THE DO PROGRAMMING LANGUAGE COMPILER

עידו הירש

ת.ז. 214290249

מנחים: ד"ר נילי נוה ומיכאל צ'רנובילסקי

תאריך: 11.11.11

תוכן עניינים

מבוא

מטרה

לבחון את הידע והכישורים שלי בפרויקט בסדר גודל כזה. לפתח אלגוריתמים חכמים ויעילים אשר יפתרו את הבעיות האלגוריתמיות השונות העולות בפרויקט זה, תוך לימוד עצמי של ידע חדש וצבירת ניסיון בנושאים שלא התעסקתי בהם בעבר, כמו מכונות מצבים, עיצוב שפה, ועוד.

תיאור המערכת

המערכת הינה מהדר, Compiler, אשר מתרגם מסמך טקסט המכיל קוד בשפה שאני עיצבתי, Do, לקוד אסמבלי 32 ביט.

שפת התכנות היא הגדרה של חוקים תחביריים וסמנטיים, שנועדו להגדיר תהליכי חישוב שיבוצעו על ידי המחשב. הגדרת שפת התכנות היא חלק בלתי נפרד מבניית המהדר – המהדר עושה שימוש בהגדרת השפה כדי לנתח את קטע הקוד שנקלט וכדי בסופו של דבר לייצר את תוכנית היעד.

כאשר מגדירים שפת תכנות, מתייחסים כאמור לשלושה מישורים: האחד מילונאי, השני תחבירי והשלישי לשוני.

המישור המילונאי מגדיר אילו מילים שייכות לשפה, ואילו לא. לדוגמא, המילה if היא מילה המקובלת בשפת C בעוד שהמילה Hel#@l0 איננה.

המישור התחבירי מגדיר אילו רצפי מילים של השפה הם חוקיים, ואילו הם לא. לדוגמא, הרצף int x = 3; הוא רצף חוקי בשפת C, בעוד שהרצף if x is 5 then, איננו חוקי בשפת C.

המישור הלשוני מתייחס למשמעות רצפי המילים, והוא מגדיר חוקים כלליים שחייבים להתקיים בכל רצף מילים בשפה. לדוגמא, חובת ההצהרה – לפני שימוש במשתנה, קיימת חובה להצהיר עליו.

ספר השפה Do

THE DO PROGRAMMING LANGUAGE

Written by Ido Hirsh

FIRST EDITION

הקדמה

קצת על Do

מקור שמה של שפת Do מגיע מקיצור שמי, Ido Hirsh, ומהמילה "תעשה!" באנגלית, מילה המעוררת מוטיבציה לעבודה ועשייה.

שפת Do דומה בסינטקס שלה לשפות התכנות C ו – ++C.

מה יהיה בספר השפה

בספר השפה תתואר שפת התכנות Do. יתוארו אבני השפה, תכולת השפה, ודקדוק השפה.

אבני השפה

קבועים – Constants

קבוע הוא ערך המופיע ישירות בקוד התוכנית.

קבוע יכול להיות מטיפוסים שונים – מספר שלם, תו.

טיפוסו של הקבוע נקבע על ידי המהדר (Compiler) בהתאם לערכו.

דוגמא: *int x = **81***

בדוגמא שלעיל 81 הוא קבוע, אשר יובן על ידי ה - Compiler כקבוע מטיפוס int, מספר שלם.

טיפוסי קבועים

- קבוע מטיפוס מספר שלם - **int**.

על מנת להגדיר קבוע מסוג int נכתוב את ערכו ישירות בקוד התוכנית:

  - דוגמא:

*a / **2***

בדוגמא זו 2 הוא קבוע ו – a הוא שם משתנה כלשהו.

- קבוע מטיפוס תו - **char**.

הגדרת קבוע מסוג char תהיה בתוך שני גרשים בודדים:

  - דוגמאות להגדרה:

char ch = '5'

הערך של **התו** 5 ייכנס אל תוך המשתנה ששמו ch.

משתנים – Variables

משתנה מייצג מקום בזיכרון בו אפשר לשמור ערכים.

מקום זה בזיכרון מיוצג על ידי שם המשתנה (Variable-name), שנקרא גם מזהה (Identifier).

שמות משתנים

1. שם משתנה הוא רצף של אותיות בשפה האנגלית, ספרות, והתו '_' (Underscore).
   רצף זה חייב להתחיל באות בשפה האנגלית או בתו '_'.
2. אורכו של מזהה אינו מוגבל.
3. אין להשתמש במילים שמורות כמזהים.
4. קיימת הבחנה בין אותיות גדולות וקטנות (Case sensitive).

טיפוסי משתנים

לכל משתנה בשפה do יש גם טיפוס (Data-type) אשר מציין את סוג הערכים שהוא יכול להכיל.

ישנם שני סוגים של טיפוסי משתנים:

- **int** – משתנה מטיפוס מספר שלם.
  - מכיל ערכים מסוג מספרים שלמים. 1, 15-, 79, 0 וכו'.
- **char** – משתנה מטיפוס תו.
  - מכיל ערכים מסוג תו. a, g, 0, 7, f וכו'.

הגדרת משתנים

הגדרה כללית של משתנה:

<Data-type> <Identifier>;

דוגמאות:

```
int x;
char c;
```

ביטויים – Expressions & Statements

Expression

יחידה תחבירית בשפת תכנות שניתן להעריכה על מנת לקבוע את ערכה. שילוב של אחד או יותר קבועים, משתנים, פונקציות, אופרטורים (Operators), ו – Expression נוספים, שהשפה מפרשת (לפי הכללים של קדימות ושיוך), ומחשבת כדי לייצר ("להחזיר") ערך. תהליך זה, עבור ביטויים מתמטיים, נקרא הערכה (Evaluation).

בפשטות, הערך המתקבל הוא בדרך כלל אחד מהסוגים הפרימיטיביים השונים, כמו ערך מספרי, ערך בוליאני וכו'.

דוגמאות ל – Expressions:

- *3 + 15*
- *4*
- *(x – 5) / y*
- *7 <= 22*
- *X % 2 == 0*
- *(x + 15) < 3 * (y – 4)*

Statement

יחידה תחבירית בשפת תכנות המבטאת פעולה כלשהי שיש לבצע. תכנית הנכתבת בשפה כזו נוצרת על ידי רצף של אחד או יותר Statements.

בשונה מ – Expression, Statement לא מוערכת לכדי ערך.

ל - Statement יכולים להיות רכיבים פנימיים (למשל Expressions).

דוגמאות ל – Statements:

- תנאים – *if, else*
- לולאות – *while*
- הצהרה על משתנה – *int x;*
- השמת ערך למשתנה – *x = 4;*

אופרטורים – Operators

חשבוניים

- חיבור - **+**
- חיסור - **–**
- כפל - *****
- חילוק - **/**
- שארית - **%**

לוגיים

- שווה ל - **==**
- גדול מ - **>**
- קטן מ - **<**
- גדול או שווה ל - **>=**
- קטן או שווה ל - **<=**
- not - **!**

קשרים לוגיים

- or - **||**
- and - **&&**

תכולת השפה

בחלק זה תתואר תכולת השפה ואיך כל חלק בשפה נכתב בצורה נכונה מבחינה דקדוקית.

כל פקודה בשפה Do תסתיים עם נקודה פסיק ;

למעט תנאים, לולאות, ו – { }

השמה

כפי שציינתי לעיל משתנה הוא מקום בזיכרון בו אפשר לשמור ערך. השמה מאפשרת לנו לשמור את הערך הרצוי במקום זה בזיכרון.

הערך יכול להיות קבוע / משתנה / ביטוי (Expression).

סימול של השמה מבוצע באמצעות סימן השווה - =

- הגדרה כללית להשמה:

<Identifier> = <Expression>;

על מנת שההשמה תהיה חוקית, טיפוס המשתנה אליו עושים את ההשמה, כלומר ה - <Data-type>

של ה - *<Identifier>* צריך לתאום לטיפוס הערך המושם, כלומר ה - <Data-type> של ה - <Expression>.

- דוגמאות:

num = -17;
ch = 'h';

תנאים ולולאות

תנאים ולולאות הם חלקי קוד המתבצעים כתלות באם ביטוי מסוים הוא אמת או שקר.

תנאים – Conditions

לתנאי יכולים להיות שני חלקים:

- if
- else

אם הביטוי נותן תוצאת אמת, הקוד שבחלק של ה – if יתבצע.

ואם נותן תוצאת שקר, הקוד שבחלק של ה – if לא יתבצע.

אם יש חלק של else, הוא יתבצע כאשר הביטוי נותן תוצאה שקרית.

חלקים אלו של ה – if וה – else יתבצעו 0 או 1 פעמים.

בכל מקרה, לאחר ביצוע התנאי התוכנית תמשיך לקוד שנמצא אחריו.

- דוגמא להגדרת תנאי בעזרת שימוש ב – if בלבד:

```
if (<Expression>)
{
        Do if <Expression> is True
}
...
```

- דוגמא להגדרת תנאי בעזרת שימוש ב – if ו - else:

```
if (<Expression>)
{
        Do if <Expression> is True
}
else
{
        Do if <Expression> is False
}
...
```

לולאות – Loops

לולאה דומה מאוד במבנה שלה לתנאי, if, אך ההבדל היחיד הוא שחלק הקוד שבתוך הלולאה מתבצע **כל עוד** התנאי תקף (כל עוד הביטוי נותן תוצאת אמת), ולאו דווקא פעם אחת או 0 פעמים. כלומר לולאה יכולה להתבצע מספר רב של פעמים.

- while

דוגמא כללית להגדרת לולאה בעזרת שימוש ב - while:

```
while (<Expression>)
{
        Do while <Expression> is True
}
...
```

True & False

- False - false הוא הערך 0.

False = 0

- True – true הוא כל ערך השונה מ – 0.

True != 0

דקדוק השפה

לאחר שהגדרתי את אבני השפה ותכולת השפה, כעת אגדיר את תחביר / דקדוק השפה. ה – Grammar של השפה.

מהי שפה?

שפה היא אוסף המשפטים שמצייתים לחוקים המוגדרים בתחביר של השפה. משפטים אלו מורכבים מהמילים / האסימונים (Tokens) המוגדרים בשפה.

תחביר השפה

תחביר השפה do, כמו רוב שפות התכנות, הוא תחביר חופשי הקשר (Context free grammar). הדקדוק מורכב מ – Terminals ו – Non-Terminals. הסימנים (Terminals) הם המילים (Tokens) שנקלטו כקלט מקטע הקוד, בעוד שהמשתנים (Non-Terminals) הם רצפי סימנים ומשתנים. תחביר השפה מוגדר באמצעות שילוב הסימנים והמשתנים, בכללים שנקראים כללי יצירה (Production rules). כללי היצירה בעצם מגדירים את המשתנים, באמצעות הסימנים המוגדרים בשפה ומשתנים אחרים.

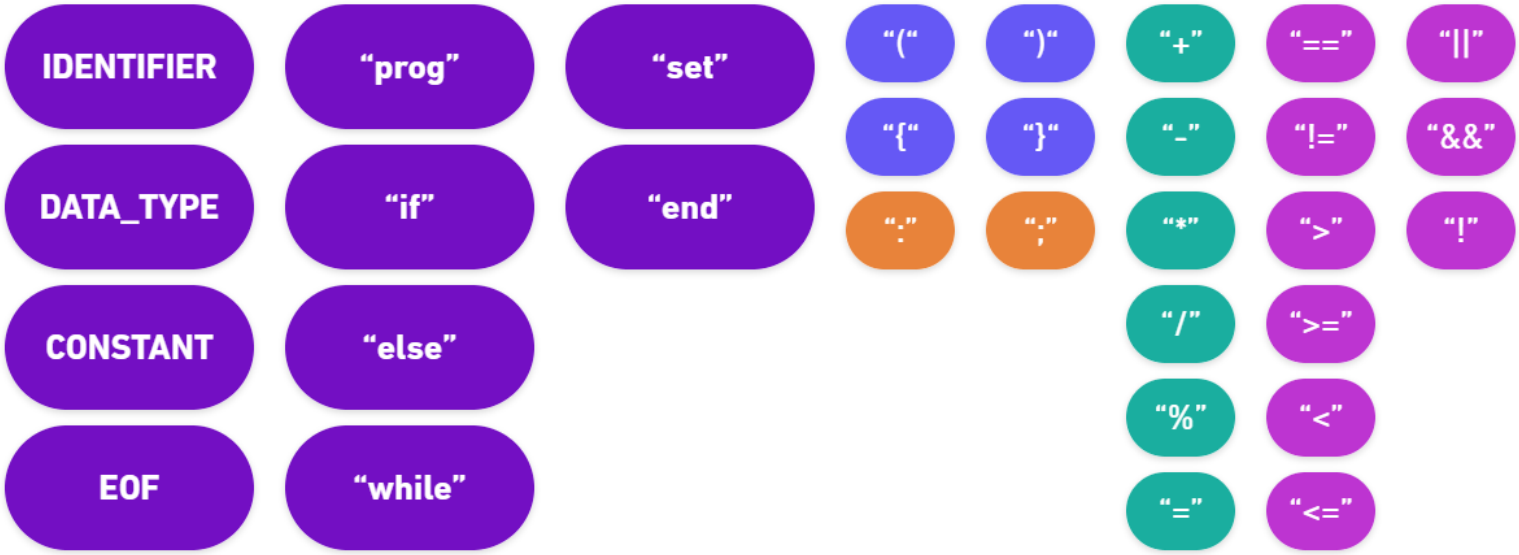
BNF

Backus-Naur Form היא צורת כתיבה פורמלית (Notation) עבור תיאור Grammars של שפות נטולות הקשר (Context free languages). צורת כתיבה זו משמשת לעיתים קרובות לתיאור שפות תכנות (שהן לרוב שפות נטולות הקשר).

BNF עוזר לכתוב בצורה חד-חד משמעית את כללי ה – Grammar של שפה מסוימת, באופן יחסית קל וקריא.

Tokens

להלן האסימונים, ה – Tokens של השפה Do:



רקע תאורטי

מהו קומפיילר

תוכנת מחשב אשר מתרגמת קוד מקור הכתוב בשפת תכנות אחת לקוד הכתוב בשפת תכנות אחרת, ללא שינוי המשמעות של קוד המקור. לרוב מתרגם משפה עילית (C, C++, Java), לשפת מכונה.

הקומפיילר גם מייעל ומשפר את קוד המקור כמה שניתן. כמו כן, מתריע על השגיאות / אזהרות שמצא, ומציע הצעות לפתרונות שלדעתו יפתרו שגיאות / אזהרות אלו.

Compiler vs. Interpreter

ישנם שני אופני עבודה עיקריים של קומפיילר: תרגום כל קוד המקור לכדי יחידת הרצה אחת (Compiler), או תרגום כל פקודה בנפרד בקוד המקור תוך כדי ריצת התוכנית (Interpreter).

כפי שצוין לעיל, Compiler, עובר על כל קוד המקור לפני הריצה, בודק את תקינותו, ומתרגם וממיר אותו ליחידת הרצה אחת בשפת מכונה. שפות שמתורגמות על ידי Compiler נקראות שפות מקומפלות. דוגמאות לשפות מקומפלות הן C, C++, Java ועוד.

בניגוד ל – Compiler, ה – Interpreter מתרגם וממיר כל פקודה בנפרד בקוד המקור לפקודות בשפת מכונה, תוך כדי ריצת התוכנית, ללא בדיקת תקינות הקוד לפני הריצה. שפות שמתורגמות על Interpreter נקראות שפות סקריפטים. דוגמאות לשפות אלו הן Python, JavaScript ועוד.

למה נדרש קומפיילר

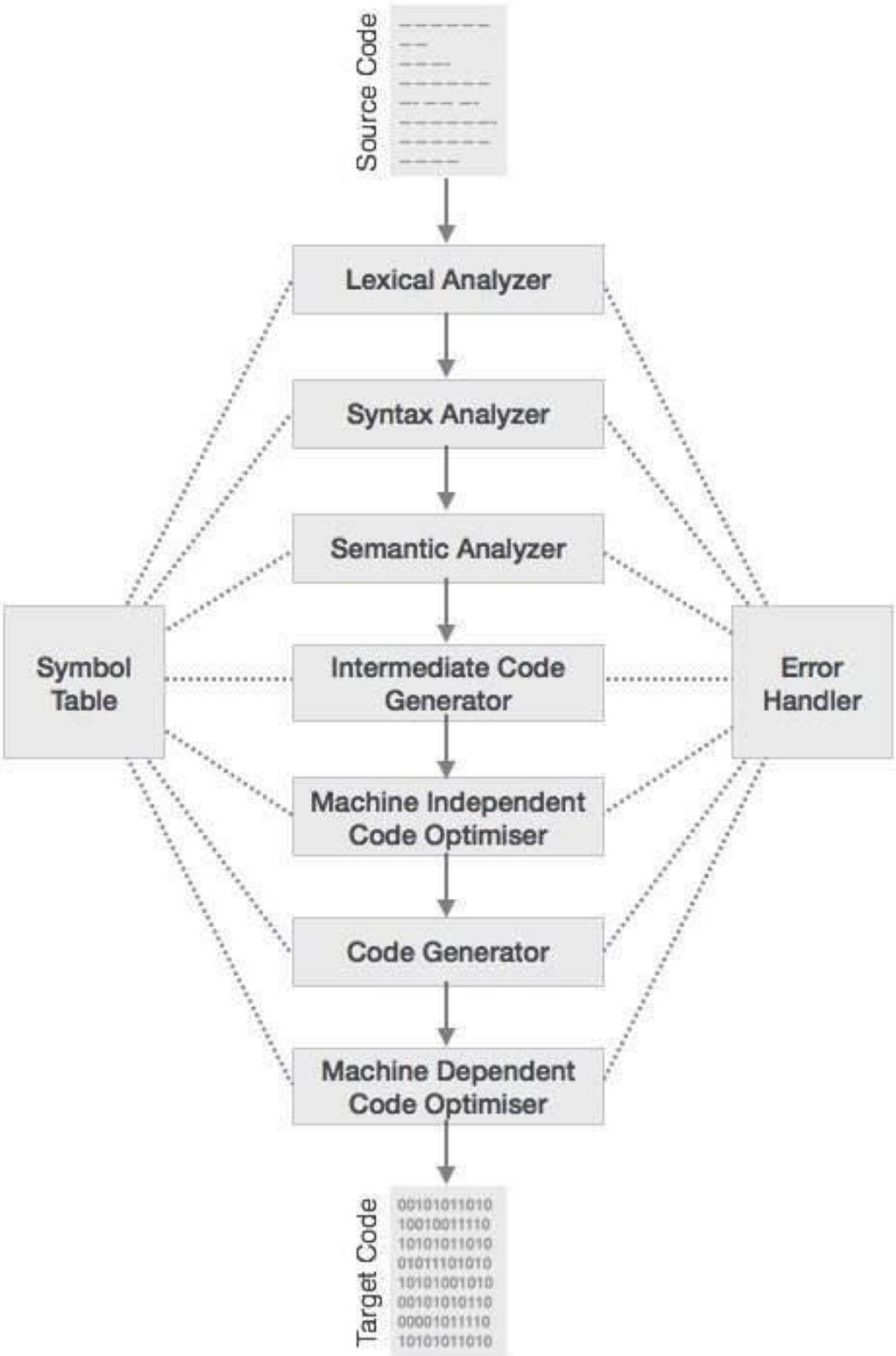
מכיוון שלבני האדם קל יותר לכתוב קוד בשפות תכנות עיליות אשר יותר קרובות אליהם (יותר קרובות לאנגלית), מאשר לכתוב קוד בשפת מכונה, אנו משתמשים בשפות עיליות אלו לכתיבת קוד. אך המחשב אינו מבין שפות עיליות אלו, הוא מבין רק קוד בשפת מכונה.

בשביל לגשר על הפער בין בני האדם לבין המכונה, צריך קומפיילר. שיתרגם את מה שאנו מתכוונים כאשר אנו כותבים קוד בשפה עילית לשפה שהמחשב יבין, שפת מכונה.

כיצד עובד קומפיילר

תהליך הקומפילציה הוא תהליך מורכב, ולכן מוטב לחלק אותו לשלבים. הקומפיילר עובד כך שכל שלב מקבל כקלט את התוצאה של השלב הקודם.

להלן דיאגרמה של השלבים:



שלבי הקומפיילר

כפי שציינתי לעיל תהליך הקומפילציה הוא תהליך מורכב, ולכן מוטב לחלק אותו לשלבים. מקובל לחלק כל שלב למודול עצמאי.

להלן חלוקה נפוצה של מודולים:

Lexical analysis

השלב הראשון בתהליך הקומפילציה הוא הניתוח המילונאי, ה – lexer.

מטרתו של שלב זה הוא לעבור על קוד המקור (source code) שהוא בעצם אוסף של תווים בתוך קובץ, ולהוציא מאוסף תווים זה טוקנים, tokens, הנמצאים בשפה. הטוקנים הם אבני השפה. לדוגמא המילה השמורה int בשפת c, או קבוע מסוג מספר שלם, או שם של משתנה כלשהו.

כבר בשלב הראשון של הקומפילציה יכולות להיווצר שגיאות. סוג השגיאות שיתפסו כאן יהיו שגיאות מילוניות. דוגמא לטעות מילונית בשפת C: int x = 1q2;

התו q לא יכול להימצא באמצע הגדרת מספר. ולכן תוצג שגיאה.

ה – lexer מעביר את ה – tokens לשלב הבא בקומפילציה, ה – parser.

Syntax analysis (Parsing)

השלב השני בתהליך הקומפילציה הוא הניתוח התחבירי, ה – parser.

מטרתו של שלב זה הוא להבין, מתוך הטוקנים שה – lexer מספק לו, האם הקוד שהמשתמש כתב, תקין מבחינה תחבירית בשפה. ה – parser עובד על פי ה – grammar של השפה אותה הוא מקמפל. הוא בודק לפי חוקי ה - grammar האם הקוד הנתון יכול להתקבל כקוד תקין בשפה. ה – grammar לרוב הוא context-free grammar.

גם בשלב זה של הקומפילציה יכולות להיווצר שגיאות. סוג השגיאות שיכולות להיתפס כאן הן שגיאות תחביריות.

דוגמא לטעות תחבירית בשפת C: x int = 5;

על פי הגדרת השפה של שפת C על מנת להגדיר משתנה צריך לכתוב את טיפוס המשתנה, אחריו שמו, ואז אם רוצים אפשר לעשות השמה של ערך. כיוון שהדוגמא לעיל לא תואמת ל – grammar של השפה, תיווצר שגיאה תחבירית.

שלב זה יוצר את ה – syntax tree של התוכנית. ה – syntax tree (נקרא גם abstract syntax tree), הוא בעצם ארגון רצף הטוקנים שמגיעים מה – lexer לתוך מבנה מאורגן בצורת עץ.

העץ נבנה על פי חוקי ה – grammar. שלב זה משמיט חלק מאבני השפה, לדוגמא סוגריים, מכיוון שהמבנה של העץ עצמו אומר לנו מה סדר הפעולות לביצוע בשלבים הבאים.

לאחר ששלב זה מסתיים, ה – syntax tree יועבר לשלב הבא, הניתוח הסמנטי.

Type checking / Semantic analysis

השלב השלישי בתהליך הקומפילציה הוא הניתוח הסמנטי.

הוא מקבל את ה – syntax tree מהשלב הקודם (parser) ומטרתו העיקרית היא לבדוק את הרצף הלוגי של התוכנית.
האם יש חוסר תאימות בין סוגי משתנים? האם יש שימוש במשתנה שלא הוכרז?

לכן דוגמא לשגיאות שיכולות להיווצר בשלב זה הן שגיאות של חוסר התאמת טיפוסים, שימוש במשתנה לא מוגדר וכו'.

שלב זה מפיק בסופו של דבר את עץ הניתוח, שהוא העץ התחבירי רק מפושט יותר, לאחר בדיקה של התאמת משתנים, שימוש במשתנים לא מוכרזים וכו'.

הבדיקה הסמנטית היא בדיקת הקלט האחרונה בתהליך הקומפילציה, ולכן עץ הניתוח שנפלט ממנה מייצג תוכנית תקינה.

Intermediate code generation

השלב הרביעי בתהליך הקומפילציה הוא שלב יצירת קוד הביניים.

שלב זה מקבל את ה – semantic tree מהשלב הקודם, כך שבשלב זה ניתן לדעת שהתוכנית תקינה.

לאחר ניתוח סמנטי, המהדר יוצר קוד ביניים של קוד המקור עבור מכונת המטרה. קוד זה מייצג תוכנית עבור מכונה מופשטת כלשהי. בין השפה העילית לבין שפת המכונה. יש ליצור קוד ביניים זה בצורה כזו שתקל על התרגום לקוד מכונת היעד.

Machine independent code optimization

שלב זה מקבל את הקוד שיוצר בשלב הקודם, ומטרתו לשפר וליעל אותו. כמו סידור מחדש של שורות קוד, והסרת שורות קוד לא נחוצות, כדי לבזבז כמה שפחות משאבים, ולהפוך את הקוד ליעיל יותר הן מבחינת זמן והן מבחינת מקום.

Code generation

שלב זה לוקח את קוד הביניים מהשלב הקודם ומתרגם אותו לקוד בשפת מכונה.

Machine dependent code optimization

השלב האחרון בתהליך הקומפילציה, מטרתו של שלב זה היא לשפר ולייעל את קוד המכונה שנוצר בשלב הקודם. בדומה ל – machine independent code optimizer, מטרתו להפוך את הקוד ליעיל יותר הן מבחינת זמן והן מבחינת מקום.

נוסף על כך, ישנם עוד מודולים שכיחים:

Register allocation

לתוכנית יש מספר ערכים שהיא צריכה לשמור במהלך הריצה שלה. ייתכן שארכיטקטורת מכונת היעד לא תאפשר לכל הערכים להישמר בזיכרון המעבד, או ה – registers. השלב של ה – machine depended code generator, מחליט אילו ערכים לשמור ב – registers, ואילו registers ישמרו ערכים אלו.

Assembly, linking and loading

אסמבלר מתרגם שפת אסמבלי לשפת מכונה. הוא יוצר מקובץ asm שמכיל שפת אסמבלי, קובץ object. קובץ object מכיל הוראות בשפת מכונה, כמו גם המידע הדרוש על איפה צריך לשים את ההוראות האלה בזיכרון.

לינקר מחבר כמה קבצי object לקובץ exe אחד.

כל הקבצים שהוא מחבר יכולים להיות מקומפלים על ידי אסמבלרים שונים.

משימתו העיקרית של הלינקר היא לקבוע את המיקום בזיכרון של כל אחד מהקבצים בעת הטעינה שלהם לזיכרון (על ידי ה – Loader), כך שההוראות מקבצי ה – obj השונים יתבצעו בסדר הגיוני בעת הריצה.

ה – loader הוא חלק ממערכת ההפעלה שאחראי על הטעינה של קבצי הרצה (exe) לזיכרון, והביצוע שלהם.

הוא מחשב את גודל התוכנית ומקצה לה מספיק מקום בזיכרון. הוא גם מאתחל מספר רגיסטרים שונים שיתחילו את תהליך הביצוע/הרצה של התוכנית.

Symbol table

ה – Symbol table או טבלת הסמלים, מכילה רשומה עבור כל Identifier (מזהה) עם שדות עבור התכונות של אותו המזהה.

טבלה זו עוזרת לקומפיילר למצוא רשומה של מזהה כלשהו בתוכנית ולקבל את הפרטים עליו באופן מהיר יחסית.

ה – Symbol table עוזרת גם ב – Scope managment.

טבלה זו לוקחת חלק בכל אחד מהשלבים שצוינו לעיל, ומתעדכנת בהתאם.

Error handler

כפי שכתבתי בתיאור השלבים של הקומפיילר, בכל שלב ושלב בתהליך הקומפילציה יכולות להיווצר שגיאות.

בשביל כך יש את ה – Error handler.

השגיאות שמתגלות מדווחות ל – Error handler והוא מדווח, ומציג אותן חזרה למתכנת בתצורה של הודעה. אם לקומפיילר יש הצעה מסוימת לפתרון הבעיה, גם היא תוצג בהודעה.

מבנה לוגי של קומפיילר

ניתן לחלק את הקומפיילר לשלושה חלקים לוגיים: back end, middle end, front end.

Front end

עובר על קוד המקור ומנתח אותו.

בחלק זה משתתפים ה – Lexical analysis, ה – Syntax analysis (Parsing) וה – Semantic analysis.

Middle end

אחראי על ייעול הקוד על מנת לשפר את ביצועי הקו.

בחלק זה משתתפים ה – Intermediate code generation וה – Machine independent code optimization.

Back end

אחראי על ייעול ויצירת הקוד המובן לשפת מכונה.

בחלק זה משתתפים ה – Machine dependent code generation וה – Machine dependent code optimization.

תיאור הבעיה האלגוריתמית

בפרק זה אציג את הבעיות האלגוריתמיות העולות בכל שלב מרכזי בתהליך הקומפילציה, אנתח אותן ואתן דוגמאות.

ניתוח מילוני

כיצד נוכל לפרש בצורה חד משמעית את רצף התווים כ – Token מסוים? למשל, כאשר נראה את התו '=', נוכל להניח שמדובר בהשמה של ערך לתוך משתנה, אך מה אם מייד אחריו יופיע עוד פעם '=' ? נצטרך להתייחס לשני התווים כ – Token אחד המייצג השוואה.

ניתוח תחבירי

כפי שציינתי לעיל, המנתח המילונאי (Lexer) מזרים טוקנים, מילים תקינות הכלולות בשפה, אל המנתח התחבירי (Parser). ה – Parser צריך למצוא היגיון בסדר הטוקנים ולבנות ממנו עץ אשר ייצג תוכנית הגיונית.

על מנת לבנות עץ מדויק, יש להגדיר "נוסחאות" מדויקות שייצגו את השפה, וההיגיון שבה. וכפי שציינתי בפרק של תיאור השפה Do, מקובל לתאר נוסחאות אלו בצורת BNF.

מאחר ומדובר בעץ, ולכל צומת בעץ יש תכונות שונות ו – "ילדים" שונים זה מזה, צריך למצוא דרך לשמור על התכונות הייחודיות של כל אחד מהם, ועדיין לשמור על היכולת להסתכל עליהם כמכלול.

עוד אתגר מעבר להגדרת השפה, הוא "מצבים מקבלים". למשל ב – C#, מה קורה כאשר המצב הנוכחי הוא Statement, והטוקן הנוכחי הוא Identifier? איך נדע האם לצפות להשמה (Assignment), כמו "X = 3;", או לקריאה לפעולה של עצם כמו "X.Foo()"? אם המצב הנוכחי הוא הכרזה על משתנה, והטוקן הנוכחי הוא Identifier, איך נדע האם לצפות ל - Semicolon, או לפסיק?

איך נוכל לדעת תמיד למה לצפות באופן מדויק?

ניתוח לשוני

עד כה ראינו רצף מאוד הגיוני. המנתח המילונאי בודק טוקנים ומוודא שכולם בשפה, המנתח התחבירי מרכיבים מטוקנים אלו משפטים ובודק שמשפטים אלו תקינים בשפה. וכעת המנתח הלשוני צריך לקבל את התוכנית בצורת העץ התחבירי ולבדוק האם היא הגיונית. פה ייבדק הרצף הלוגית של התוכנית, האם יש שימוש במשתנה שלא הוכרז? האם יש חוסר תאימות בין סוגי משתנים?

בכדי לפענח את העץ התחבירי, נצטרך למצוא שיטה יעילה לעבור עליו, ולפשט אותו. כיצד נזהה את הטיפוס של כל צומת בעץ? כיצד נסרוק את העץ בצורה שתאפשר לנו לקבל את הערכים הנורשים מאחיו, הוריו וילדיו?

העץ לאחר הניתוח הלשוני נבדל מעץ הניתוח התחבירי בכך שהוא מכיל אך ורק את מה שנדרש על ידי המתרגם לתרגום הקוד. העץ התחבירי פשוט יותר גם מבנית וגם רעיונית. הוא ממוקד וללא צמתים מקשרים, ומטרתו היחידה היא לייצג את התוכנית במבנה שיאפשר בקלות יחסית לתרגמו לייצוג ביניים.

ושוב, כפי שציינתי כבר קודם, המנתח הלשוני (הבדיקה הסמנטית) היא בדיקת הקלט האחרונה בתהליך ההידור, ולכן העץ שנפלט ממנה מייצג תוכנית תקינה.

סקירת אלגוריתמים בתחום הבעיה

תהליך הניתוח התחבירי, ה – Parsing, הוא התהליך המשמעותי והמורכב ביותר מבחינה אלגוריתמית ורעיונית בתהליך הקומפילציה. כעת אציג שיטות שונות ואלגוריתמים שונים הנפוצים בשלב זה.

מונחים

כמה מונחים שאשתמש בהם בתיאור האלגוריתמים.

Derivation

בעברית, גִזרָה, היא בעצם רצף של Production rules, על מנת לקבל את מחרוזת הקלט.

במהלך תהליך ה – Parsing אנו בעצם מקבלים שתי החלטות עבור קלט מסוים:

1. החלטה על ה – Non-terminal אשר יוחלף.
2. ההחלטה על כלל הייצור, שבאמצעותו יוחלף ה – Non-terminal.

על מנת להחליט על איזה Non-terminal יוחלף בכלל הייצור, יכולות להיות לנו שתי אפשרויות:

Left-most Derivation

אפשרות זו קובעת כי תמיד ה – Non-terminal השמאלי ביותר הוא זה שיוחלף.

Right-most Derivation

אפשרות זו קובעת כי תמיד ה – Non-terminal הימני ביותר הוא זה שיוחלף.

דוגמא

<u>נתון ה - Grammar הבא:</u>

```
E → E + E
E → E * E
E → id
```

עבור מחרוזת הקלט *"id + id * id"* כך יראו שני סוגי ה – Derivation:

Left-most derivation

```
E → E * E
E → E + E * E
E → id + E * E
E → id + id * E
E → id + id * id
```

Right-most derivation

```
E → E + E
E → E + E * E
E → E + E * id
E → E + id * id
E → id + id * id
```

Left Factoring

אם יותר מ – Production rule אחד מתחיל באותה קידומת, אז ה – Parser לא יכול לבצע הכרעה באיזה מהחוקים הוא צריך לבחור בשביל לנתח את הקלט הנוכחי.

דוגמא

אם כלל ייצור מסוים נראה כך:

$$A \Longrightarrow \alpha\beta \mid \alpha\gamma \mid \ldots$$

המנתח לא יודע להחליט אחרי איזה חוק לעקוב, כיוון ששני החוקים מתחילים באותו Terminal (או Non-terminal).

על מנת להסיר בעיה זאת משתמשים בטכניקה שנקראת Left factoring.

Left factoring ממירה את ה – Grammar כך שלא יהיו חוסר הוודאויות האלו. היא עובדת כך שעבור כל קידומת שמשומשת יותר מפעם אחת יוצרים כלל חדש וההמשך של הכלל הישן משורשר לכלל החדש.

דוגמא

הכלל הקודם יוכל כעת להיראות כך:

$$A \Rightarrow \alpha A'$$

$$A' \Rightarrow \beta \mid \gamma \mid \ldots$$

עכשיו למנתח יש רק כלל אחד עבור הקידומת המסוימת הזאת, מה שמקל עליו לקבל החלטות.

Parsing Algorithms

על מנת ליצור Parse Tree עליו יתבסס תהליך הקומפילציה, ישנם כמה אלגוריתמים הנקראים Parsing Algorithms.
אלגוריתמים אלה מתחלקים לשני סוגים עיקריים.

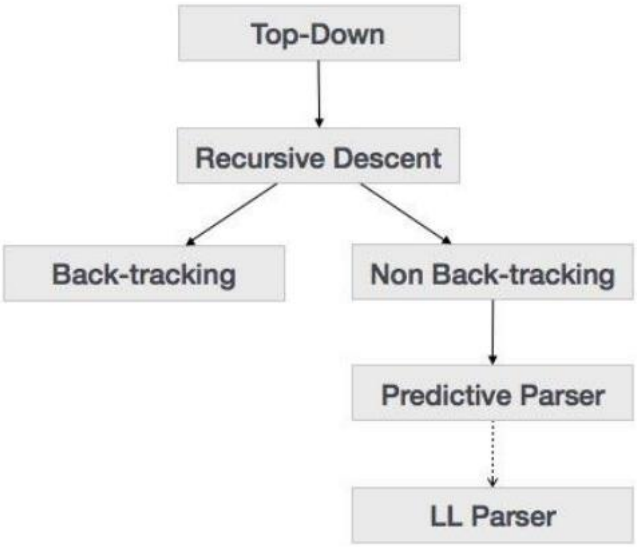
1. Top Down Parsing (TDP)
2. Bottom Up Parsing (BUP)

שפות תכנות הן בדרך כלל Context-free languages. נהוג לפרש CFL באמצעות מכונות מצבים, ובאופן יותר ספציפי מכונות מצבים המשתמשות במחסנית (Pushdown machines). לכן האלגוריתמים שכעת אציג ישתמשו באוטומט מחסנית לרוב, על מנת לבצע את פעולת ה – Parsing.

Top Down Parsing (TDP)

טכניקה בה עוברים מהחלקים העליונים לחלקים התחתונים של העץ התחבירי, על ידי שימוש בכללי השכתוב של Grammar השפה. עוברים מה – Grammar לקלט.

דיאגרמה המתארת מספר סוגים של Top Down Parsing:



כעת אציג ואסביר על כמה אלגוריתמים שמשתמשים בגישה של Top Down.

Definite Clause Grammar Parsers

Definite Clause Grammar (DCG) הוא דרך להביע תחביר של שפה, בין אם שפה טבעית או פורמלית. DCGs מזוהים בדרך כלל עם Prolog, שפת תכנות לוגית שבניגוד לשפות תכנות רבות אחרות, מיועדת בעיקר כשפת תכנות הצהרתית, כך שההיגיון של התוכנית מתבטא במונחים של יחסים, המיוצגים כעובדות וכללים. ביצוע "חישוב" בה מתבצע על ידי הפעלת שאילתה על היחסים הללו.

דוגמא ל – DCG ב – Prolog ←

```
sentence --> noun_phrase, verb_phrase.
noun_phrase --> det, noun.
verb_phrase --> verb, noun_phrase.
det --> [the].
det --> [a].
noun --> [cat].
noun --> [bat].
verb --> [eats].
```

Recursive Decent Parsing

צורה נפוצה של TDP. בשיטה זו, כיוון שהיא שיטה שמתבסת על הגישה של Top Down, עץ הניתוח נוצר מלמעלה למטה, והקלט נקרא משמאל לימין. שיטה זו משתמשת בפונקציות עבור כל Terminal ו – Non-terminal שנמצא ב – Grammar השפה. Recursive descent parser יוצר את עץ הניתוח תוך מעבר רקורסיבי על הקלט, מה שיכול לגרום לו לסבול מ – Back tracking. (האם יהיה או לא יהיה Back tracking תלוי ב – Grammar השפה, אם ה - Grammar הוא Left factored, הוא ימנע מ – Back tracking).

גרסה של Recursive decent parsing שלא משתמשת ב – Back tracking נקראת Predictive parsing.

Back tracking

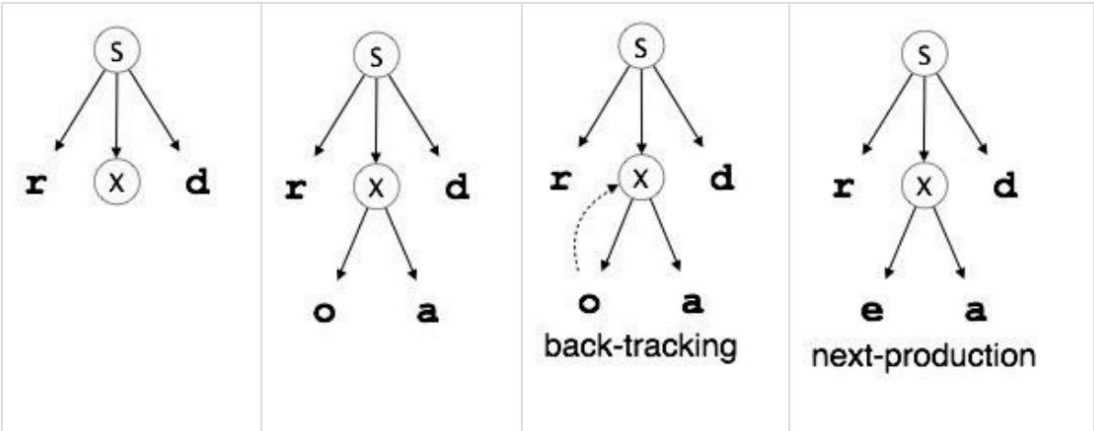
כאשר ה – Parser משתמש בשיטה של Recursive decent parsing (וה – Grammar הוא לא Left factored), ייוצרו מצבים במהלך הניתוח של הקלט בהם המנתח יגיע למבוי סתום, וזאת כנראה בגלל שעשה בחירה לא נכונה של כלל מסוים בדרך. לכן, המנתח חוזר חזרה למקום האחרון בו ביצע הכרעה, ושם בוחר באופציה האחרת. החזרה הזאת למקום האחרון בו ביצע הכרעה, על מנת לבצע הכרעה שונה, נקראת Back-tracking. רק כאשר ניסה את כל האפשרויות ולא הצליח להתאים את הקלט לכללי השפה, ניתן להבין שהקלט הוא לא תקין מבחינת השפה.

דוגמא

נתון ה – Grammar הבא:

```
S → rXd | rZd
X → oa | ea
Z → ai
```

עבור מחרוזת הקלט "*read*" כך יראה תהליך הניתוח:



Predictive parsing

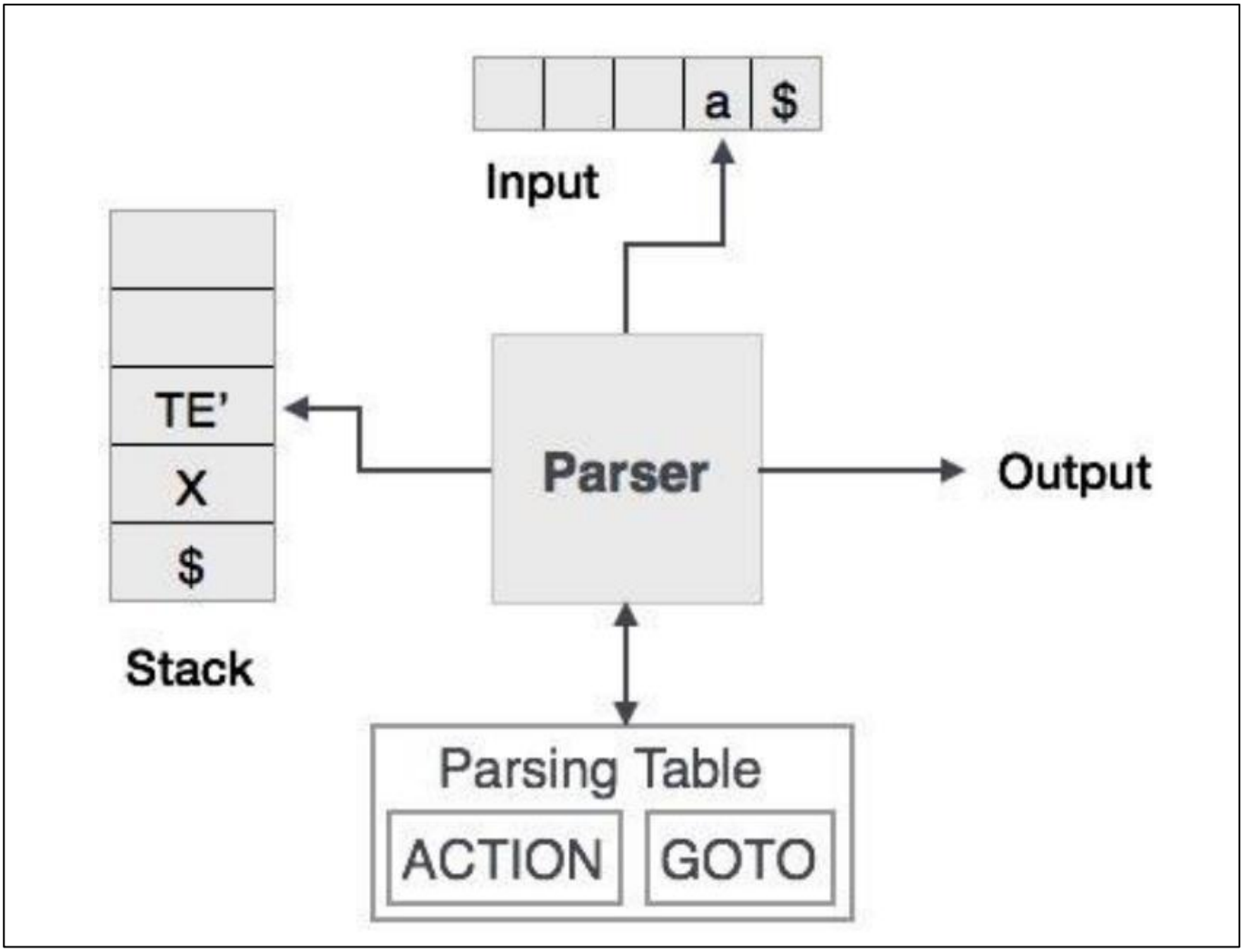
Predictive Parser הוא Recursive descent parser אשר יש לו את היכולת לחזות באיזה Production הוא צריך להשתמש בשביל להחליף את הקלט. עקב כך הוא לא סובל מ – Back tracking.

בשביל להשיג יכולת חיזוי זו, ה – Predictive parser "מציץ" לסמלים הבאים בקלט. בשביל שהוא יהיה ללא Back tracking ה - Predictive parser מגביל את ה – Grammar כך שהוא יכול להיות רק מתת-קבוצה של CFGs הנקראת LL(k) Grammars.

Predictive parser משתמש במחסנית (Stack) ובטבלת ניתוח (Parsing table) בשביל לנתח את הקלט ולייצר את עץ הניתוח. הוא פונה ומשתמש בטבלת הניתוח בשביל לקבל החלטה עבור כל צמד של קלט ואיבר במחסנית.

בניגוד ל – Recursive descent parsing שם עבור קלט מסוים יכולים להיות מספר כללים, ב – Predictive parsing יש לכל היותר כלל אחד עבור כל קלט מסוים. כך שבמקרים בהם אין אף כלל שתואם את הקלט, תהליך הניתוח נכשל.

דיאגרמה הממחישה את עבודתו של ה – Predictive parser

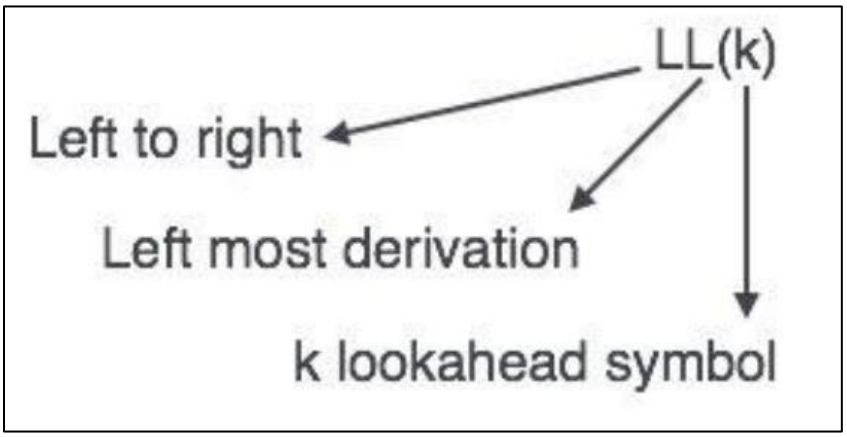


LL parser

LL parser מקבל LL grammars. כפי שציינתי לעיל, LL grammars הם תת-קבוצה של Context free grammars.

ניתן ליישם LL parsing באמצעות שני האלגוריתמים שהצגתי קודם, כלומר, Recursive descent או Predictive (באמצעות עזרה של טבלה).

LL parser נכתב גם כ – LL(k), כך שה – L הראשונה מייצגת שהקלט נקרא משמאל לימין, ה – L השנייה מייצגת left-most derivation, ו – k מייצג את מספר הסמלים עליהם "מציצים" קדימה (Look ahead).



Early parser

Early parser, נקרא אחר שמו של מי שהמציא אותו, Jay Early, הוא Parsing algorithm המשתמש בטכניקה של dynamic programming על מנת לנתח את מחרוזת הקלט.

האלגוריתמים הקודמים שתיארתי, לדוגמא Recursive descent, מבוססים על חיפוש רקורסיבי של מבנים תחביריים אפשריים אשר יקבלו את מחרוזת הקלט. שיטה זו של חיפוש יכולה לגרום לכך שחלקים מהמבנה התחבירי הכללי אשר מקבלים חלק מסוים ממחרוזת, מיוצרים שוב ושוב. החזרה הזאת על פתרונות חלקיים בתוך תהליך המבנה התחבירי הכולל, היא התוצאה של ה – Back tracking הדרוש בשיטת חיפוש זאת, מה שיכול להוביל לזמן ביצוע אקספוננציאלי של האלגוריתם. Dynamic programming נותן חלופה יעילה יותר בה החלקים שכבר יוצרו יישמרו לצורך שימוש חוזר בתהליך השלם של הניתוח. כך לא יהיה צורך לחזור על אותם חישובים שוב ושוב, מה שמייעל את האלגוריתם.

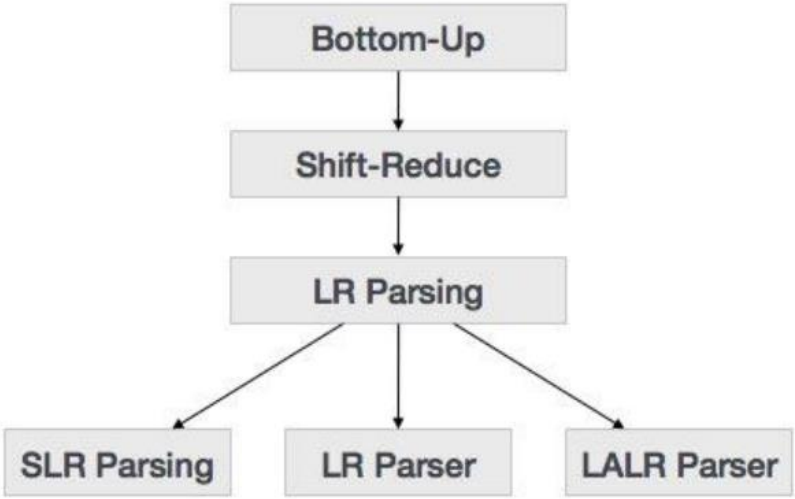
שמירה על פתרונות חלקיים אלו מתבצעת באמצעות מבנה נתונים הנקרא טבלה, *chart*. לכן גרסאות שונות של ה - Early parser נקראות גם *chart parsing*.

*** דוגמא ***

Bottom Up Parsing (BUP)

טכניקה בה עוברים מהחלקים התחתונים לחלקים העליונים של העץ התחבירי, על ידי שימוש בכללי השכתוב של Grammar השפה. עוברים מהקלט אל ה - Grammar.

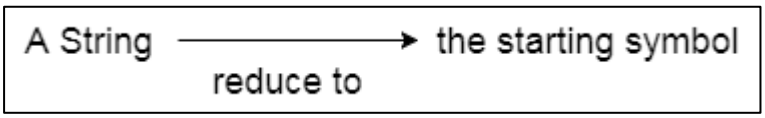
דיאגרמה המתארת מספר סוגים של Bottom Up Parsing:



כעת אציג ואסביר על כמה אלגוריתמים שמשתמשים בגישה של Bottom Up.

Shift Reduce

Shift Reduce הוא התהליך של הפחתת מחרוזת הקלט ל – Starting non-terminal של ה – Grammar.



שיטה זו משתמשת בשני שלבים הייחודיים ל – Bottom up parsing. שלבים אלו נקראים Shift step ו – Reduce step.

Shift step

שלב זה מבצע מעבר לסמל, Symbol, הבא שמגיע מהקלט, והוא נקרא גם Shifted symbol. סמל זה נדחף (PUSH) למחסנית. ה – Shifted symbol מטופל כצומת אחת של עץ הניתוח.

Reduce step

כאשר המנתח מוצא כלל ייצור שלם, כלומר הסמלים שעל המחסנית תואמים לאחד מה – RHS (Right Hand side) של כללי הייצור של ה – Grammar, ומחליף אותו ב – Non-terminal שנמצא ב – LHS של אותו כלל ייצור, שלב זה נקרא Reduce step. שלב זה בעצם עושה POP למחסנית עבור כל הסמלים שתואמים ל – RHS שמצא, ודוחף למחסנית את ה – LHS התואם.

דוגמא

נתון ה – Grammar הבא:

$S \rightarrow S+S$

$S \rightarrow S-S$

$S \rightarrow (S)$

$S \rightarrow a$

עבור מחרוזת הקלט a1-(a2+a3) כך יראה תהליך הניתוח:

Stack contents	Input string	Actions
$	a1-(a2+a3)$	shift a1
$a1	-(a2+a3)$	reduce by S → a
$S	-(a2+a3)$	shift -
$S-	(a2+a3)$	shift (
$S-(	a2+a3)$	shift a2
$S-(a2	+a3)$	reduce by S → a
$S-(S	+a3) $	shift +
$S-(S+	a3) $	shift a3
$S-(S+a3	) $	reduce by S → a
$S-(S+S	) $	shift)
$S-(S+S)	$	reduce by S → S+S
$S-(S)	$	reduce by S → (S)
$S-S	$	reduce by S → S-S
$S	$	Accept

LR Parser

LR Parser הוא Parser מסוג Bottom up, לא רקורסיבי המשתמש בשיטה של Shift Reduce. LR Parser ידוע גם כ – LR(k) Parser, כך שבדומה ל – LL(k) Parser, ה – L מסמנת קריאה של הקלט משמאל לימין (Left to Right), ה – R מסמנת Right most derivation, וה – k מסמן את מספר הסמלים שה – Parser "מציץ" עליהם קדימה (Look ahead).

*** דוגמא ***

Precedence Parser

בשמו המלא, Operator-Precedence Parser, הוא Parser פשוט המשתמש בשיטה של Shift Reduce, ומסוגל לנתח תת-קבוצה של LR(1) Grammars. ה – Precedence Parser משתמש ב – Precedence (קדימות) של האופרטורים בביטוי מסוים על מנת להחליט איך לנתח אותו.

דוגמא

כפל '*' קודם לחיבור '+', לכן עבור הביטוי 3 * 2 + 1, האלגוריתם ייחשב ראשית את הערך של 3 * 2, ורק אז יתווסף 1. כך התוצאה תהיה 7 כמו שהיא צריכה להיות, ולא 9 אם קודם היינו עושים 2 + 1, ורק אז מכפילים ב – 3.

CYK Parser

Cocke–Younger–Kasami algorithm, נקרא כך אחר שם של המגלים שלו, John Cocke, Daniel Younger, Tadao Kasami. אלגוריתם זה הוא אלגוריתם מסוג Bottom up והוא משתמש ב – Dynamic programming.

דוגמא

עבור המשפט 'abac' = w, הוא יבדוק אם ניתן לייצר:

1. a, b, c
2. ab, ba, ac
3. aba, bac
4. abac

כל בדיקה תתבצע בתקף על הבדיקה הקודמת.

האלגוריתם עושה זאת על ידי יצירת טבלה בגודל N * N, כאשר N הוא כמות המילים (אסימונים) במשפט, ומתבסס על התוצאה של השורה הקודמת.

יעילות האלגוריתם היא: $O(n^3)$

דוגמא לתהליך ניתוח של CYK Parser:

5 letters	C,S,A				
4 letters	-	A,S,C			
3 letters	-	B	B		
2 letters	A,S	B	S,C	A,S	
1 letter	B	A,C	A,C	B	A,C
	b	a	a	b	a

→ AB | BC

→ BA | a

→ CC | b

→ AB | a

1. ba, aa, ab, ba
2. baa (b U aa | ba U a), aab (a U ab | aa U b), aba (a U ba | ab U a)
3. baab (baa U b | b U aab | ba U ab), aaba (aab U a | a U aba | aa U ba)
4. baaba (baa U ba | ba U aba | b U aaba)

Recursive Ascent Parser

Recursive ascent parser היא טכניקה ליישום LALR Parser (Look Ahead LR Parser) שמשתמשת בפונקציות רקורסיביות, מאשר בטבלאות.

הוא עובד כך שכל פונקציה של ה – Parser מייצגת מצב מסוים אחד במכונת המצבים. בתוך כל פונקציה מתקבלת ההחלטה על איזה פעולה לעשות בהתאם ל – Token הנוכחי. ברגע שה – Token זוהה, הפעולה שתילקח מתבסס על המצב הנוכחי. יש שתי פעולות יסודיות שיכולות להילקח, Shift ו – Reduce.

*** דוגמא ***