עבודת גמר

לקבלת תואר טכנאי תוכנה

הנושא: **קומפיילר**

המגיש : איתן רפאל צ'רטוף

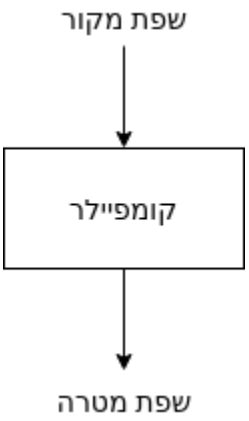
ת.ז. המגיש: 215310715

שמות המנחים : מיכאל

אפריל 2024 תשפ"ד

תקציר

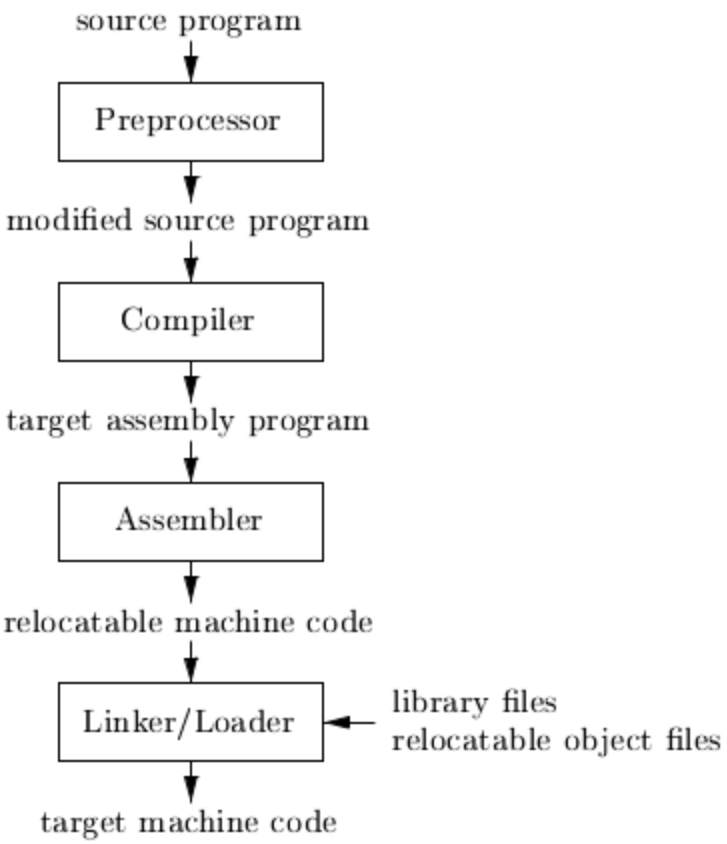
שפת תכנות היא קבוצה של סימונים המשמשת לכתיבת תוכניות מחשב. העולם היום תלוי על שפות תכנות, מכיוון שכל התוכניות הרצות בעולם נכתבו באחת מן כל שפות התכנות. אך, בשביל להריץ תוכנית הכתובה משפת תכנות, צריך לתרגם אותה לצורה שהמחשב יכול להריץ.
המערכות שיכולות לעשות תרגום שכזה נקראות *קומפיילרים* (או בעברית תקינה, *מהדרים*. במהלך כל התיק אני אשתמש במילה קומפיילר). במילים פשוטות, קומפיילר הינו תוכנית המקבלת תוכנית הכתובה בשפת תכנות - *שפת המקור* - ומתרגם את התוכנית הזאת לתוכנית מקבילה לתוכנית המקורית, רק בשפה אחרת - *שפת המטרה*.



עיבוד שפה

כאשר מעבדים שפת תכנות לשפת מכונה, הקומפיילר הוא רק חלק מהתהליך. עוד חלקים מהתהליך הם:

- מעבד מקדים - תכנית הקולטת נתונים מקדימים בשביל שהפלט שלה ישמש בתכנית אחרת. סוג תכנית זו תקרא תמיד לפני תכנית אחרת שתשתמש בפלו תכנית זאת, לכן השם עיבוד מקדים.
- מעבד שפת שף - מתרגם שפת סף לשפת מכונה
- מקשר - תכנית המחברת תוכניות מחשב שעברו הידור לשפת מכונה לתוכנית אחת.



מטרת הקומפיילר בתהליך היא לקחת את הפלט של המעבד המקדים (שלא אמור להיות שונה בצורה גדולה מקוד המקור), ולתרגם אותו לשפת סף.

מושגים

קומפיילר/מהדר

תוכנית המקבלת תוכנית הכתובה בשפת תכנות - *שפת המקור* - ומתרגם את התוכנית הזאת לתוכנית מקבילה לתוכנית המקורית, רק בשפה אחרת - *שפת המטרה*.

מתורגמן/Interpreter

תכנית המבצעת ישירות הוראות שנכתבו בשפות תכנות מבלי לדרוש שהן תורגמו לשפת מכונה. בדרך כלל האינטרפרטר כולל קבוצה של הוראות שאפשר לבצע ורשימה של הוראות אלו לפי הסדר שהתכניתן רצה שההוראות יפעלו.

האינטרפרטר מתרגם ומבצע את התכנית הרצויה שורה אחרי שורה, לכן בדרך כלל האינטרפרטרים יהיו איטיים מקומפיילרים, המתרגמים את כל התכנית.

מושגי תכנות

שפות תכנות

שפת תכנות היא קבוצה של סימונים המשמשת לכתיבת תוכניות מחשב. שפות תכנות נוצרו לראשונה בשביל להקל על בני אדם ליצור תכניות מחשב. אך, בשביל להשתמש בשפות האלה, צריכים תכנית התתרגם את התכנית לשפת מכונה.

כאשר מדברים על שפות תכנות, נהוג לחלק אותם לשני קטגוריות:

- **שפות תכנות עיליות (high level) -** שפת תכנות המיועדת לשימוש ע"י מתכנתים אנושיים. שפות תכנות עיליות משתמשות במבנים תחביריים האלולים להזכיר שפות טבעיות, לכן הן קלות לכתיבה וקריאה ע"י בני אדם. שפות אלה משתמשות בכמות הפשטה גדולה, לא רק בתחביר והסמנטיקה של התכנית, אלה גם במה שקורה ברקע, לדוגמה, שפות מודרניות מנהלות לבד את זיכרון התכנית. שפות תכנות עיליות מודרניות הופכות את תהליך הפיתוח לפשוט ומובן יותר. רמת ההפשטה של השפה מגדירה כמה "עילית" השפה.
- **שפות תכנות נמוכות (low level) -** שפת תכנות המספקת הפשטה מעטה, לכן תהיה משומשת ע"י מכונות ולא ע"י תוכניתנים. שמדברים על מושג ההפשטה בתכנות בקשר לשפות תכנות, בדר"כ מדברים על ההפשטה בין ארכיטקטורת סט ההוראות של המחשב (ISA) לבין השפה. כלומר, מכיוון ששפות סף הם "קרובות" לסט ההוראות של המחשב (מבחינה תחבירית), ההפשטה שלהם היא מעטה ולכן הם low-level, אולם שפה כמו python היא high-level בזכות כמות ההפשטה שהיא מספקת לתכניתן.

משתנים

מקום אחסון בעל שם וסוג ערך שמור. אפשר להתייחס למשתנים בעזרת שמם או כתובת הזיכרון שלהם. בזמן ריצת תכנית מחשב אפשר להכין משתנים, להגדיר להם ערך, לשנות ערך זה, למחוק את המשתנים ועוד.
דוגמאות להגדרת משתנים שישמשו כמידע על בן אדם:

```
// גיל המוגדר כמספר שלם
int age;
// שם המוגדר כמחרוזת של אותיות
string name;
// מספר אהוב המוגדר כשבר
float fav_number;
```

תנאים

הוראה הבודקת תנאי מסוים. תנאים הם דרך לבדוק תנאי מסוים בזמן ריצת התכנית, ואפשר להשתמש במשתנים בתנאים. תנאים בדר"כ כתובים בהוראות if - else, הכוונה היא אם קורה משהו, תעשה משהו, ואם לא תעשה משהו אחר:

```
if (condition):
  statement
else:
  statement
```

לולאות

לולאות משומשות בשביל להריץ חלק של הקוד שוב ושוב עד שתנאי מסוים מתקיים. יש שני סוגים עיקריים של לולאות, ה - for loop וה - while loop. בדר"כ משתמשים בfor שאנחנו יודעים את מספר האיטרציות, ובwhile שאנחנו לא. דוגמא לשני תוכניות המדפיסות 10 כוכביות, אחת לאחר השנייה:

```
int i;
for(i = 0; i < 10; ++i) {
  printf("*");
}

int i = 0;
while(i < 10) {
  printf("*");
  ++i;
}
```

תיאור הנושא

מכיוון שמטרת הקומפיילר הינה לתרגם שפת תכנות לשפת מכונה, הקומפיילר צריך לדעת איזה שפה הוא מתרגם, בשביל שיוכל לעבוד אליו. השפה שאנחנו נתרגם היא שפת *Quest*, שפה חדשה שנוצרה במיוחד לפרויקט הזה. השפה היא Turing Complete, כלומר השפה בעלת יכולת לדמות מכונת טיורינג, ובעל משתנים, לולאות, תנאים ועוד...

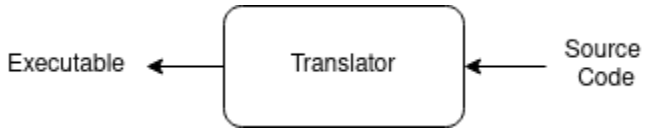
תכולת השפה

רקע תיאורטי

תוכניות תרגום

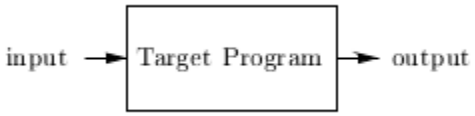
המחשב לא יכולה לקרוא שפה מדוברת ישירות, אפילו לא שפות תכנות, המחשב מבין בינארית. תוכניות תרגום, או מתרגמים, הם תוכניות המתרגמות שפה אחת לשפה אחרת. בדר"כ משתמשים בתוכניות אלה בשביל לתרגם שפה מובנת לבני אדם, כלומר שפה עילית, לשפה מופשטת יותר, כמו שפת סף או בינארית, כלומר שפה תחתונה.

מתרגמים יכולים לתרגם תוכנית הכתובה בשפה עילית לתוכניות שאפשר להריץ על מכונה, ואותה תוכנית יכולה לקבל קלט, לעבד אותו, ולהוציא פלט אחר.

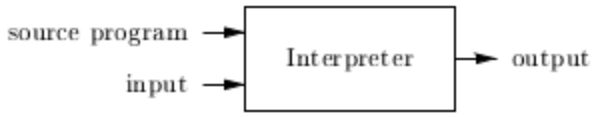


יש סוגים שונים של מתרגמים, כמו *קומפיילרים*, *אינטרפרטים* ו*אסמבלרים*, אך מתרגם יכול להיות כל תוכנית העומדת בתנאים, לא רק שלושת סוגי המתרגמים האלה.

הקומפיילר הינה תוכנית המתרגמת תוכנית משפה אחת לשפה אחרת, ובנוסף למצוא ולהתמודד עם שגיאות כאשר נמצאו. מה שמבדיל את הקומפיילר מסוגים אחרים של מתרגמים היא העובדה שהוא מעבד את כל התוכנית פעם אחת, ובדר"כ מוציא קובץ המתורגם לשפת המטרה.
עבודת הקומפיילר לתרגם תוכנית שנכתבה בשפת המקור ולתרגם אותה לשפת המטרה, ובנוסף למצוא ולהתמודד עם שגיאות מתי שאפשר. נשמע קל נכון? תגלו בהמשך את התשובה לבד...



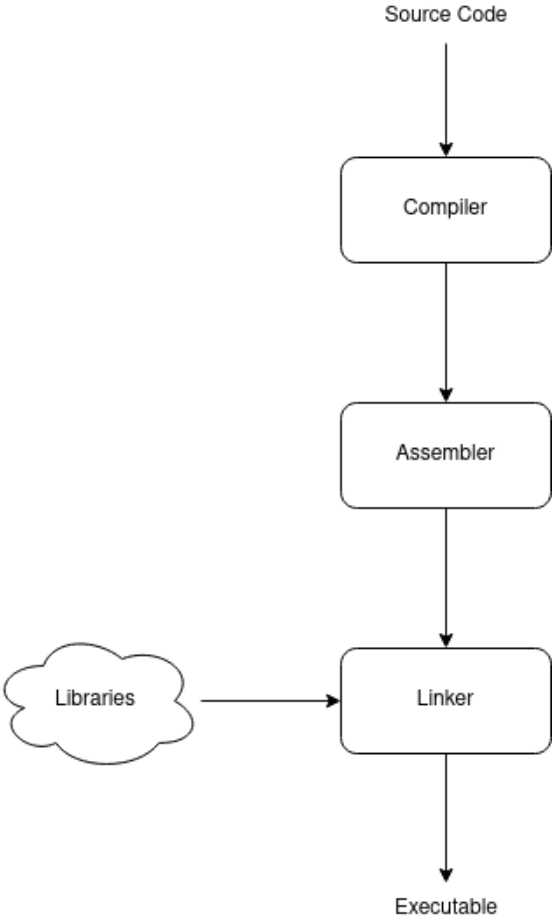
האינטרפרטר הוא עוד סוג של מתרגם. במקום להפיק תכנית כסוג של תרגום, האינטרפרטר נראה כיאלו הוא מריץ כל הוראה בקוד המקור אחד אחרי השני, בהתייחס לפלט. כלומר הוא מריץ את הפקודות אחד אחרי השני בלי קומפילציה של התוכנית.



ההבדל בניהם בא לידי ביטוי בתהליך התרגום. הקומפיילר עובד לפי העקרון "הכל או כלום", כלומר הוא מקמפל את התכנית, וכאשר רואה שגיאה שאי אפשר להתמודד איתה, הוא נעצר ומתאר את השגיאה. אולם האינטרפרט מריץ את התוכנית בצורה המתאימה עד שמבחין בשגיאה, ולאחר מכן מתאר את השגיאה שהגיע אליה. בנוסף אפשר להתייחס לתכנית עצמה. אם שפת המטרה הייתה שפת מכונה, בדר"כ התכנית שתרגם הקומפיילר תהיה יותר מהירה מהתכנית של האינטרפרטר. למרות זאת, אינטרפרטים טובים יותר במציאה וטיפול בשגיאות, מכיוון שהם מריצים את התוכנית הוראה אחת אחרי השנייה.

איך שפה מתורגמת לשפת מכונה

שאנחנו מתרגמים שפה לשפת מכונה בעזרת קומפיילר, הקומפיילר הוא לא השלב היחיד בתהליך התרגום. בדר"כ, קומפיילר מתרגמים את שפת המקור לשפת סף, ומשם ממשיך התהליך. התהליך השלם מתואר בתרשים הבא:



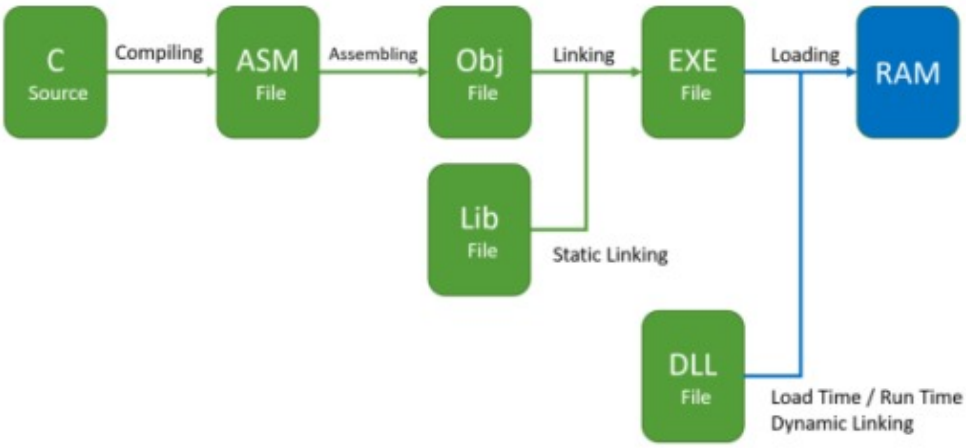
- הערה: בקומפיילרים מודרניים העיבוד המקדים הוא חלק מהקופיילר, לכן התהליך לא נראה כאן.

- מעבד שפת שף (assembler) - מתרגם שפת סף לשפת מכונה
- מקשר (linker) - תכנית המחברת תוכניות מחשב שעברו הידור לשפת מכונה לתוכנית אחת.
- ספריות (libraries) - חלק תוכנה read only שמוסף לתוכנית

התהליך הולך כך:

1. הקומפיילר מתרגם את השפה לשפת סף
2. שפת הסף מתורגמת על ידי האסמבלר
3. המקשר מקשר בין כל הספריות והתוכניות
4. מתקבל קובץ הרצה

שפת מכונה היא תלויה בהרבה דברים, כמו מערכת הפעלה ומשאבי המחשב. למשל, אם היינו רוצים להריץ תוכנית במערכת ההפעלה windows, היינו צריכים להפיק קובץ הרצה בפורמט exe. הנה תרשים המתאר איך תכנית בשפת c יהיה מתורגם לתכנית בwindows:



מבנה של קומפיילר

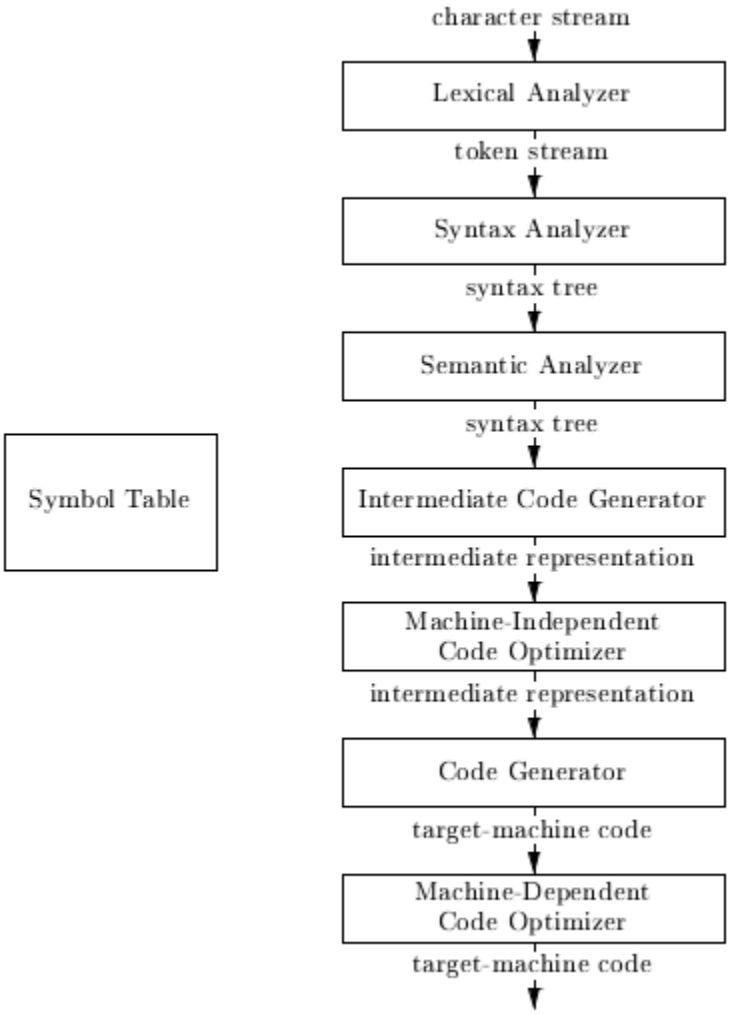
עד כו התייחסנו אל הקומפיילר כקופסה שחורה הממירה תכנית מקור לתוכנית מטרה השקולה לה. אם נפתח קצת את הקופסא הזאת נראה שישנם שני חלקים עיקריים: *אנליזה (analysis)* ו*סינתזה (synthesis)*.

בחלק האנליזה התכנית מפרקת את תכנית המקור למבנים מתאימים וכופה אליהם חוקים מילוניים, תחביריים וסמנטים. בנוסף חלק האנליזה הופך את תוכנית המקור למבנה ביניים המתאר את התכנית ברמה קרובה יותר לתכנית היעד. אם התכנית שמה לב בשגיאות מילוניות, תחביריות וסמנטיות, אליה להגיב ולהתמודד אם אותם שגיאות, ובמידת הצורך גם לתאר אותם למשתמש. בנוסף, התכנית אוספת מידע על תכנית המקור ושמה אותו במבנה שנקרא l*טבלת הסימנים*. לאחר מכאן, טבלת הסימנים ומבנה הביניים נעברים על חלק הסינתזה.

בחלק הסינתזה התכנית יוצרת את תכנית המטרה בעזרת מבנה הביניים וטבלת הסימנים.

בדר"כ, חלק האנליזה נקרא ה-front end, וחלק הסינתזה נקרא ה-back end.

כאשר נפתח את הקופסה השחורה עוד יותר ונראה את הפרטים הקטנים, נבחין שהתכנית עובדת כסדרה של שלבים, כאשר כל אחד משנה את מבנה התכנית למבנה אחר. התרשים הבא מתאר את כל אחד מהשלבים האלה, את הפלט שלהם ואת הקלט שלהם:



- הערה: אופטימיזציה היא תהליך אופציונלי, לכן תרשימים שונים יכולים להראות רק את אחד משני תהליכי האופטימיזציה, או אף אחד מהם.

ניתוח מילוני (lexical analysis)

השלב הראשון של התכנית נקרא *המנתח המילוני* או *הלקסר (lexer)*. הלקסר קורא את זרם התווים המהווים את תוכנית המקור, ומקבץ תווים למחרוזות בעל ערך בשם *lexemes*. בשביל כל אחד מה-lexemes הלקסר יוצר *אסימון (token)* במבנה הבא:

$$< name, attribute/value >$$

הלקסר יוצר זרם של אותם אסימונים ומעביר אותם לשלב הבא.
בתוך האסימון יש שני ערכים, שם האסימון והערך שלו. שם האסימון הוא סמל מופשט המתאר את סוג האסימון, וערך האסימון הוא ערך שימושי המאוחסן בתוך האסימון. לדוגמה, יכול להיות לנו את מחרוזת התווים הבאה:

Position = initial + rate * 60

תווים יכולים להיות מתאימים לאסימונים באופן הבא:

- Position יהיה ממופה לאסימון <id, 1> כאשר id הוא סמל מופשט המתאר שם של משתנה, ו-Position הוא שם הערך
- = יהיה ממופה לאסימון <=>
- Initial יהיה ממופה לאסימון <id, 2>
- + יהיה ממופה לאסימון <+>
- Rate יהיה ממופה לאסימון <id, 3>
- * יהיה ממופה לאסימון <*>
- 60 יהיה ממופה לאסימון <60>

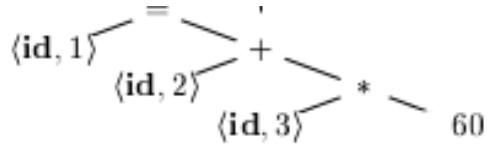
ולכן, זרם האסימון יהיה:

$$\langle \mathbf{id}, 1\rangle \ \langle = \rangle \ \langle \mathbf{id}, 2\rangle \ \langle + \rangle \ \langle \mathbf{id}, 3\rangle \ \langle * \rangle \ \langle 60\rangle$$

ניתוח תחביר (syntax analysis/parsing)

השלב השני של התכנית נקרא *המנתח המילוני* או *הפרסר (parser)*. הפרסר משתמש בזרם האסימונים מהשלב הראשון בשביל ליצור עץ תחבירי המתאר את המבנה התחבירי של זרם האסימונים. בעץ תחבירי כל צומת מתארת תהליך מסוים, והילדים של כל צומת מתארים את סימני התהליך.

בהתאמה לדוגמה מהשלב הראשון, העץ התחבירי שיופק מזרם האסימונים יכול להיות:



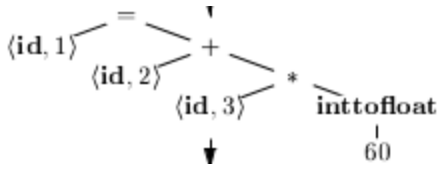
כאשר שורש העץ (=) מתאר את תהליך אחסון הערך של הבן הימני שלו (העץ ששורשו +) לתוך הבן השמאלי שלו, + מתאר את תהליך החיבור בין הבן השמאלי לבן הימני, ו-* מתאר את תהליך הכפל בין הבן השמאלי והבן הימני.

חוקי המנתח התחבירי מוגדרים בתכנית בקוד, אך יש מסמכים כמו מסמכי BNF המתארים את תחביר השפה. קוד הקומפיילר אעקוב אחרי מסמך הBNF בשביל מקור חוקי התחביר שלו.

ניתוח סמנטי (semantic analysis)

השלב השלישי של התכנית נקרא *המנתח הסמנטי*. הוא משתמש בעץ התחביר ההופק בשלב הקודם בשביל לבדוק עקביות סמנטית בתכנית המקור בעזרת חוקים סמנטים המוגדרים בקוד. בנוסף, הוא אוסף נתונים על התכנית בעזרת העץ התחבירי ושומר אותם או בטבלת הסימנים או בעץ סמנטי.

לדוגמה, לפי העץ התחבירי מהשלב הקודם, העץ הסמנטי יכול להיות:



כאשר השינוי היחיד הוא הגדרת שינוי סוג המספר 60, שהיה מסוג int אך שונה לfloat, מכיוון שהצומת משמאלו היא מסוג float. בנוסף, טבלת הסימנים הייתה נראת כך:

1	position	···
2	initial	···
3	rate	···

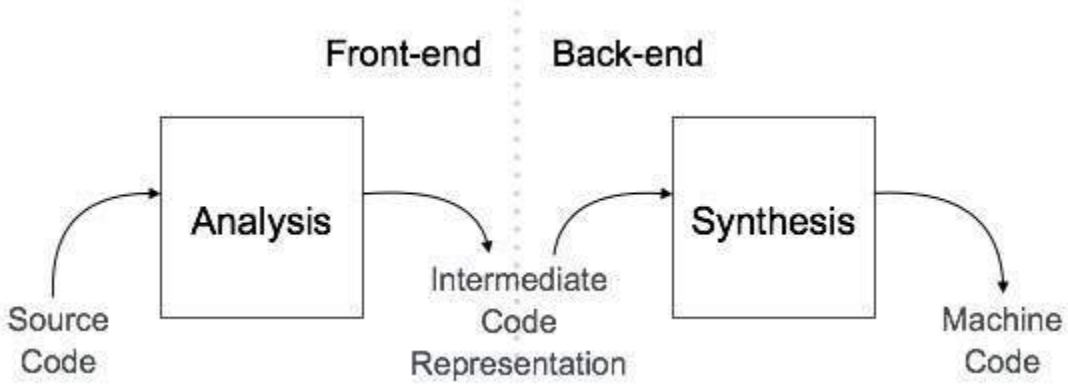
כאשר כל אחד מהתאים בטבלה מתאים למשתנה בתכנית.

חלק חשוב בשלב הוא בדיקת סוגי משתנים, כאשר הקומפיילר בודק שלכל operator יש operands מתאימים. לדוגמה, להרבה שפות יש חוק המגדיר ש-index של מערך חייב להיות מסוג שלם, ואם הוא לא מסוג שלם, הקומפיילר מדווח על זאת. לפעמים הקומפיילר יכול להתמודד סוגי משתנים לא מתאימים, כפי שנראה בדוגמה הקודמת.

יצירת קוד ביניים (intermediate code generation)

השלב הרביעי של התכנית הוא יצירת קוד הביניים, בוא הקומפיילר אבנה אחד או יותר מבני קוד ביניים, שיכולים להיות מבנים שונים המגשימים מטרות שונות. דוגמה למבנה אחד של קוד ביניים הוא העץ הסמנטי.

קוד הביניים הינו מבנה המתאר את את תכנית המקור ברמה נמוכה, כמו תכנית מופשטת של התכנית המקורית. המטרה העיקרית של קוד הביניים להיות קל ליצור ולתרגם, ולהשאיר את התכנית המקורית באותה מבנה שהייתה. לכן, מבנים שונים מתאימים לקומפיילרים שונים.



אופטימיזציה (code optimization)

בשלב החמישי של התכנית, אופטימיזציה, התכנית מנסה לשפר את קוד הביניים כך שיהיה קוד מטרה טוב יותר מתרגום ישיר של קוד המקור, אך עם השארת הלאגוריתם כפי שהיא. בדר"כ כאשר מדברים על אופטימיזציה מתכוונים לשיפור התכנית ביחס למהירות, אך גם משתנים אחרים יכולים להכנס כמו שימוש באחסון או שימוש במשאבים.

למרות זאת, שלב האופטימיזציה הוא שלב אופציונלי לקומפיילר.

יצירת קוד (code generation)

בשלב השישי והאחרון של הקומפיילר התכנית צריכה ליצור את קוד המטרה בעזרת קוד הביניים. יצירת הקוד יכולה להיות תרגום ישיר של קוד הביניים, או להשתנות בהתאם ליחסים שונים של התכנית.

אם שפת המטרה היא שפת מכונה, רגיסטרים וזיכרון צריכים להיות מתאימים להוראות בשפה. ואז קוד הביניים מתורגם למחרוזת הוראות המתארות את אותו אלגוריתם שהיה מתואר בקוד המקור.
לדוגמה, אם שפת המטרה היא שפת סף, קוד המטרה יכול להיראות כך:

```
LDF  R2,  id3
MULF R2,  R2, #60.0
LDF  R1,  id2
ADDF R1,  R1, R2
STF  id1, R1
```

טבלת סימנים (symbol table)

טבלת סימונים (Symbol Table) משמשת ככלי חיוני לניהול מידע אודות המשתנים, הפונקציות, הנהלים והטיפוסים המוגדרים בקוד המקור.

טבלת הסימונים היא מבנה נתונים דינמי המאחסן מידע מפורט על כל אובייקט סימנטי בקוד המקור, כולל:

- **שם האובייקט:** שם ייחודי המזהה את האובייקט בקוד המקור.
- **סוג האובייקט:** משתנה, פונקציה, נתונים, טיפוס מוגדר על ידי המשתמש, ועוד.
- **ערך האובייקט:** ערך התחלתי (במידה וקיים) או טווח ערכים אפשריים.
- **מיקום האובייקט:** מיקום האובייקט בזיכרון (במידה וקיים) או מידע אחר הקשור למיקומו.
- **מידע נוסף:** מידע רלוונטי נוסף, כגון סוגי נתונים של פרמטרים של פונקציה, תכונות של משתנים, ועוד.

טבלת הסימונים משמשת לאורך כל תהליך הקומפילציה, החל מניתוח תחבירי ועד יצירת קוד מכונה. השימושים העיקריים בטבלת הסימונים כוללים:

- **זיהוי ופתרון שגיאות:** הקומפילטור משתמש בטבלת הסימונים כדי לזהות שגיאות תחביריות וסמנטיות בקוד המקור, כגון שימוש כפול באותו שם משתנה, הצהרה על משתנה שאינו מוגדר, או קריאת פונקציה שאינה קיימת.
- **בניית קוד ביניים:** טבלת הסימונים מספקת מידע חיוני עבור בניית קוד ביניים יעיל, כגון הקצאת זיכרון למשתנים, יצירת קישורים בין פונקציות, וביצוע אופטימיזציות שונות.
- **יצירת קוד מכונה:** טבלת הסימונים משמשת ליצירת קוד מכונה המתאים לקוד המקור, תוך התחשבות בארכיטקטורת המחשב ובמערכת ההפעלה.

טבלת הסימונים היא מרכיב קריטי בקומפילטור, והיא תורמת רבות ליעילות ודיוק תהליך הקומפילציה.

חשוב לציין שקיימות גישות שונות למימוש טבלאות סימונים בקומפילטורים שונים. גישות אלו נבדלות זו מזו במבנה הנתונים, באלגוריתמי הניהול, ובמידת המידע המאוחסן בטבלה.

ניתוח מילוני

הניתוח המילוני הוא השלב הראשון בתהליך בעל שלושת שלבים שהקומפיילר משתמש בשביל להבין את תכנית המקור. המנתח במילוני, או לקסר, מקבל את זרם התווים שמרכיבים את התכנית, מעבד אותם ומוציא זרם של אסימונים המרכיבים את התכנית.

תפקיד המנתח המילוני

למה אנחנו צריכים להפוך את מחרוזת התווים המרכיבה את התכנית לזרם של אסימונים? אותו זרם אסימונים שהלקסר מעבד עוזר לנו להפשיט ולנתח את השפה בשלבי הניתוח הבאים. זרם האסימונים נשלח על המנתח התחבירי, שם הוא משתמש באסימונים המופשטים בשביל ליצור עץ תחבירי, שבדר"כ חלקו בנוי מאותם אסימונים. קומפיילרים מסוימים אוספים מידע על משתנים בניתוח המילוני ושומרים את אותו מידע בטבלת הסימנים.

אסימון, תבנית ולקסמה

- אסימון (token) הוא זוג המורכב משם אסימון וערך מאפיין אופציונלי. שם האסימון הוא סמל מופשט המייצג סוג של יחידה לקסיקלית, לדוגמה מילת מפתח מסוימת, או רצף תווים בקלט המציין מזהה. שמות האסימונים הם סימני הקלט שהמנתח התחבירי מעבד. המאפיין הוא ערך המגדיר את האסימון כאשר לקסמה מתאים ליותר מתבנית אחת, לדוגמה, אם יש לנו אסימון עם השם **number** (כלומר אסימון המתאים למספר) אותו אסימון יכול להתאים לכמה מספרים. לכן, נתאים לו מאפיין שיהיה ערך המספר, וכך נדע מה הערך המספרי של **number**.

- תבנית היא תיאור של הצורה בה עשויים להופיע הלקסמות של אסימון. במקרה של מילת מפתח כאסימון, התבנית היא רק רצף התווים המרכיבים את מילת המפתח. עבור מזהים ואסימונים אחרים, התבנית היא מבנה מורכב יותר שמוצא התאמה על ידי מחרוזות רבות.

- לקסמה היא רצף תווים בתוכנית המקור שתואם לתבנית עבור אסימון ומזוהה על ידי המנתח הלקסיקלי כמקרה ספציפי של אותו אסימון.

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped"

התרשים מלמעלה מתאר דוגמאות לאסימונים.

תהליך הניתוח המילוני

מכיוון שלקסר הוא החלק בקומפיילר שקורא את תכנית המקור, יש לו עוד תת-מטרות שהוא צריך לעשות בשביל להכין זרם נכון של אסימונים. חלק אחד מן תת המטרות האלה יהיה להתעלם מהערות שיכולות להימצא בקוד ורווחים (space, blank, tab, newline וכו..), חלק אחר יהיה לשמור מונה של השורות של התכנית. יש עוד הרבה תת-מטרות כאלה, אך הם תלויים בהגדרת השפה.

הגדרת השפה

להלן הגדרות לשפה פורמלית:

- *א"ב* מסוים הוא כל סט סופי של סימנים. לדוגמה, הסט {0, 1} מגדיר את הא"ב הבינארית. נהוג לסמן את הא"ב בסימן Σ.
- *מחרוזת* מעל א"ב מסוים הינה רצף של סימנים הלקוחים מא"ב מסוים. נהוג לסמן אורך של מחרוזת (מספר הסימנים במחרוזת) s כ-$|s|$. המחרוזת הריקה, ε, היא המחרוזת בעל האורך 0.
- *שפה* היא כל סט של מחרוזות מעל א"ב מסוים. נהוג לסמן שפה באות L, ונהוג להגדיר שפה כך:

$$L = \{s \in \sum^* \mid Condition\}$$

  - כאשר s הינו מחרוזת
  - *Σ הינו אוסף כל המילים מעל הא"ב Σ
  - Condition הינו תנאי מסוים שהמילים בשפה מוגבלים אליו

כמה תהליכים חשובים על שפה מוגדרים בטבלה הבאה:

Operation	Definition and Notation
Union of L and M	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L and M	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \cup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \cup_{i=1}^{\infty} L^i$

- Regular expression הם אנוטציה המגדירה תבניתים של lexemes. בעזרתם נוכל להגדיר את התבניות של אסימונים בשפה. לדוגמה, אם נרצה להגדיר תבנית המתארת את כל האנוטציות האפשריות בשביל שם משתנה ב-C, נגדיר אותם כך (בעזרת Regular Expression):

*letter_(letter_ | digit)**

תהליכים חשובים בשביל regular expression מוגדרים בטבלה הבאה:

LAW	DESCRIPTION
$r\|s = s\|r$	$\|$ is commutative
$r\|(s\|t) = (r\|s)\|t$	$\|$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s\|t) = rs\|rt;\ (s\|t)r = sr\|tr$	Concatenation distributes over $\|$
$\epsilon r = r\epsilon = r$	ϵ is the identity for concatenation
$r^* = (r\|\epsilon)^*$	ϵ is guaranteed in a closure
$r^{**} = r^*$	$*$ is idempotent

- Regular definition הינה אנוטציה הנותנת לתת שמות לRegular expressionים מסוים (אקרה Regular expression מהלך גם בשם Regex ול-Regular definition כ-Regdef). ההגדרה הפורמלית של Regdef הינה:

כאשר Σ הינה א"ב של סימנים, אז Regdef הינה רצף של הגדרות מהמבנה:

$$d_1 \rightarrow r_1$$
$$d_2 \rightarrow r_2$$
$$\dots$$
$$d_n \rightarrow r_n$$

כאשר:

- d_i הינו סימן חדש, לא ב-Σ ולא דומה לכל d אחר
- r_i הינו Regex מעל $\sum \cup \{d_1, d_2, d_3, \dots, d_{i-1}\}$

עכשיו בעזרת Regdef, אנחנו יכולים להגדיר את שמות המשתנים בצורה יותר פורמלית כך:

$$\begin{aligned} letter\_ &\rightarrow \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{\_} \\ digit &\rightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\ id &\rightarrow letter\_\ (\ letter\_ \mid digit\ )^* \end{aligned}$$

להלן הגדרות על Regdef שנשתמש בעתיד:

EXPRESSION	MATCHES	EXAMPLE
c	the one non-operator character c	a
\c	character c literally	*
"s"	string s literally	"**"
.	any character but newline	a.*b
^	beginning of a line	^abc
$	end of a line	abc$
[s]	any one of the characters in string s	[abc]
[^s]	any one character not in string s	[^abc]
r*	zero or more strings matching r	a*
r+	one or more strings matching r	a+
r?	zero or one r	a?
$r\{m,n\}$	between m and n occurrences of r	a{1,5}
$r_1 r_2$	an r_1 followed by an r_2	ab
$r_1 \mid r_2$	an r_1 or an r_2	a|b
(r)	same as r	(a|b)
r_1/r_2	r_1 when followed by r_2	abc/123

בעזרת כלים אלו אנחנו יכולים להגדיר מבנים ללקסימות והאסימנים המתאימים להם. לדוגמה, הנה הגדרה פורמלית לאסימונים בשפה, שיהיו מנותחים ע"י מנתח מילוני:

$$
\begin{array}{rcl}
digit & \rightarrow & [\texttt{0-9}] \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (\texttt{.}\ digits)?\ (\ \texttt{E}\ [\texttt{+-}]?\ digits\ )? \\
letter & \rightarrow & [\texttt{A-Za-z}] \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \texttt{if} \\
then & \rightarrow & \texttt{then} \\
else & \rightarrow & \texttt{else} \\
relop & \rightarrow & \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{=} \mid \texttt{<>}
\end{array}
$$

בנוסף נרצה להגדיר עוד אסימון בשביל למחוק רווחים:

$$ws \rightarrow (\ \textbf{blank} \mid \textbf{tab} \mid \textbf{newline}\ )^{+}$$

להלן טבלה המגדירה lexemes ואת האסימונים שלהם:

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

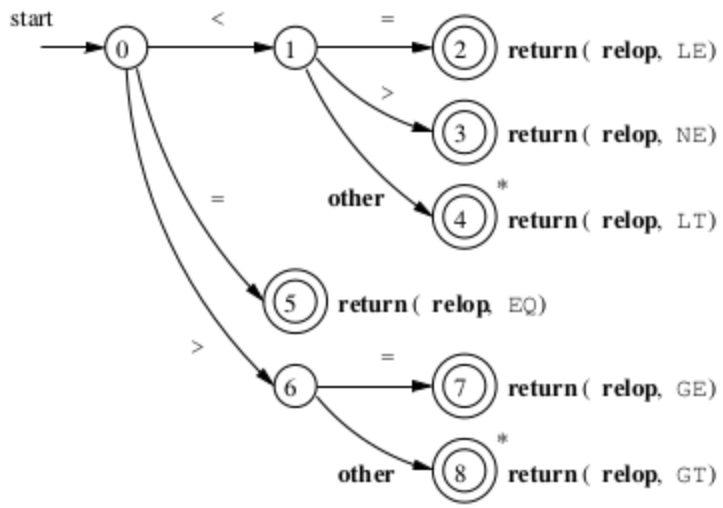
מכונת מצבים

מכונת מצבים (Finite State Machine - FSM) היא מודל חישובי מופשט המשמש לתיאור מערכות שמתנהלות דרך סדרת מצבים.

רכיבים עיקריים של מכונת מצבים:

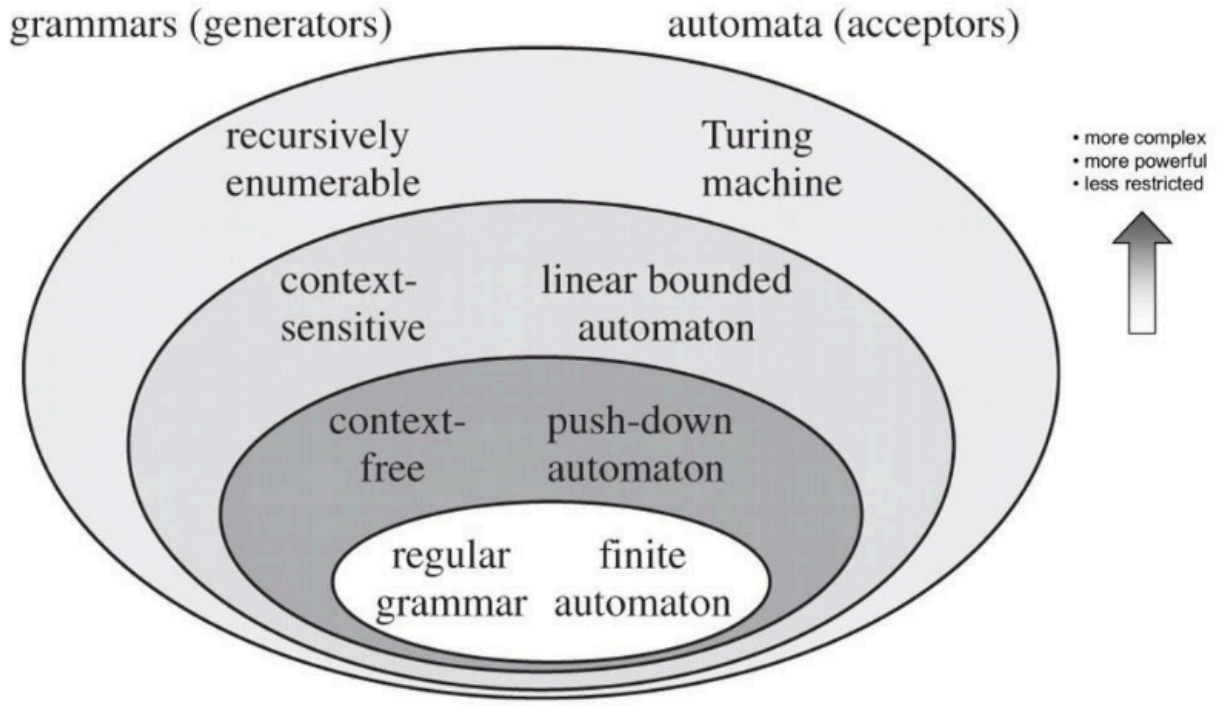
- **מצבים:** קבוצה סופית של מצבים, המייצגים את כל התצורות האפשריות של המערכת. מתואר ע"י עיגול בדר"כ בדיאגרמה, או ע"י האותיות q ו-Q.
- **מצב התחלתי:** מצב מיוחד שבו המערכת מתחילה. מתואר ע"י חץ בלי מצב מצב מתחיל, או ע"י האות q_0
.
- **מצבים מקבלים:** קבוצת מצבים מיוחדים שמסמנים שהמערכת ביצעה את הפעולה בהצלחה. מתוארים בדר"כ ע"י עיגול כפול בדיאגרמה, או ע"י האות F.
- **פונקציות המעברים:** כללים המגדירים כיצד המערכת עוברת ממצב אחד למצב אחר בתגובה לקלט. מתוארים בדר"כ ע"י חץ בעל מצב מתחיל ומקבל בדיאגרמה, או ע"י האות δ.
- **פונקציית פלט:** פונקציה שמספקת פלט עבור כל מצב.

לדוגמה, הנה מכונת מצבים המגדירה את מכונת המצבים לאסימון **relop**, שהוגדר לפני:



סוגי אוטומטים

ישנם כמה סוגים של אוטומטים, כאלה שיכולים לקבל קבוצות שונות של שפות. סוגי האוטומטים מוגדרים ע"פ ה-Chomsky Hierarchy, היררכיה של אוטומטים, כאשר כל אוטומט רחב יותר עוטף את שאר האוטומטים בהיררכיה. היא מגדירה איזה אוטומט "חזק יותר" משאר האוטומטים ואיזה סוגי שפות כל אחד מהאוטמטים האלה יכול להגדיר. ההיררכיה הוגדרה לראשונה ע"י Noam Chomsky, וחברו Marcel-Paul Schützenberger עזר לו להגדיר את ההיררכיה. המאמר בוא הגדירו את ההיררכיה מופיע בביליוגרפיה.



אפשר לראות בדיאגרמה שהאוטומט החלש ביותר, אוטומט סופי, יכול להגדיר רק שפה המוגדרת לפי RegRx, ושהאוטומט החזק ביותר, turing machine, שהינו המודל המרכזי בתיאור אופן פעולת המחשב המודרני, יכול

להגדיר שפות רקורסיביות. למעשה יהיה ניתן לראות שבמהלך תהליך הניתוח, ננתח את השפה בעזרת אוטומטים יותר ויותר מסובכים, עד שנגיע לשפה שהיא turing-complete.

אוטומט סופי דטרמיניסטי/לא דטרמיניסטי

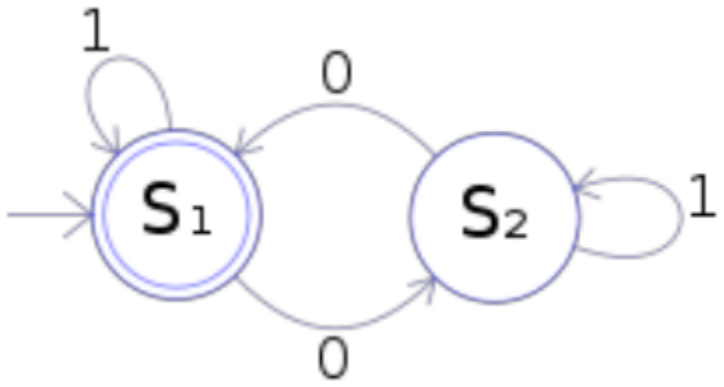
אוטומט סופי דטרמיניסטי ואוטומט סופי לא דטרמיניסטי (בהתאמה DFA ו NFA) הם סוגי אוטומטים סופיים.
שניהם מורכבים מהרכיבים (שהוגדרו לפני זאת):

- מצבים - q
- מצב התחלתי - q_0
- מצבים מקבלים - F
- פונקציות מעבר - δ

האוטומט הסופי יכול להיות מוגדר ע"י RegEx.

האוטומט הסופי הדטרמיניסטי הוא אוטומט שלכל מצב יש פונקצית מעבר עבור כל סימן בא"ב שלו.
האוטומט הסופי הלא דטרמיניסטי הוא אוטומט שלא לכל מצב יש פונקצית מעבר עבור כל סימן בא"ב שלו. כלומר, יש לפחות מצב אחד שאין לו פונקצית מעבר עבור סימן בא"ב.
ההבדל בין האוטומט הסופי דטרמיניסטי והאוטומט הסופי הלא דטרמיניסטי הוא בהגדרתם, כאשר יש לפחות מצב אחד שאין לו פונקציית מעבר עבור סימן מסוים, הוא לא טדרמיניסטי, אך כאשר לכל המצבים יש פונקציות מעבר עבור כל הסימנים, האוטמט הוא דטרמיניסטי.

להלן דוגמה לאוטומסט סופי דטרמיניסטי המגדיר שפה מעל הא"ב {0, 1} המקבל רק מחרוזות עם מספר זוגי של אפסים:



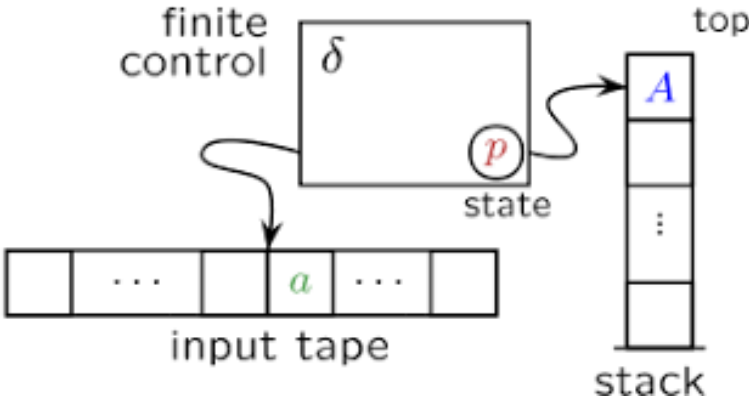
אוטומט מחסנית

אוטומט מחסנית (Push Down Automaton) הוא אוטומט סופי אשר נעזרת במחסנית לביצוע מעברים. פונקצית המעבר לא מוגדרת אך ורק על גבי הסימן המתקבל, אלא גם על ידי מה שנמצא בראש המחסנית. הפעולות היחידות האפשריות על המחסנית הם Push ו-Pop של איברים הנמצאים בא"ב של המחסנית. הרכיבים המגדירים PDA הם:
Q - אוסף סופי של מצבים
Σ - הא"ב של הקלט
Γ - הא"ב של המחסנית

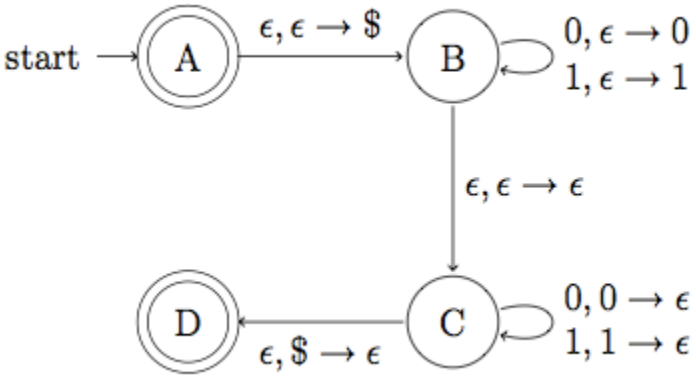
δ - פונקציות המעבר המוגדרות ע"י הקלט והמחסנית
q_0 - המצב ההתחלתי של האוטומט
Z - הסמל ההתחלתי על המחסנית
F - אוסף המצבים המקבלים

אוטומט מחסנית יכול להגדיר שפות חופשיות הקשר.

להלן דיאגרמה המתארת בצורה מופשטת מה זה אוטומט מחסנית:



להלן דיאגרמה המתארת אוטומט מחסנית המקבל פלינדרומים מהא"ב {0, 1}:

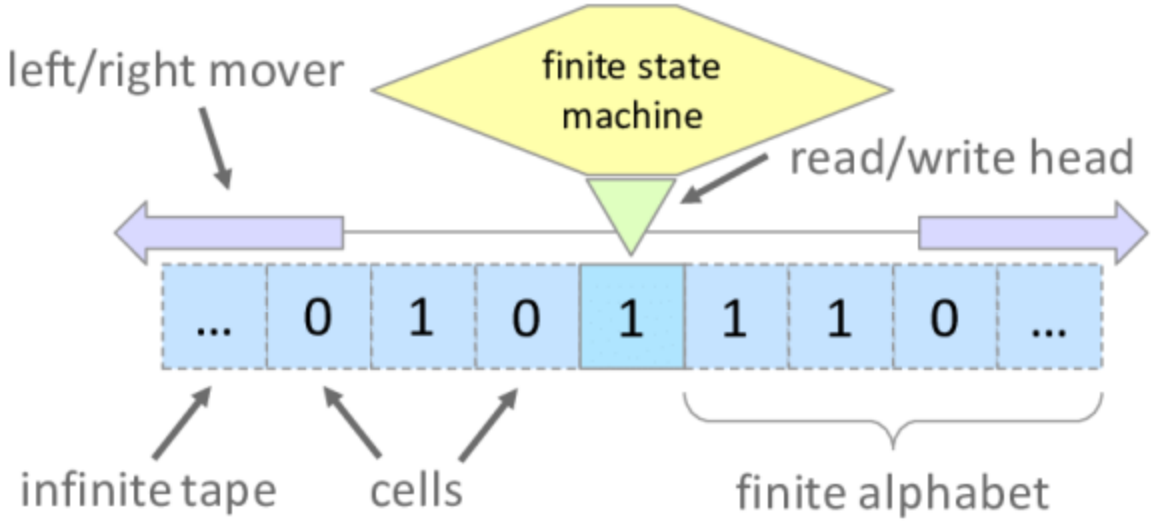


מכונת טיורינג

מכונת טיורינג היא מודל מתמטי המתאר מכונה מופשטת המשתמשת בסימנים על גבי סרט וסט של חוקים בשביל אלגוריתם. למרות פשטות המכונה, אפשר להגדיר העזרתה כל אלגוריתם חישובי שניתן להגדיר במחשבים מודרנים. המודל הומצא ע"י אלן טיורינג בשנת 1936.

מכונת טיורינג נחשבת לאוטומט החזק ביותר, אך היא לא יכולה להגדיר כל שפה שקיימת. הוכחה לזה קיימת בשני מאמרים שיצאו בשנת 1936, אחד ע" אלן טיורינג והשני ע"י אלאנזו צ'רצ', העונים על השאלה החישובית, ה-Entscheidungsproblem.

להלן דיאגרמה אבסטרקטית המתארת מכונת טיורינג:



במהלך הספר נראה איך נוכל לבנות אוטומט סופי דטרמיניסטי שבעזרתו נתרגם את תכנית המקור לזרם של אסימונים.

ניתוח תחבירי

ניתוח סמנטי

יצירת קוד ביניים

אופטימיזציה

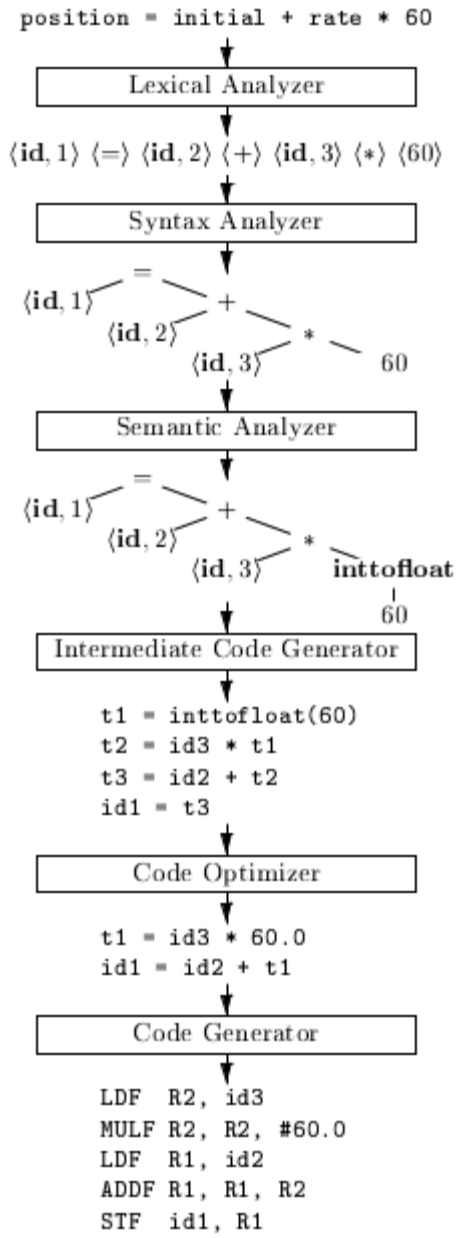
יצירת קוד

תיאור הבעיה האלגוריתמית

מטרת הפרויקט היא ליצור קומפיילר המתרגם שפת מקור לשפת מטרה, לכן הבעיה האלגוריתמית היא לא תרגום שפת מקור לשפת מטרה, אלה הבעיות האלגוריתמיות העולות בבניית קומפיילר, כלומר, הבעיה בכל שלב בקומפילציה.

לשם כך נפריד בין שלבי הקומפילציה ונתאר את הבעיה העולה בכל שלב. בנוסף נתאר את הבעיות העולות בבנייה מבנים חשובים לקומפיילר, כמו הבעיות העולות בבניית ה-symbol table. בכל שלב יהיה מתואר מה הקלט בכל שלב, מה הפלט בכל שלב, וסקירה של הבעיה העולה בבניית אלגוריתם כזה. לא יתוארו כל הפרטים הקטנים במהלך הקומפילציה, אלה רק התהליכים המרכזיים המופשטים.

התרשים הבא מתאר את שלבי הקומפילציה, מקבלת שפת המקור למנתח המילולי עד הוצאת שפת המטרה. חשוב לדעת שאין דרך אחת נכונה לבצע כל שלב ושלב, אך בתיאור הבעיה אתייחס למבנים ולאלגוריתמים שאני בחרתי בכל שלב ושלב הקומפילציה.



ניתוח מילוני

- **קלט:** מחרוזת של אותיות שהינם התוכנית הכתובה בשפת המקור.
- **פלט:** זרם של אסימונים המתאימים לתוכנית.

הבעיה היא, איך ניצור אלגוריתם שיכול לתרגם רצף של תווים המתאימים לשפת המקור לאסימונים? כלומר, כאשר נראה את התו '+' או את המספר 425, איך נוכל להתאים אותם לאסימון? איך נוכל להבדיל בין רצף התווים "+=" לרצף התווים "=="? ובנוסף, איך נדע איזה ערך אנחנו צריכים לייחס לאסימון?

ניתוח תחבירי

- **קלט:** זרם של אסימונים

- **פלט:** עץ המתאר את התוכנית בצורה הגיונית, מסודרת, ונכונה בצורה תחבירית.

שאלת השאלות היא איך ניצור את העץ התחבירי הזה? בשביל לבנות עץ כזה צריך להגדיר איך אסימונים מתחברים ויוצרים תחביר, כלומר את תחביר השפה. לכן האלגוריתם עוקב אחרי חוקי התחביר של השפה ויוצר מזרם האסימונים עץ תחבירי מתאים. לצורך הגדרת תחביר השפה ניצור מסמך בפורמט BNF המתאר את תחביר השפה.

בעיה גדולה בבניית העץ היא לדעת איך להתמודד עם אסימונים במצבים מסוימים. לדוגמה, איך האלגוריתם יודע מה לעשות כאשר האסימונים האחרון שקיבל הוא identifier (אסימון המתאר שם משתנה) (האסימון האחרון מתאר את "המצב" של המנתח) והאסימון הבא הינו plus (אסימון המתאר את הסימן החשבוני פלוס)? איך האלגוריתם ידע שהמצב הנוכחי שלו הוא מקובל תחבירית, ושאין שגיאה תחבירית בתוכנית?

ניתוח סמנטי

- **קלט:** עץ תחבירי
- **פלט:** עץ סמנטי מופשט

שלב הניתוח הסמנטי בקומפילציה הוא תהליך מורכב שמטרתו לוודא שהקוד המקור תקין מבחינה תחבירית וסמנטית. הבעיה האלגוריתמית העיקרית בשלב הניתוח הסמנטי היא זיהוי ופתרון טעויות סמנטיות.

טעויות סמנטיות הן טעויות שאינן קשורות לתחביר הקוד, אלא למשמעותו. לדוגמה, ייתכן שקיים בקוד משפט המנסה לבצע פעולה על סוג נתונים לא חוקי, או שייתכן שקיים בקוד משפט המשתמש במשתנה שאינו מוגדר. חשוב לזהות ולתקן טעויות סמנטיות בשלב הניתוח הסמנטי, מכיוון שהן עלולות להוביל לתוצאות בלתי צפויות ואף לקריסת התוכנה.

בשביל לפענח את הטעויות בקוד נהוג לסרוק את העץ התחבירי וליצור עץ מופשט יותר, המתאר את התכנית בצורה מופשטת אך שם דגש על משמעות התכנית. כלומר האלגוריתם יכול לעשות דברים כמו להסיר את הצמתים בעץ המתארים סוגריים, מכיוון שהם לא תורמים סמנטית לתכנית ולתרגום שלה.

יצירת קוד ביניים

- **קלט:** עץ סמנטי מופשט
- **פלט:** ייצוג של קוד המקור ברמה נמוכה יותר ע"י מבנה מופשט

שלב בניית קוד הביניים בקומפילציה הוא תהליך שבו הקוד המקור של התוכנה מומר לקוד ביניים. קוד ביניים הוא ייצוג של קוד המקור ברמה נמוכה יותר, אך עדיין בר-הבנה על ידי מכונה.

קוד הביניים יכול להיות מיוצג ע"י הרבה מבנים, וקומפיילרים מודרנים בדר"כ ישתמשו ביותר ממבנה אחד בתהליך הקומפילציה. מטרת קוד הביניים היא להיות נאמן לקוד המקור ולתאר כל דבר שימושי לקוד המקור, אך להיות יעיל לעבודה. בחירה של קוד ביניים מתאים דורש הבנה של שפת המקור ושפת המטרה, לכן כאשר נתרגם את השפה לשפה גבוהה, נרצה להשתמש במבנה במרמה גבוהה יותר ממתי שנתרגם את השפה לשפה נמוכה יותר.

אופטימיזציה

- **קלט:** קוד ביניים
- **פלט:** קוד הביניים רק יעיל יותר

שלב האופטימיזציה בקומפילציה הוא תהליך מורכב שמטרתו לשפר את ביצועי קוד התוכנה. הוא עושה זאת על ידי ביצוע שינויים בקוד הביניים, כגון הסרת קוד מיותר, שינוי סדר הפקודות, או שינוי סוגי נתונים.

מצד אחד, קיימות טכניקות אופטימיזציה רבות שיכולות לשפר משמעותית את ביצועי קוד התוכנה. עם זאת, יישום טכניקות אלו עלול להוביל לשינויים משמעותיים בקוד, מה שעלול לגרום לבעיות בתפקוד התוכנה. בנוסף, אופן האופטימיזציה והסוגים השונים של אופטימיזציה תלויים בייצוג קוד הביניים, לכן קומפיילרים לפעמים משתמשים ביותר מייצוג קוד ביניים אחד.

יצירת קוד

- **קלט:** קוד ביניים
- **פלט:** קוד המטרה

שלב יצירת הקוד בקומפילציה הוא תהליך שבו קוד הביניים מומר לקוד המטרה. השלב הזה מתרגם את קוד הביניים לקוד המטרה, ויהיה מתאים למבנים שונים בקוד הביניים.

אם שפת המטרה היא שפת מכונה, רגיסטרים וזיכרון צריכים להיות מתאימים להוראות בשפה. ואז קוד הביניים מתורגם למחרוזת הוראות המתארות את אותו אלגוריתם שהיה מתואר בקוד המקור.

טבלת סימנים

טבלת הנתונים שומרת מסמכים של שמות נתונים ואוספת תכונות על אותם משתנים. לכן הבעיה היא איך המידע והנתונים נאספים ונשמרים. טבלת הסימנים אמורה להיות בנויה כך שהיא מתאימה לנתונים הנאספים ושהמידע יהיה נאסף בצורה מהירה.

התכונות של המשתנים יכולים להיות:

- סוג המשתנה
- איפה נשמר בזיכרון
- ה-scope של משתנים

לכן חלק מהבעיה היא איך נאסף המידע הזה.

התמודדות עם שגיאות

בכל תהליך הקומפילציה צריך להתמודד עם שגיאות, לכן קיים מתמודד השגיאות (error handler). מתמודד השגיאות הוא רכיב בקומפילטור שאחראי לטיפול בשגיאות במהלך הקומפילציה. הבעיה היא להתמודד עם שגיאות בצורה יעילה.

סקירת אלגוריתמים בתחום הבעיה

האלגוריתם הנבחר לפתרון

במהלך הפרק אציין לגבי כל שלב בקומפילציה באיזה מבני נתונים ובאיזה אלגוריתמים השתמשתי בשביל לתרגם את קוד המקור לקוד המטרה. בשביל כל בחירה אציין בנוסף למה בחרתי בא, עם התייחסות למשתנים כמו זמן ריצה, יעילות, פשטות, שימוש חוזר ועוד...

חשוב לדעת שאני לא אציין שני שלבים, יצירת קוד הביניים ואופטימיזציה. לא מוסבר על קוד ביניים מכיוון שבחרתי להשתמש בAST כמבנה קוד הביניים שלי, שמובנה כבר בניתוח הסמנטי. לא הסברתי על אופטימיזציה מכיוון שאין אופטימיזציה לקוד. למרות זאת, קוד המקור מתורגם והתרגום הוא בזמן ריצה O(n).

ניתוח מילוני (lexical analysis)

בשלב זה, המנתח מקבל את התכנית כמחרוזת, והוא צריך להוציא זרם של אסימונים המתאימים למילון השפה.

מכונת המצבים

בשלב הניתוח המילוני החלטתי להשתמש באוטומט סופי דטרמיניסטי (DFA) בשביל לתאם lexemes לאסימונים מתאימים. השכבה הזאת מנתחת את שפה רגולרית, לכן אוטומט סופי מתאים לשלב זה.

כפי שהוסבר בחלק מבני הנתונים (כאן), התאמה של מילה למצב מקבל וכך לסוג אסימון לוקחת זמן ריצה O(1).

להלן כל מילה שיכולה להיות קיימת בשפה, בפורמט הבא:
(<ערך סוג האסימון>) <מספר מצב>:(<מילה/סוג>)

```
(null): 0 (1)
(TOK_IDENTIFIER): 1 (4)
(TOK_NUMBER_CONSTANT): 2 (5)
(CHAR): 3 (11)
(CHAR_DENY): 4 (1)
(STRING): 5 (7)
(TOK_UNKNOWN): 6 (1)
b: 7 (4)
bo: 8 (4)
boo: 9 (4)
bool: 10 (8)
br: 11 (4)
bre: 12 (4)
brea: 13 (4)
break: 14 (9)
```

```
c: 15 (4)
ca: 16 (4)
cas: 17 (4)
case: 18 (10)
ch: 19 (4)
cha: 20 (4)
char: 21 (11)
co: 22 (4)
con: 23 (4)
cons: 24 (4)
const: 25 (12)
cont: 26 (4)
conti: 27 (4)
contin: 28 (4)
continu: 29 (4)
continue: 30 (13)
d: 31 (4)
do: 32 (14)
dou: 33 (4)
doub: 34 (4)
doubl: 35 (4)
double: 36 (15)
e: 37 (4)
el: 38 (4)
els: 39 (4)
else: 40 (16)
f: 41 (4)
fa: 42 (4)
fal: 43 (4)
fals: 44 (4)
false: 45 (17)
fl: 46 (4)
flo: 47 (4)
floa: 48 (4)
float: 49 (18)
fo: 50 (4)
for: 51 (19)
i: 52 (4)
if: 53 (20)
in: 54 (4)
int: 55 (21)
r: 56 (4)
re: 57 (4)
```

```
ret: 58 (22)
s: 59 (4)
sh: 60 (4)
sho: 61 (4)
shor: 62 (4)
short: 63 (23)
si: 64 (4)
sig: 65 (4)
sign: 66 (4)
signe: 67 (4)
signed: 68 (24)
sw: 69 (4)
swi: 70 (4)
swit: 71 (4)
switc: 72 (4)
switch: 73 (25)
t: 74 (4)
tr: 75 (4)
tru: 76 (4)
true: 77 (26)
ty: 78 (4)
typ: 79 (4)
type: 80 (4)
typed: 81 (4)
typede: 82 (4)
typedef: 83 (27)
u: 84 (4)
un: 85 (4)
uns: 86 (4)
unsi: 87 (4)
unsig: 88 (4)
unsign: 89 (4)
unsigne: 90 (4)
unsigned: 91 (28)
v: 92 (4)
vo: 93 (4)
voi: 94 (4)
void: 95 (29)
w: 96 (4)
wh: 97 (4)
whi: 98 (4)
whil: 99 (4)
while: 100 (30)
```

```
p: 101 (4)
pr: 102 (4)
pri: 103 (4)
prin: 104 (4)
print: 105 (31)
.: 106 (46)
,: 107 (44)
~: 108 (126)
!: 109 (33)
&: 110 (38)
|: 111 (124)
^: 112 (94)
*: 113 (42)
/: 114 (47)
+: 115 (43)
-: 116 (45)
%: 117 (37)
>: 118 (62)
<: 119 (60)
=: 120 (61)
..: 121 (1)
...: 122 (62)
->: 123 (63)
&&: 124 (64)
||: 125 (65)
^^: 126 (66)
<<: 127 (67)
>>: 128 (68)
==: 129 (69)
>=: 130 (70)
<=: 131 (71)
+=: 132 (72)
-=: 133 (73)
*=: 134 (74)
/=: 135 (75)
%=: 136 (76)
&=: 137 (77)
|=: 138 (78)
^=: 139 (79)
!=: 140 (80)
~=: 141 (81)
<<=: 142 (82)
>>=: 143 (83)
```

```
(: 144 (40)
): 145 (41)
[: 146 (91)
]: 147 (93)
{: 148 (123)
}: 149 (125)
:: 150 (58)
;: 151 (59)
```

בנוסף, מטריצת הסמיכויות ב-plain text, עם שני השורות הראשונות כמספר השורות ועמודות המטריצה:

```
152
128
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 109 6 -1 -1 117 110 4 144 145 113 115 107 116 106
114 2 2 2 2 2 2 2 2 2 2 150 151 119 120 118 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 146 -1 147 112 1 -1 1 7 15 31 37 41 1 1 52 1 1 1
1 1 1 101 1 56 59 74 84 92 96 1 1 1 148 111 149 108 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 2 2 2
2 2 2 2 2 2 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
4 4 4 4 4 4 4 4 4 4 4 4 4 4
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 5 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 8 1 1 11
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 9 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 10 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 12 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 13 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 14 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 16 1 1 1 1 1 1 19 1 1 1 1 1 1 22 1 1
1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
17 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 18 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 20 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 21
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 23 1 1 1 1
```

```
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
24 26 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 25 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 27 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 28 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 29 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 30 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 32 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 33 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 34 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 35 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 36 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 38 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
39 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 40 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 42 1 1 1 1 1 1 1 1 1 1 46 1 1 50 1 1
1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 43 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
44 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 45 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 47 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 48 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 49 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 51
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 53 1 1 1 1 1 1 1 54 1 1 1
1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 55 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 57 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 58 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 69 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 61 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 62
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 63 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 65 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 66 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 67 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 68 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 70 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 71 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 72 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 73 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 75
1 1 1 1 1 1 78 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 76 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 77 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 79 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 80 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 81 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 82 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 83 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 85 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
86 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 87 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 88 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 89 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 90 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 91 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 93 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 94 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 95 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 97 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 98 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 99 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

```
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 100 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
102 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 103 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 104 1 1 1
1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 105 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 -1 -1 -1 -1 1 -1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 121 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 141 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 140 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 124 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 137 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 138 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 125
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 139 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 126 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 134 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 135 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 132 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 133 123 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 136 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 130 128 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 127 131 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 129 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 122 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 142 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 143 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
-1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1
```

פסאודו קוד המתאר איך lexeme מותאם לאסימון:

```
1. תתחיל ב-q0
2. כל עוד לא הגעתה למצב מקבל ומצב 1-
    2.1. תסתקל על הערך במצב הנוכחי ובעמודה המתאימה לסימן
    2.2. תעבור למצב המתאים לערך
3. החזר אסימון מתאים למצב
```

קבלת כל אסימון והעברתו למנתח התחבירי

זרם האסימונים שמקבל המנתח התחבירי יהיה במבנה הנתונים תור. בחרתי להשתמש בתור מכמה סיבות:

- הכנסה לתור היא O(1)
- תור אשמור על סדר האסימונים

פסאודו קוד המתאר את יצירת זרם האסימונים

```
1. אתחל אסימון בעל ערך NULL
2. כל עוד האסימון לא מסוג EOF
    2.1. תקרא לפונקציית קבלת האסימון הבא ותכניס את הערך המוחזר לאסימון
    2.2. אם האסימון לא ידוע
        2.2.1. תזרוק שגיאה ותפסיק את התוכנית
    2.3. תשמור את האסימון בתור
```

ניתוח תחביר (syntax analysis/parsing)

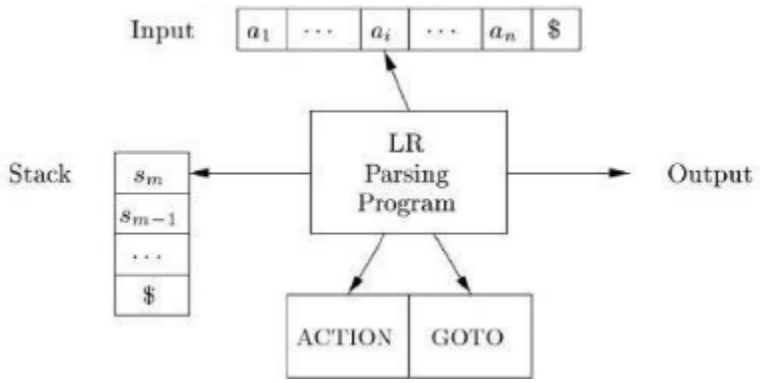
שלב זה מקבל זרם של אסימונים והמנתח המילוני ומוציא עץ תחבירי.

מכונת המצבים

בשלב הניתוח התחבירי החלטתי להשתמש באוטומט מחסנית בשביל להתאים אסימונים לnon-terminal בעזרת חוקים תחביריים. השכבה הזאת מנתחת את שפה חסרת הקשר (CFG), לכן אוטומט מחסנית מתאים לשלב זה. אלגוריתם הפרסור הוא אלגוריתם LR.

אלגוריתם הניתוח התחבירי מורכב מכמה חלקים:

- טבלת הפרסור, המחולקת לשני טבלאות
  - טבלת אקשן (action table)
  - טבלת גוטו (goto table)
- מחסנית
- זרם אסימונים



Action table

טבלת האקשן מתארת פונקציות המעבר שהאוטומט עושה בפורמט הבא:

<action><modifier (if any)>

כל שורה מייצג מצב וכל עמודה מייצג טרמינל מתאים. יש ארבעה סוגים שונים של אקשנים:

- Shift
- Reduce
- Accept
- Error

Goto table

מציין למנתח לאיזה מצב הוא צריך לעבור לאחר ביצוע Reduce.

פסאודו קוד לפרסור LR

1. תן ל-a להיות הטרמינל הראשון, ול-s להיות ראש הסטאק תמיד
2. כל עוד action_table[s, a] לא accept
   - 2.1. אם action_table[s, a] הינו shift t
     - 2.1.1. תדחוף t לסטאק
     - 2.1.2. תן ל-a להיות הטרמינל הבא
   - 2.2. אם action_table[s, a] הינו reduce A -> b

2.2.1. תוציא $|b|$ מהמחסנית
2.2.2. תן ל-t להיות ראש המחסנית
2.3. אם האקשן הינו accept
2.3.1. כלום
2.4. תזרוק שגיאה ותעצור את התכנית

Parser stack

מחסנית הפרסר תשמור בתוכה שלמים המייצגים את המצב הנוכחי. המצב הראשון יהיה '$' המתאר EOF.
אפשר לראות שהאקשן accept מופיע אך ורק פעם אחת בעמודה המתאימה לטרמינל $.

עץ תחבירי

בנוסף לבדיקה של תקינות התוכנית, נרצה לבנות עץ ניתוח שייצג את פעולתה.

בנייתה עץ תעשה תוך כדי תהליך הניתוח, כך שכל פעם שנבצע Reduce נוסיף לעץ את הNon-terminal מצד שמאל של החוק ונקבע שהבנים שלו יהיו כל הTerminalים והnon-terminalים שמימין של חוק הדקדוק, כך לאט-לאט נבנה העץ התחבירי.

תחביר השפה

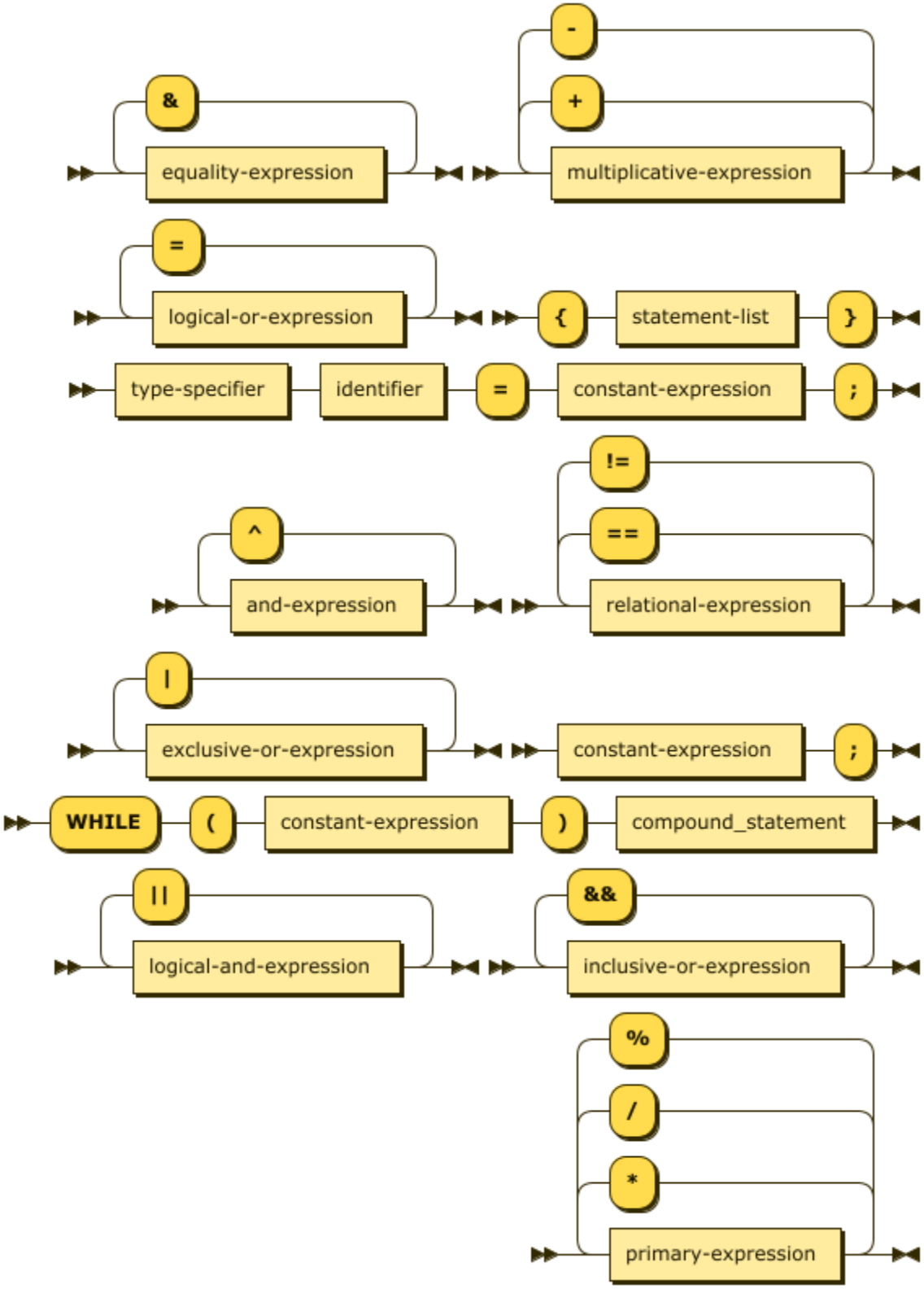
להלן ה-BNF של השפה:

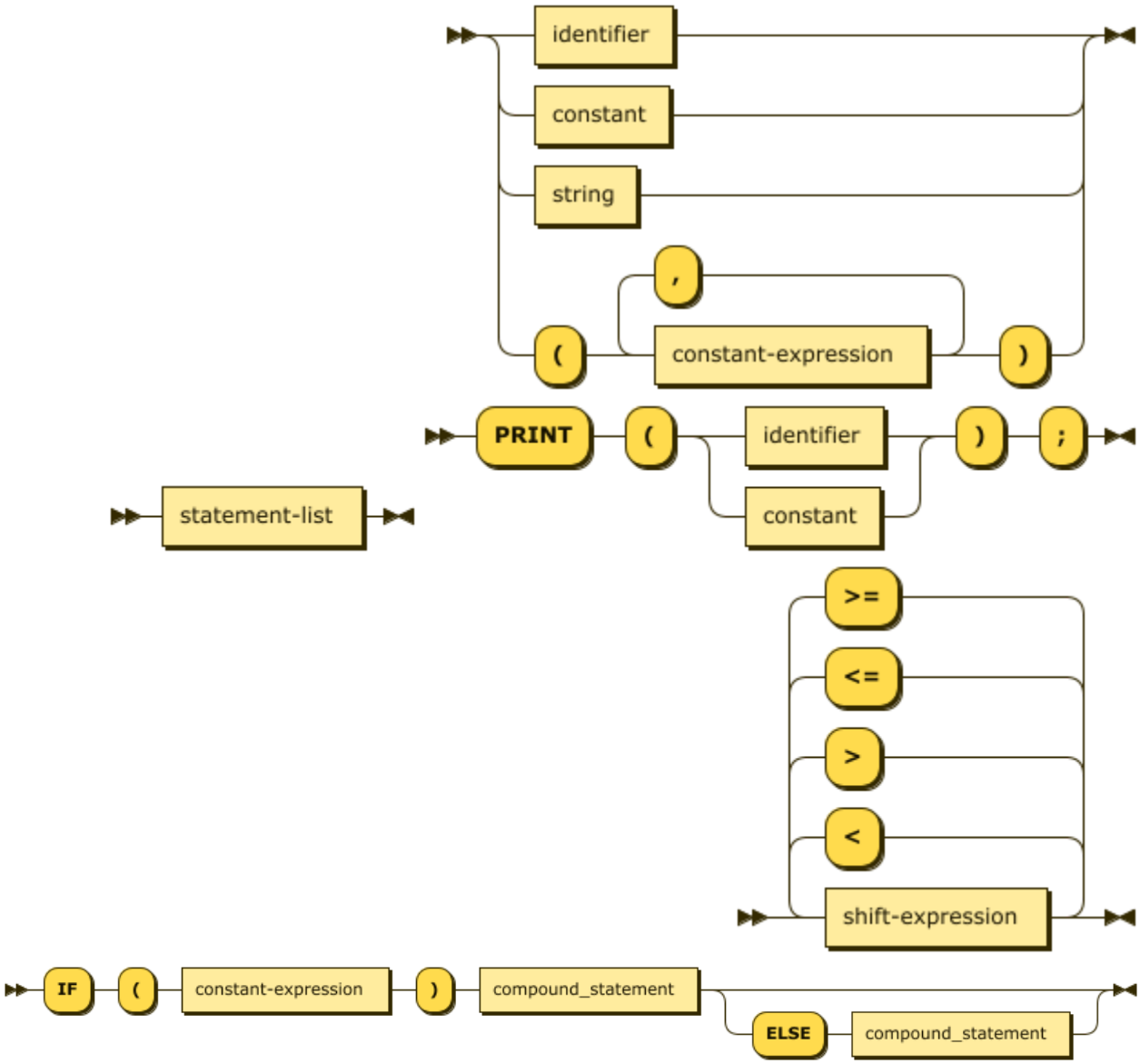
```
program ::= statement-list

statement-list ::= statement
                  | statement statement-list

statement ::= expression-statement
             | compound-statement
             | selection-statement
             | iteration-statement
            | print-statement
             | declaration

declaration ::= type-specifier identifier '=' constant-expression ';'

type-specifier ::= 'CHAR'
                    | 'INT'

expression-statement ::= constant-expression ';'

constant-expression ::= assignment-expression
```

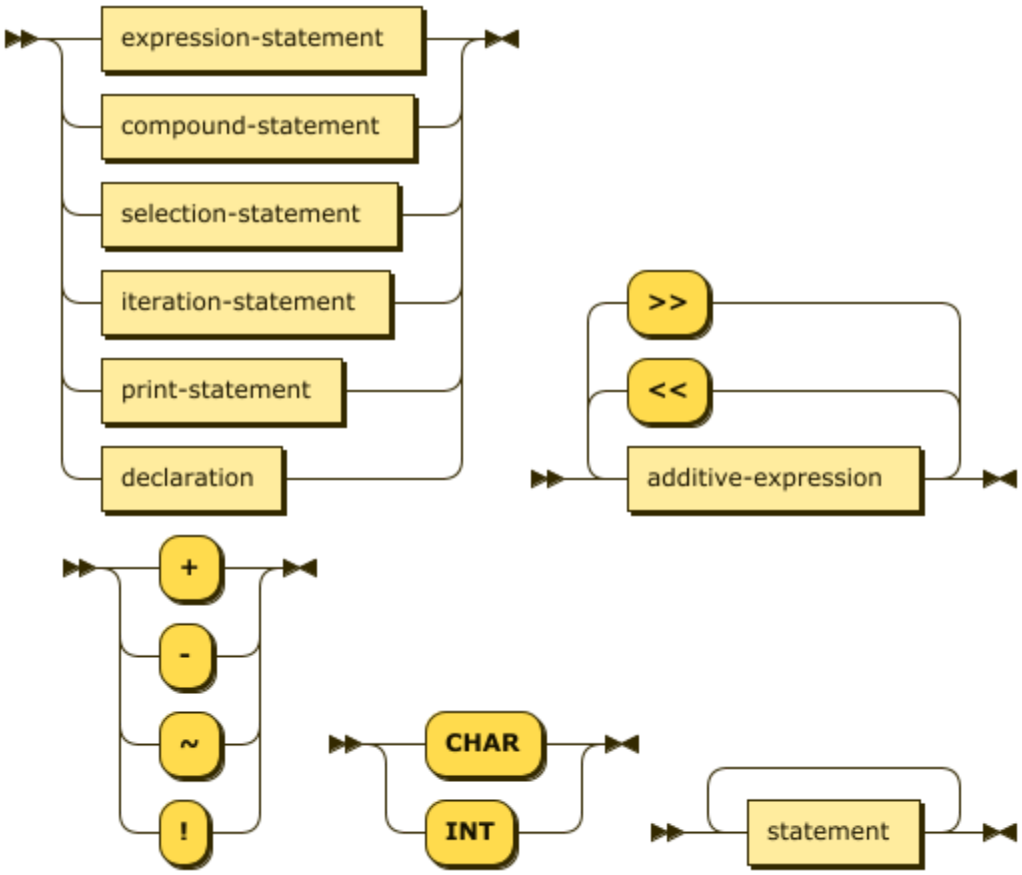
```
assignment-expression ::= logical-or-expression
                        | assignment-expression assignment-operator
logical-or-expression

logical-or-expression ::= logical-and-expression
                        | logical-or-expression '||' logical-and-expression

logical-and-expression ::= inclusive-or-expression
                        | logical-and-expression '&&'
inclusive-or-expression

inclusive-or-expression ::= exclusive-or-expression
                              | inclusive-or-expression '|'
exclusive-or-expression

exclusive-or-expression ::= and-expression
                              | exclusive-or-expression '^' and-expression

and-expression ::= equality-expression
                  | and-expression '&' equality-expression

equality-expression ::= relational-expression
                        | equality-expression '==' relational-expression
                        | equality-expression '!=' relational-expression

relational-expression ::= shift-expression
                        | relational-expression '<' shift-expression
                        | relational-expression '>' shift-expression
                        | relational-expression '<=' shift-expression
                        | relational-expression '>=' shift-expression

shift-expression ::= additive-expression
                  | shift-expression '<<' additive-expression
                  | shift-expression '>>' additive-expression

additive-expression ::= multiplicative-expression
                        | additive-expression '+' multiplicative-expression
                        | additive-expression '-' multiplicative-expression

multiplicative-expression ::= primary-expression
                              | multiplicative-expression '*'
primary-expression
```

```
                              | multiplicative-expression '/'
primary-expression
                              | multiplicative-expression '%'
primary-expression

primary-expression ::= identifier
                       | constant
                       | string
                       | '(' expression ')'

expression ::= constant-expression
            | expression ',' constant-expression

assignment-operator ::= '='

unary-operator ::=   '+'
                  | '-'
                  | '~'
                  | '!'

compound-statement ::= '{' statement-list '}'

selection-statement ::= 'IF' '(' constant-expression ')' compound_statement
                              | 'IF' '(' constant-expression ')'
compound_statement 'ELSE' compound_statement

iteration-statement ::= 'WHILE' '(' constant-expression ')'
compound_statement

print-statement ::= 'PRINT' '(' identifier ')' ';'
                  | 'PRINT' '(' constant ')' ';'
```

בנוסף railroad diagram של התחביר:


&
equality-expression
-
+
multiplicative-expression
=
logical-or-expression
{
statement-list
}
type-specifier
identifier
=
constant-expression
;
^
and-expression
!=
==
relational-expression
|
exclusive-or-expression
constant-expression
;
WHILE
(
constant-expression
)
compound_statement
||
logical-and-expression
&&
inclusive-or-expression
%
/
*
primary-expression


identifier
constant
string
,
(
constant-expression
)
PRINT
(
identifier
constant
)
;
statement-list
>=
<=
>
<
shift-expression
IF
(
constant-expression
)
compound_statement
ELSE
compound_statement



כל הטבלאות התוארו יהיו בנספח

ניתוח סמנטי (semantic analysis)

יצירת קוד (code generation)

טבלת סימנים (symbol table)

מטפל השגיאות (error handler)

ארכיטקטורת הפתרון

Symbol table

טבלת סמלים היא מבנה נתונים המשמש את המהדר לאחסון מידע על הסמלים השונים, כגון מזהים, קבועים, נהלים ופונקציות, בקוד המקור של תוכנית. היא שומרת מידע חיוני על כל סמל, כולל שמו, סוגו, תחומו (scope) ותכונות אחרות שלו. טבלת הסמלים משומשת במהלך כל שלבי הקומפילציה, ונבנת בשלבי הניתוח. מטרות טבלת הסמלים:

- זיהוי סוג משתנים, כתובת הזיכרון ושמות של משתנים.
- ניהול משתנים בהתייחס לתחום.
- משומש בשביל להתמודד עם שגיאות.
- הטבלה מסדרת את הסמלים והתכונות שלהם מה שעוזר לניהול התכנית.
- משתמשים בטבלה בשביל ליצור את הקוד הסופי.

התמודדות עם שגיאות

בנוסף לכל הרכיבים האלו, הקומפיילר צריך לדעת איך להתמודד עם שגיאות. לכל מבנה שונה יהיה שגיאות שונות, לדוגמא, במנתח המילולי, יכול להיות שגיאה לקסיקלית, שבא אותו מנתח מזה רצף תווים לא מזוהה ואינה יודעת איך להתמודד אם אותו רצף. נחלק את השגיאות שיכולות להיות לנו לארבעה קטגוריות:

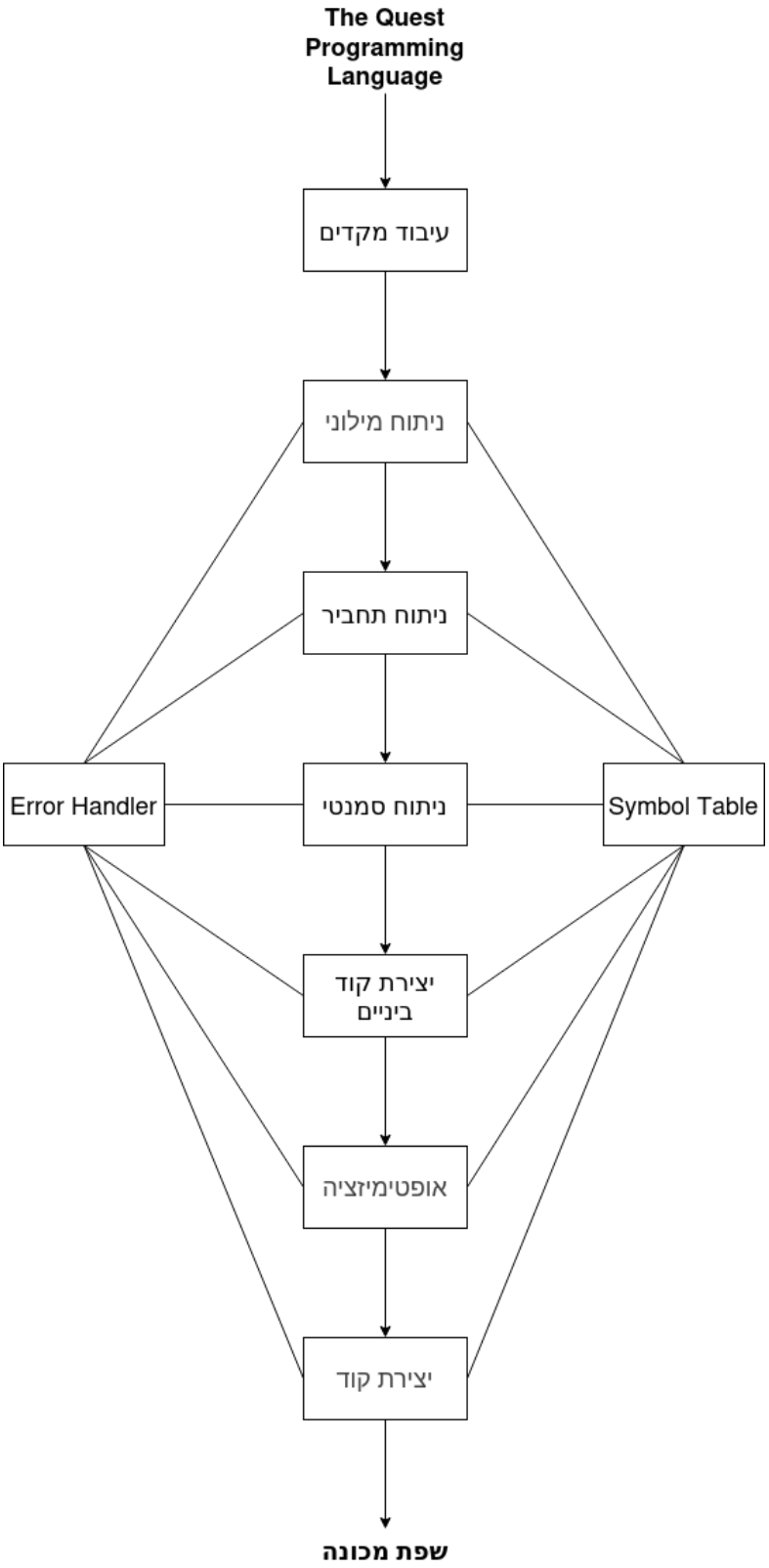
1. **שגיאות לקסיקליות** - המנתח המילוני קורא רצף תווים לא מזוהה ולא יודע איך להתמודד עם אותו הרצף, לדוגמה, שם של סוג משתנה רשום בצורה שגויה.
2. **שגיאות תחביריות** - המנתח התחבירי מזהה בתחביר החומרה, כלומר אסימון במקום שהוא לא צריך להיות, לדוגמא, סוג משתנה במקום לא רצוי או סוגריים פותחות ללא סוגר מתאים.

3. **שגיאות סמנטיות** - המנתח הסמנטי מזהה שגיאה סמנטית, לדוגמא חיבור בין שני סוגי משתנה שונים בלי דרך למצוא תוצאה תואמת.
4. **שגיאות לוגיות** - שגיאות בלוגיקה של הקוד, לדוגמא לולאה אין-סופית.

בשביל לזהות את אותם שגיאות, נשתמש בעוד רכיב הנקרא **מטפל השגיאות (Error Handler)**.
למטפל השגיאות יהיה שלושה ייעודים:

- זיהוי שגיאות
- דיווח שגיאות (במידה ורצוי)
- טיפול בשגיאות (במידה ואפשר)

תרשים סופי


The Quest Programming Language
עיבוד מקדים
ניתוח מילוני
ניתוח תחביר
Error Handler
ניתוח סמנטי
Symbol Table
יצירת קוד ביניים
אופטימיזציה
יצירת קוד
שפת מכונה

תרשים מקרי שימוש

מבנה נתונים

set

מבנה הנתונים המופשט *סט* הינו אוסף השומר נתונים שונים אחד מהשני, בדיוק כמו סט במתמטיקה. תכונות המפתח של סט הינו:

- **ייחודיות** - כל האלמנטים בסט הם ייחודיים בלי שום הכפלות.
- **אי-סידוריות** - לאיברים בקבוצה אין סדר ספציפי.

המימוש של הסט בקוד נוצרה בעזרת שני מבנים:

- Set - הסט עצמו
  - מצביע לצומת ראש
  - מצביע לפונקציית ההשוואה
  - גודל הסט
- Set node - צומת המגדיר איבר בסט, יוצר רשימה מקושרת
  - ערך גנרי
  - מצביע לצומת הבאה ברשימה

הקוד:

```
typedef struct GENERIC_SET_NODE_STRUCT {
   void *data;                            // Pointer to the element data
(can hold any type)
   struct GENERIC_SET_NODE_STRUCT *next;  // Pointer to the next node in
the set
} set_node_T;

// Define the set structure
typedef struct GENERIC_SET_STRUCT {
   set_node_T *head;                          // Pointer to the head node
of the set
   int (*compare)(const void*, const void*);   // Pointer to a comparison
function
   size_t size;                               // Size of the set (number
of elements)
} set_T;
```

hashset

אימפלמנטציה של סט בעזרת hash table (טבלת גיבוב).

טבלת גיבוב, הידועה גם כטבלת ערבול, היא מבנה נתונים מיוחד במדעי המחשב שמאפשר אחזור וניהול של נתונים בצורה יעילה. טבלת הגיבוב מורכבת משני רכיבים עיקריים:

- **אחסון**: רכיב המכיל את הנתונים.
- **פונקציית גיבוב**: פונקציה זו מקבלת מפתח (לדוגמה, שם) ומחזירה אינדקס במערך. האינדקס הזה מצביע על התא שבו מאוחסן ערך המפתח (לדוגמה, מספר טלפון).

טבלת הגיבוב מאפשרת גישה מהירה לנתונים. הסיבה לכך היא שפונקציית הגיבוב מחשבת ישירות את מיקום הנתונים במערך, ללא צורך בחיפוש סדרתי.

המימוש המיוחד של סט בעזרת טבלת גיבוב מחייבת בנוסף גם פונקציית השוואה בין איברים.

הקוד:

```
typedef struct HASHSET_NODE_STRUCT {
   void *data;                          /* The data stored in the node */
   struct HASHSET_NODE_STRUCT *next;  /* Pointer to the next node in the
bucket */
} hashset_node_T;
typedef struct HASHSET_STRUCT {
   int size;               /* Number of elements in the set */
   int capacity;           /* Maximum number of elements in the set */
   float load_factor;      /* Maximum ratio of number of elements to number
of buckets */
   hashset_node_T **buckets; /* Array of bucket pointers */
   unsigned int (*hash_func)(void *); /* Pointer to a hash function to hash
data */
   int (*compare_func)(void *, void *); /* Pointer to a comparison function
to compare data */
} hashset_T;
```

queue

תור (Queue) הוא מבנה נתונים מופשט שמתפקד כמו תור המתנה. בדומה לתור פיזי של אנשים, בתור נתונים האיבר הראשון שנכנס לתור הוא גם הראשון שיוצא ממנו. עיקרון זה ידוע בשם "נכנס ראשון יוצא ראשון" (FIFO).

תכונות מפתח:
First in first out (FIFO): האיבר שנכנס ראשון לתור יצא ראשון מהתור, האיבר שנכנס שני יצא שני...

פעולות עיקריות:

- **הכנסה (enqueue)**: הוספת איבר חדש לסוף התור.
- **הוצאה (dequeue)**: הוצאת האיבר הראשון מהתור.
- **בדיקה אם ריק (isEmpty)**: בדיקה האם התור ריק מאיברים.
- **בדיקת ערך בראש התור (peek)**: הצגת ערך האיבר הראשון בתור מבלי להוציא אותו.

ישנן דרכים שונות ליישם תור, ביניהן:

- **רשימה מקושרת**: רשימה מקושרת חד כיוונית, בה האיברים מאוחסנים בצמתים והקשרים ביניהם מצביעים על סדר הכניסה לתור.
- **מערך**: מערך דינמי שבו איברים חדשים מתווספים בסוף ומאוחסנים לפי סדר הכניסה.
- **תור מעגלי**: מערך שבו האינדקסים "מתעטפים" כך שהאיברים תמיד מאוחסנים באותו גודל מערך, ללא בזבוז מקום.

אני בחרתי לממש את התור ברשימה מקושרת, כך נשמר הפשטות של התור בלי ויתור על יעילות. מימוש התור:

```
/**
* Generic Queue Node Structure
*
* Contains data and a pointer to the next node in the queue.
*/
typedef struct GENERIC_QUEUE_NODE_STRUCT {
   void* data;                    /* Data stored in the node */
   struct GENERIC_QUEUE_NODE_STRUCT *next;  /* Pointer to the next node in
the queue */
} queue_node_T;

/**
* Generic Queue Structure
*
* Contains a pointer to the head and tail nodes of the queue,
* as well as the current size of the queue.
*/
typedef struct GENERIC_QUEUE_STRUCT {
   queue_node_T *head;   /* Pointer to the head node of the queue */
   queue_node_T *tail;   /* Pointer to the tail node of the queue */
   int size;            /* Current size of the queue */
} queue_T;
```

stack

מחסנית (Stack) היא מבנה נתונים מופשט הפועל בצורה דומה למחסנית רובה: האיבר שנכנס ראשון למחסנית יוצא ממנה אחרון. עיקרון זה ידוע בשם "נכנס אחרון יוצא ראשון" (LIFO).

תכונות מפתח:

- **Last In First Out (LIFO)**: האיבר הראשון שיכנס אצא אחרון.

פעולות עיקריות:

- **דחיפה (push)**: הוספת איבר חדש לראש המחסנית.
- **שליפה (pop)**: הוצאת האיבר העליון מהמחסנית.
- **בדיקה אם ריקה (isEmpty)**: בדיקה האם המחסנית ריקה מאיברים.
- **הצצה (peek)**: הצגת ערך האיבר העליון במחסנית מבלי להוציא אותו.

ישנן דרכים שונות ליישם מחסנית, ביניהן:

- **רשימה מקושרת**: רשימה מקושרת חד כיוונית, בה האיברים מאוחסנים בצמתים והקשרים ביניהם מצביעים על סדר הכניסה למחסנית.
- **מערך**: מערך דינמי שבו איברים חדשים מתווספים לראש ומאוחסנים לפי סדר הכניסה.

אני בחרתי לממש את המחסנית ברשימה מקושרת, כך נשמר הפשטות של המחסנית בלי ויתור על יעילות. מימוש המחסנית:

```
/**
* Structure of a stack node
*/
typedef struct STACK_NODE_STRUCT {
 void *data;                        // The data stored in the stack node
 struct STACK_NODE_STRUCT *next;  // Pointer to the next stack node
} stack_node_T;

/**
* Structure of a generic stack
*/
typedef struct GENERIC_STACK_STRUCT {
 stack_node_T *top;                // Pointer to the top of the stack
 size_t size;                      // Number of items in the stack
} stack_T;
```

ניתוח מילוני (lexical analysis)

אוטומט סופי דטרמיניסטי המתאים למנתח המילוני

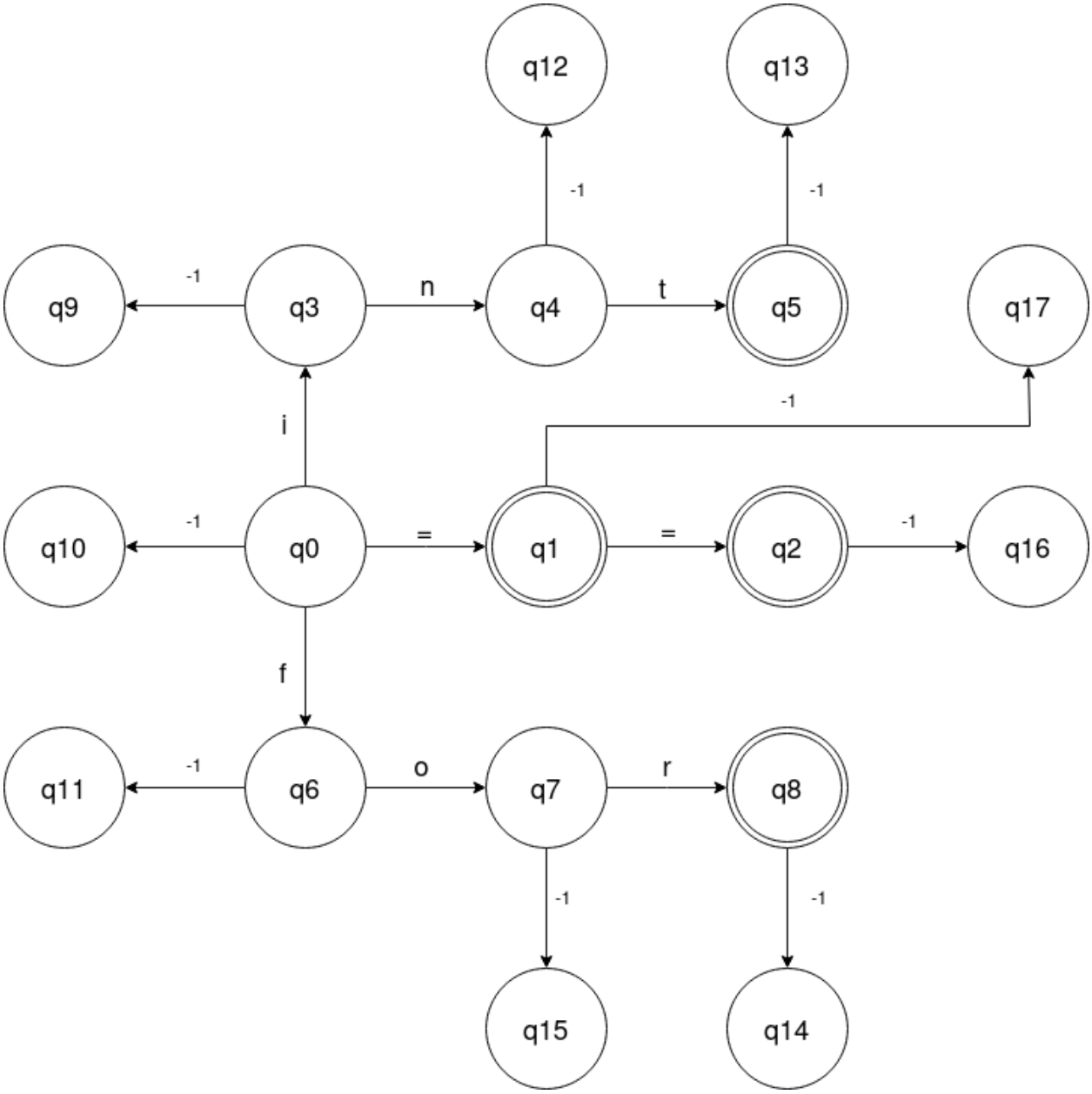
בשביל זיהוי lexemeים בצורה אופטימלית במהלך הניתוח המילוני החלטתי להשתמש באוטומט סופי דטרמיניסטי, שנותן לי לזהות מילה ממילון הטבלה בזמן O(1).

הקלט יהיה אוסף של אותיות (מילה) מתוך הא"ב של השפה, שהיא בנויה מקבוצה של סימנים, לדוגמה, בשפת Quest, הא"ב יהיה כל תווי הascii. האוטמט עובר על המילה ורואה בשביל כל מצב איזו פונקציית מעבר מתאימה לו, כלומר במקרה הזה. כאשר אין יותר סימנים במילה, והמילה היא מילה במילון השפה, הזיהוי הסתיים בהצלחה. תהליך זה מראה שזיהוי מילה הוא אכן בזמן O(1) (כאשר אורך מילה מקסימלית בשפה הוא קבוע).

האוטומט יהיה ממומש בעזרת גרף, כאשר הגרף יהיה ממומש ע"י מטריצת סמיכויות. נממש מבנה זה בשפת C בעזרת מערך דו-ממדי, כאשר:

- כל 128 תווי ה-ascii יוצגו ע"י עמודות המערך הדו-ממדי, כאשר כל אינדקס מותאם לערך האות (לפי טבלת הascii).
- כל המצבים ייוצגו ע"י שורות המערך הדו-ממדי, המצב האחרון שנהיה בוא יהיה תואם לאסימון.
  - אנחנו נגדיר מצב מתחיל שבו תמיד נתחיל את האנליזה, שהוא יהיה המצב הראשון שנקרא לו q0.
- כל ערך במטריצה מגדיר את פונקציית המעבר.
  - ערך שלם אי-שלילי מגדיר את המצב הבא.
  - הערך 1- מגדיר שהמילה לא במילון השפה ושצריך להפסיק את חיפוש המילה.

לדוגמא, נגדיר DFA שלוקח את המילים: int, for, ==, =



נמיר את זה למערך דו ממדי (נתעלם מזה שאינדקס העמודה מתאים לערך האות):

	=	i	n	t	f	o	r
0	1	3	-1	-1	6	-1	-1
1	-1	2	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	4	-1	-1	-1	-1

4	-1	-1	-1	5	-1	-1	-1
5	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	7	-1
7	-1	-1	-1	-1	-1	-1	8
8	-1	-1	-1	-1	-1	-1	-1

אסימון (Token)

ניתוח תחביר (syntax analysis/parsing)

Grammer
lr(0) automaton
Action table
Goto table
lr stack

עץ תחביר (Parse Tree)

ניתוח סמנטי (semantic analysis)

sdt
ast

יצירת קוד (code generation)

טבלת סימנים (symbol table)

מטפל השגיאות (error handler)

סביבת העבודה ושפת התכנות

סביבת העבודה

OS: Debian GNU/Linux 12 (bookworm) x86_64
Host: VivoBook_ASUSLaptop X415JA_X415JA 1.0
Kernel: 6.1.0-18-amd64
Uptime: 4 days, 4 hours, 48 mins
Packages: 2944 (dpkg), 7 (flatpak), 24 (snap)
Shell: bash 5.2.15
Resolution: 1366x768, 1920x1080
DE: Xfce 4.18
WM: Xfwm4
WM Theme: empty
Theme: Xfce [GTK2], Adwaita [GTK3]
Icons: Tango [GTK2], Adwaita [GTK3]
Terminal: x-terminal-emul
CPU: Intel i5-1035G1 (8) @ 3.600GHz
GPU: Intel Iris Plus Graphics G1
Memory: 6875MiB / 7682MiB

עורכי הקוד

VSCodium v1.88.1
NVIM v0.9.4

שפת תכנות וקומפיילר

כל הקוד נכתב ב-C, עם ה-gcc קומפיילר.

gcc: v12.2.0 (Debian 12.2.0-14)

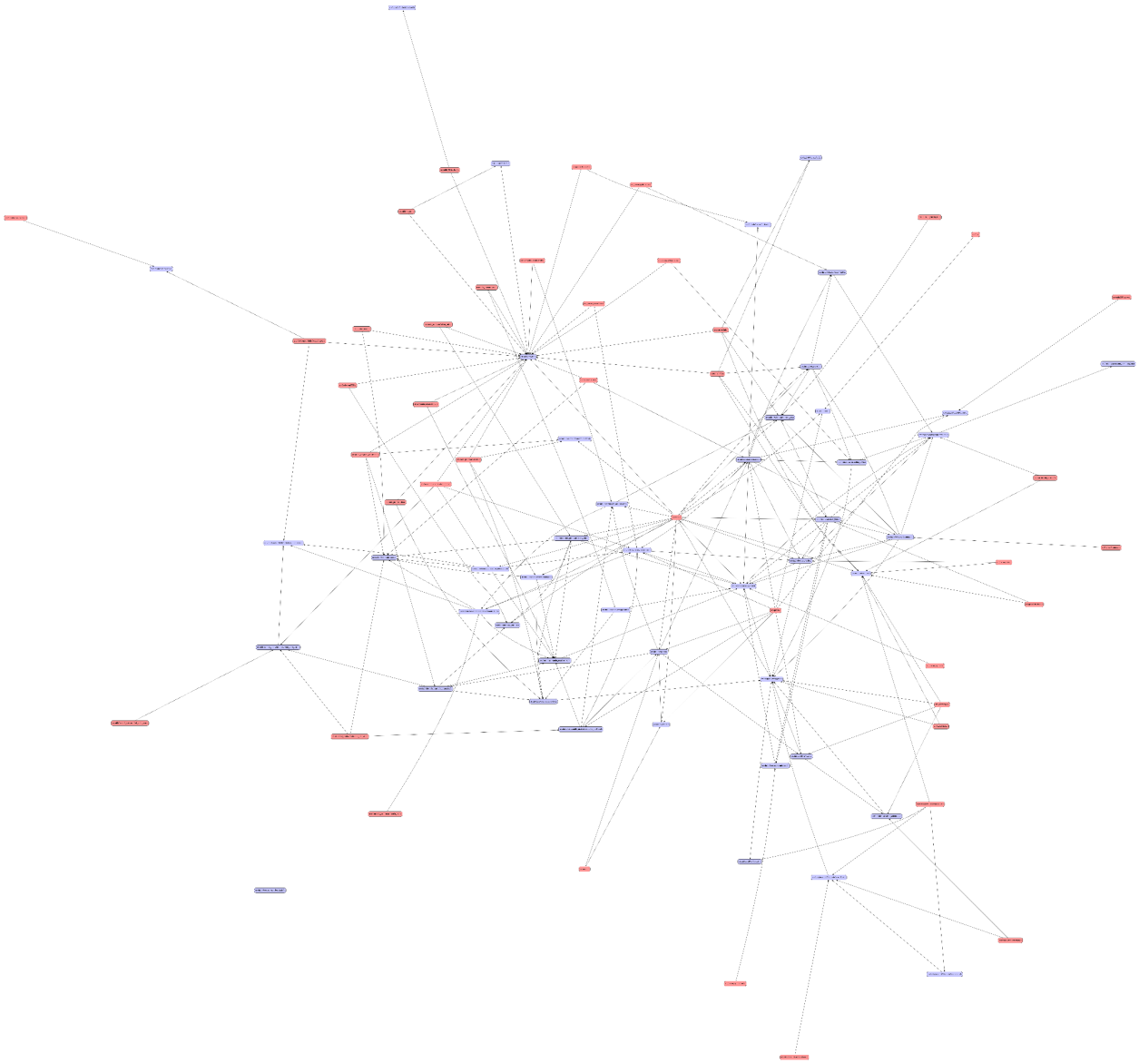
שפת התכנות

אלגוריתם ראשי

תיאור ממשקים חיצוניים

תרשים מחלקות

מכיוון שאין מחלקות ב-C, הכנתי dependency diagram:



הדיאגרמה מופיע כנספח.

מודולים ופונקציות ראשיות

מבט על

שימו לב שקיימות תיקיות include, שהם תיקיות השומרות קבצי header. בהמשך אני אתייחס לפעמים לקובץ ה-c וקובץ ה-h כאחד.

- code_gen
  - TTS.c
  - code_generator.c
  - operand.c
  - register.c
  - translation_rule.c
  - translations.c
- include
  - code_gen
    - TTS.h
    - code_generator.h
    - nasm_macros.h
    - operand.h
    - register.h
    - translation_rule.h
    - translations.h
  - io.h
  - lang.h
  - lexer
    - lexer.h
    - lexer_automata.h
    - token.h
    - tokens.h
  - macros.h
  - parser
    - action_table.h
    - bnf.h
    - goto_table.h
    - grammer.h

            - lr_item.h
            - lr_stack.h
            - non_terminal.h
            - non_terminals.h
            - non_terminals_bnf.h
            - parse_tree.h
            - parser.h
            - rule.h
            - slr.h
            - symbol.h
            - symbol_set.h
        - quest.h
        - semantic_analizer
            - AST.h
            - definitions.h
            - sdt.h
            - semantic_analyzer.h
            - semantic_rule.h
- io.c
- lang.c
- lexer
    - lexer.c
    - lexer_automata.c
    - token.c
- main.c
- parser
    - action_table.c
    - bnf.c
    - goto_table.c
    - grammer.c
    - lr_item.c
    - lr_stack.c
    - non_terminal.c
    - parse_tree.c
    - parser.c
    - rule.c
    - slr.c
    - symbol.c
    - symbol_set.c
- quest.c
- semantic_analizer
    - AST.c
    - definitions.c
    - sdt.c

    - semantic_analyzer.c
    - semantic_rule.c
- utils
    - DS
        - generic_set.c
        - hashset.c
        - include
            - generic_set.h
            - hashset.h
            - queue.h
            - stack.h
        - queue.c
        - stack.c
    - err
        - err.c
        - err.h
        - errors.h
    - hashes
        - hashes.c
        - hashes.h
    - lexer_DFA
        - include
            - lexer_DFA.h
            - transitions.h
        - lexer_DFA.c
        - transitions.c
    - symbol_table
        - include
            - symbol_table.h
            - symbol_table_tree.h
    - symbol_table.c
    - symbol_table_tree.c

להלן הסברים עבור כל קובץ חשוב, עם הסברים על מבנים והפונקציות הראשיות שבניהם.
בשביל להסביר על כל struct אשתמש בטבלה הבא:

	שם מבנה	
הסבר	סוג	שם משתנה

ובשביל להסביר על פונקציה אשתמש בטבלה הבא:

שם פונקציה	קלט	פלט	יעילות	תיאור

code_gen

TTS

	TREE_TRANSLATION_SCHEME_STRUCT	
שם משתנה	סוג	הסבר
tok_translation	translation_rule_T **	מערך דינמי השומר מצביעים ל-structים המגדירים תרגומים של אסימונים
n_tok	size_t	מספר האיברים במערך
non_term_translation	translation_rule_T **	מערך דינמי השומר מצביעים ל-structים המגדירים תרגומים של non-terminals
n_non_term	size_t	מספר האיברים במערך

שם פונקציה	קלט	פלט	יעילות	תיאור
init_tts	translation_rule_T **tok_translation, size_t n_tok, translation_rule_T **non_term_tra	tts_T*	O(1)	יוצר משתנה מסוג tts_T* ומחזיר אותו.

			nslation, size_t n_non_term	
יוצר משתנה מסוג tts_T* בנוסף למיקום במערך המתאים לערך האסימון או לערך ה-non terminal.	O(1)	tts_T*	translation_rule_T **tok_translation, size_t n_tok, translation_rule_T **non_term_translation, size_t n_non_term	create_tts

code_generator

	CODE_GENERATOR_STRUCT	
הסבר	סוג	שם משתנה
מערך דינמי השומר מצביעים ל-structים המגדירים תרגומים של אסימונים	register_pool_T **	registers
מספר האיברים במערך	tts_T *	tts
מערך דינמי השומר מצביעים ל-structים המגדירים תרגומים של non-terminals	symbol_table_tree_T *	sym_tbl
מונה בשביל יצירת labelים	uint32_t	label_counter
מחרוזת תווים המייצגת את הקוד הנוצר	char *	output

תיאור	יעילות	פלט	קלט	שם פונקציה

יוצר משתנה מסוג code_gen* ומחזיר אותו.	O(1)	code_gen_T *	register_pool_T **registers, tts_T *tts, symbol_table_tree_T *sym_tbl	init_code_gen
יוצר את הקוד ומחזיר מחרוזת של תכנית המטרה	O(n)	char *	ast_node_T *ast, code_gen_T *cg	generate_code

operand

	OPERAND_TYPES_ENUM	
הסבר	**סוג**	**שם משתנה**
מגדיר סמל	uint = 0	SYMBOL
מגדיר רגיסטר	uint = 1	REGISTER

	OPERAND_UNION	
הסבר	**סוג**	**שם משתנה**
מצביע לסמל	symbol_T *	sym
מצביע לרגיסטר	register_T *	reg

	OPERAND_STRUCT	
הסבר	**סוג**	**שם משתנה**
סוג המשתנה ב-union	operand_type_E	type

operand	operand_U *	מגדיר אופרנד

שם פונקציה	קלט	פלט	יעילות	תיאור
init_operand_symbol	symbol_T *	operand_T *	O(1)	יוצר מצביע לאופרנד שיש בתוכו סימן
init_operand_register	register_T *	operand_T *	O(1)	יוצר מצביע לאופרנד שיש בתוכו רגיסטר

register

	REGISTER_STRUCT	
שם משתנה	סוג	הסבר
reg	register_E	סוג רגיסטר
size	register_size_E	גודל רגיסטר
name	char *	שם רגיסטר

	REGISTER_POOL_STRUCT	
שם משתנה	סוג	הסבר
value	uint64_t	הערך ברגיסטר
in_use	uint64_t	האם כל בית ברגיסטר בשימוש או לא
type	register_type_E	סוג הערך של הרגיסטר

שם פונקציה	קלט	פלט	יעילות	תיאור
init_register	uint8_t type, uint8_t size, char *name	register_T *	O(1)	איתחול הרגיסטר עם הקלט
init_register_pool	uint8_t type	register_pool_T *	O(1)	אתחול בריכת הרגיסטרים המשומשים בקוד
get_register	register_pool_T **regs, uint8_t type, uint8_t size	register_T *	O(n)	מוצא רגיסטר חופשי בבריכת הרגיסטרים המתאים לקלט
get_register_name	uint8_t type, uint8_t size	char *	O(1)	מחזיר את שם הרגיסטר לפי מטריצת שמות הרגיסטרים
get_byte_data_reg_name	uint8_t type, uint8_t bits	char *	O(1)	מחזיר את שם הרגיסטר לפי מטריצת שמות הרגיסטרים מסום DATA ובגודל BYTE
reg_free	register_pool_T **regs, register_T *reg	void	O(1)	משחרר את השימוש ברגיסטר בבריכת הרגיסטרים

Translation_rule

	TRANSLATION_RULE_STRUCT	
שם משתנה	סוג	הסבר

symbol	symbol_T *	הסמל המתאים לתרגום
translation	char *(*translation)(ast_node_T *ast, stack_T *astack, stack_T *code_stack, register_pool_T **regs)	מצביע לפונקציה שמתרגמת את הסימן בהתאם לפרמטרים של הפונקציה

שם פונקציה	קלט	פלט	יעילות	תיאור
init_translation_rule	l_T *symbol, char *(*translation)(ast_node_T *ast, stack_T *astack, stack_T *code_stack, register_pool_T **regs)	translation_rule_T *	O(1)	מאתחל מצביע לחוק תרגום

translations

שם פונקציה	קלט	פלט	יעילות	תיאור
translation_… (כל פונקציית תרגום)	ast_node_T *ast, stack_T *astack, stack_T *code_stack, register_pool_T **regs	char *	O(1)	מתרגם סימן בהתאמה לפרמטרים

lexer

lexer

	LEXER_STRUCT	
הסבר	סוג	שם משתנה
קוד המקור	char *	src
גודל קוד המקור	size_t	src_size
התו שהלקסר עליו	char	c
האינדקס של התו ממצביע קוד המקור	unsigned int	i
האוטומט של הלקסר	lexer_automata_T *	automata

תיאור	יעילות	פלט	קלט	שם פונקציה
מאתחל משתנה מסוג מצביע ללקסר	O(1)	lexer_T *	char *src	init_lexer
עובר לתו הבא במחרוזת התכנית	O(1)	void	lexer_T* lex	lexer_advance
עובר על רווחים	O(1)	void	lexer_T *lex	lexer_skip_whitespace
עובר על תווים לא מוגדרים בסימיני השפה	O(1)	void	lexer_T *lex	lexer_skip_bullshit
מחזיר את האסימון הבא בתכנית	O(1)	token_T *	lexer_T *lex	lexer_next_token

lexer_automata

	LEXER_AUTOMATA_STRUCT	
הסבר	סוג	שם משתנה
מטריצה המגדירה את האוטומט	short **	automata
מערך המגדיר את סוגי המצבים	token_type_E *	state_type
מספר המצבים	unsigned int	n_state
מספר הסימנים מתחילת טבלת הASCII	unsigned int	n_symbols

תיאור	יעילות	פלט	קלט	שם פונקציה
מאתחל את האוטומט של הלקסר	O(1)	lexer_automata_T *	const char *auto_src, const char *states_src	init_lexer_automata

token

	TOKEN_STRUCT	
הסבר	סוג	שם משתנה
ערך האסימון	char *	value
סוג האסימון	token_type_E	type

תיאור	יעילות	פלט	קלט	שם פונקציה

init_token	char *value, int type	token_T *	O(1)	מאתחל אסימון
token_cmp	const token_T *tok1, const token_T *tok2	int	O(1)	משווה בין שני אסימונים
token_cmp_generic	const void *tok1, const void *tok2	int	O(1)	משווה בין שני אסימונים עם קלט גנרי

parser

action_table

	ACTION_TABLE_STRUCT	
שם משתנה	סוג	הסבר
actions	char ***	מטריצה של מחרוזות המתארת את ה-actionים שהפרסר לוקח
terminals	token_T **	מערך של טרמינלים המתארים את הטרמינלים במטריצה
n_terminals	size_t	מספר הטרמינלים
n_states	size_t	מספר המצבים

שם פונקציה	קלט	פלט	יעילות	תיאור

מאתחל את טבלת האקשנים	O(1)	action_tbl_T *	token_T **terminals, size_t n_terminals, size_t n_states	init_action_tbl
מחזיר את האינדקס של הטרמינל במערך ובהתאמה במטריצה	O(n)	int	action_tbl_T *tbl, token_T *term	action_tbl_find_terminal
מדפיס את טבלת האקשנים לקובץ	O(n)	void	action_tbl_T *tbl, char *dest	action_tbl_print_to_file
מדפיס את טבלת האקשנים לקובץ רק יפה	O(n)	void	action_tbl_T *tbl, char *dest	action_tbl_pretty_print_to_file

bnf

תיאור	יעילות	פלט	קלט	שם פונקציה
יוצר non-terminals בקוד מקובץ bnf	O(n)	void	char *src, char *dest	bnf_make_non_terminals

goto_table

	GOTO_TABLE_STRUCT	

שם משתנה	סוג	הסבר
gotos	int **	מטריצה של שלמים המתארת את ה-gotoים שהפרסר לוקח
non_terminals	non_terminal_T **	מערך של non-terminals המתארים את ה-non-terminals במטריצה
n_non_terminals	size_t	מספר ה-non-terminals
n_states	size_t	מספר המצבים

שם פונקציה	קלט	פלט	יעילות	תיאור
init_goto_tbl	non_terminal_T **non_terminals, size_t n_non_terminals, size_t n_states	goto_tbl_T *	O(1)	מאתחל את טבלת ה-gotoים
goto_tbl_find_non_terminal	goto_tbl_T *tbl, non_terminal_T *nterm	size_t	O(n)	מחזיר את האינדקס של הnon-terminal במערך ובהתאמה במטריצה
goto_tbl_print_to_file	goto_tbl_T *tbl, char *dest	void	O(n)	מדפיס את טבלת הgotoים לקובץ
goto_tbl_pretty_print_to_file	goto_tbl_T *tbl, char *dest	void	O(n)	מדפיס את טבלת הgotoים לקובץ רק יפה

grammer

	GRAMMER_STRUCT	
שם משתנה	סוג	הסבר
rules	set_T *	סט של חוקים
symbols	set_T *	סט של סימנים

שם פונקציה	קלט	פלט	יעילות	תיאור
init_grammer	set_T *rules, set_T *symbols	grammer_T *	O(1)	מאתחל את הדקדוק
terminals_in_symbol_set	set_T *symbols	token_T **	O(n)	מחזיר את כל הטרמינלים מסט של סימנים
terminals_in_symbol_set_and_dollar	set_T *symbols	token_T **	O(n)	מחזיר את כל הטרמינלים מסט של סימנים פלוס סימן של דולר המסמן התחלה
n_terminals_in_symbol_set	set_T *symbols	size_t	O(n)	מחזיר את מספר הטרמינלים בסט של סימנים
non_terminals_in_symbol_set	set_T *symbols	non_terminal_T **	O(n)	מחזיר את כל הnon-terminals מסט של סימנים
n_non_terminals_in_symbol_set	set_T *symbols	size_t	O(n)	מחזיר את מספר הnon-terminals בסט של סימנים
find_right_grammer_index	const grammer_T *gram,	size_t	O(n)	מחזיר את האינדקס של חוק בדקדוק שיש לו

	symbol_T **right, size_t right_size			את אותו הצד השמאלי של החוק

lr_item

	LR_ITEM_STRUCT	
שם משתנה	סוג	הסבר
rule	rule_T *	מצביע לחוק של הפריט
dot_index	size_t	האינדקס של הנקודה של הפריט
lookaheads	set_T *	ה-lookaheadים של הפריט

שם פונקציה	קלט	פלט	יעילות	תיאור
init_lr_item	rule_T *rule, size_t dot_index, set_T *lookaheads	int	O(1)	מאתחל את הפריט
lr_item_cmp	const lr_item_T *item1, const lr_item_T *item2	int	O(1)	משווה את הפריט
lr_item_cmp_generic	const void *item1, const void *item2	int	O(1)	משווה את הפריט הגנרי

lr_item_set_cmp	const set_T *set1, const set_T *set2	int	O(1)	משווה סט פריטים
lr_item_set_cmp_generic	const void *item1, const void *item2	set_T *	O(1)	משווה סט פריטים גנרי
first	const grammer_T *gram, const symbol_T *sym	set_T *	O(n)	מוצא את ה-first set של סימן מסוים
follow	const grammer_T *gram, const non_terminal_T *nt	set_T *	O(n)	מוצא את ה-follow set של non-terminal
closure	grammer_T *grammer, set_T *items	set_T *	O(n)	מוצא את ה-clouser של סט של פריטים
go_to	grammer_T *grammer, set_T *items, symbol_T *symbol	set_T *	O(n)	מוצא את ה-goto של סט של פריטים וסימן מסוים
closure_lookahead	grammer_T *grammer, set_T *items	set_T *	O(n)	מוצא את ה-clouser של סט של פריטים עם lookahead
go_to_lookahead	grammer_T *grammer, set_T *items, symbol_T *symbol	set_T *	O(n)	מוצא את ה-goto של סט של פריטים עם lookahead וסימן מסוים
lr0_items	grammer_T *grammer, lr_item_T *starting_item	set_T *	O(n)	מוצא את הסט של הסטים של פריטים של תחביר lr(0)
lr1_items	grammer_T *grammer, lr_item_T *starting_item	set_T *	O(n)	מוצא את הסט של הסטים של פריטים של תחביר lr(1)

lr_stack

	LR_STACK_STRUCT	
הסבר	סוג	שם משתנה
הגג של הסטאק	int *	top
גודל הסטאק	size_t	size
הגודל המקסימלי של הסטאק	size_t	capacity
כמות הביתים להקצות כאשר הסטאק מלא	size_t	alloc_size

תיאור	יעילות	פלט	קלט	שם פונקציה
מאתחל את סטאק הפריטים	O(1)	lr_stack_T *	size_t alloc_size	init_lr_stack
דוחף איבר מגג הסטאק	O(1)	void	lr_stack_T *s, int data	lr_stack_push
מוציא איבר מגג הסטאק	O(1)	int	lr_stack_T *s	lr_stack_pop
מסתקל על האיבר בגג הסטאק	O(1)	int	lr_stack_T *s	lr_stack_peek
מסתקל על האיבר בגג הסטאק במרחק מסוים	O(1)	int	lr_stack_T *s, int n	lr_stack_peek_inside
בודק האם הסטאק מלא	O(1)	int	lr_stack_T *s	lr_stack_full
מוחק את כל הסטאק	O(n)	void	lr_stack_T *s	lr_stack_clear

non_terminal

	NON_TERMINAL_STRUCT	
הסבר	סוג	שם משתנה
סוג הnon-terminal	non_terminal_E	type
ערך הnon-terminal	char *	value

תיאור	יעילות	פלט	קלט	שם פונקציה
מאתחל את הnon-terminal	O(1)	non_terminal_T *	char *value, non_terminal_E type	init_non_terminal
משווה בין שניnon-terminal	O(1)	int	const non_terminal_T *nt1, const non_terminal_T *nt2	non_terminal_cmp

parse_tree

	PARSER_TREE_NODE_STRUCT	
הסבר	סוג	שם משתנה
סימן הצומת	symbol_T *	symbol

rule_index	ssize_t	האינדקס של החוק במערך החוקים
children	struct PARSE_TREE_NODE_STRUCT **	ילדי הצומת
n_children	size_t	מספר הילדים

	PARSE_TREE_STRUCT	
שם משתנה	סוג	הסבר
root	parse_tree_node_T *	שורש העץ התחבירי
rules	rule_T **	מערך החוקים
n_rules	size_t	מספר החוקים

שם פונקציה	קלט	פלט	יעילות	תיאור
init_parse_tree	parse_tree_node_T *root, rule_T **rules, size_t n_rules	parse_tree_T *	O(1)	מאתחל את העץ התחבירי
parse_tree_free	parse_tree_node_T *tree	void	O(1)	מוחק את העץ התחבירי
init_parse_tree_node	symbol_T *sym, ssize_t rule_index, parse_tree_node_T **children,	parse_tree_T *	O(1)	מאתחל צומת עץ תחבירי

			size_t n_children	
מאתחל עלה עץ תחבירי	O(1)	parse_tree_T *	symbol_T *sym	init_parse_tree_leaf
עובר על העץ התחבירי בסדר preorder	O(n)	void	parse_tree_node_T *tree, int layer	parse_tree_traverse_preorder
עובר על העץ התחבירי בסדר postorder	O(n)	void	parse_tree_node_T *tree, int layer	parse_tree_traverse_postorder

parser

	PARSER_STRUCT	
הסבר	**סוג**	**שם משתנה**
ה-action table	action_tbl_T *	action
ה-goto table	goto_tbl_T *	go_to
מערך דינמי של חוקים	rule_T **	rules
מספר החוקים	size_t	n_rules
סטאק הפרסר	lr_stack_T *	stack

שם פונקציה	קלט	פלט	יעילות	תיאור
init_parser	slr_T *slr	parser_T *	O(1)	מאתחל את המנתח התחבירי
parse	parser_T *prs, queue_T *queue_tok	parse_tree_T *	O(n)	מפרסר
parser_shift	parser_T *prs, int data	void	O(1)	עושה shift לסטאק
parser_reduce	parser_T *prs, rule_T *rule	symbol_T *	O(1)	עושה reduce לסטאק

rule

	RULE_STRUCT	
שם משתנה	סוג	הסבר
left	non_terminal_T *	הצד השמאלי של החוק התחבירי
right	symbol_T **	הצד הימני של החוק התחבירי
right_size	size_t	מספר הסימנים בצד הימני

שם פונקציה	קלט	פלט	יעילות	תיאור
init_rule	non_terminal_T *left, symbol_T **right, size_t right_size	rule_T *	O(1)	מאתחל חוק התחבירי
rule_cmp	const rule_T *rule1, const rule_T *rule2	int	O(1)	משווה בין שני חוקים
rule_cmp_generic	const void *rule1, const void *rule2	int	O(1)	משווה בין שני חוקים גנרים
find_first_nt	const rule_T *rule, int offset	int	O(n)	מוצא את הnon-terminal הראשון בצד ימין בחוק

slr

	SLR_STRUCT	
שם משתנה	סוג	הסבר
lr0_cc	set_T **	הסט הקאנוני של של כל הפריטים של lr(0)
lr0_cc_size	size_t	גודל הסט הקאנוני
action	action_tbl_T *	ה-action table
go_to	goto_tbl_T *	ה-goto table

grammer	grammer_T *	תחביר השפה

שם פונקציה	קלט	פלט	יעילות	תיאור
init_slr	set_T *lr0, grammer_T *gram	slr_T *	O(1)	מאתחל את ה-slr
slr_write_to_bin	slr_T *slr, char *dest	void	O(n)	כותב את ה-slr לקובץ בינארי
slr_read_from_bin	char *src	slr_T *	O(n)	קורא את ה-slr לקובץ בינארי
init_default_slr	void	slr_T *	O(1)	מאתחל את ה-slr הדיפולטי

symbol

	SYMBOL_TYPES_ENUM	
שם משתנה	סוג	הסבר
TERMINAL	uint = 0	מסמן טרמינל
NON_TERMINAL	uint = 1	מסמן non-terminal

	SYMBOL_UNION	
שם משתנה	סוג	הסבר
terminal	token_T *	אסימון

non-termial	non_terminal_T *	non_terminal

	SYMBOL_STRUCT	
הסבר	סוג	שם משתנה
סוג הסימן	symbol_type_E	sym_type
ה-union המגדיר את הסימן	symbol_U *	symbol

תיאור	יעילות	פלט	קלט	שם פונקציה
מאתחל סימן	O(1)	symbol_T *	symbol_U *symbol, symbol_type_E type	init_symbol
מאתחל סימן עם טרמינל	O(1)	symbol_T *	token_T *tok	init_symbol_terminal
מאתחל סימן עם non-terminal	O(1)	symbol_T *	non_terminal_T *nt	init_symbol_non_terminal
בודק אם הסימן שווה	O(1)	int	const symbol_T *sym1, const symbol_T *sym2	symbol_equals
משווה בין שני סימנים	O(1)	int	const symbol_T *sym1, const symbol_T *sym2	symbol_cmp
משווה בין שני סימנים	O(1)	int	const void *sym1, const void *sym2	symbol_cmp_generic

symbol_set

	SYMBOL_SET_STRUCT	
שם משתנה	סוג	הסבר
set	symbol_T **	סט של סימנים
size	size_t	גודל הסט של סימנים

שם פונקציה	קלט	פלט	יעילות	תיאור
init_symbol_set	void	symbol_set_T *	O(1)	מאתחל את סט הסימנים
init_symbol_set _with_symbols	symbol_T **syms, const size_t size	symbol_set_T *	O(n)	מאתחל את סט הסימנים עם סימנים קיימים
add_symbol	symbol_set_T *set, symbol_T *item	int	O(n)	מוסיף סימן אם הוא לא קיים בסט
remove_symbo l	symbol_set_T *set, symbol_T *item	int	O(n)	מוחק סימן בסט

semantic_analyzer

AST

	ABSRACT_SYNTAX_TREE _NODE_STRUCT	

שם משתנה	סוג	הסבר
symbol	symbol_T *	סימן הצומת
children	struct ABSTRACT_SYNTAX_TREE_NODE_STRUCT **	הילדים של הצומת בעץ הסמנטי המופשט
n_children	size_t	מספר הילדים
st_entry	symbol_table_entry_T *	איבר ה-symbol table המתאים לצומת

שם פונקציה	קלט	פלט	יעילות	תיאור
init_ast_node	symbol_T *symbol, ast_node_T **children, size_t n_children	ast_node_T *	O(1)	מאתחל צומת בעץ הסמנטי
init_ast_leaf	symbol_T *symbol	ast_node_T *	O(1)	מאתחל צומת בעלה הסמנטי
ast_add_to_node	ast_node_T *ast, ast_node_T *child	void	O(1)	מוסיף ילד לצומת
traverse_ast	ast_node_T *ast, int layer	void	O(n)	מדפיס את העץ

definitions

שם פונקציה	קלט	פלט	יעילות	תיאור

defenition_…	stack_T *astack, parse_tree_node_T *tree, stack_T *st_s	void	O(1)	מתרגם צמתים בעץ התחבירי לעץ הסמנטי

sdt

	SDT_STRUCT	
שם משתנה	סוג	הסבר
definitions	semantic_rule_T **	מערך דינמי עם חוקים תחביריים
n_defenitions	size_t	גודל המערך

שם פונקציה	קלט	פלט	יעילות	תיאור
init_sdt	semantic_rule_T **definitions, size_t n_defenitions	sdt_T *	O(1)	מאתחל sdt
init_default_sdt	rule_T **rules, size_t n_rules	sdt_T *	O(n)	מאתחל את ה-sdt הדיפולטי

semantic_analyzer

שם פונקציה	קלט	פלט	יעילות	תיאור
build_ast	parse_tree_T *tree, quest_T *q	ast_node_T *	O(n)	בונה את העץ התחבירי

semantic_rule

	SEMANTIC_RULE_STRUCT	
שם משתנה	סוג	הסבר
rule	rule_T *	חוק תחבירי המתאים להגדרה הסמנטית
definition	void (*)(stack_T *astack, parse_tree_node_T *tree, stack_T *st_s)	ההגדרה הסמנטית המתאימה לחוק התחבירי

שם פונקציה	קלט	פלט	יעילות	תיאור
init_sementic_rule	rule_T *rule, void (*)(stack_T *astack, parse_tree_node_T *tree, stack_T *st_s)	semantic_rule_T *	O(1)	מאתחל את החוק הסמנטי

מבני נתונים

generic_set

	GENERIC_SET_NODE_STRUCT	
שם משתנה	סוג	הסבר
data	void *	הערך הגנרי של הסט
next	struct GENERIC_SET_NODE_STRUCT *	הצומת הבאה בסט

GENERIC_SET_STRUCT		
הסבר	סוג	שם משתנה
צומת הראש של הסט	set_node_T *	head
מצביע לפונקציית ההשוואה של הסט	int (*)(const void*, const void*)	compare
גודל הסט	size_t	size

תיאור	יעילות	פלט	קלט	שם פונקציה
מאתחל את הסט הגנרי	O(1)	set_T *	int (*)(const void*, const void*)	set_init
מוסיף איבר לסט	O(1)	int	set_T* set, void* data	set_add
מוסיף סט לסט	O(n)	int	set_T* set, set_T *more_set	set_add_all
מוסיף מערך ערכים לסט	O(n)	int	set_T* set, void **more_set, size_t size	set_add_arr
מוחק איבר מהסט	O(n)	int	set_T* set, void* data	set_remove
בודק אם ערך מסוים נמצא בסט	O(n)	int	set_T* set, void* data	set_contains
הופך את הסט	O(n)	void	set_T* set	set_flip
מוחק את הסט	O(n)	void	set_T* set	set_free

hashset

	HASHSET_NODE_STRUCT	
הסבר	**סוג**	**שם משתנה**
ערך הצומת	void *	data
הצומת הבאה	struct HASHSET_NODE_STRUCT *	next

	HASHSET_NODE_STRUCT	
הסבר	**סוג**	**שם משתנה**
גודל הסט	int	size
הגודל המקסימלי של הסט	int	capacity
ה-load factor של הסט	float	load_factor
הצמתים בסט	hashset_node_T **	buckets
פונקציית ה-hash של הסט	unsigned int (*)(void *)	hash_func
פונקציית ההשוואה של הסט	unsigned int (*)(void *)	compare_func

שם פונקציה	קלט	פלט	יעילות	תיאור
init_hash_set	int initial_capacity, float load_factor, unsigned int (*hash_func)(void *), int (*compare_func)(void *, void *)	hashset_T *	O(1)	מאתחל את ה-hashset
hash_set_add	hashset_T *set, void* data	int	O(1)	מוסיף ערך לסט
hash_set_add_all	hashset_T *set, hashset_T *other_set	int	O(n)	מוסיף סט לסט אחר
hash_set_contains	hashset_T *set, void* data	int	O(1)	בודק אם ערך מסוים קיים בסט
hash_set_resize	hashset_T *set, int new_capacity	int	O(n)	משנה את גודל הסט
hash_set_compare	void *a, void *b	int	O(1)	פונקציית ההשוואה של הסט
hash_set_hash	void *data	unsigned int	O(1)	פונקציית ה-hash של הסט

queue

GENERIC_QUEUE_NODE_STRUCT		
שם משתנה	סוג	הסבר

data	void *	ערך הצומת
next	struct GENERIC_QUEUE_NODE_STRUCT *	הצומת הבאה

	GENERIC_QUEUE_STRUCT	
שם משתנה	סוג	הסבר
head	queue_node_T *	ראש הטור
tail	queue_node_T *	זנב הטור
size	int	גודל הטור

שם פונקציה	קלט	פלט	יעילות	תיאור
queue_init	void	queue_T *	O(1)	מאתחל את ה-queue
is_empty	queue_T* q	int	O(1)	בודק עם התור ריק
queue_enqueue	queue_T* q, void* data	void	O(1)	מכניס לתור ערך
queue_dequeue	queue_T* q	void *	O(1)	מוציא מהתור ערך
queue_peek	queue_T* q	void *	O(1)	מחזיר את הערך בראש התור

queue_clear	queue_T* q	void	O(n)	מוחק את כל התור
queue_size	queue_T* q	int	O(1)	מחזיר את גודל התור

stack

	STACK_NODE_STRUCT	
שם משתנה	סוג	הסבר
data	void *	ערך בצומת
next	struct STACK_NODE_STRUCT *	הצומת הבאה

	GENERIC_STACK_STRUCT	
שם משתנה	סוג	הסבר
top	stack_node_T *	ראש הסטאק
size	size_t	גודל הסטאק

שם פונקציה	קלט	פלט	יעילות	תיאור
stack_init	void	stack_T *	O(1)	מאתחל את הסטאק

stack_push	stack_T* s, void* data	void	O(1)	דוחף לראש לסטאק
stack_pop	stack_T* s	void *	O(1)	להוציא מראש הסטאק
stack_peek	stack_T* s	void *	O(1)	מחזיר את ערך בראש הסטאק
stack_clear	stack_T* s	void	O(n)	מוחק את הסטאק
stack_flip	stack_T* s	void	O(n)	הופך את הסטאק
stack_size	stack_T* s	size_t	O(1)	מחזיר את גודל הסטאק

שגיאות

שם פונקציה	קלט	פלט	יעילות	תיאור
thrw	int err	void	O(1)	זורק שגיאה מתאימה לקלט ועוצר את התכנית

hashes

שם פונקציה	קלט	פלט	יעילות	תיאור
hash_JOAAT	char *key, size_t length	uint32_t	O(1)	מימוש של ה-"Jenkins One

At A Time" Hash				

lexer_DFA

lexer_DFA

	LEXER_DFA_STATE_STRUCT	
הסבר	סוג	שם משתנה
האינדקס של מצב ה-DFA	unsigned int	index
ה-lexeme של המצב	char *	lexeme
סוג המצב	token_type_E	type

	LEXER_DFA_STRUCT	
הסבר	סוג	שם משתנה
מטריצה המגדירה את הDFA	short **	DFA

toks	token_T **	מערך דימני של אסימונים המתאימים ל-DFA
n_toks	unsigned int	גודל מערך האסימונים
states	lexer_dfa_state_T **	מערך דינמי של מצבי ה-DFA
last_states	lexer_dfa_state_T **	המופע האחרון של כל מצב מכל סוג
n_states	unsigned int	מספר המצבים
filename	char *	שם הקובץ בו נשמר ה-DFA
flags	unsigned short	דגלים המגדירים את ההגדרות של ה-DFA
n_flag_states	unsigned char	מספר המצבים המוגדרים ע"ג הדגלים

שם פונקציה	קלט	פלט	יעילות	תיאור
init_dfa	token_T **toks, const size_t n_toks, const char *DFA_filename, const char *DFA_states_filename, const char *DFA_states_details_filename, unsigned short flags	int	O(n)	מאתחל את ה-DFA
init_default_dfa	void	int	O(n)	מאתחל את ה-DFA הדיפולטי

transitions

שם פונקציה	קלט	פלט	יעילות	תיאור
add_…_transition	lexer_dfa_T *dfa, lexer_dfa_state_T *src_state, int dest_state_index, int c	void	O(1)	פונקציות המוסיפות סוגים שונים של פונקציות מעברים ב-DFA

symbol_table

symbol_table.h

SYMBOL_TABLE_ENTRY_STRUCT		
שם משתנה	סוג	הסבר
name	char *	שם הערך של הרשומה
type	int	סוג הרשומה
value	void *	ערך הרשומה
declaration_type	entry_type_E	סוג הצהרה
next	struct SYMBOL_TABLE_ENTRY_STRUCT *	הרשומה הבאה

	SYMBOL_TABLE_STRUCT	
הסבר	סוג	שם משתנה
גודל הטבלה	size_t	size
הגודל המקסימלי של הטבלה	size_t	capacity
ה-load factor של ה-symbol table	float	load_factor
מערך דינמי השומר את הרשומות	symbol_table_entry_T **	buckets
פונקציית ה-hash של ה-symbol table	unsigned int (*)(char *, size_t length)	hash

symbol_table_tree.h

	SYMBOL_TABLE_TREE_NODE_STRUCT	
הסבר	סוג	שם משתנה

table	symbol_table_T *	ה-symbol table של הצומת
children	struct SYMBOL_TABLE_TREE_NODE_STRUCT **	ילדי הצומת
n_children	size_t	מספר ילדי הצומת

	SYMBOL_TABLE_TREE_STRUCT	
שם משתנה	סוג	הסבר
root	symbol_table_tree_node_T *	שורש עץ ה-symbol table

שם פונקציה	קלט	פלט	יעילות	תיאור
init_symbol_table_tree	symbol_table_tree_node_T *root	symbol_table_tree_T *	O(1)	מאתחל את עץ ה-symbol table
init_symbol_table_tree_node	symbol_table_T *table, symbol_table_tree_node_T **children, size_t n_children	symbol_table_tree_T *	O(1)	מאתחל צומת בעץ ה-symbol table
init_symbol_table_tree_leaf	symbol_table_T *table	symbol_table_tree_T *	O(1)	מאתחל עלה בעץ ה-symbol table
symbol_table_tree_node_add	symbol_table_tree_node_T *sttn,	void	O(1)	מוסיף ילד לצומת בעץ ה-symbol table

			symbol_table_tree_node_T *sttn_child	
symbol_table_tree_print	symbol_table_tree_T *stt	void	O(n)	מדפיס את עץ ה-symbol table

מודולים ראשיים

io

שם פונקציה	קלט	פלט	יעילות	תיאור
read_file	const char *filename	char *	O(n)	קורה את הערך בתוך קובץ
write_file	const char *filename, const char *content	void	O(1)	כותב לקובץ
write_file_append	const char *filename, const char *content	void	O(1)	מוסיף לקובץ
print_to_stdout	char *content	void	O(1)	כותב ל-stdout
print_to_stderr	char *content	void	O(1)	כותב ל-stderr
get_new_filename	const char *filename, const char *new_name	char *	O(1)	מחזיר את שם הקובץ המתאים

lang

שם פונקציה	קלט	פלט	יעילות	תיאור
init_quest	const char *filename	quest_T	O(1)	מאתחל את המבנה המחזיק מבנים חשובים

quest

	QUEST_STRUCT	
שם משתנה	סוג	הסבר
srcfile	char *	שם קובץ תכנית המקור
destfile	char *	שם קובץ תכנית המטרה
src	char *	תכנית המקור
lexer	lexer_T *	הלקסר
parser	parser_T *	הפרסר
sdt	sdt_T *	ה-sdt
code_gen	code_gen_T *	יוצר הקוד

שם פונקציה	קלט	פלט	יעילות	תיאור

מקמפל	O(n)	void	quest_T *q	compile
מקמפל קובץ	O(n)	void	const char *filename	compile_file

התוכנית הראשית

```
function main()
    input_file = get_input_file_name()
    output_file = get_output_file_name()

    try
        source_code = read_source_code(input_file)

        tokens = tokenize(source_code)  // Break code into tokens
        syntax_tree = parse(tokens)  // Analyze grammatical structure

        target_code = generate_code(syntax_tree)  // Generate target machine code
        write_output(target_code, output_file)

    catch error as e
        report_error(e)

end function
```

מה התוכנית הראשית עושה

- בשורות 1 ו-2 התוכנית מקבלת את שמות קבצי הקלט והפלט.
- שורה 6 קוראת את קוד המקור, ומכניסה את הקוד לתוך משתנה כמחרוזת של תווים.
- שורה 8 קוראת לפונקציה tokenize, כלומר למנתח המילוני, ומכניסה לתוך המשתנה tokens את כל האסימונים המעובדים מהניתוח.
- שורה 9 קוראת לפונקציה parse, כלומר למנתח התחבירי והסמנטי, המקבלת תעבורה של אסימונים, ומכניסה לתוך המשתנה syntax_tree עץ תחביר.
- שורה 11 קוראת לפונקציה generate_code, כלומר ליוצר קוד הביניים והמטרה, ויוצרת את הקוד.
- שורה 12 כותבת את קוד המטרה לקובץ הפלט.
- שורה 5 ו14 אחראיות לתפוס כל שגיאה, ולתפל בא כראוי, כלומר היא מתנהגת כמו הerror handler
- בנוסף, כל הפונקציות השתמשו בsymbol table.

מדריך למשתמש

רפלקציה

ביבליוגרפיה

קוד הפרויקט