים ושטחיים בלבד כגון מחיקת שורות קוד, שינוי פרוצדורות לפונקציות על פני שינויים מהותיים בעריכת התכנית.

תכנון תכניות איכותיות

מומחים ומתחילים מסכימים שתכנית איכותית מאפשרת עריכת הפעולות המוזכרות לעיל בצורה קלה ונוחה. שיקולים אלו אמורים להשפיע על תכנון התכנית. ההבדלים ביניהם מתבטאים במידת החשיבות שהם מייחסים לפעולות אלו.

דיון

אמונות התלמידים המתחילים משקפות את השלב בו הם נמצאים בתהליך הבניית הידע שלהם אודות תכנות. הממצאים מראים שאצל התלמידים המתחילים לא נוצר לפי שעה הקונפליקט בין הרצון לעבוד עם טיפוסי נתונים פשוטים, לבין הצורך בטיפוסי נתונים מורכבים כאשר פותרים בעיה מורכבת (במקרה של פתרון בעיה מורכבת, דווקא טיפוסי נתונים מורכבים עשויים לפשט את התכנית). בדומה, לא נוצר עדיין אצל מתחילים הקונפליקט בין הרצון לעבוד עם טיפוסי נתונים פשוטים, לבין העובדה ששימוש בטיפוסי נתונים מורכבים מקל על ניפוי שגיאות ובדיקה, או הרחבת התכנית.

מתוך ההצדקות של מתחילים לגבי מקורות שנעזרו בהם להערכת התכניות, ניתן להבין כי בשלב זה הם לא גילו יכולת של רפלקציה עצמית על העדפתם סגנון של תכנית ואופן ארגונה. נימוקיהם התבססו בעיקר על היבטים טכניים של שפת התכנות, וכללים שהוכתבו על ידי המורה. לא הייתה מעבר לזה התייחסות משמעותית ליתרונות של סגנון תכנות מסוים או ארגון מסוים של הנתונים ושל התכנית.

למעשה, חוסר ברפלקציה עצמית אצל המתחילים מנע את ההתפתחות בשינוי מבנה הידע שלהם.

המומחים לעומת זאת, פיתחו מבנה ידע מתקדם בהרבה מהמתחילים וגילו יכולת של חשיבה פורמלית יותר אודות סגנונות תכנות.

אחת הסיבות להבדלים בין מומחים למתחילים נעוצה בין השאר בהבדל בניסיון המצטבר שלהם בתכנות. ניסיונם של המתחילים בתכנות הסתכם במטלות התכנות שעשו במסגרת לימוד הקורס, והם לא נחשפו עדיין לבעיות בעולם האמיתי. מתחילים קוראים תכניות מחשב רק כשמבקשים מהם לעשות זאת, או כשעליהם לציין את פלט התכנית עבור קלט נתון. מומחים לעומת זאת קוראים תכניות כאשר עליהם לערוך בהם שינוי, או כדי לבדוק אם ניתן להיעזר בהם לפתרון בעיות חדשות. ניפוי שגיאות נעשה על ידי תלמידים על תכניות שהם כתבו זה עתה. מומחים לעומת זאת מנפים שגיאות גם מתכניות שאחרים כתבו. מתחילים מתנסים מעט מאד בשינוי תכניות לעומת מומחים.

מסקנות והמלצות

המורה חייב להיות קשוב לאמונות תלמידים, ולתכנן בהתאם את ההוראה. מומלץ לתכנן מגוון פעילויות ואירועי למידה בגישה קונסטרוקטיביסטית שיתרמו להבניית הידע של הלומד, וימחישו את עבודת המומחה ב"עולם האמיתי". אפשר לבנות פעילות לימודית ברוח הפעילות שנערכה לצורך ביצוע המחקר.

רשימת מקורות לנושא "קשיי מתחילים בתיכנות ויסודות מדעי המחשב"

להלן רשימת מאמרים נבחרים המיועדים מומלצים לקריאה. המאמרים מתייחסים לקשיי מתחילים מנקודות מבט שונות. בחלק מהמאמרים הדגש הוא של שגיאות ותפיסות מוטעות (חלופיות) הקשורות לתחביר ומשמעות של שפות תכנות. חלקם עוסקים בקשיים בפתרון בעיות. חלקם מדגישים את הסיבות לקשיי מתחילים ואחרים מדגישים המלצות לטיפול או מניעת קשיים אלו.

1. Bayman, P., and Mayer, R.E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements, *Communication of the ACM*, Vol. 26(9), 677-679.
2. Bonar, J. & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programming, *Human Computer Interaction*, Vol. 1(2), 133-161.
3. Carpenter, T.P., Corbitt, M.R., Kepner, H.S., Lindquist, M.M. and Reys, R.E.(1980). Solving Verbal Problems: Results and Implications from National Assessment. *Aritmetic Teacher,* Vol. 28(1), 8-13.
4. Clement, J. (1982). Algebra Word Problem Solutions: Thought Processes Underlying a Common Misconception, *Journal for Research in Mathematics Education,* Vol. 13(1), 16-30.
5. Clements, D.H., Guilo, D. (1984). Effects of computer programming on young children. In the *proceedings of Logo 84*.
6. Cunniff, N., Taylor, R.P. and Black, J.B. (1989). In Soloway E. and Spohrer J.C. (eds.), *Studying the novice programmer*. Lawrence Erlbaum Associates, Publishers, New Jersey, 419-429.
7. Du Boulay, B. (1986). Some Difficulties of Learning to Program. J. of Educational Computing Research, Vol. 2(1), 57-73.
8. Du Boulay, B., O'Shea, T. and Monk, J. (1981). The black box inside the glass box: Presenting computing concepts to novices, International J. of Man-Machine Studies, Vol. 14, 237-244.
9. Ebrahimi, A. (1994). Novice programmer errors: Language constructs and plan composition. Int. J. Human-Computer Studies, Vol. 41, 457-480.
10. Fluery A.N. (1993). Student beliefs about Pascal programming. *J.Educational Computing Research*, Vol. 9(3), 355-371.
11. Gray, W.D., Goldberg, N.C. and Byrnes, S.A. (1993). Novices and programming: Merely a difficult subject (why?) or a means to mastering metacognitive skills? A review of Soloway & Spohrer's, Studying the novice programmer. *Journal of Educational Research on Computers*. Vol. 9(1), 131-140.
12. Gugerty, L., Olson, G.M. (1986). Comprehension differences in debugging by skilled and novice programmers. In Soloway E. and Iyengar S. (eds.), *Empirical Studies of Programmers*, Norwood, NJ: Ablex Publishing Co., 13-27.

13. Haberman, B. and Ben-David Kolikant, Y. (2001). Activating "black boxes" instead of opening "zippers" - a method of teaching novices basic CS concepts, *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education,* 41-44.

14. Hembree, R. (1992). Experiments and Relational Studies in Problem Solving: A Meta-Analysis. *Journal for Research in Mathematics Education,* Vol. 23(3), 242-290.

15. Joni, S.A. and Soloway, E. (1986). But my program run! Discourse rules for novice programmers, *J. Educational Computing Research*, Vol. 2(1), 95-128.

16. Kessler, C.M., Anderson, J.R. (1986). A model of novice debugging in LISP. In Soloway E. and Iyengar S. (eds.), *Empirical Studies of Programmers*, Norwood, NJ: Ablex Pub. Co., 198-212.

17. Linn, M.C. (1985). The cognitive consequences of programming instruction in classrooms. *Educational Researcher*, Vol. 14(5), 14-16, 25-29.

18. Linn, M.C., and Clancy, M.J. (1992). The case for case studies of programming problems. *Communications of the ACM*, Vol. 35(3), 121-132.

19. Littlefield, J. & Rieser, J. (1993). Semantic Features of Similarity and Children's Strategies for Identifying Relevant Information in Mathematical Story Problems, *Cognition and Instruction,* Vol. 11(2), 133-188.

20. Littman, D.C., Pinto, J., Letovsky, S. and Soloway, E. (1986). Mental Models and Software Maintenance. In Soloway, Iyengar (eds.), *Empirical Studies of Programmers*. Norwood: Ablex Pub Co., 80-98.

21. Onorato, L.A., Schvaneveldt, R.W. (1987). Programmer/Nonprogrammer Differences in Specifying Procedures to People and Computers. In Soloway E. and Iyengar S. (eds.), *Empirical Studies of Programmers*, Norwood, NJ: Ablex Publishing Co., 128-137.

22. Pea, R.D. (1986). Language-independent conceptual "bugs" in novice programming. *J. Educational Computing Research*, Vol. 2(1), 25-36.

23. Perkins, D.N., Hancock, C., Hobbs, R., Martin, F., Simmons, R. (1986). Conditions of learning in novice programmers, *J. Educational Computing Research,* Vol. 2(1), 37-55.

24. Perkins, D.N., Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. In Soloway E., and Iyengar S. (eds.), *Empirical Studies of Programmers*. Norwood: Ablex Pub., 213-229.

25. Perkins, D.N., Schwartz, S., Simmons, R. (1988). Instructional strategies for the problems of novice programmers. In Mayer R.E. (ed.), *Teaching and Learning Computer Programming, multiple research perspectives*, Lawrence Erlbaum Associates, Inc., 153-178.

26. Puntam, R.T., Sleeman, D., Baxter, J.A. and Kuspa L.K. (1989). A summary of misconceptions of high-school Basic programmers. In Soloway E. and Spohrer J.C. (eds.), *Studying the novice programmer*. Lawrence Erlbaum Associates, Publishers, NJ, 301-314.

27. Kurland, D.M., Pea, R.D., Clement, C. and Mawby, R. (1989). A study of the development of programming ability and thinking skills in high school students. In Soloway E. and Spohrer J.C. (eds.), *Studying the novice programmer*. Lawrence Erlbaum Associates, Pub, NJ, 83-112.

28. Rath, A. and Brown, D.E. (1995). Conceptions of human-computer interaction: A model for understanding student errors. *J. Educational Computing Research,* Vol. 12(4), 395-409.

29. Rosnick, P. (1981). Some Misconceptions Concerning the Concept of Variable, *The Mathematics Teacher*, Vol. 74(6), 418-420.

30. Samurcay, R. (1985). Learning programming: an analysis of looping strategies used by beginning students, *For the Learning of Mathematics*, Vol. 5(1), 37-43.

31. Samurcay, R. (1989). The concept of variable in programming: Its meaning and Use in problem solving by novice programmers. In Soloway E. and Spohrer J.C. (eds.), Studying the novice programmer. Lawrence Erlbaum Associates, Publishers, New Jersey, 161-178.

32. Scherz, Z., Goldberg, D., and Fund, Z. (1990). Cognitive implications of learning Prolog - mistakes and misconceptions, *J. Educational Computing Research*, Vol. 6(1), 89-110.

33. Sleeman. D., Putnam. R.T., Baxer. J., and Kuspa, L. (1986). Pascal and High School Students: A Study of Errors, *J. of Educational Computing Research*, Vol. 2(1), 5-23.

34. Sleeman. D., Putnam. R.T., Baxer. J. and Kuspa, L. (1988). An introductory Pascal class: a case study of students' errors, In Mayer, R. E. (eds.), *Teaching and Learning Computer Programming - Multiple Research Perspectives.* Lawrence Erlbaum Associates, Publishers, New Jersey, 237-257.

35. Soloway, E. (1985). From problems to programs via plans: the concept and structure of knowledge for introductory LISP programming, *J. of Educational Computing Research*, Vol. 1(2), 157-172.

36. Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations, *Communication of the ACM,* Vol. 29(9), 850-857.

37. Soloway, E. and Ehrlich, K. (1984). What do novices know about programming? In Badre, A. and Shneiderman, B. (eds.), *Directions in Human Computer Interaction,* Ablex Publishing Corporation, Norwood, New Jersey, 27-54.

38. Soloway, E., Lochhead, J. and Clement, J. (1982). Does computer programming problem solving ability? Some positive evidence on algebra word problems. In Seidel, R.J., Anderson, R.E. & Hunter, B. (eds.), *Computer Literacy,* Academic press, New Jersey, 171-185.

39. Slone, K.D. and Linn, M.C. (1988). Instructional conditions in Pascal programming classes. In Mayer R.E. (ed.), *Teaching and Learning Computer Programming, multiple research perspectives*, Lawrence Erlbaum Associates, Inc., 207-236.

40. Spohrer, J.C. (1992). MARCEL: Simulating the novice programmer. Ablex Publishing Corporation, Norwood, New Jersey.

41. Spohrer, J.C., Pope, E., Lipman, M., Sack, W., Freiman, S., Littman, D., Johnson, L. and Soloway, E. (1985). Bugs in novice programs and misconceptions in novice programmers. In Duncan, K. and Harris, D. (eds.), *Computers in Education*. North-Holland: Elsevier Science Publishers B.V., 543-552.

42. Spohrer, J.C., Soloway, E. and Pope, E. (1985). Where the bugs are. *Proceedings Computer-Human Interaction*, 47-53.

43. Spohrer, J.C. and Soloway, E. (1986). Analyzing the High Frequency Bugs in Novice Programs. In Soloway E. and Iyengar S. (eds.), *Empirical Studies of Programmers.* Norwood, NJ: Ablex, 230-251.

44. Spohrer, J.C. and Soloway, E. (1986). Alternatives to construct-based programming misconceptions. *Conference proceedings on Human Factors and Computing Systems*, 183-191.

45. Spohrer, J.C. and Soloway, E. (1989). Novice mistakes: Are the folk wisdoms correct?. In Soloway E. and Spohrer J.C. (eds.), *Studying the novice programmer*. Lawrence Erlbaum Associates, Publishers, New Jersey, 401-416.

46. Spohrer, J.C., Soloway, E. and Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. In Soloway E. and Spohrer J.C. (eds.), *Studying the novice programmer*. Lawrence Erlbaum Associates, Publishers, New Jersey, 355-400.

47. פז, ת. (1996). *מדעי המחשב לתלמידי תיכון ברמות גמר: תהליכי למידה והוראה*. חיבור לתואר שני, הטכניון.

48. ליברמן, נ. (1996). טעות לעולם חוזרת / עוזרת: טעויות מושגיות בשפת תכנות פרוצדורלית. *מחשבים בחינוך*, גליון 36, עמודים 22-27.

49. ליברמן, נ. (1997). שגיאות תלמידים בבחינות הבגרות במדעי המחשב: איסוף סיווג ואבחון השגיאות, והמלצות דידקטיות. *מחשבים בחינוך*, גליון 40, עמודים 18-23.