שאלה 2 (30 נקודות): DFA

בשפה KennY יש רק שני סוגים של משפטי השמה: $x := n$ ו- $x := y + n$, כאשר n מספר שלם ו-x, y הם משתנים (יכולים להיות אותו משתנה).

בנוסף ישנם תנאים בוליאניים ומשפטי בקרה בדומה לשפת WHILE. תחביר מלא של השפה נתון על-ידי הדקדוק הבא:

$$
\begin{aligned}
S &\rightarrow x := n \mid x := y + n \mid \text{skip} \\
&\mid \text{if } E \text{ then } S \text{ else } S \\
&\mid \text{while } E \text{ do } S \\
E &\rightarrow x \diamond y \qquad (\diamond \in \{<, >, \leq, \geq, =, \neq\})
\end{aligned}
$$

קני (ממציא השפה) רוצה לעקוב אחרי כל הערכים האפשריים של כל משתנה בתכנית, ולכן בחר להשתמש בסריג קבוצת החזקה של המספרים השלמים $L = \langle \mathcal{P}(\mathbb{Z}), \subseteq \rangle$. לשם כך הוא כתב את פונקציות המעבר הבאות:

s	$[\![s]\!]^{\#}\sigma^{\#}$
$x := n$	$\sigma^{\#}[x \mapsto \{n\}]$
$x := y + n$	$\sigma^{\#}[x \mapsto \{v + n \mid v \in \sigma^{\#}(y)\}]$
skip	$\sigma^{\#}$
if e	$\sigma^{\#}$

(הערה: זכרו שמצב אבסטרקטי $\sigma^{\#}$ שייך לסריג חזקה L^k – עבור תכנית עם k משתנים – ויש בו ערך אבסטרקטי אחד לכל משתנה בתכנית; וכן שהפעולה $\sigma^{\#}[x \mapsto e]$ מחליפה את הערך שמתאים למשתנה x במצב $\sigma^{\#}$ לערך אבסטרקטי חדש e.)

א. הראו שפונקציות המעברים של קני הן מונוטוניות.

מחלקים למקרים לפי סוג המשפט s.

עבור המקרים skip ו if e – טריוויאלי מכיוון שפונקצית המעבר היא זהות.

עבור משפטי השמה: יהיו $\sigma^{\#}_{1,2}$ שני מצבים אבסטרקטיים כך ש $\sigma^{\#}_1 \sqsubseteq \sigma^{\#}_2$. אז לפי ההגדרה של סריג חזקה ושל סריג קבוצת חזקה, לכל משתנה v מתקיים $\sigma^{\#}_1(v) \subseteq \sigma^{\#}_2(v)$. נסמן $\sigma^{\#\prime}_i = [\![s]\!]\sigma^{\#}_i$ עבור $i = 1,2$.

לכל משתנה $v \neq x$ מתקיים ממילא $\sigma^{\#\prime}_i(v) = \sigma^{\#}_i(v)$ ולכן הסדר נשמר. נראה שהסדר נשמר גם עבור x.

משפט $x := n$ – במקרה זה $\sigma^{\#\prime}_1(x) = \sigma^{\#\prime}_2(x) = \{n\}$ ולכן ברור כי $\sigma^{\#\prime}_1(x) \subseteq \sigma^{\#\prime}_2(x)$.

משפט $x := y + n$ – ידוע כי $\sigma^{\#}_1(y) \subseteq \sigma^{\#}_2(y)$ ולכן גם כאשר מוסיפים לכל מספר בקבוצה את הקבוע n נשמרת ההכלה: $\{v + n \mid v \in \sigma^{\#}_1(y)\} \subseteq \{v + n \mid v \in \sigma^{\#}_2(y)\}$.

ב. הראו תכנית דוגמה שעבורה הרצת האנליזה של קני לא עוצרת.

כל תכנית שמכילה לולאה שבה משתנה מקודם בכל איטרציה תראה התנהגות כזו.

ג. $x := 1$

ד. **while** $x < y$ **do** $x := x + 1$

הערך האבסטרקטי בכותרת הלולאה יהיה בהתחלה $\{1\}$, אחרי איטרציה מלאה של האנליזה הוא יהיה $\{1,2\}$, אחר-כך $\{1,2,3\}$ וכן הלאה; לעולם לא נגיע לנקודת שבת.

ה. כדי להתגבר על בעיית אי-העצירה, קני מוכן להתפשר ולעקוב אחרי **שלושה ערכים שונים לכל היותר** עבור כל משתנה בתכנית. הערכים אינם ידועים מראש ויכולים להיות שונים מתכנית לתכנית וכן בין משתנים שונים באותה תכנית.

משמעות הדבר היא, שאם בנקודה כלשהי בתכנית משתנה מסוים יכול לקבל (על פי האנליזה המקורית) ארבעה ערכים שונים או יותר, אז באותה נקודה ניתן יהיה להניח (באנליזה החדשה) שאותו משתנה יכול לקבל *כל ערך שלם שהוא*.

(1) הגדירו את הסריג המתאים, את יחס הסדר ($\sqsubseteq$) ואת אופרטור ה-join ($\sqcup$).

(2) הגדירו פונקציות מעבר חדשות והראו שהן מונוטוניות.

(3) הסבירו מדוע האנליזה המתוקנת **תמיד עוצרת**.

(הערה חשובה: קני מתעלם מערכי האמת של תנאים בולאניים באנליזה שלו, ולכן גם באנליזה המתוקנת מותר לעשות כך.)

(1) הסריג מכיל את כל הקבוצות של מספרים שלמים עד גודל 3, ובנוסף איבר $\top$.
יחס הסדר הוא עדיין הכלת קבוצות, כאשר $\top$ מכיל כל קבוצה ולא מוכל באף קבוצה סופית.

$$a \sqsubseteq b = \begin{cases} a \subseteq b & a, b \neq \top \\ \text{true} & b = \top \\ \text{false} & a = \top, b \neq \top \end{cases}$$

אופרטור join מוגדר כך שאם גודל האיחוד גדול מ-3, התוצאה היא $\top$.

$$a \sqcup b = \begin{cases} a \cup b & a, b \neq \top, |a \cup b| \leq 3 \\ \top & a, b \neq \top, |a \cup b| > 3 \\ \top & a = \top \text{ or } b = \top \end{cases}$$

(2) כל המעברים נשארים אותו דבר, למעט המקרה של $x := y + n$ שבו יש צורך בטיפול מיוחד ב $\top$.

$$[\![x := y + n]\!]\sigma^\# = \begin{cases} \sigma^\#[x \mapsto \{v + n \mid v \in \sigma^\#(y)\}] & \sigma^\#(y) \neq \top \\ \sigma^\#[x \mapsto \top] & \sigma^\#(y) = \top \end{cases}$$

הוכחות המונוטוניות עבור המשפטים שלא השתנו ($x := n$, if e, skip) זהות לסעיף א.
עבור ההגדרה החדשה שוב יש להבדיל בין ערכים "רגילים" לבין $\top$.
אם $\sigma_2^{\#\prime}(y) = \top$ אז $\sigma_1^{\#\prime}(x) = \top$ ולכן בבירור $\sigma_1^{\#\prime}(x) \sqsubseteq \sigma_2^{\#\prime}(x)$.
אם $\sigma_2^\#(y) \neq \top$ ממילא גם $\sigma_1^\#(y) \neq \top$ (כי $\sigma_1^\#(y) \sqsubseteq \sigma_2^\#(y)$), וההוכחה היא שוב כמו בסעיף א.

(3) מכיוון שפונקצית המעברים מונוטוניות ובנוסף בסריג המתוקן **אין שרשראות אינסופיות** (בניגוד לסריג של קני; כאן השרשראות הארוכות ביותר הן באורך 4), האנליזה מתכנסת בזמן סופי.

שאלה 5 (25 נקודות): Code Generation

חלק א (10 נקודות) – טיפול בשגיאות

למדנו בכיתה שאחת הדרכים לממש טיפול בחריגות (exceptions) היא באמצעות שמירת כתובת הexception handler הנוכחי ברשומת ההפעלה. בפתרון זה, בעת זריקת חריגה, נבדוק אם ה exception handler הנוכחי מטפל בחריגה שנזרקה. אם כן, נבצע אותו. אם לא, נמצא את רשומת ההפעלה הקודמת לרשומת ההפעלה הנוכחית, נמצא את הexception handler ששמור בה ונבדוק האם הוא מטפל בחריגה. נמשיך לחפש כך עד שימצא exception handler מתאים (או שנגיע לdefault exception handler).

טימי לא מרוצה מכך שבמנגנון טיפול בחריגות זה אנו נאלצים לבזבז במשאבים ב"חיפוש" ה exception handler הנכון בין רשומות ההפעלה שבמחסנית. טימי מציע לשנות את אופן קימפול exception handlers כך שכל exception handler יכיל בתוכו את כל הexception handler שקודמים לו. לטענת טימי, בפתרון הזה תמיד יספיק לבדוק רק את הexception handler הנוכחי.

לצורך מימוש פתרון זה, נניח שקיימת מחלקת חריגה Exception ממנה יורשות כל מחלקות החריגה האחרות.

כלומר, בהנתן הקוד הבא:

```
1.  void f (int x) {
2.     try {
3.        int y = 0;
4.        try {
5.           if ( x > y ) {
6.              <throw exception>
7.           }
8.        }
9.        catch (Exception1 e) {
10.          <handle exception e1>
11.       }
12.    }
13.    catch (Exception2 e) {
14.       <handle exception e2>
15.    }
16. }
17.
18. void main() {
19.    f();
20. }
```

הקומפיילר ייצר קוד ששקול ל:

```
1'.  void f (int x) {
2'.     try {
3'.        int y = 0;
4'.        try {
5'.           if ( x > y ) {
6'.              <throw exception>
7'.           }
8'.        }
9'.        catch (Exception e) {
10'.          if (typeof(e) is Exception1) {
11'.             <exception handler of line 10>
12'.          }
13'.          if (typeof(e) is Exception2) {
14'.             <exception handler of line 14>
15'.          }
16'.          abort(); // crashes program
17'.       }
18'.    }
19'.    catch (Exception e) {
20'.          if (typeof(e) is Exception2) {
21'.             <exception handler of line 14>
22'.          }
23'.          abort(); // crashes program
24'.    }
25'. }
26'.
27'. void main() {
28'.    f();
29'. }
```

(Exception1 ו Exception2 יורשות מException)

א. (6 נק') הפתרון המוצע אינו יכול לעבוד בתצורה זו. הסבירו את הבעיה בפתרון של טימי והציעו תכנית שמדגימה את הבעיה.

הפתרון כלל מספר בעיות אפשריות. כל אחת מהן התקבלה כפתרון לשאלה:

(1) נשים לב שעל מנת לחסוך לגמרי את ה"חיפוש" ברשומות ההפעלה, אנחנו צריכים שהexception handler החדש שאנחנו מייצרים יכיל את כל הexception handlers שקדמו לו. בפרט זה אומר שהוא צריך להכיל גם את הexception handler מהמתודה הקוראת. נניח למשל שהמתודה f נקראת משתי מתודות g וh, כך שg וh מגדירות exception handlers משלהן. בעת קמפול f לא נדע אם f נקראה מתוך g או מתוך h, ולכן לא נדע איזה exception handler צריך להיכלל בתוך הexception handler שלה. לא ניתן להכליל גם את ה exception handler של g וגם את של h בתוך f מכיוון שייתכן והם סותרים אחד את השני.

(2) תיתכן חפיפה בשמות המשתנים בין scopeים שונים ומתודות שונות. במקרה כזה אם נעתיק את הhandler אנו עלולים לקבל טיפול לא נכון בשגיאה מאחר והhandler ייש למשתנה הלא נכון. באופן דומה, יתכן שהhandler יפנה למשתנה שאינו נגיש בscope הנוכחי.

(3) אם exception handler שקודם לexception handler הנוכחי מטפל באותו סוג שגיאה, אם בפתרון של טימי פשוט מעתיקים את כל הexception handlers אזי השגיאה שתיזרק תטופל פעמיים.
נשים לב שנעתיק את הhandlers באותו סדר שבו הם היו מטופלים בפתרון המקורי. זה בא לידי ביטוי בדוגמה בתרגיל בכך בהhandler של 10 מופיע לפני הhandler של שורה 14 בקוד החדש שנוצר.
(4) בגלל שהעתקנו את הhandler, אם הטיפול נועד לאפשר המשך ריצה תקין, אנחנו נמשיך את הריצה ממקום לא נכון. למשל אם הhandler של שורה 14 נכתב כך שאחריו אמורים להמשיך לרוץ, אזי בקוד המתוקן המשך הריצה יהיה משורה 18' במקום 24'.

פתרונות נוספים שהתקבל באופן חלקי:

(5) בגלל ביצוע הabort, לא נגיע אף פעם לdefault exception handler. ה default exception handler קודם לכל exception handler אחר בתכנית. לפי ההצעה של טימי, נכליל את כל הexception handler שקדמו. השימוש בabort נועד לעצור את החיפוש מאחר וה exception handler החדש שניצור מכסה את כל הexception handlers האפשריים.

פתרונות שלא התקבלו:

(6) אם הexception handler עצמו זורק חריגה אז היא לא תיתפס. פתרון זה לא נכון כי אופן הטיפול בחריגות אומר שאם הexception handler זורק חריגה אזי היא תיתפס על ידי הexception handler הקודם במחסנית ותטופל שם באופן נכון כמו בקוד המקורי.

(4 נק') ברצוננו "להרוויח" כמה שיותר מההצעה של טימי. הציעו תיקון פשוט ככל האפשר להצעה של טימי שיפתור את הבעיה הקיימת בהצעה הנוכחית.
שימו לב: יתכן והפתרון שתציעו לא יחסוך לגמרי את החיפוש בין רשומות ההפעלה (כפי שצויין בתחילת השאלה), אך עליו לחסוך כמה שיותר.

פתרונות מקובלים בהתאם לבעיה שמצאתם בסעיף א:

(1) על מנת להרוויח כמה שיותר מהפתרון של טימי, ניישם אותו רק כאשר הexception handler הקודם ידוע. למשל במתודה f שבדוגמה קיימים nested exception handlers ולכן הexception handler שקודם לexception handler הפנימי הוא בהכרח ידוע (והוא ה exception handler החיצוני), לכן נוכל להפעיל את הפתרון של טימי.
במקרה של קריאה למתודה ממספר מתודות שונות שלכל אחת מהן exception handler משלה (כפי שתואר בסעיף א), לא נפעיל את הפתרון של טימי.
נוכל להריץ אנליזה על גרף הקריאות של התכנית (גרף הקריאות מכיל את כל המתודות בתכנית וקשת ממתודה א למתודה ב מייצגת במתודה א קוראת למתודה ב) כדי לקבוע האם הexception handler הקודם הוא יחיד וידוע (גם במקרה של קריאה למתודה ממספר מתודות שונות). אם האנליזה תקבע שהוא יחיד וידוע, נוכל להפעיל את הפתרון של טימי.
בכל מקרה בו לא ניתן להשתמש בפתרון של טימי, נאלץ לבצע חיפוש ברשומות ההפעלה.
(2) נבדוק לאיזה משתנים הhandler ניגש. נעתיק את הhandler אך ורק אם הוא ניגש רק למשתנים שנגישים בscope הנוכחי ושלא קיימת כפילות בשמם (כלומר שהם לא מוגדרים גם ב scope הנוכחי וגם בscope הקודם).
(3) נזכור באיזה טיפוסי חריגות כבר טיפלנו בhandler הנוכחי ונעתיק רק handlers שמטפלים בסוגי חריגות שלא טופלו עדיין.
(4) נוסיף בסוף כל handler מועתק פקודת goto אל המיקום הנכון ממנו אמורה להמשיך הריצה.

חלק ב (15 נקודות) – חלוקת רגיסטרים לCallee-saved וCaller-saved

נתון הקטע קוד הבא:

```
int f(int x, int y) {
        int a,b,c;
        a = x + 2;
        b = a * y;
        c = y – 3;
        return a * b * c;
}
int g() {
        int x,y,z,w;
        x = 13;
        y = 42;
        z = f(x,y);
        w = f(y,x);
        y = f(w,x);
        x = f(y,z);
        return x + y;
}
```

הקומפיילר החליט להקצות רגיסטרים למשתנים באופן הבא:

- x -> r0
- y -> r1
- z -> r2
- w -> r3
- a -> r0
- b -> r1
- c -> r4
- f.x -> r5
- f.y -> r3

1. (9 נק') הציעו חלוקה של הרגיסטרים לcaller-saved וcallee-saved שתצמצם למינימום את מספר הכתיבות והקריאות של משתנים למחסנית בסך כל הקריאות למתודה f מתוך g.

לצורך פתרון השאלה נבחן מי משתמש באיזה רגיסטר ומתי.
כמו כן, נבדוק מתי כל אחד מהרגיסטרים חי (באמצעות אנליזת חיות).
שימו לב שלכל רגיסטר צריך להיות מישהו שאחראי לו (או הcaller או הcallee), גם אם מעשית רגיסטר זה לא יגובה. האחראי לרגיסטר הוא זה שמחליט לא לגבות את הרגיסטר.
נבחין כי אם רגיסטר נמצא רק בשימוש המתודה הקוראת, אזי נעדיף שהוא יהיה callee-saved מכיוון שהמתודה הנקראת תדע שהיא לא דורסת אותו ולכן תדע שאין צורך לגבותו. באופן דומה, אם רגיסטר נמצא רק בשימוש המתודה הנקראת, נעדיף שהוא יהיה caller-saved.

- r2 רק בשימוש בg ולכן הוא יהיה callee-saved.
- r4 וr5 רק בשימוש בf ולכן הם יהיו caller-saved.
- r3 בשימוש בשתי המתודות אך מכיוון ואף פעם אין צורך לשמר את ערכו (כי הוא אף פעם לא חי לאורך קריאה לf) הוא יהיה caller-saved.
- r0 וr1 בשימוש בשתי המתודות וב2 מתוך 4 קריאות יש צורך לשמר את ערכם (הקריאות בשורות 12 ו13 עבור r0 והקריאות בשורות 12 ו15 עבור r1); אם הם יהיו callee-saved נצטרך לגבותם בכל הקריאות, לכן הם יהיו caller-saved (כדי שנוכל לחסוך את הגיבוי ב2 קריאות).

2. (6 נק') באפשרותכם להחליף את הרגיסטר המוקצה לאחד המשתנים עם רגיסטר אחר מבין הרגיסטרים הקיימים. הציעו החלפה שתצמצם עד כמה שניתן את מספר הכתיבות והקריאות של משתנים למחסנית בקריאות למתודה f.

נחליף את הרגיסטר של a מr0 לr5. נשים לב שf.x לא חי לאחר ההגדרה של a, לכן לא נצטרך לגבות את ערכו.
כמו כן, כעת r0 נמצא רק בשימוש הcaller ולכן נוכל להחליט שהוא callee-save ולא לגבות אותו. ההחלפה הזאת חסכה לנו שני גיבויים של r0 וזהו החסכון המקסימלי שניתן להגיע אליו.
לחילופין, ניתן להחליף את הרגיסטר של b מr1 לr5 ומאותה סיבה נחסוך שני גיבויים.

פתרון נוסף שהתקבל באופן חלקי הוא להחליף את x מr0 לr5. אנליזה על f תקבע שהיא לעולם לא כותבת אליו ולכן לא משנה את ערכו של הרגיסטר r5. הבעיה בפתרון זה היא שהוא מניח שהארגומנט מועבר באמצעות רגיסטר. אם הארגומנט מועבר דרך המחסנית אז מעשית כן ביצענו כתיבה לr5 ולכן נצטרך לגבות את ערכו לפני הקריאה.
הצעות להחליף את הרגיסטר של y מr1 לr3 קיבלו ניקוד דומה (החלפה זו לא מתנגשת עם המשתנה w מאחר והם לא חיים באותו זמן).
שימו לב: גם אם רגיסטר (r5 במקרה הזה) מועבר דרך המחסנית הוא בכל מקרה יצטרך להיקרא לרגיסטר לאחר מכן לצורך העבודה איתו.